Volume 23 (2004), number 2 pp. 173–187 COMPUTER GRAPHICS *forum*

Modeling Trees with Hypertextures

D. Strnad and N. Guid

Department of Computer Science, The University of Maribor, Maribor, Slovenia
damjan.strnad@uni-mb.si, guid@uni-mb.si

Abstract

In the paper, a new method for modelling trees at medium detail is presented. The method is based on a volumetric representation of trees, generated by an iterated function system (IFS). Alleviating the modeling restrictions of fractal techniques, extensions to the standard IFS are introduced. Practical aspects of modeling and rendering of trees, such as data structures and bounding volumes, are discussed. The advantages of the new method are described at the end together with some results.

Keywords: fractals, hypertextures, iterated function systems, trees, volume rendering.

ACM CCS: I.3.3 [Computer Graphics]: Picture/image generation; I.3.5 [Computer Graphics]: Computational geometry and object modelling; I.3.7 [Computer Graphics]: Three-dimensional graphics and realism.

1. Introduction

Realistic simulation of nature incorporates realistic modeling of its complex geometry. Among natural objects particularly trees present the challenge of modeler's creativity. Two groups of methods for modeling trees have been presented in the past. The first group is concentrated on veracious production of detail for close range views of trees [1,2]. The second one generates tree models for distant views and gives more attention to the overall model structure, fast generation, and/or compact representation [3–7].

Various modeling techniques are employed in these approaches. Weber and Penn use growth simulation, controlled through a set of real valued parameters [4], while Holton simulates the development of tree structure using strands [6]. Oppenheimer employs fractal generation techniques, in particular iterated function systems [8]. The biologically motivated rewriting systems or L-systems, introduced by Lindenmayer and Prusinkiewicz [7], present a well-established tree modeling technique. While they used turtle interpretation to bind L-systems to their geometric representation, Gervautz and Traxler adopted the CSG graphs for this task [5]. Traditional modeling also proved to form a powerful combination with rule-based graph models [2].

The model representation used so far was a set of geometric primitives [4,6], a set of higher-level components [2,7], a constructive solid geometry [5], or a system of particles [3].

In this paper, we present a new method for modeling trees using indirect hypertextures [9]. The method is based on a three-dimensional variation of an iterated function system (IFS) [10,11] as a generation process. We call the trees, that we form with our algorithm, the *hypertrees* to emphasize the hypertextural nature of resulting volume densities.

The organization of this paper is as follows. In Section 2, we briefly review the principles of hypertextures and IFS. In Section 3, we introduce hypertrees and describe extensions to the IFS needed for modeling them. These extensions increase the flexibility and versatility of the general IFS algorithm, what have been its major stumbling blocks in the past. Then, in Section 4, we describe the volumetric representation of hypertrees and discuss practical aspects of hypervolume storage within specialized data structures in Section 5. In Section 6, the complete modeling algorithm is given. Operations on hypertree volumes are the topic of Section 7, while Section 8 is devoted to the rendering of hypertrees using raymarching algorithm. After that, we address the problem of level of detail in Section 9. We proceed with the description of a software framework binding it all together in Section 10, and with a

Submitted May 2003
Revised November 2003
Accepted February 2004

short analysis of results in Section 11. We conclude by relating our technique with the approaches of other authors and investigate possible future enhancements in Section 12.

2. Motivation and Required Basics

Hypertextures [9] are a half-way representation between geometry and textures and facilitate the solution of some problems connected with the complex geometry models, like the level of detail or model compactness. During the last decade, hypertextures have become a principal tool for modeling and visualization of soft objects, primarily amorphous natural phenomena, such as fog, smoke and clouds. However, hypertextures have also proven to be appropriate for modeling objects with complex geometry (e.g. fur balls [12], knitted clothes [13]). Undoubtedly, trees fall into this category of objects.

Some trees can be represented as fractal objects with limited depth and many more trees can be adapted to that perception considering some limitations. Iterated function systems are a technique for fractal generation that has been successfully applied to model botanical structures in the past. There are known applications of IFS to produce directly rendered three-dimensional fern [14]. Also, Oppenheimer used IFS to combine twisted helixes into fractal trees, but his approach was geometry-based and designed for close-up views of trees [8]. Our approach instead uses IFS to produce intermediate hypertexture representation that can be afterward rendered using one of many volume rendering methods.

2.1. Hypertextures

The history of procedural volumetric modeling in the form of hypertextures starts with Perlin [9]. Often, there are misconceptions about the definition of hypertextures, especially about their relation to the volume densities. We treat hypertextures as procedural volume densities, because this is the most distinguishing difference in comparison to the classical method of acquisition of volumetric data by three-dimensional scanning (e.g. medical CT). Additionally, hypertextures may define other properties of volume elements besides density, such as color or normal for shading.

A hypertexture algorithm defines density at each point in three-dimensional space or its bounded subregion. Common algorithms for modeling soft objects do that directly for each infinitesimal point, where the calculations of density in different points are independent and can be carried out in an arbitrary order. This is possible because of the properties of underlying pseudo-random base functions, such as a noise and a turbulence. Their advantages are small space requirements and possible on-the-fly evaluation during rendering. The drawback is that a very limited group of hypertextures can be presented in such a direct way.

Indirect (sometimes also called explicit) hypertexture algorithms, on the other hand, have the recognizable property of being order-dependent, usually in an iterative manner. The complete volume representation must be calculated in advance and stored before rendering can begin. This can only be done for a finite resolution of the volume in question, so that the resulting volume data structure can be processed by the computer. The rendering time is therefore heavily dependent on fast access to the stored data, whereas the organization of the data structure affects the quality of results.

We need to emphasize the difference between fractal hypertextures used to model soft objects and fractal hypertrees. The former are fractal in their range, while the hypertrees are fractal in their domain. This is a reason why hypertrees cannot be independently point-evaluated by a direct algorithm but rather generated all-at-once by an indirect algorithm.

2.2. Fractals and Iterated Function Systems (IFSs)

Fractals were introduced to the computer graphics community by Mandelbrot [15]. Mathematically, fractals are shapes of real-valued geometrical dimension with the property of infinite self-similarity. This means that a fractal under magnification looks as if it is built up of a set of reduced copies of itself. Nature is overfilled with objects and phenomena with fractal properties. However, these objects and phenomena are only statistically self-similar and even that only for a limited depth. This holds for trees as well.

An IFS is a simple geometrical model for the representation of fractal objects. We review in short the major outlines of a classical IFS algorithm [10], concretized to the three-dimensional case of our interest.

Let $\{w_j\}$ be a set of m affine and contractive transformations, operating on bounded three-dimensional objects. Let X denote such an object. The transformation w_j is a contraction if and only if the following property holds for all pairs of the points $x, y \in X$:

$$d(w_j(x), w_j(y)) \leq c_j d(x, y),$$

where $c_j \in [0, 1)$ is the contractivity factor and $d(x, y)$ is the Euclidean distance between the points x and y.

An IFS is completely described by the set $\{w_j\}$ and a set of corresponding positive probabilities $\{p_j\}$. The exact shape of the fractal object, represented by any particular IFS, is the so-called *attractor* A, with the following property [14]:

$$A = \bigcup_{j=1}^{m} w_j(A).$$

According to the established terminology, we shall call an object, being subdue to any of transformations w_j, a *generator*. One way of visualizing the attractor is the random iteration algorithm [10,14], described next.

Let X_0 denote an initial generator. We obtain the transformed generator X_1 by randomly selecting a transformation w_k from the set $\{w_j;\ j = 1, \ldots, m\}$ using probabilities

$\{p_j\}$ and applying $X_1 = w_k(X_0)$. Let A_n be the union set of transformed generators after the composite application of n transformations:

$$A_n = \bigcup_{i=1}^{n} X_i,$$

where

$$X_i = w_k(X_{i-1}); \quad k = \text{rand}(1, m).$$

As the number n of compound transformations increases, the rendering of A_n converges to the shape of attractor. In practice, we limit the process of fractal generation by setting a finite number of iterations at which to stop. During each iteration, the transformation w_j is randomly chosen with probability p_j from the set and applied to the generator. When the process finishes, the constructed A_n is rendered.

The advantageous property of IFS is that a set of required transformations can be determined by observation of the sample fractal object and the use of so-called *collage theorem* [14]. According to the collage theorem, if we isolate the disjoint self-similar subparts that make up the fractal object and determine the transformations among them and the whole, we get the set of required IFS transformations. The mutual covering of subparts determines the quality of the fractal object realized by the IFS transformations.

3. Hypertrees

Trees in nature are composed of a trunk and a multitude of branches. Branches themselves are like smaller trees, consisting of even smaller trees. This similarity goes several levels deep, depending on the tree family. Branches need not be exact copies of the whole tree, but they may only closely resemble it. We are speaking of statistically self-similar fractals of a limited depth.

A limited set of tree families can be described by the deterministic and affine IFSs. However, the relaxation of restrictions on IFS transformations can yield a much wider range of modeling opportunities, producing objects that digress from pure fractals. Here, we describe the extensions needed for modeling hypertrees.

3.1. Nonlinear IFS

The transformations w_j can be composed of many elementary transformations. As long as the composite transformation is a contraction, there is no need to place affinity or contractivity restriction on component transformations. The use of nonlinear transformations can thus greatly enhance the modeling versatility.

The nonlinear IFS was used previously by Gröller [16]. He described the nonlinear effect of transformations as a distortion of a rectilinear grid. In this way, he introduced tapering and twisting into IFS.

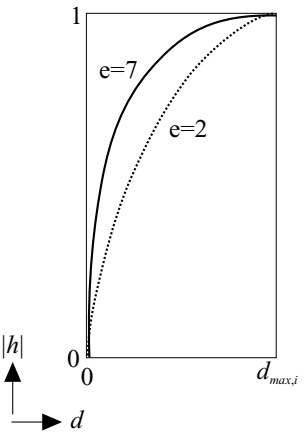

Figure 1: *Nonlinear shear.*

To model hypertrees, we rather use combinations of linear transformations (scalings, translations and rotations), and nonlinear shears. While scalings, translations and rotations are used to size, position and to orientate the fractal subparts, respectively, nonlinear shears are used to bend them in two coordinate directions.

The nonlinear shear uses three parameters—the maximum displacements in each of the bending directions and the bending exponent. The actual displacement of a generator is given by multiplying its "height" along the bending axis (i.e. its third coordinate) with the maximum displacement in the corresponding direction, so the generator with height 0 is not displaced at all, while the generator with some predefined maximum height is maximally displaced (see Figure 1). The displacement d between these endpoints is given with:

$$d = d_{\max,i}|h|^e, \quad h \in [0, 1],$$

where $d_{\max,i}$ is the maximum displacement in the coordinate direction $i \in \{x, y, z\}$, h is the generator's relative "height" with respect to the maximum height and e is the bending exponent.

Shears have magnifying scaling factors and must therefore be accompanied by appropriate downsize scalings to meet the contraction condition.

3.2. Nondeterministic IFS Transformations

To alleviate the determinism of an IFS transformation, the values of its parameters, such as the angle of rotation for a rotation, can be randomly chosen from some prescribed range around the mean value. However, parameter values for all randomized transformations must be determined prior to the beginning of generation process and remain constant until the end of it to prevent the hypertree from "breaking up." Random parameters allow the same set of transformations to produce visually different hypertrees with similar basic structure.

3.3. Limited Fractal Depth

Trees have limited fractal depth, in nature from 4 to 7. To generate the branching structure of a tree, we insert a *condensation* transformation with a condensation set forming the stem of the tree. The condensation transforms any given generator into the stem volume (usually a cone-like region), thereby acting as a reseting transformation. Two condensations may define the forked stem as overlapping sheared cones, producing the so-called dichotomous branching.

We use the condensation to limit the fractal depth of the tree by enforcing it after a certain number of iterations. Subsequent transformations will replicate the stem as branches. The same process essentially generates the whole tree including the leaves, which are only distinguished from the branching structure by different coloring. Therefore, the method is not applicable for generating trees with distinguished foliage at medium detail (e.g. palm trees). In this context, our method is similar to the method of Reeves [3], which models leaf layers as clouds of spherical particles. Even the standard geometrical approaches, that first generate the branching structure of the tree and then add the leaves in a separate step, use no highly sophisticated way to place the leaves, which are themselves usually modeled as simple triangles at medium detail.

Following the collage theorem, we may derive many different tree partitions, but two characteristic cases of definition of transformation sets are most recurrent. In the first case, each major branch is considered as a separate subpart of the whole tree, contributing one transformation to the set. This can be a complex task to perform for a modeler. The second case is to use approximate coverings that compromise for less accurate copies of the original. In the extreme, we may define the tree as a slightly reduced, raised and arbitrarily rotated copy of itself plus few imposing low branches. The advantage of this approach is that it demands much less effort from the modeler. Its disadvantages are poor versatility of shaping and smaller efficiency of rendering.

In the continuation, we refer to the number of transformations between the last enforced condensation and some transformation w_j as the *level of execution* or the *depth* of that transformation.

3.4. Level Dependent Self-Similarity

The central problem about shaping hypertrees is that any change to IFS transformation reflects throughout the tree structure, since, in a classical IFS, we only define the relations of the first level subparts to the whole tree and the relations are consistently reproduced on lower levels. A limited per-level control of hypertree shape can be achieved in two ways:

- using transformation constraints, and
- using transformations with variable environment-dependent parameters.

$$
\begin{array}{ccccccc}
w_1^1 & & & & & & \\
w_1^2 & w_2^1 & & & & & \\
w_1^3 & w_2^2 & w_3^1 & & & & \\
\dots & & & & & & \\
w_1^n & w_2^{n-1} & w_3^{n-2} & \dots & w_n^1 &
\end{array}
$$

Figure 2: *The pipeline of transformations.*

The constraining rules for transformations allow different levels of tree structure to dissent from each other, but all subparts of the tree on the same level are still similar. Examples of such a constraint are "dummy transformations," which transform the generator, but the resulting object does not contribute to the attractor set. A transformation can also be marked as dummy from certain level on, so the corresponding branch is generated but with a different depth than the rest of the tree.

The introduction of the last allowed level of execution is similar to the dummy level of transformation. The difference is that the branches corresponding to the transformation with a limited level of execution m are not generated at all on levels deeper than $m + 1$.

Transformations with variable parameters are a more powerful technique, because they distinguish the intralevel tree structure. The transformation parameter in this case can be the rotation angle, the shear exponent or any other transformation specific argument. The parameter value depends on the context in which the transformation is executed, such as the depth of transformation, the indices of precedent transformations, etc.

The employment of transformations with varying parameter values is not straightforward. Consider having two such transformations A^i and B^j with parameter values being functions of the level of execution, denoted by indices i and j. To perform the sequence of transformations ABA, we first need to execute the transformation A^1 (i.e. the transformation A with parameter values valid for the first level), then execute transformations A^2 and B^1 in that order and finally execute the sequence A^3, B^2 and A^1, registering the transformed generators between any two executions. In general, any sequence of transformations $w_1 w_2, \dots, w_n$, where n stands for the fractal depth of the hypertree, turns into n sequences, depicted schematically as seen in Figure 2.

In Figure 2, the upper index signifies the transformation's level of execution, so appropriate parameter values are used. In each row, the transformations are executed left-to-right and generators are registered at the end of each row.

If all of the transformations w_i have varying parameters with different values at all levels, then the number of executed transformations is $\frac{n(n+1)}{2}$ or $\frac{n+1}{2}$-times as many as with constant transformations. Fortunately, in practice, we rarely

define different parameter values except for the first two or three levels, which usually define the tree branches. If parameter values remain constant after the mth level, then only the last m transformations in each row of the above scheme need to be recalculated. This reduces the number of required executions to a factor m or less compared to the constant case.

3.5. Noise Perturbation

The too ideal structure of hypertrees can be distorted with a three-dimensional vector-valued noise or turbulence. We can inject the perturbations at various stages of hypertree modeling. The best solution is to perturb generator positions during IFS generation process. Perturbations can also be added as a post-process by applying them as a shift operation to the completed hypertree. The alternative solution is to shift the sampled points during rendering instead. This is similar to displacing samples when animating procedural textures [17]. All of these approaches produce the same effect—they form the continuously distorted branching structure.

4. Volume Density Representation of Hypertrees

Standard rendering methods of fractal attractors make an instant projection of transformed generators and draw pixels directly to the image buffer. Pixel color is proportional to its hit ratio [10] or distance from the viewpoint [16].

When generating hypertrees, we want to store the volume density representation of the hypertree. To do that, we enclose the hypertree object with a three-dimensional box called *hypervolume*. The hypervolume is partitioned into rectilinear grid of voxels with the grid resolution possibly different for each of the three coordinate directions. The hypertree resolution determines, how much detail we can capture in the representation. Values around 300 are used for an average quality.

The spatial span of any hypertree (i.e. its bounding box) is defined by used IFS transformations, but since that information is rarely known in advance, the hypervolume should be large enough to safely capture the complete hypertree during the IFS generation process. The later established exact bounding box may therefore be smaller, but if the hypervolume is closely superimposed, then the loss of resolution owing to marginal void space will be negligible.

A numerical information, attached to a voxel, represents the spatial density μ in a volume region belonging to that voxel. The range of values for the density could be set to [0, 1], so the floating point data type would be used for memory storage, requiring 4 bytes in single precision. For practical reasons, we have decided to use integer density values between 0 and 255, needing only one byte per voxel.

Before tree generation starts, all voxels are set to zero density. During the IFS iteration process, each generator transformation increments the density value of a hit voxel (or voxels,

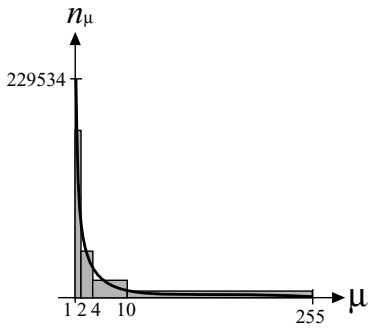

Figure 3: *Introduction of density classes.*

if the generator is not a point) by 1, limiting the maximum density value to 255. Higher number of iterations gives larger number of hit voxels with higher density. Such generation process is fast. The resulting volume density approximates the space occupancy of the hypertree, appurtenant to the underlying IFS model.

Any three-dimensional object may be used as a generator, but we decided to use points and line segments only. With true three-dimensional objects, such as a sphere, the problem of when to increase some voxels density must be solved. While points are a straightforward generator type, we use line segments to give the impression of oblong leaves. The length of a line segment must therefore be preserved under IFS transformations. We do that by rescaling the transformed line segment to the original length.

It is possible to store additional information besides density at each voxel, e.g. the information about which transformation caused the hit or that transformation's level of execution. The normal direction for rendering that voxel, calculated from the chain of used transformations, could also be stored. However, approximative on-the-fly calculation of rendering normals is beneficial, because their storage would lead to high memory cost.

The density resolution (i.e. the number of density levels) plays an important role. The previously described linear connection between voxel's hit count and its density proves to be unsuitable since it is not consistent with the distribution of number of voxels with certain density, which is exponential (see Figure 3). Such hypertrees are almost completely bare of leaves. Therefore, we do an internal recalculation of voxel's density after the generation process is over but before the rendering begins. We distribute all integer density values into a given smaller number of *density classes* so that a number of voxels falling into each class is approximately the same (see Figure 3).

Each voxel is then characterized by the relative density of its density class rather than directly by the density originating from the voxel's hit count. If we choose to have only two density classes, so that any hit voxel gets the maximum density

while all the other get zero density, we have binary or solid hypertrees. In practice, we use up to 128 density classes, which softens the appearance of trees.

5. Data Structure

Three-dimensional arrays are straightforward to use for storing hypertrees, but they set high demands on memory space. If storing a single voxel information would take one byte, then a volume density cube with resolution 512 in all directions would occupy 128 MB of memory. This number even increases if we store additional information with the voxel. For this reason, we estimate that classical arrays are suitable for storing hypertrees of resolution 300 and lower, offering fast access to the low detail data, which will presumably occur in animated sequences.

Hypertrees of higher resolution will more likely be used for still picturing, so we may trade fast access for more compact deposition. The common approach for compacting volume data are the octrees, but they are not appropriate for storing hypertrees for at least two reasons, i.e. sparseness of hypertrees and their nonbinary description.

Therefore, we derived an *ad hoc* data structure with a memory cost less than 5% of that of a three-dimensional array, but which is consequently slower when accessing data. We consider a fact that hypertrees are quite sparse volume densities, where only few percents of all voxels are relevant (i.e. voxels with nonzero density). The idea is to store only important information—the number of hits per each voxel excluding empty voxels.

During the IFS iteration, only indices of hit voxels are registered, forming an *index list* with presumably repeated elements. After the generation, some postprocessing is necessary to compact this index list and determine the hit ratios. We call this postprocess a *hypertree validation*.

First, the index list is quick-sorted. Then, the duplicates are removed from the index list, leaving only one appearance of each voxel index and storing the number of repetitions of that index in another list, called the *hit count list*.

When storing additional information with each voxel, a rule needs to be defined about how to combine that information when the lists are being compacted. Consider the case when a level of transformation is stored together with the hit voxel's index into the index list. When we compact the index list, only one level can be assigned to each voxel. In our implementation, the shallowest level for each voxel prevails, but different rules can be applied (e.g. most frequent level). Such extension of the hypertree generation algorithm makes minimum impact on both time and memory consumption, because depth information can be encoded in 2 or 3 bits.

The two lists together are sufficient to answer the query about any voxel's density class. Given a voxel index, its density class can be determined by looking it up in the index list and converting the corresponding value in the hit count list to the density class. Searching through the index list is fastest using a bisection. The golden cut and other search techniques require programming overhead that does not compensate for earlier termination. If some index is not found in the index list, then the voxel density is 0.

The memory consumption of the index list is temporarily higher before the list is compacted. After the validation, the memory use of both lists together is only a fraction of the memory used by a conventional three-dimensional array, despite the fact, that the elements of the index list require 4 bytes of storage.

With the described organization, we can generate and store hypertrees with resolutions over 1000 without running into space limit imposed by today's average computer configuration. Of course, the increased time consumption is the price we pay for indirect data access.

Querying for empty voxels consumes the most time and it may be too slow for some contexts, so we decided to use a compromised solution for medium hypertree resolutions (300–600 voxels). For such situations, we encode the information, which voxels are empty and which voxels are not, into a bit array. The index list is then only searched-through if the voxel is known not to be empty. The size of the bit array is of magnitude smaller than the size of the equivalent voxel array and may therefore sustain at least double resolution. It can be even much smaller, because we need to encode only information about voxels with indices between the known lowest and highest nonempty voxel index.

Considering the fact, that the whole hypertree is a union of mostly disjoint subparts, it is sometimes reasonable to maintain separate lists for every subpart. If the complete hypertree represents the first level in structure hierarchy, each of its direct subparts represents the second one. Second level subparts can further be stored as a list of the third level subparts, etc. The advantages of this approach are following:

- it enables us to find the optimal balance between compact storage and faster data access and
- keeping the hypertree in such organization is also very useful for determining shorter traversal paths for ray-marching.

The drawbacks speaking in favour of the single index list, are:

- there is a certain presented overhead when storing the hypertree as a set of separate subparts, because additional organizational information has to be saved along with the hypertree and because the subparts usually share smaller number of redundantly duplicated voxels,

- only approximate estimation of the required storage size for each subpart is possible, whereas the storage size for the whole hypertree can be usually exactly determined and is proportional to the number of IFS iterations,
- management and implementation of hierarchical representation is hard and can easily lead to unexpectedly meagre results.

6. The Hypertree Algorithm

To summarize the complete hypertree generation algorithm, we present its pseudo code here:

1. allocate an empty index list
2. generate values for random transformation parameters
3. $i = 0$
4. while ($i < N$)
   a. X = condensation()
   b. perturb the position of X using noise
   c. record indices of voxels occupied by X into the trunk index list
   d. select $k - 1$ transformations from w_j using probabilities p_j
   e. for $s = 1$ to $k - 1$
      i. if s-th selected transformation is dummy continue next loop
      ii. generate the transformation pipeline W using the first s of previously selected transformations
      iii. $Y = W(X)$
      iv. perturb the position of Y using noise
      v. record indices of voxels occupied by Y into the proper index list
      vi. record additional information (s, index of last transformation, ...)
   f. $i = i + k$
5. compact the index list(s) and generate the hit count list(s)
6. transform the hit counts into density classes, assign full density class to the voxels of the branching structure
7. bit-encode the information about nonempty voxels

7. Operations on Hypertrees

Volumetric representation of hypertrees allows us to easily combine multiple hypervolumes to embrace a wider range of attainable shapes. Some of the operations are a part of constructive volume geometry (CVG) and form the algebra of operators on hypertrees [18]. Other operations are used to reshape the hypertrees in a natural way.

Since hypertrees are medium-detail volume density models that describe the space occupation, none of the operations on them requires the intersection and other calculations traditionally connected with the solid geometry operations.

The union of hypertrees is the most intuitive CVG operation. It combines hypervolumes generated by different IFS codes and can thus produce shapes not achievable with a single IFS. A case from reality is a coniferous tree whose secondary branches are all bent downward because they are not strong enough to support their own weight. While such a tree could eventually be approximated by a single IFS with variable transformation parameters and dummy transforms, it can easier be constructed by combining two or more hypertrees. The IFS that considers the particularity of secondary level branches in the mentioned example produces only one half of the tree (assuming the dangling branches fill the lower halfspace). We can therefore use two IFSs to separately generate the left and right half of the hypertree and then unite them using the union operator.

Unifying two hypertrees H_1 and H_2 to produce the union hypertree $H = H_1 \cup H_2$ means combining their voxel densities. If $\mu_H(x)$ denotes the density of the voxel x in a hypertree H, then the rule for the union of hypertrees is:

$$\mu_{H_1 \cup H_2}(x) = \max\left(\mu_{H_1}(x), \mu_{H_2}(x)\right),$$

where the unification function is the maximum of two given densities. Other unification functions could also be used for reasons described below.

It is most reasonable to join two hypertrees with the same resolution, fractal depth, hypervolume and condensation. Because of that, one might consider running two or more IFSs consecutively on the same volume data set. This is unfeasible with the maximum unification function, since the information, about which voxel's density is higher, is only valid when the whole hypertree is generated. If addition is used as a unification function, however, then this is actually a better solution because of lower memory cost of a single hypervolume.

We can also use operations that directly set or clear voxel density values. With them, we can simulate processes like cutting off of branches or trimming of trees. To trim the hypertree, certain voxels have to be locked for hits or cleared afterward, so such deformation can be implemented as an internal constraint of IFS or performed as postprocessing. The currently open problem here is the occurrence of dangling parts, e.g. parts of branches inside the trimmed hypervolume with cutoff joints. The problem is hard to solve because there is no connectivity information, but the anomaly is usually imperceptible in densely foliated trees.

8. Rendering

8.1. Raymarching

Today, direct volume rendering of uniform volume data sets is a well developed field of computer graphics with many algorithms of general application. Surface reconstruction

methods are inapplicable to hypertrees because they could never faithfully describe the procedurally scattered structure. We use classical raymarching [9,12] to render hypertrees, but we employ some adjustments to produce visually diverse and interesting results.

The common front-to-back accumulation integral to gather the total opacity of the projection ray is applied. In each sample point along the ray, the sampled density μ is transformed to opacity α using the following equation:

$$\alpha = 1 - (1 - \mu)^{k\Delta s}.$$

This equation takes into account the possibly variable stepsize Δs and empirical normalization coefficient k which is used to condense or thin the hypertree. At each sample point, the local illumination model is also evaluated. The resulting sample color is multiplied by the sample opacity and attenuated by the tree's self-shadow, which is calculated by shooting shadow rays toward all of the light sources and accumulating the opacity. Shadowing is also the most time consuming part of the rendering process because the number of shadow rays is several times bigger than the number of projection rays.

8.2. Bounding Volumes and Ray Coherence

Raymarching is a time-consuming volume rendering method, whose duration depends primarily on stepsize along the ray. We have realized that using minimum voxel side as stepsize is a quality choice because taking smaller stepsize makes no relevant progress in the quality of rendered image.

The efficient calculation of traversal paths for projection rays is crucial for fast raymarching, especially when using the compact data structure where unnecessary queries for void voxels are most time-consuming. The standard approach is to raymarch the part of the ray that intersects with the hypervolume. A more elaborate technique can save a lot of time spent on redundant sampling in regions known to be void. At the start, we may intersect the ray with the axes-aligned bounding box of the hypertree, which was marked during the generation process. Since the bounding box is always smaller than the complete hypervolume, the time saving is guaranteed.

The fractal scheme of hypertree generation allows specialized solution for calculating economical traversal paths of projection rays. As we described earlier, the hypertree can be stored as a list of separate subparts at some level in which case their bounding boxes are determined during generation. If this is not the case, we can still find them later by subjecting the bounding box of the whole hypertree to the corresponding IFS transformations.

The union of secondary bounding boxes plus the bounding box of the condensation (which is usually quite narrow) may encapsulate the hypertree in a more economical way than the primary bounding box. However, transforming the axes-aligned bounding box by nonlinear transformations is inefficient owing to the following reasons:

- the secondary bounding boxes will in general not be rectilinear and axes-aligned, which makes them harder to traverse, and
- to set secondary bounding boxes rectilinear and axes-aligned as well, we have to bound the nonaligned bounding boxes and thereby embrace unsuitably large areas.

The better choice in this situation is to use the bounding spheres although they generally perform inferiorly compared to the classical bounding boxes. Their advantages are lower storage cost, fast intersection calculation, and, in our case, easier derivation of secondary bounding spheres.

To gain the secondary bounding sphere by transforming the primary bounding sphere with the corresponding IFS transformation, we only transform the primary sphere's center and recondition the radius. The calculation of the new radius is trivial for linear transformations. It is also simple for nonlinear shears, where the radius of the primary bounding sphere has to be scaled by $(1 + \frac{d_{\max}}{r})^2$. Here, $d_{\max}$ is the maximum displacement and r is the radius of the primary sphere. Because the ratio between $d_{\max}$ and r is in practice around 0.1, the scaling factor of shear itself is not very high.

If perturbations are incorporated in the tree generation, the radius of secondary bounding spheres needs to be increased by the largest possible offset. Raymarching traversal paths for a single projection ray are then constructed as a set of ray intervals lying inside the secondary bounding spheres and the condensation bounding box, cross-sectioned by the primary bounding box. The efficiency of this method depends on the number and types of used IFS transformations. A higher number of smaller separate subparts is more suitable because it yields more disjunct bounding volumes.

Depending on the rendering scenario, caching of rays could be an admissible technique to economize the rendering time. For high-resolution rendering, consecutive projection rays travel a considerable portion of passage through the hypervolume close to each other, thus sampling the series of same voxels (i.e. the principle of ray coherence). This is especially true if antialiasing through supersampling is involved (see Subsection 8.4).

Since querying for voxel densities is a remiss task due to the use of space-economical data structure, looking-up the cache for stored voxel densities is a potential time-saving. We only cache projection rays, since they originate in the same camera point and separate later. Caching of shadow rays would require tracking them back from the light sources, while they usually terminate early on the way from the sampling point because of density saturation.

A ray cache can be established as an array of voxel index and density pairs that were queried upon at the last raymarch. The size of this array equals maximum possible number q of sampling positions along the hypervolume. In our case, this number can be calculated as:

$$q = \frac{\sqrt{v_x^2 + v_y^2 + v_z^2}}{s_{\min}},$$

where v_x, v_y and v_z are the hypervolume extents in the directions x, y, and z, respectively, and $s_{\min}$ is the minimum voxel side length used as the raymarch step.

For the first projection ray, queried voxel indices and corresponding sampled densities are registered in sequential cache entries. For each query within the next ray, the corresponding cache entry is first examined, to eventually avoid the need to search by bisection. As soon as the lookup into the ray cache fails (i.e. there is no corresponding voxel index entry in it), the remainder of the ray is sampled past the ray cache and the sampled densities overwrite together with voxel indices the corresponding cache entries. Such cache organization may seem suboptimal, because a single misplaced sampling (e.g. because of the projection shift) stops the cache look-up for the remainder of the ray, even though there may still be following hits in the cache. From practical experiments, however, we concluded that any more sophisticated cache administration uses up the time gained in evading the search through the hypertree data structure.

8.3. Shading

When observing the trees in nature under moderate illumination, we can establish that shadows are the prevalent factor of three-dimensional comprehension. If there is no direct illumination from the light source, trees can be rendered using shadow attenuated ambient term only. If the tree is directly illuminated, we may use the shadow attenuated diffuse term to simulate the direction dependent illumination of the exposed tree side and the specular term to add the specular light reflection off smooth leaves.

It is sometimes inadequate to approximate the ambient term by a constant value. Reeves suggested the exponential attenuation of ambient illumination towards the interior of the tree based on a perpendicular distance from the sampling point to some predefined tree's outer shell [3].

Diffuse and specular reflections require accurate and sensible calculation of normals. The common gradient method which works fine for amorphous hypertextures is neither reliable nor justified here because we may be dealing with isolated voxels and highly disconnected structure. This is especially true for point-generated hypertrees, which bear no directional information. If a line segment is used as a generator, then some information about the structure flow can be saved along with each voxel. The normal can then be calculated using that information together with other data like sampling position within the voxel.

A simpler method of assigning normals to voxels is to do it procedurally by accounting only the main (i.e. first level) structure flows, as depicted in Figure 4. The structure direction in any sample point is calculated by interpolation. The

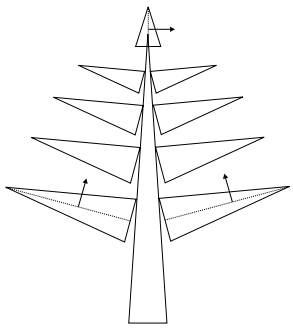

Figure 4: *The rendering normals with respect to the first level tree structure flow.*

side benefit of this approach is that it is also applicable to point-generated hypertrees. The drawback is that the method ignores the second and higher level structure orientations, but this is usually tolerable for medium detail visualizations. If secondary bounding boxes are maintained for hypertree representation, then normals to these bounding boxes can be used for shading, as previously done by Hart and DeFanti [19].

When evaluating the local illumination model, the basic nonshaded sample color is given by a *transfer function* that maps density, sampled position or any other voxel information to the color, using one of many possible coloring schemes. Coloring schemes may consider various physical aspects of real trees, for example darker inner areas or differently colored shoots.

A useful realization of a coloring scheme is to derive it as a function of voxel's fractal depth. For each hit voxel, the level of the used transformation is registered during the hypertree generation. The coloring scheme is then accomplished by prescribing a color span for each level.

The branching structure of the hypertree is generated by a limited number of replications of the condensation set. A given number of transformations (usually two or three) after each (possibly enforced) condensation transformation determine the voxels that belong to the lignified branches which are colored differently from the rest of the hypertree. Two color spans are thus usually sufficient—one for the branches and the second for the leafs, although intermediate voxel levels may be differently colored to visualize fruits.

8.4. Antialiasing

The hypertree generation process inherently produces the signal with unbounded frequency content. The problem of aliasing can therefore only be resolved with low-pass postfiltering of the volume data sets, similar to the visualization of presampled image arrays.

The antialiasing method used normally with raymarching is supersampling, where we shoot many projection rays for

each image pixel and average their colors into the resulting pixel color. This extends the rendering time for as many times as the supersampling factor, but it is usually necessary for quality visualization.

Additional rays may pierce the projection plane in uniformly distributed or jittered points. The necessary level of antialiasing is a subjective measure, but it also depends on several objective factors, such as distance to the tree, resolution of the tree, and, of course, the resolution of the image. The required supersampling factor s can be estimated by investigating the size of voxel's projection onto the image plane. A reasonably good estimation of the required supersampling factor is:

$$s = \left\lceil \sqrt{\frac{area(pixel)}{area(projection)}} \right\rceil .$$

9. Level of Detail

The required level of detail (LOD) has always been an important question in computer graphics. It has been recognized that different methods of modeling are applicable to various levels of detail. For high detail picturing, the exact and ample geometrical representation is necessary, while for the low one, the adequate solution can be phototexturing. For intermediate levels of detail, the impoverished geometry is commonly used, consisting of lesser simplified versions of original geometric constructs. There is however no reliable procedure prescribing how to deduce the lower level representation out of the higher level one.

One advantage of hypertrees over geometrical tree representations is that they can be relatively easily and almost continuously adjusted to the required level of detail by increasing or reducing their resolution. If we treat voxels as homogenous density regions, we can deduce a hypertree of lower resolution by fusing voxels. The density of new voxels is the weighted average of densities of contributing voxels, where weights are given by corresponding voxels' shares. The latter are calculated considering the resolution ratios between old and new hypervolume resolutions, called the *reduction factors*. In normal cases of resolution reduction, reduction factors should be strictly between one and two. Converting the hypervolume in rows, reduction factors are separately accumulated to determine the starting and ending positions of the new voxel within the grid of old voxels. Figure 5 shows such situation in two dimensions. From the situation on the left side of Figure 5, we can derive the formula for the new voxel's density μ' as:

$$\begin{aligned}\mu' = {} & ((i - x_s)(j - y_s)\mu_{i,j} + (x_e - i)(j - y_s)\mu_{i+1,j} \\ & + (i - x_s)(y_e - j)\mu_{i,j+1} \\ & + (x_e - i)(y_e - j)\mu_{i+1,j+1})/(r_x r_y).\end{aligned}$$

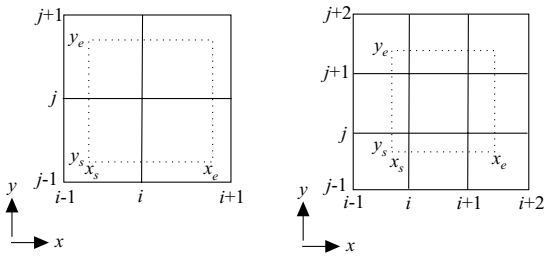

Figure 5: *Density averaging during resolution reduction.*

Here, $\mu_{i,j}$ denotes the density of the voxel (i,j), while x_s, x_e, y_s and y_e are the aforementioned starting and ending positions in x and y directions, respectively. The reduction factors are denoted by r_x and r_y. Only four adjacent voxel densities are averaged in the simplest two-dimensional case, while in the worst case nine of them need to be considered, as shown on the right side of Figure 5. In three dimensions, the maximal number of averaged densities is 27. If the reduction factor is allowed to be greater than *2* (i.e. the resolution is more than halved), this number may be larger.

We use empirical solutions for combining other voxel related information, such as voxel depth, which is set to be the most represented voxel depth in fused voxels. Because new voxels are afterward treated as homogenous regions, a chain reduction of hypertree resolution can yield a result that differs from the result of a single conversion. However, practical experiments involving the animation of camera with automatic LOD adjustment certify the applicability of described technique, since the human eye does a similar fusion of detail when moving away from the observed object.

The optimal level of detail can be calculated by choosing the most appropriate voxel size based on the projected voxel area. We do not go into an elaborate study of this question, but some research in this field was already done [20].

10. The Framework for Hypertree Definition and Rendering

Everything we have described so far has been implemented inside the design framework, which links together the definition, generation and rendering of hypertrees. We have defined a special syntax for the definition of hypertrees.

Figure 6 shows a fragment of hypertree definition. Such text file is parsed by the framework, allowing the user to adjust the parameters before the hypertree generation begins. The definition of a hypertree includes its resolution, number of iterations, fractal depth and number of branch levels, followed by the description of the generator, the coloring scheme and transformations. Reserved words RESOLUTION, ITERATIONS, TREEDEPTH and BRANCHDEPTH are used to describe the tree properties.

```
RESOLUTION 300 300 300
ITERATIONS 500000
TREEDEPTH 10
BRANCHDEPTH 1
HYPERVOLUME -1 1 -1 1 -1 1
POINTINIT 0 0 0

DEPTHCOLOR 0-2 0.5±0.0 0.5±0.0 0.3±0.0
DEPTHCOLOR 3-7 0.25±0.250.75±0.25 0.25±0.25
DEPTHCOLOR 8-10 1.0±0.0 0.0±0.0 0.0±0.0
NAME CHERRY

TRANSFORM 1 0
TRANSLATE 0 1 0
SCALE (0.7,0.5,0.7) (0.7,0.3,0.7)
(0.7,0.5,0.7)
SHEAR_Y 0 (0.5,-0.2) (0.5,2)
ROTATE (10,60,90) (-30,0) 0
TRANSLATE 0 -0.5 0

...
CONDENSATION 3
CONE -1.0 0.5 0.05 0.01 CONE_Y
```

Figure 6: *An excerpt from hypertree definition file.*

We use the descriptor DEPTHCOLOR to associate voxels of certain depth with the corresponding color span. The parameters of DEPTHCOLOR are the range of voxel depths and corresponding RGB range specification.

Parameters of noise perturbations are given after the PERTURB keyword. The hypervolume extent and displacement of the tree inside it follow the HYPERVOLUME and DISPLACEMENT keywords, respectively. Additionaly, the tree can be given a distinctive name using the NAME descriptor.

Two types of generators are supported at present—points and line segments, described by POINTINIT and LINEINIT descriptors, respectively.

As described before, the transformations on generators can be composed of several elementary transformations as long as the final result is a contraction. Each IFS transformation w_i is in our syntax denoted by TRANSFORM and additionally described by two numerical parameters—the probability p_i and the last allowed level of execution c_i. The probabilities p_i are cumulative, i.e. they do not necessary sum to one, but only reflect the relations among probabilities of selecting the corresponding transformations during IFS iteration. Any transformation can be designated as dummy from level d_i on by writing DUMMY d_i at the end of the line.

What follows the TRANSFORM line are descriptions of the basic transformations that make up the IFS transformation. Each basic transformation type (given by TRANSLATE, ROTATE, SCALE or SHEAR_x keyword) is described with appropriate parameter values, which are optionally grouped into pairs or triplets, depending on whether a different parameter value is valid for the first, second or third and subsequent levels.

The condensation transformations are declared separately with the CONDENSATION descriptors. They are usually composed of a single CONE transformation, which maps into the conical region given by bottom line, top line, bottom radius and top radius for one of the three major axes.

Hyperforest is a group of hypertrees, described in a separate text file which only contains links to hypertree files. However, the volume data set of these hypertrees is common so the hyperforest defines its resolution and storage type (using the RESOLUTION and STORAGETYPE keywords).

The generated hypertree can be saved for later use or rendered right away. The rendered image can also be saved in jpeg format.

11. Results

Hypertree generation and rendering are tested on several fictitious models of hypertrees and some models gained by contemplation of trees in nature. Various combinations of storage structures, raymarching adaptations and shading models are used. The time needed to perform different operations and the quality of results are considered.

Several data structures are used, such as standard three-dimensional array, the intermediate data structure with bitwise encoded information about empty voxels, and the adapted data structure with index and hit count lists only. The latter have been also implemented in two versions—as a single index list for the complete tree and as a partitioned index list for the first level subparts.

Within raymarching, we have used different schemes of bounding volumes, starting with a single axes-aligned bounding box, axes-aligned secondary bounding boxes and secondary bounding spheres (except for the bounding box for the condensation). We have also evaluated the effect of ray-caching on rendering time.

Different shading models are applied to various illumination scenarios. We have investigated all possibilities from self-shadowed hypertrees with mere ambient illumination to hypertrees with full Phong illumination.

Finally, we examined the effect of antialiased rendering on the quality of results. All tests have been performed on a PC with Athlon 1.4 GHz and 512 MB of RAM.

Figure 7 shows four examples of hypertrees. The top hypertrees have been modeled after real trees, while the bottom hypertrees are imaginary. In all cases, resolution is equal to

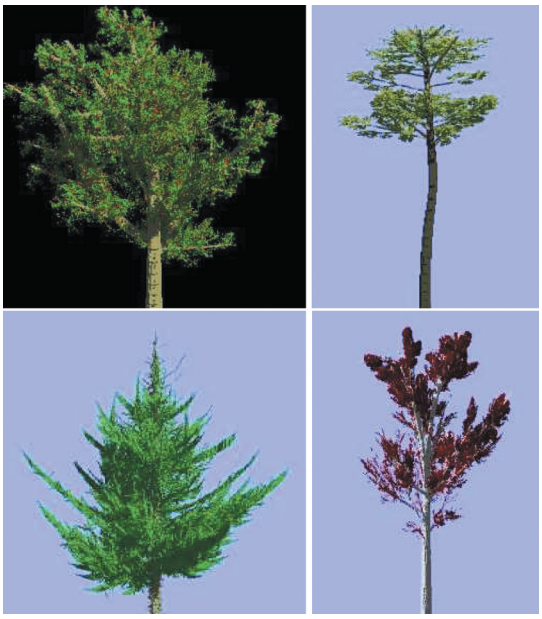

Figure 7: *Examples of hypertrees.*

Figure 8: *Noise-perturbed hypertree.*

300 voxels in all directions. The top left hypertree is generated by 10 IFS transformations (2 of them dummy) in 500 000 iterations and has a fractal depth of 10, wherein only 1 level after the condensation are branches. Levels 8 to 10 are colored red to simulate the fruit. Only 4 transformations are used in generating the top right hypertree, 2 of them are dummy. The number of iterations is 1 million, the fractal depth is 8 and 4 levels represent branches. The bottom left hypertree is generated by 100 000 iterations of 8 nondummy transformations with fractal depth 5 and 1 level of branches. The generator is a line segment of length 0.05. The bottom right hypertree uses 2 condensations and 7 transformations in an IFS iteration of 1 million cycles. The fractal depth is 6, 3 levels belong to branches.

Figure 8 gives an example of noise-perturbed hypertree at isotropic resolution of 1000. The number of IFS transformations used is 9, the number of iterations is 1 million, the fractal depth is 4 and 3 levels are dedicated to branches. The perturbation amplitude in y-direction is 0.01 and is a double of perturbation amplitudes in x- and z-directions.

Figure 9: *Hypertrees produced by a union of four hypertrees (left) and a clear operation (right).*

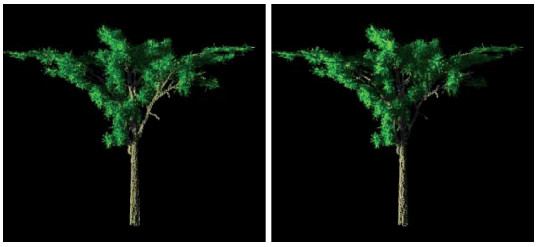

Figure 10: *Shading models: diffuse, diffuse + specular.*

Figure 9 shows hypertrees treated by techniques from Section 7. The hypertree on the left is a union of four hypertrees, each representing a quarter of the whole. The top half of the hypertree on the right has been cleared in a postprocess, as if the hypertree has been horizontally docked.

For most of the images, the recommended shading model is attenuated ambient plus diffuse shading with shadowing only. The Lambert shading with artificially calculated normals should be employed to make the impression of a clear sunny day. If we want to stress the directed illumination, we can also use specular reflection (e.g. for night scenes of trees illuminated by lamps). In Figure 10, examples of diffuse (left), and specular shading models (right) are shown.

The importance of antialiasing is evident from Figure 11 with two renderings of the same hyperforest: The non-antialiased version and the antialiased version with supersampling factor 3. We found that, in practice, $s = 3$ suffices for most situations.

Application of level of detail reduction is depicted in Figure 12. The hypertree in question is defined by nine IFS transformations. Two millions of iterations produce the hypertree with fractal depth 4 and 2 levels of branches. The resolution of hypertrees in Figure 12 degrades in rows from top left to bottom right in the following order: 1000, 800, 600 and 400.

Figure 11: *Aliased vs. antialiased rendition of hyperforest.*

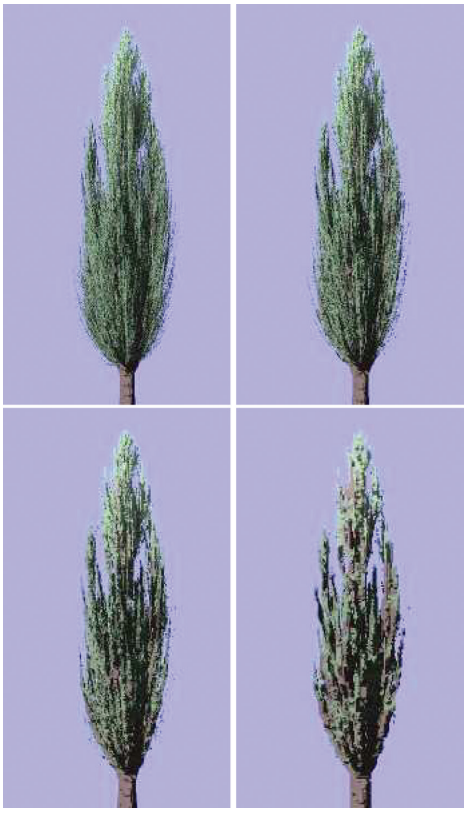

Figure 12: *Hypertree level-of-detail.*

Table 1 shows orientational results for different data structures and rendering scenarios. The rendered trees are from Figure 7 and are in the table designated as A - top left B - top right, C - bottom left and D - bottom right. Columns in the table correspond to different data structures used. Hypertree creation time (CT), rendering time (RT) and memory cost (MC) are shown in subcolumns. All results are displayed as pairs. The first values result from using trees with uniform resolution of 100 voxels and no supersampling. The values in parentheses were obtained with resolution of 300 voxels and supersampling factor 2. Image size is 300 × 300 pixels in all cases. It is evident that a combination of index list and bit array is almost as fast as the three-dimensional array. The difference in rendering time is even compensated by shorter creation time.

Antialiasing extends the rendering time by almost exactly s^2, where s is the supersampling factor. Caching of projection rays has not given the expected acceleration even with high-resolution images and narrow field of view.

There is no simple answer to the question of the best bounding volume scheme. We have had cases with bounding spheres being from 40% faster to twice slower than simple axes-aligned bounding boxes (AABB). The amount of speedup depends on the structure of the tree and how well it can be captured by the bounding volumes. The administration needed to operate secondary bounding boxes in current implementation rarely justifies their use, though it does accelerate rendering in special cases, but always adds to the creation time. Figure 13 shows an example of a tree for which both the bounding spheres and secondary bounding boxes approach have proved successful. The corresponding generation and rendering times are gathered in Table 2.

It was shown by Deussen et al. [21] that trees can be rendered in real-time using points and lines. Volume rendered hypertrees can presently not be visualized at that speed, except at low resolutions and with no antialiasing, so their use in interactive animations is limited to low detail afforestation.

12. Conclusions and Future Work

Hypertrees are a new form of complex natural geometry generation, appropriate for medium detail visualizations. Using today's computers, they can be generated and rendered within few seconds for average resolution and image size. The major benefits of hypertrees are their compact representation both in terms of model description and storage, and ease of LOD derivation.

The modeling technique and model representation are loosely coupled because the selection of one does not dictate the selection of the other. The hypertree volume may well be constructed by a process other than IFS. Hypertrees could, for example, be represented as a volumetric interpretation of L-systems. Such approach would alleviate the main disadvantage of current fractal generation compared to geometric models, which is the confinement of modeling freedom, but retain the benefits of hypertextural representation.

Practical limitations of hypertrees in the present form are nonintuitive model description and lack of a real-time rendering method. We intend to solve the first problem by building the semi-automatic generator of hypertree definitions. Such generator would take as input a set of numerical parameters to describe the structure of the tree, similar to the approaches of other authors [4]. Incorporating the graphical interface, this would help the user to determine the required transformations and allow the opportunity of a fast preview. As usually it would have a built-in database of referential parameter values for typical representatives of tree species.

Table 1: *Creation times, rendering times and memory costs of different hypertrees and data structures*

	Index List			Index List + Bit Array			3D Array + Bit Array		
	CT(s)	RT(s)	MC	CT(s)	RT(s)	MC	CT(s)	RT(s)	MC
A	0.5(0.5)	1.0(16.7)	209K (860K)	0.5(0.5)	0.6(8.0)	331K (4.0M)	0.4(1.9)	0.5(7.7)	2.9M (80M)
B	2.6(2.3)	0.8(11.5)	80K (350K)	2.6(2.4)	0.4(5.7)	202K (3.5M)	2.2(3.7)	0.4(5.6)	2.9M (80M)
C	0.4(0.8)	1.0(15.5)	210K (1.4M)	0.4(0.9)	0.5(7.5)	332K (4.6M)	0.3(2.0)	0.5(7.1)	2.9M (80M)
D	2.1(1.4)	0.5(7.8)	56K (315K)	2.2(1.5)	0.3(4.1)	178K (3.5M)	1.0(2.5)	0.3(4.1)	2.9M (80M)

Figure 13: *A testing hypertree for various bounding schemes.*

Table 2: *Creation and rendering times for different bounding schemes (in seconds)*

	AABB	Spheres	Secondary AABB
Creation time	3.7	3.8	4.5
Rendering time	11.3	7.7	10.0

Further extensions of IFS are also possible. Recurrent IFS [22] and additional nonlinear transformations (e.g. nonlinear tappering) could enhance the modeling creativity.

References

1. J. Bloomenthal. Modeling the mighty maple. In *Proceedings of SIGGRAPH '85*, vol. 19(3), pp. 305–311, 1985.
2. B. Lintermann and O. Deussen. Interacive modeling of plants. *IEEE Computer Graphics and Applications*, 19(1):56–65, 1999.
3. W. T. Reeves and R. Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *Computer Graphics*, 19(3):313–322, 1985.
4. J. Weber and J. Penn. Creation and rendering of realistic trees. In *Proceedings of SIGGRAPH '95*, vol. 29, pp. 119–128, 1995.
5. M. Gervautz and C. Traxler. Representation and realistic rendering of natural phenomena with cyclic CSG-graphs. *Visual Computer*, 12(1):62–74, 1995.
6. M. Holton. Strands, Gravity and botanical tree imagery. *Computer Graphics Forum*, 13(1):57–67, 1994.
7. P. Prusinkiewicz and A. Lindenmayer. The algorithmic beauty of plants. Springer-Verlag, Berlin, 1996.
8. P. E. Oppenheimer. Real time design and animation of fractal plants and trees. *Computer Graphics*, 20(4):55–64, 1986.
9. K. Perlin and E. M. Hoffert. Hypertexture. In *Proceedings of SIGGRAPH '89*, vol. 23(3), pp. 253–262, 1989.
10. S. Demko, L. Hodges and B. Naylor. Construction of fractal objects with iterated function systems. *Proceedings of SIGGRAPH '85*, vol. 19(3):271–278, 1985.
11. C. E. Zair and E. Tosan. Fractal modeling using free form techniques. *Computer Graphics Forum*, 15(3):269–279, 1996.
12. J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. In *Proceedings of SIGGRAPH '89*, vol. 23(3), pp. 271–280, 1989.
13. K. Daubert and H. P. Seidel. Hardware-based volumetric knit-wear. *Computer Graphics Forum*, 21(3):575–584, 2002.
14. M. Barnsley. *Fractals everywhere*. 2nd edition, Academic Press, New York, 1993.
15. B. B. Mandelbrot. *The fractal geometry of nature*. 2nd edition, Freeman, San Francisco, 1982.
16. E. Gröller. Modeling and rendering of nonlinear iterated function systems. *Computers & Graphics*, 18(5):739–748, 1994.
17. D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin and S. Worley. *Texturing & modeling—A procedural approach*. 2nd edition, Academic Press, Neq York, 1998.
18. M. Chen and J. V. Tucker. Constructive volume geometry. *Computer Graphics Forum*, 19(4):281–293, 2000.
19. J. C. Hart and T. A. DeFanti. Efficient antialiased rendering of 3D linear fractals. *Computer Graphics*, 25(4):91–100, 1991.

20. M. Weiler, R. Westermann, C. Hansen, K. Zimmerman and T. Ertl. Level-of-detail volume rendering via 3D textures. In *Proceedings of IEEE Volume Visualization and Graphics Symposium*, pp. 7–13, 2000.

21. O. Deussen, C. Colditz, M. Stamminger and G. Drettakis. Interactive visualization of complex plant ecosystems. In *Proceedings of the IEEE Visualization '02*, pp. 219–226, 2002.

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