Raytracing 3D linear graftals

Ph. Bekaert, Y. D. Willems

Department of Computing Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
philippe@cs.kuleuven.ac.be

Abstract

Many objects in nature, like trees, mountains and seashells, have a property called selfsimilarity. Sometimes this property is very pronounced, other natural phenomena exhibit this property to a lesser degree. During the past years much attention has been paid to fractals, purely selfsimilar objects. We present a formalism, based on the wellknown object-instancing graph, to represent objects which are not necessarily purely selfsimilar. We show that Iterated Function Systems and some famous variants can be described elegantly in this formalism. We also present an algorithm to raytrace such objects.

Keywords: rendering, fractals, formal languages, raytracing

1 Introduction

B.B.Mandelbrot [9] pointed out that many objects in nature, like trees and mountains, share the property that they seem to consist of downscaled copies of themselves, a property called selfsimilarity. He called such objects "fractal" objects. One distinguishes between deterministic and random (or statistical) selfsimilarity. In the latter case the parameters of the downscaling transformations are not constant, but subject to statistical spread. One also distinguishes linear deterministic fractals, where the downscaling transformations are linear, from nonlinear fractals, like the Mandelbrot set. Random fractals have been used with success for the simulation of terrain in computer graphics [4; 10] whereas deterministic fractals are well suited for the computer generation of e.g. plausible trees and seashells.

Linear deterministic fractals are attractors of Iterated Function Systems (IFSs) [1]. An IFS is a set $\{w_i \mid i = 1 \ldots n \text{ with } n \in \mathbb{N}\}$ of contractive affine linear transformations w_i. A transformation is contractive if it has, roughly spoken, the property of downscaling all possible geometrical figures. Each such set defines a unique figure, called the attractor of the IFS. The collage theorem [1] provides us with a means of constructing such a set of transformations for a given geometrical figure. The use of IFS's in computer graphics is presented in [2; 3].

Lindenmayer-systems (L-systems) [8] are based on the theory of formal languages. L-systems are parallel rewriting systems that are used for describing the growth of plants and trees in a biologically motivated way. An introduction to L-systems and some excellent computer generated pictures of plants and trees can be found in [12]. The figures resulting from an L-system description are in general not purely selfsimilar and thus, as such, not fractals. However, they clearly show features of fractals: some, but not all, parts of such figures are copies of the complete figure or parts of it. Such figures are called graftals. [12] shows how an L-system description of a purely selfsimilar object can be converted into an IFS.

In this article we present an L-system-like formalism for describing all possible linear graftals and an algorithm for raytracing them. The formalism is based on the notion of object instancing graph, a well-known concept in computer graphics. In §2 we present the formalism and in §3 the raytracing of the objects described by the formalism.

2 The Formalism

2.1 Object instancing

A database of graphical objects to be visualized often has a hierarchical structure. Down in the hierarchy are simple, primitive objects from which more complex objects are composed. These composed objects can be used for composing even more complex objects. The composition is done by transforming a primitive object to the location where it is needed. This transforming is called "instantiating". Such an approach has several benefits: it makes the modelling of objects easier and it saves memory.

The way objects are defined in terms of other objects can be represented by a directed graph, the object instancing graph. The nodes in the graph represent graphical objects and the edges are transformations. An edge from node A to node B denotes that a graphical object A is composed of an instance of a graphical object B according to the transformation w associated with the edge (fig.1).

Such an object instancing graph is normally acyclic. If the graph contains cycles, the graphical object denoted by the topnode of the graph has parts that consist of copies of itself and thus is a graftal (fig.2). A sufficient criterion for convergence is that the composition of all transformations in each cycle is a contractive transformation.

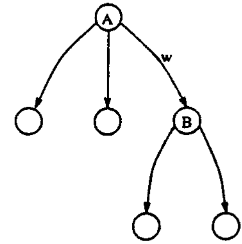

Figure 1: An acyclic object instancing graph

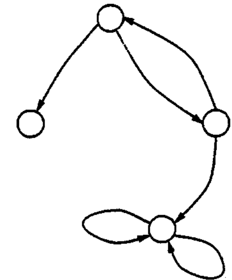

Figure 2: A cyclic object instancing graph

In the next section we propose an L-system-like formalism that describes object instancing graphs in the most general case.

2.2 IFS codes with restrictions

An object instancing graph can be represented by the following parallel rewriting system:

- Let V be an alphabet. An alphabet is a set of symbols. These symbols label the nodes of the object instancing graph;
- Let W be a set of transformations closed under composition: the composition of two transformations of W

is always a transformation of W too. In this paper we think of the full set of linear affine transformations in the three dimensional euclidian space. It was stated in §2.1 that such transformations are associated with the edges of the graph;

- Let $(V \times W)^*$ and $(V \times W)^+$ be the set of resp. all and all non-empty words, written with couples $(A, w) \in (V \times W)$. We write such a couple (A, w) as $w(A)$ and, if w is the unity transformation, also as A. Such a couple corresponds to an instance of a graphical object, labeled A, according to the transformation w. A word $w(A)v(B)$ denotes a graphical object composed of an instance of A and B according to the transformations w resp. v.

An arbitrary object instancing graph can be represented by a quadruple (V, W, ω, P) with $\omega \in (V \times W)^+$, called the axiom, and $P \subset V \times (V \times W)^*$, called the production rules.

There can be, but does not have to be, a geometry associated with each symbol A in V. This means that A represents a sphere or a cube or some other primitive geometrical shape. In general this will be the case only for the terminal nodes, the leaves, of the object instancing graph. Such a terminal node corresponds to a symbol for which no production rule, except the identity production $A \rightarrow A$, is given. There must always be a geometry associated with each terminal node. This does not have to be the case for non-terminal nodes since non-terminal nodes denote objects which are composed of other objects.

The axiom ω is the object represented by the top of the graph. It will normally be a couple (O, e) with no geometry associated with O and e the identity transform. We allow the axiom to be any possible non-empty word, but the general case can be reduced

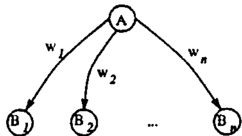

Figure 3:

to this special case by introducing a production rule $O \rightarrow \omega$ that produces the axiom ω from O.

A production rule

$$A \rightarrow w_1(B_1) \dots w_n(B_n)$$

with predecessor A denotes the edges that leave from the node labelled A of the object instancing graph: there are n edges connecting A with $B_1 \dots B_n$ with transformations $w_1 \dots w_n$ associated with them. Thus, A consists of instances of $B_1 \dots B_n$ according the transformations $w_1 \dots w_n$ (fig.3). $w_1(B_1) \dots w_n(B_n)$ is called the successor of the production rule. If no production rule is given for a symbol, we assume the identity production rule $A \rightarrow A$. Such a symbol is a terminal symbol.

Application of a set of production rules $A_i \rightarrow \dots$ to a word of couples (A_i, w_i) is done by substituting each symbol A_i in the word by the successor of the production rule with predecessor A_i. We agree that the successor be a non-empty word, although in general also empty successors are allowed. Application of $A \rightarrow w_1(B_1) \dots w_n(B_n)$ to $w(A)$ thus yields

$$\begin{aligned} & w(w_1(B_1) \dots w_n(B_n)) \\ = & (w \circ w_1)(B_1) \dots (w \circ w_n)(B_n) \end{aligned}$$

with $w \circ w_i$ denoting the composition of w after w_i.

We acquire a (generally long) word representing the graphical object described by the object instancing graph by repeatedly applying the production rules P on the axiom ω of the parallel rewriting system. If

the object instancing graph is acyclic, this process will end, since after a finite number of iterations the resulting word will contain only terminal symbols for which the identity production rule applies. If the graph is cyclic, we have to look for an appropriate stopcriterion.

When rendering the graphical object represented by this word, we are free to choose a geometry for symbols with no associated geometry Our convention is to ignore such symbols when there are symbols with an associated geometry in the graph, i.e. leaves, and to choose a random geometry which encloses the limit-figure in case there are no symbols with associated geometries. In the latter case the object instancing graph has no leaves and represents an object that consists of only instances of (parts of) itself, as is the case for e.g. the attractor of an IFS code.

The examples in §2.3 will make clear why we call the quadruples (V, W, ω, P) IFS codes with restrictions.

2.3 Examples

2.3.1 IFS-codes

The attractor of an IFS-code $\{w_1, \dots, w_n\}$ [1] is a figure composed of only instances of itself according to the transformations $w_1, \dots, w_n$. The object instancing graph is shown in figure 4. The graph consists of one node A and n loops and is represented by the following parallel rewriting system:

$$\begin{aligned} \omega &: A \\ p_1 &: A \rightarrow w_1(A) w_2(A) \dots w_n(A) \end{aligned}$$

The first derivations are:

$$\begin{gathered} A \\ w_1(A) \dots w_n(A) \\ (w_1 \circ w_1)(A)(w_1 \circ w_2)(A) \dots \\ \dots (w_n \circ w_{n-1})(A)(w_n \circ w_n)(A) \\ \dots \end{gathered}$$

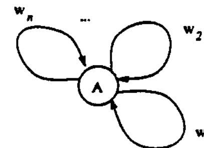

Figure 4: Object instancing graph of the attractor of an IFS-code

The derivation of length k consists of the k-th order images of A. The graphical objects represented by the derivations form a series which converges to the attractor of the IFS-code, whatever geometry is associated to A.

2.3.2 IFS-codes with condensation set

Figure 5 shows the object instancing graph of the attractor of an IFS-code with condensation set (CIFS-code) [2]. The graph has two nodes A and B. Node A has n loops annotated with the transformations $w_1, \dots, w_n$ of the code and one edge to node B. B is a terminal node and thus must have an associated geometry. If w_B is the transformation corresponding to the edge from A to B, then $w_B(B)$ is the condensation set of the CIFS-code. The graph is represented by the following parallel rewriting system:

$$\begin{aligned} \omega &: A \\ p_1 &: A \rightarrow w_B(B) w_1(A) \dots w_n(A) \\ p_2 &: B \rightarrow B \end{aligned}$$

The derivation of length 1 is

$$w_B(B) w_1(A) w_2(A) \dots w_n(A)$$

and of length 2

$$\begin{array}{cc} & w_B(B) \\ w_1(w_B(B)) & w_1(w_1(A)) \dots w_1(w_n(A)) \\ w_2(w_B(B)) & w_2(w_1(A)) \dots w_2(w_n(A)) \\ & \dots \\ w_n(w_B(B)) & w_n(w_1(A)) \dots w_n(w_n(A)) \end{array}$$

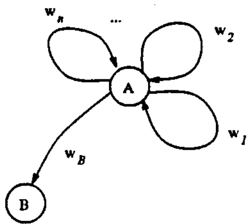

Figure 5: Object instancing graph of the attractor of a CIFS-code

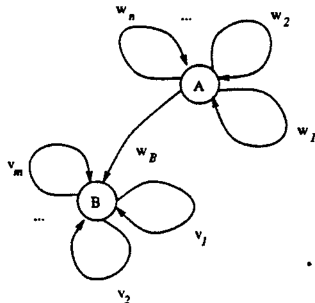

Figure 6: Object instancing graph corresponding to a HIFS-code

The derivation of length k contains all images up to order k of the condensation set $w_B(B)$ according to the transformations $w_1, \ldots, w_n$ of the CIFS code. If we ignore the symbol A when rendering the graphical objects represented by the derivations, we get in the limit the attractor of the CIFS code.

2.3.3 Hierarchical IFS-codes

The condensation set of a CIFS-code can be the attractor of an IFS-code in its turn. In that case the code is called a hierarchical IFS code (HIFS) [2]. A HIFS-code $\{\{v_1, \ldots, v_m\}, \{w_1, \ldots, w_n\}\}$ implies an object instancing graph as shown in figure 6. The difference with the CIFS object instancing graph of figure 5 is that now node B also has loops and, since it is not a terminal node, does not have to have an associated geometry. A HIFS-code corresponds to the following parallel rewriting system:

$$\begin{aligned} \omega &: A \\ p_1 &: A \rightarrow w_B(B)w_1(A)w_2(A)\ldots w_n(A) \\ p_2 &: B \rightarrow v_1(B)v_2(B)\ldots v_m(B) \end{aligned}$$

There are no edges leading from B to A. The transformations w_i in the resulting derivations will always be to the left of the transformations v_j: when rendering, the transformations w_i will never be applied before the transformations v_j. The derivations represent a series of graphical objects that converge to the attractor of the HIFS-code.

One easily extends to the case of more than two hierarchical levels, and the case with an arbitrary condensation set in the lowest level. One also sees how to generate schemes for arbitrary object instancing graphs with two or more geometries and many arbitrary cycles.

The attractor of the HIFS-code is part of the attractor of the IFS-code $\{v_1, \ldots, v_m, w_1, \ldots, w_n\}$. By imposing restrictions on the order in which transformations may be composed, one obtains an object that is not purely selfsimilar anymore like the attractor of plain IFS-codes. The production rules allow to impose arbitrary restrictions, whence the name IFS-codes with restrictions.

[12] shows a way to convert L-systems for purely selfsimilar figures to an IFS code. The same method can be applied to convert a much broader range of L-systems into our IFS-codes with restrictions.

In the next section we show how the raytracing of objects described by our IFS-codes with restrictions can be done.

3 A raytracing algorithm

The attractor of an IFS code with restrictions is part of the attractor of a plain IFS code. The raytracing can be done in a way similar to the one proposed in [6]. The main difference with the algorithm proposed there is that our method also accounts for systems with geometries in it. This kind of systems is most useful for modelling.

3.1 Ray-graftal intersection test

As pointed out in [6], fractals, and also graftals, are well suited for raytracing by constructing a hierarchy of bounding volumes, a wellknown raytracing acceleration technique [5]. Such a hierarchy is constructed by applying the transformations of the IFS-code with restrictions in all possible ways to an initial bounding volume $\mathcal{G}$ that encloses the limit-figure, taking into account the restrictions in the code. §3.2 explains how such an initial bounding volume $\mathcal{G}$ can be generated automatically.

There are several ways to construct children-volumes from a given volume in the hierarchy. Premultiplication is needed for the parent volume to enclose the children [7]. Also, all transformations in the code must be contractive to get a good hierarchy of bounding volumes.

There is no need to keep the whole hierarchy in memory all the time, not even for constructing it all at once. It is sufficient to construct only that part of the hierarchy that is needed for a given ray and to start over for the next ray. This is done by maintaining a list of volumes in the hierarchy that can give rise to an intersection and refine it until the list becomes empty or we are sure that we will not find a closer intersection or a given stopcriterion is met. The refinement is done by constructing the children of the closest volume on the list and move those that are hit by the ray to the list. The list initially contains the initial bounding volume.

The algorithm presented here works only for codes with axiom $\omega = (O, e)$. It was stated in §2.2 that the general case can be reduced to this case by introducing a new production rule $O \rightarrow \omega$. For understanding it one must see that there is a one to one correspondence between images of the initial bounding volume $\mathcal{G}$, and images of the graphical object represented by the axiom ω. Instead of keeping a list of images of the initial bounding volume, we keep couples deduced from the axiom $\omega = (O, e)$, which is equivalent.

We assume that the ray, for which ray-graftal intersection is tested, hits the initial bounding volume $\mathcal{G}$. This is a necessary condition for having an intersection with the graftal:

1. initialize the list with one couple, being the axiom (O, e);
2. until the list is empty, do
   (a) find the couple (X, w) on the list for which the intersection distance to $w(\mathcal{G})$ is smallest;
   (b) remove this couple from the list;
   (c) if $w(\mathcal{G})$ is "small enough", then do
      i. if there are no geometries in the code, we take the intersection with $w(\mathcal{G})$ as an intersection with the graftal. Stop;
      ii. if there is a geometry $\mathcal{X}$ associated to the symbol X, and the ray hits $w(\mathcal{X})$, remember this intersection if it is closer than all other intersections found until now.
   (d) otherwise, for each couple (Y_i, v_i) in the successor of the production rule $X \rightarrow (Y_1, v_1)\ldots(Y_n, v_n)$, do
      i. if Y_i is a terminal symbol with associated geometry $\mathcal{Y}_i$, and

the ray hits $(w \circ v_i)(\mathcal{Y}_i)$, remember this intersection if it is closer than all other intersections found until now;

ii. if the ray hits $(w \circ v_i)(\mathcal{G})$, add the couple $(Y_i, w \circ v_i)$ to the list.

3. no or closest intersection found. Stop.

The advantage of this algorithm is that it is able to raytrace attractors of 3D IFS, CIFS and HIFS codes and many more in a quite efficient way. To improve efficiency when testing shadow rays against the graftal, we can also stop when the first intersection is found, since we are not interested in the closest intersection: one only wants to know if there is an intersection or not. Another optimization is to remove all couples (Z, t) from the list for which the distance to $t(\mathcal{G})$ is larger then the distance to a newly found intersection point in step 2(c)ii and 2(d)i.

Step 2(c)ii may seem unnecessary at first sight. It ensures that the geometries of nonterminal symbols will be rendered the moment the stopcriterion "small enough" is met. It is of no importance when we consider codes with geometries associated to terminal symbols only, which is the most useful case.

The stopcriterion "small enough" needs some more explanation: it can be

- "a fixed recursion level is reached": in that case we keep not only the couples and intersection distances on the list, but also the recursion level. The recursion level of the axiom (O, e) is 0 and if the recursion level of a given item on the list is n, the one of its children is $n+1$;
- "the size of $w(\mathcal{G})$ is smaller than a predefined size". The best way to calculate an approximation for the size of $w(\mathcal{G})$ is to calculate the contractivity of the transformation w and multiply with the size of $\mathcal{G}$. See [6; 7];
- "$w(\mathcal{G})$ is smaller than one pixel": this implies calculating the width of a beam originating at the eyepoint and fitting one pixel on the screen, eventually after reflections and refractions, and comparing this width with the size of $w(\mathcal{G})$. See [6]. This method is used in our implementation.

Raytracing also requires a normal to be computed at the intersection points:

- if there are geometries in the system, intersections are always intersections with images of these geometries (step 2(c)ii and 2(d)i). In this case we take the normal at the point of intersection on the image of a geometry;
- if there are no geometries in the code, we compute a weighted average of normals at intersection points with the $w(\mathcal{G})$ volumes that led to this intersection (step 2(c)i), the way it is done in [6].

A nice variant is to test for intersection not with $(w \circ v_i)(\mathcal{Y}_i)$ in step 2(d)i, but with the image of an alternative geometry when $(w \circ v_i)(\mathcal{G})$ is "small enough". For a CIFS-code this reduces to the method in [11] for rendering fractal trees when we take a geometry for a branch as condensation set and for a leaf as alternative geometry.

3.2 Initial bounding volume

The construction of a good initial bounding volume $\mathcal{G}$ is of crucial importance for the efficiency of the algorithm. A good initial bounding volume should

- be a bounding volume: it should enclose the limit-figure of the code. If not, not only the parts of the limit figure that lay outside the initial volume will not be rendered, but generally also a whole series of images of these parts. This "fractal clipping" can be useful in some cases but is generally an unwanted effect. This may seem bizarre since we don't know the limit-figure in advance.
- be as small as possible: smaller bounding volumes will give rise to less ray-bounding volume intersection tests in the algorithm of §3.1. Tests showed that the execution time is significantly reduced when the right initial bounding volume is chosen: e.g. the Barnsley fern in figure 7 can be rendered 27% faster when using a box instead of a sphere.

Figure 7: Barnsley fern: A HIFS with 6 transformations in 3 levels. No geometries.

The following algorithm produces a good bounding volume of any possible geometry (a sphere, box, ...) for a given code. It takes as argument a couple (X, w) and is initially called with the axiom $\omega = (O, e)$. It assumes that an arbitrary geometry $\mathcal{H}$ (e.g. the unit sphere) has been chosen beforehand for symbols with no associated geometry:

1. if there is no geometry associated with X, and $w(\mathcal{H})$ is "small enough", then the result is the bounding volume of requested geometry enclosing $w(\mathcal{H})$;
2. otherwise, if $w(\mathcal{X})$ with $\mathcal{X}$ the geometry associated to X, is "small enough", or X is a terminal symbol, then the result is the bounding volume of requested geometry of $w(\mathcal{X})$;
3. otherwise, for each couple (Y_i, v_i) in the successor of the production rule $X \rightarrow (Y_1, v_1) \ldots (Y_n, v_n)$, do

   (a) calculate (recursively) the bounding volume $\mathcal{G}_i$ of $(Y_i, w \circ v_i)$.

   The result is the smallest bounding volume of requested geometry of the volumes $\mathcal{G}_i$.

The criterion "small enough" is similar to the same criterion in the algorithm of §3.1 except that comparing with pixelsizes makes no sense here. In our implementation "small enough" means that the image of a volume is smaller than a predefined fraction of the size of the volume itself.

This brute-force algorithm has the advantage that it is applicable for any possible geometry for the bounding volume, not only a sphere, and, for spheres, it seems to give smaller volumes than the algorithm from [6] (the Barnsley fern in figure 7 was rendered three times faster with our algorithm). It has the disadvantage of being a quite intensive calculation, but since this calculation has to be performed only once, before the raytracing, this is not so much a problem.

A criterion for deciding between alternative geometries for the initial bounding volume can be the average projected area, which is said to be one fourth of the total surface area for convex geometries [5]. In our implementation we compare spheres and boxes and chose the one with smallest average projected area.

3.3 Results

We implemented a specialization of the general algorithms in §3.1 and §3.2 for the example cases of §2.3, which are the cases of most practical use. Figures 7 to 9 show some pictures made with our implementation in the public-domain raytracer *rayshade*.

Figure 7 shows a variation of the famous Barnsley fern [1], a prototype of a hierarchical IFS, in this case with 6 transformations in 3 levels. Figure 8 shows a temple-like construction and is described by a CIFS-code with 4 transformations and a composite condensation set. The scene in figure 9 is described by a hierarchical code with also 3 levels and in total 4 transformations with a geometry for a branch in the lowest level and an alternative geometry for a leaf, as proposed in §3.1 for drawing trees à la [11]. All these scenes are composed of several hundreds of tousands of primitive objects, making it impossible to compute a list of primitive objects and raytrace these using e.g. space partitioning to improve efficiency.

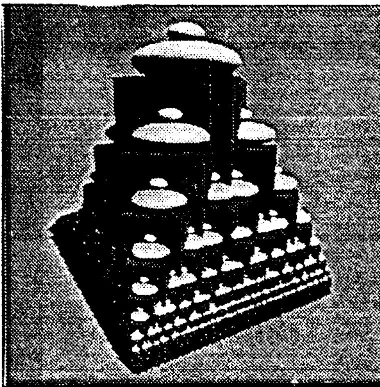

Figure 8: Temple: a CIFS with 4 transformations and composite condensation set.

The raytracing of these images takes about 15 minutes and 100 Kbytes of memory on an IBM RS/6000-320 system. The examples were chosen to show that quite complex pictures can be made from an extremely short description, a property of fractal-like objects. The raytracing can be done in reasonable time and with almost negligible memory usage.

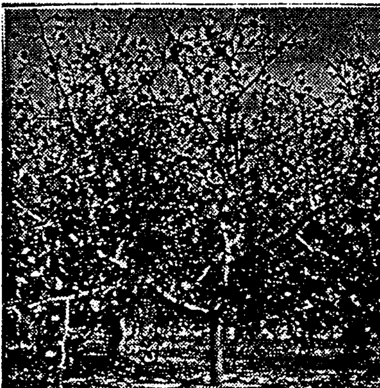

Figure 9: Trees: described by a code with 4 transformations in 3 hierarchical levels, a condensation set and alternative geometry.

4 Conclusion

In §2 we presented a formalism that can describe arbitrary linear graftals. IFS-codes, CIFS-codes, HIFS-codes emerge as simple examples. In §3 we presented an algorithm for raytracing objects described by this formalism, which is very general. We implemented a specialized version, suitable for the interesting cases of multilevel HIFS and CIFS codes. Our experience is that we now have a very powerful and easy to use tool for describing very complex objects that are not necessarily purely selfsimilar. The rendering of these objects is feasible.

As yet, however, we did not implement the most general algorithm. Our conclusions are based on a specialized version. One other point that may require further work is to investigate under which conditions the conversion of L-systems into our formalism is possible and/or useful. There are also some variants of L-systems (stochastic L-systems, ...) which might find an analog in our formalism. Nothing is known, as far as the author is aware of, about the usefulness of such variants. Since initial bounding volumes play an important role for the efficiency of the raytracing, some more work can be done on this subject too.

References

[1] M. Barnsley. *Fractals Everywhere*. Academic Press Inc, 1988.

[2] M. F. Barnsley, L. Hodges, and B. Naylor. Harnessing chaos for image synthesis. *Computer Graphics*, 22(4):131, 1988.

[3] S. Demko. Construction of fractal objects with iterated function systems. *Computer Graphics*, 19(3):271, 1985.

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

[5] A. S. Glassner, editor. *An Introduction To Ray Tracing*. Academic Press Ltd., 1989.

[6] J. C. Hart and Th. DeFanti. Efficient antialiased rendering of 3d linear fractals. *Computer Graphics*, 25(4):91, July 1991.

[7] D. Hepting, P. Prusinkiewicz, and D. Saupe. Rendering methods for iterated function systems. In H. O. Peitgen, J. M. Henriques, and L. F. Penedo, editors, *Fractals in the fundamental and applied sciences*, pages 183–224. Elsevier Science Publishers (North-Holland), 1991.

[8] A. Lindenmayer. Mathematical models for cellular interactions in development, parts i and ii. *Journal of Theoretical Biology*, 18:280–315, 1968.

[9] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Co. (San Fransisco), 1982.

[10] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. *Computer Graphics*, 23(3):41–50, 1989.

[11] P. E. Oppenheimer. Real time design and animation of fractal plants and trees. *Computer Graphics*, 20(4):55–64, 1986.

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