Representation and Realistic Rendering of Natural Phenomena with Cyclic CSG-Graphs*

Michael Gervautz Christoph Traxler

Institute of Computer Graphics
Technical University of Vienna
Karlsplatz 13/186-2
1040 WIEN
AUSTRIA

email: gervautz@cg.tuwien.ac.at
phone: +43(1)58801/4579
fax.: +43(1)5874932

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

Abstract

In this paper we introduce a method for ray tracing of recursive objects defined by parametric rewriting systems. Constructive Solid Geometry (CSG) is used as underlying object representation. Thus the formal languages of our rewriting systems are subsets of the infinite set of CSG expressions. Instead of deriving such expressions to build up large CSG trees we translate the systems into cyclic CSG graphs, which can be used directly as an object representation for ray tracing. For this purpose the CSG concept is extended by three new nodes. Selection nodes join all the rules for one grammar symbol, control the flow by selecting proper rules, and are end-points of cyclic edges. Transformation nodes map the rays in affine space. Calculation nodes evaluate a finite set of arithmetic expressions to modify global parameters, which effect flow control and transformations. The CSG graphs introduced here are a very compact data structure, much like the describing data set. This property meets our intention to avoid both restrictions of the complexity of the scenes by computer memory and the approximation accuracy of objects.

Key Words: L-Systems, CS G Graphs, Ray Tracing, Natural Phenomena, Fractals

1 Introduction and motivation

Several methods for modeling and realistic visualization of natural phenomena have been developed in the history of computer graphics. These iterative or recursive algorithms are more or less related to Mandelbrot's Fractal Geometry of Nature [MAND82] and are capable of generating very complex objects out of a very small data set. This common property was called database amplification in [SMIT84]. Though most of these methods were designed to generate a specific class of objects, some of them have been proved to be more powerful. This is especially true for parallel rewriting systems, which were first used by the biologist Lindenmayer for the simulation of the development of plants [LIND90]. Parametric Lindenmayer Systems, briefly called PL-systems, are the most important extension of this method.

PL-systems operate on strings of modules, which form the alphabet of the grammar, and are associated with a finite number of parameters. A set of rewriting rules for each module determines a string of modules that substitutes one module in the current string. In such a derivation step the arithmetic expression for each parameter of the module is evaluated. The resulting values can control the selection of rules for the current substitution step, and determine the derivation sequence in this way. It is an essential aspect of PL-systems that the geometry of the generated figures is affected by the parameter values as well. The resulting module string is interpreted from left to right by a virtual construction tool called turtle, which builds up an object representation according to the chosen visualization method. Thus the final shape of an object is the result of a series of derivation steps, which can be seen as development stages of its geometry and topology or simply as a growth process. PL-systems are described in detail in [LIND90] and [PRUS90]. They allow the definition of a variety of recursive objects, the class of linear fractals, for example. Carpenter's method for the generation of fractal terrain can be expressed as a rewriting system as first described by Smith in [SMIT84]. Many other techniques for the simulation of plant development can be mapped onto PL-systems, like the model of deReffye [REFF88].

Ray tracing is the preferred rendering technique for realistic visualization and is also suitable for outdoor scenes. For recursive objects the ray-tracing algorithms can be classified in two classes: those, where the representation of the scene has to be built up completely before any intersection calculations can be done, and those, where this representation evolves during intersection calculation. Ray tracing of objects defined by PL-systems belongs to the first class. The representation of the scene has to be built up completely by the turtle before ray-tracing, because the turtle cannot interact with the intersection algorithm. Both, the complexity of the scene and the accuracy of single objects are therefore restricted by memory space.

Building up the scene during visualization has the advantage that only those parts of a scene are created, which might be intersected by a ray. Such a method was applied for the ray tracing of fractal terrain by Kajiya and was introduced in [KAJI83]. A prism, better known as cheese-cake extent, is used as a bounding volume for a part of the fractal, that is subdivided by Carpenters method if a ray hits the extent. In this case four new prisms are created, which enclose four smaller parts of the fractal. The size of each extent must be estimated from stochastic properties, because it should enclose a particular part of the fractal surface that is not known in advance. The triangles inside the prisms serve as primitive elements for a certain approximation degree. The final degree may depend on the projection area of the triangle on the screen. If it fits the size of a pixel no further subdivisions are necessary. Bouville [BOUV85] has improved this method by the use of ellipsoids instead of prisms.

A ray tracing method of the second class for linear fractals defined by IFS was introduced by Hart and deFanti in [HART91]. An IFS is a finite set of contractive transformations that are iteratively applied on a primitive object to generate an approximation of the fractal, which is also called *attractor* of the IFS [BARN88]. In the method of Hart and deFanti visualization is done by transforming the rays using the inverse transformations of the IFS and intersecting them with an appropriate bounding volume. Geometrically this has the same effect as transforming the bounding volume, which also serves as a primitive object. The iteration stops when the current ray does not hit anything or the corresponding bounding box is small enough. Antialiasing is achieved by storing the normal vectors of the whole iteration sequence and calculating a weighted average for the final normal vector.

The method which is introduced here is related to both, the ray tracing technique of Hart, deFanti [HART91], and Kajiya [KAJI83], and to PL-systems. Rewriting systems are a more powerful and flexible modeling technique than IFS. The systems we use are based on Constructive Solid Geometry (CSG), because this is an efficient object representation for ray tracing. Thus the formal languages of our rewriting systems are subsets of all possible CSG expressions. Instead of deriving such expressions we translate the systems into cyclic CSG graphs, which are directly used as representations for ray tracing.

The following section gives a definition of parametric rewriting systems for CSG expressions, and describes their translation into CSG graphs. How ray tracing with CSG graphs is done is explained in the third section. In section four we introduce a parametric CSG-system (PCSG-system) for a branching structure and a general system for the generation of fractal terrain. In the fifth section problems with conventional optimization techniques are discussed. The last section gives an overview of our further research.

2 Building cyclic CSG graphs

Constructive Solid Geometry (CSG) is one of the most efficient object representations for ray tracing. Many types of simple solids can be combined by three Boolean operators to build up complex scenes. Thus each scene is defined by a binary expression consisting of CSG operators, primitives and brackets. The operator $\cup$ creates the union of two volumes, operator $\cap$ their intersection and the operator \ subtracts the volume of the second operand from the volume of the first. Transformations are either associated with primitive objects or incorporated into CSG expressions as unary operators. For rendering, the expressions are translated into binary trees, called CSG trees, which are traversed by the ray-intersection procedure. The transformation matrices are usually stored in the leaves of a CSG tree, so that a ray has to be mapped from world space into primitive object space only once. This significantly saves computation time during rendering. All intersections of a ray with primitives in the leaves of a CSG tree are collected in a list. Finally the Boolean combinations can be done directly with intersection intervals of the ray. The three-dimensional problem of combining volumes is thereby reduced to the one dimensional problem of combining intervals of a ray [ROTH82].

An alternative data structure for the representation of CSG expressions is a directed acyclic graph (DAG), where primitive objects are stored only once, a feature that was called object instancing in

[RUBI80]. This implies that transformations are stored as internal nodes of a CSG-DAG, which have only one successor. This structure saves memory, if the description of the primitive is large but when used with ray tracing, a ray has to be transformed more than once on his way through the graph to a primitive node.

2.1 PL-systems for CSG expressions

Because CSG expressions are strings, it is possible to derive them from a parameterized L-System (PL-system). As we have noted above, it is an essential aspect of PL-systems that the geometry of the generated objects evolves from the derivation sequence. For this reason we will specify transformations as unary operators in our CSG expressions, which depend on parameters. Considering transformations, the set of all valid CSG expressions is the formal language of the following Chomsky system (fig. 1).

```
w : E                         // the axiom
p1 : E → E op E               // op ∈ {∪, ∩, \}
p2 : E → trans E              // trans ∈ {rotx(α), roty(β), rotz(χ), move(dx,dy,dz), scale(fx,fy,fz), uscale(f), ...}
p3 : E → (E)
p4 : E → obj                  // obj ∈ set of all primitive objects
```

Fig. 1: The Chomsky system defining the set of all valid CSG-expressions.

In this system there are two types of symbols: variables and terminals (denoted by capitals and small letters respectively). Only variables can be substituted and therefore there are only rules for the symbol E. To obtain a valid CSG expression the last rule is applied to all remaining E symbols, so all the variables are substituted by terminals and no further derivation step can be done. The formal language of every PL-system for CSG expressions has to be a subset of the formal language of the Chomsky system described above and will be called *PCSG-system* in the following text. This implies that the derivation sequence can not be stopped arbitrarily like in PL-systems, where the turtle ignores modules that do not belong to its command set. Here the resulting string is translated into a CSG tree by a parser, which needs a syntactic correct CSG expression. For this reasons we have to be careful when designing a PCSG-system. First, rules can only be applied to modules (*generating rules*), and second, at least one rule must exist for every module, that finally substitutes all variables with a string of terminals (*terminating rules*). The first PCSG-system we introduce (fig. 2) generates the well known Sierpinski tetrahedron, one of the oldest linear fractals.

```
w:S(6)                                                              // the axiom
p1:S(c) : c=0 → tetrahedron                                         // terminating rule
p2:S(c) : c>0 → move(0.5,0.5,0.5) uscale(0.5) S(c-1) ∪              // generating rule
                  move(-0.5,-0.5,0.5) uscale(0.5) S(c-1) ∪
                        move(0.5,-0.5,-0.5) uscale(0.5) S(c-1) ∪
                              move(-0.5,0.5,-0.5) uscale(0.5) S(c-1)
```

Fig. 2: An attributed grammar for the Sierpinski tetrahedron.

Two rules for one module with only one parameter is sufficient for this system. The parameter *c* is initialized in the axiom and determines the order of the approximation. First rule p2 is applied to the axiom, because its condition *c>0* is true. The resulting string is a ∪-combination of four modules, which are translated and uniformly scaled by the transformations *move()* and *uscale()*. In each derivation step the parameter *c* is decreased for all modules. As long as *c* meets the condition of p1 this rule is used again. At the end, when *c* becomes zero the condition of p2 is no longer true but that of p1. Now all modules are replaced with the primitive object *tetrahedron*. This terminates the derivation sequence and the resulting string is a valid CSG expression. Interpreted geometrically each tetrahedron is substituted by four copies of itself, with halved edge length of the original and which are moved into each of its corners. Picture 1 shows an approximation of the sixth order. Now we have seen how using PCSG expressions. We can produce CSG-expressions, which can then be converted into CSG trees. In the next chapter we will explain how to make cyclic CSG graphs out of these systems.

2.2 Translation of PCSG-systems into cyclic CSG graphs

Let us first interpret the right hand side of each rule (the string following the derivation symbol →) as a CSG expression and their modules as special elements. In this case we can build a CSG tree out of each right hand side. The unary transformation operators are represented by internal nodes as described for CSG-DAGs above with the extension that they are able to access parameters. We will call them *Transformation nodes*, *T-nodes* for short. For each module in the alphabet of a given PCSG-system exactly one *Selection node*, *S-node* for short, is created, which contains all the rules of the module. Thus the number of successors of a S-node is equal to the number of rules for the module it represents. When translating the right hand side of all rules for a particular module into CSG trees, we connect an edge for each instance of a module in the right hand side with the corresponding S-node. In this way a cyclic CSG graph is formed. Instead of selecting successors by evaluation of separate conditions, we supply each S-node with only one function, which returns the index of one successor.

So far, what is missing is a mechanism for the modification of parameters. In PL-systems their values are determined separately for each instance of a module by arithmetic expressions. This implies that each instance of a module has its local set of parameter values, which allows distinct substitutions in one derivation step. For parameter handling we introduce another kind of nodes called *Calculation node*, briefly *C-node.* They are related with a finite set of assignment expressions and evaluate them one by one. Like T-nodes they can considered to be as unary operators, since they are nodes with only one successor. Therefore it is not necessary that the successor of a C-node is a S-node as PL-systems imply but it can be any type of node. In the textual representation of rules, a C-node is defined by brackets, which enclose the assignment expressions separated by semicolons.

Figure 3b shows the CSG graph, that generates the Sierpinski tetrahedron. In figure 3a the corresponding PCSG-system using the new notation is shown. The selection function in this example evaluates its first argument, which is a condition, and returns its second argument if this condition is true and otherwise its third argument. The axiom consists of a C-node, that initializes the parameters and the start node.

```
{c=6} S                                          // definition of the start node and initialization of c
S if(c > 0,2,1)                          // declaration of the module S and its selection function
1: S → tetrahedron                               // terminating rule
2: S → {c=c-1} move(0.5,0.5,0.5) uscale(0.5) S ∪    // generating rule
                move(-0.5,-0.5,0.5) uscale(0.5) S ∪
                     move(0.5,-0.5,-0.5) uscale(0.5) S ∪
                          move(-0.5,0.5,-0.5) uscale(0.5) S
```

Fig. 3a: The Sierpinski tetrahedron in PCSG-notation.

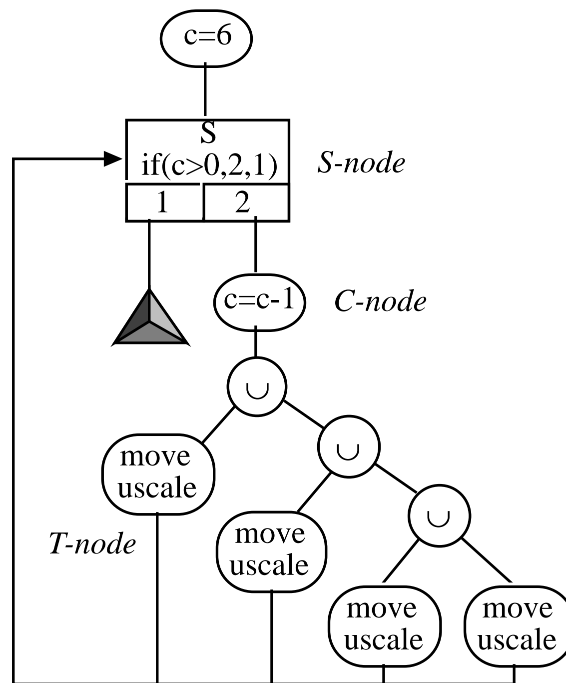


Fig. 3b: A CSG graph representing the Sierpinski tetrahedron

3 Ray Tracing with CSG graphs

Ray tracing is done simply by traversing the CSG-graph in a recursive manner. As in conventional ray-tracing with CSG-trees, the ray is intersected with all primitive objects and the intersection information is combined afterwards by the Boolean operations in the operator nodes. The newly introduced S-nodes also have a quite simple task to perform. S-nodes evaluate their selection function according to the parameters and pass the ray to the appropriate successor node. C-nodes change the parameter values according to the assignment instructions given.

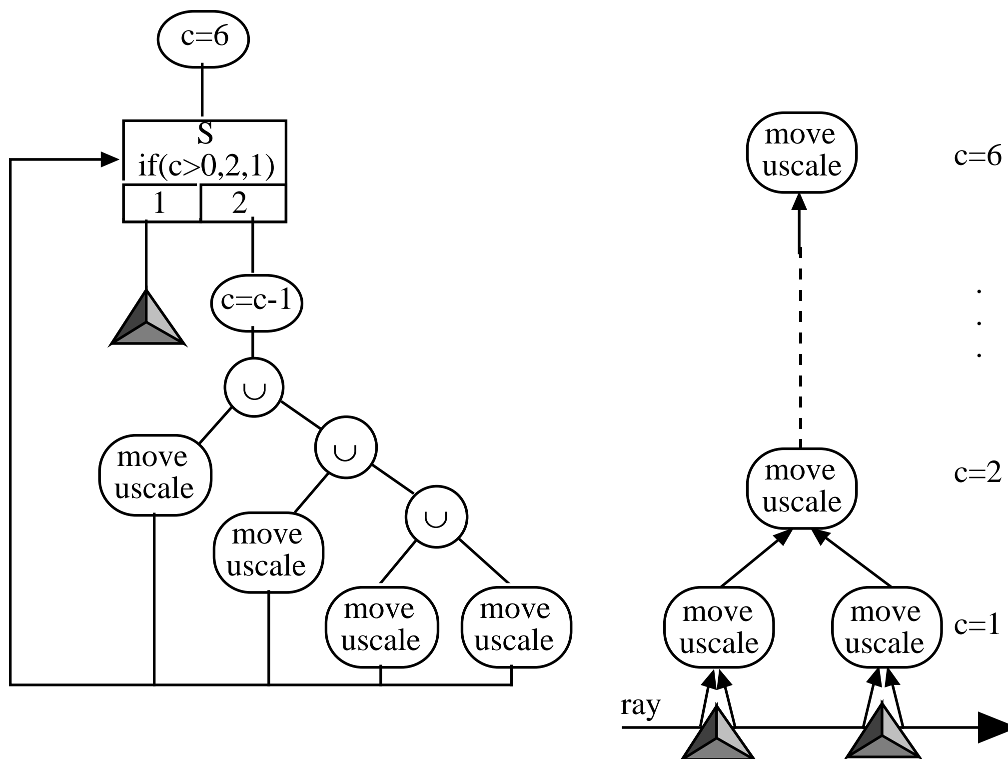


Fig. 4: During ray tracing an inverse binary tree of transformations is built up to calculate the normal of the first hit.

The task of T-nodes is a little bit more difficult than the others. In CSG trees rays are mapped with the transformation matrix that has been assigned to a primitive object. This is called world-to-object mapping. The normal vector at the first hit point resulting from an intersection with a primitive in object space is transformed back to the world coordinate system by the inverse transposed matrix. Similarly, a T-node transforms each ray with its matrix, which is calculated from the passed parameter values. The transformation of the normal vectors is not trivial anymore because more than one transformation has been performed on the ray. In every T-node on the way to a primitive the ray was transformed. Thus we have to store the matrices and map the normal vector of the first hit after all the cycles of the CSG graph have been terminated for the current ray. For this purpose we build up a list of matrices. A reference to this chain is passed through the graph together with the ray so that each T-node can append its matrix to the list. When a primitive object is reached, all its intersections with the current ray are associated with the current reference to the chain. Consider the frequent case where the predecessor and both successors of an operator node are T-nodes. This corresponds to a branch in the list. Both T-nodes append their matrices to the same element of the chain. The structure, which is built up in this way can be conceived as binary tree with inverted pointers (see fig. 4).

Ray-tracing CSG-graphs in this way leads to very realistic images of natural phenomena. CSG-graphs can be used for representing objects described by IFS-codes as well as plants and fractal terrain. Some examples are given in the next chapter.

4 CSG graphs for plants and fractal terrain

Let us first introduce a PCSG-system for a simple sympodial branching structure. In figure 5a the full definition of the tree is given, figure 5b shows the corresponding CSG graph, and picture 7 shows a resulting image. The trunk is built up by the second rule of module TR. The two parts of the third rule generate left and right branches with distinct angles, which are combined and attached to their mother branch by a $\cup$-node. The transformation above this $\cup$-node performs a rotation around the z-axis to avoid that all branches lie in the same plane. The left and right part of the structure are successively scaled so that the limbs get smaller and shorter depending on their order, determined by the parameter *cnt*. If *cnt* gets zero, the first rule is applied, which represents the twigs with the highest order. The order of a limb determines if its segments bear leaves or not. If *cnt* is beyond the threshold *noleaves,* the second rule of module SG is selected, which builds up a limb out of scaled cylinders with two leaves attached to them. These segments are defined as a CSG expression. The arrangement of the leaves according to the effect of phyllotaxis is achieved by successive rotations of the cylinders. The iterative rotations of the branches around the z-axis scatter the orientation of the leaves in various different directions. To simulate the effect of heliotropism it is necessary to correct the orientation of leaves. For this purpose two angles must be calculated dependent on the current orientation of a leave and the direction of the light source.

Initialization of the parameters

```
{       cnt = 11;                  // order of the branching structure
        trunk = 2,                 // number of trunk segments (cycles of rule with index 2)
        fTR = 0.96;                // scaling factor for trunk segments
        gamma = -72.0;             // divergence angle
        alpha1= -60.0;             // branching angles
        alpha2 = 20.0;
        fBR1 = 0.73;               // scaling factors for left and right branches
        fBR2 = 0.67;
        noleaves = 3;          // if cnt is less than noleaves the limbs bear leaves
        segments = 2;              // number of segments for a limb with leaves
        fxSG = 1/7;                // scaling factors for segment cylinders
        fySG = 1/7;
        dSG = 1/segments;          // height of a segment cylinder
        fLV = 0.6;                 // scaling factor for leaves
} TR                               // the start node
```

The rules for TR and SG

TR   if(trunk > 0, 2, if(cnt > 0, 3, 1))   // module for the tree
1: TR → SG   // terminating rule
2: TR → SG ∪ {trunk=trunk-1} move(0,0,1) uscale(fTR) TR   // rule for the generation of the trunk
// rule for the generation of the branching structure
3: TR → SG ∪ move(0,0,1) rotz(gamma)({cnt=cnt-1} rotx(alpha1) uscale(fBR1) TR ∪
{cnt=cnt-1} rotx(alpha2) uscale(fBR2) TR)
SG   if(cnt < noleave, if(segments > 0, 3, 2), 1)   // module for segments
1: SG → scale(fxSG,fySG,1) cylinder   // this rule is selected for limbs without leaves
2: SG → segment   // terminating rule for limbs with leaves

3: SG → segment ∪ {segments=segments-1} move(0,0,dSG) rotz(-90) SG   // generating rule for limbs with leaves
// CSG expression for segments with leaves
segment = scale(fxSG,fySG,dSG) cylinder ∪ move(0,0,dSG) (uscale(fLV) leaf ∪ flipxz uscale(fLV) leaf)

Fig. 5a: A PCSG-system for a simple sympodial branching structure.

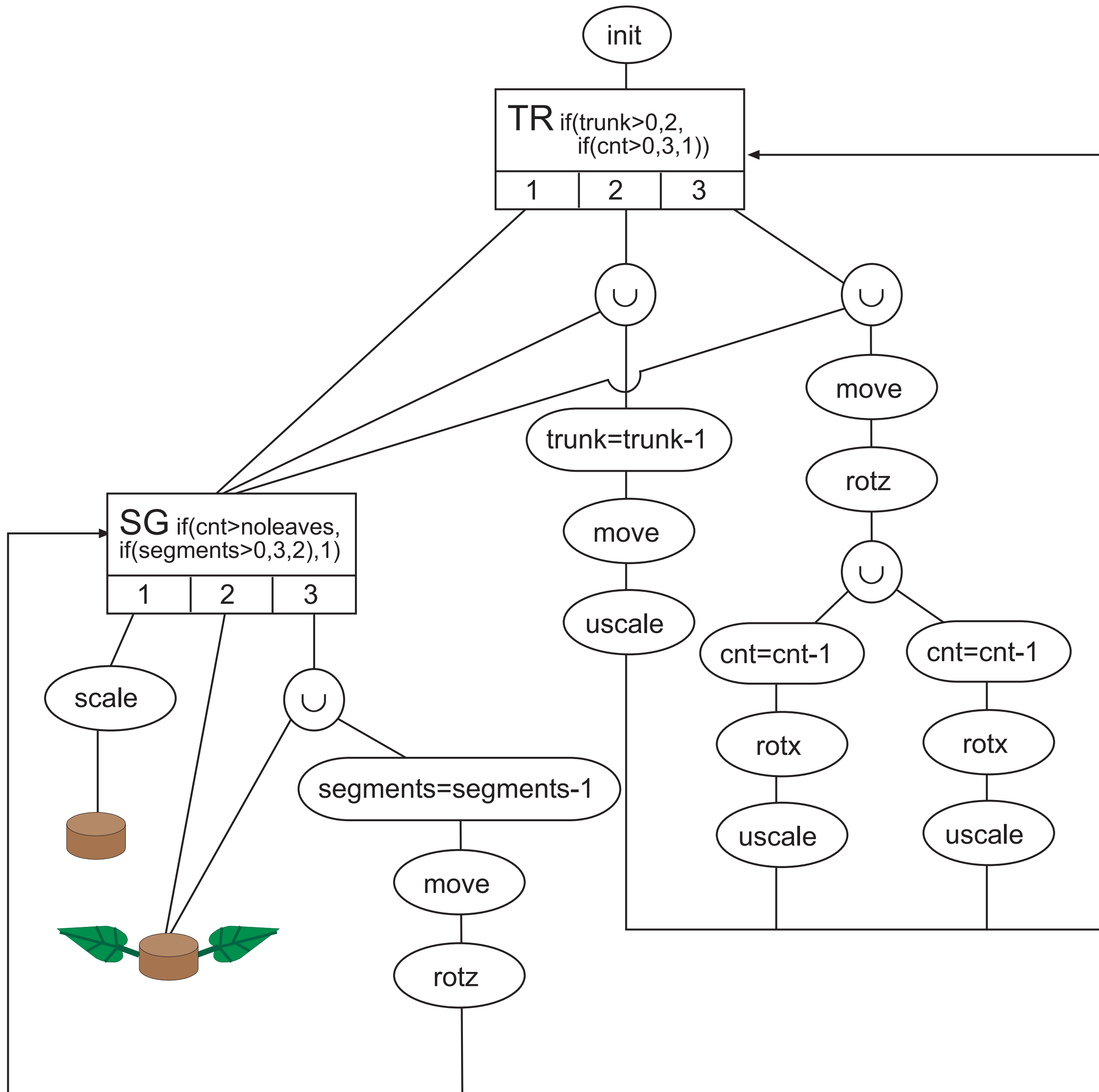


Fig. 5b: The CSG graph that generates the sympodial branching structure

A good example for the advantage of modeling with CSG is the construction of leaves out of two hollow spheres and a cylinder. In figure 6 this construction is reprinted in the CSG notation. A much simpler shape for leaves can be created by intersecting the cylinder with only one hollow sphere. Both methods are shown in picture 3.

```
connected_balls =   (move(-D,0,0) sphere ∪ move(D,0,0) sphere) \
                    (move(-D,0,0) uscale(1-thickness) sphere ∪ move(D,0,0) uscale(1-thickness) sphere)
leaf = move(0, dy, dz) scale(fx, fy, fz) cylinder ∩ connected_balls
```

Fig. 6: The construction of a simple leaf using CSG-operations.

Three modules were used to generate the palm tree of picture 4. This results in a CSG graph with three S-nodes, which can be seen as a combination of three graphs as well. The rules of the first module build up the bent trunk, those of the second module the palm top and those of the third the twigs. The conifer tree of picture 6 is generated in a similar way. One sub-graph builds up the trunk and attaches the branches to it, which are generated by another sub-graph. The flower of picture 5 was defined by two modules. One represents the stage of vegetative growth and the other the stage of florescence. Picture 8 shows the result of a stochastic PCSG-system, where the rules are selected by a random function. The rules for vegetative growth, the generation of branches and the termination of an apex with a tip are selected with different probabilities.

Finally we introduce the PCSG-system, which generates a fractal terrain by the method of Carpenter [CARP80]. For this purpose we use triangles as primitive objects. To obtain a rectangular base we merge two triangles T1 and T2 which are represented by two different S-nodes in the graph, and their subdivision is represented two sub-graphs. As Carpenter et.al. [CARP80] suggested, the subdivision of a triangle is done by dividing each edge (figure 7) so that we get four smaller triangles which are subdivided again. Because the center triangle is of the other type (it is mirrored) it has to be represented by the other S-node. This yields to a CSG graph with two S-nodes, each representing a triangle and one edge of each S-node leading to the other S-node.

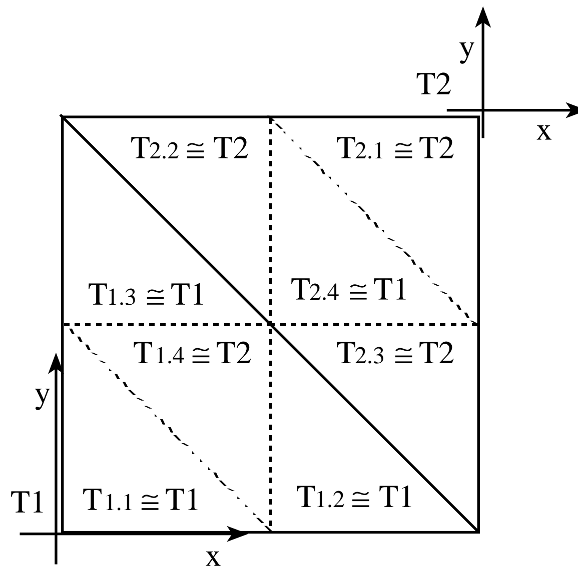


Fig. 7: The subdivision of the two triangles T1 and T2

The height of the new triangle points is calculated by the midpoint displacement method with exponential distributed random numbers. During subdivision triangles are only moved and scaled in the xy-plane. The z-coordinates of their points are stored in parameter values and determined by the C-nodes. The terminating rule shears and shifts the triangles along the z-axis to put them into their final shape and position.

The implementation of midpoint subdivision with CSG graphs leads to problems with shared edges. Since triangles are subdivided at different times and in unpredictable order we must guarantee to get

the same displacement value for each edge, which belongs to two triangles. The problem is solved by pseudo random numbers that depend on both, the start and end point of an edge projected onto the xy-plane. These endpoints are calculated by C-nodes and are used in a linear combination to evaluate an unique seed value for the pseudo random generator. The shape of the fractal terrain depends on the values of the four coefficients for the linear combination. In the PCSG-system given in figure 8 we use the ´#define´ directive to specify macros like in C. Only the specification of the C-node for the triangle 1.1 (see figure 7) is printed in that figure, because the other seven are defined likewise. A result is shown in picture 9. Here the CSG graph was combined with a reflective cube with a bump map to model a sea. Picture 2 shows a Landsberg relief as described by Mandelbrot in [BARN88, Appendix A]. Here the displacement values for the midpoints is scaled down by the constant number 1.8 in each subdivision step.

Predefinition of the C-node for triangle 1.1 (see figure 7)

```
#define calcT1_1
    {   cnt = cnt-1;
        delta = delta*frc;                               // variance
        kl = kl/2.0;                                     // new length of the cathetii
        sd = a*x2 + b*x1 + c*y2 + d*y1;                  // seed value
        rnd = ((-ln(rand(sd)*ex+1.0))+add)*delta;        // displacement value
        z2 = (z1+z2)/2.0 + rnd;                          // new height for p2
        sd = a*x3 + b*x1 + c*y3 + d*y1;                  // next seed value
        rnd = ((-ln(rand(sd)*ex+1.0))+add)*delta;        // next displacement value
        z3 = (z1+z3)/2.0 + rnd;                          // new height for p3
        x2 = x1-kl;                                      // new x-coordinate for p2
        y3 = y1-kl;                                      // new y-coordinate for p3
    }
```

Initialization of the parameters

```
{   cnt = 10;                    // order of the terrain
    frc = 0.47;                  // scaling factor for the variance
    delta = 1.0;                 // initial variance
    add = 0.6;                   // addition value for the random numbers
    ex = exp(1.0)-1.0;           // for the mapping of the random numbers into [1,e]
    kl = 1.0;                    // lenght of a cathetus
    a = 255;                     // coefficients for the linear combination
    b = pow(255,2);
    c = pow(255,3);
    d = pow(255,4);
} MNT
```

```
                                        The rules
MNT    1                                                          // a constant selection function
1:MNT →     {                                                     // initialization of the points for triangle T1
                x1 = 0.0;       y1 = 0.0;       z1 = 0.0;
                x2 = 1.0;       y2 = 0.0;       z2 = 0.0;
                x3 = 0.0;       y3 = 1.0;       z3 = 0.0;
            } T1 ∪
            {                                                     // initialization of the points for triangle T2
                x1 = 0.0;       y1 = 0.0;       z1 = 0.0;
                x2 = 1.0;       y2 = 0.0;       z2 = 0.0;
                x3 = 0.0;       y3 = 1.0;       z3 = 0.0;
            } move(1,1,0) T2
T1      if(cnt > 0, 2, 1)                                         // triangle on the upper right part of the xy-plane
1:T1 →      calc_TRI1 move(0,0,z1) zshear(z2-z1,z3-z1) tri1  // terminating rule for triangle T1
2:T1 →      calcT1_1 scale(0.5,0.5,1) T1 ∪                        // subdivision of triangle T2
                calcT1_2 move(0.5,0,0) scale(0.5,0.5,1) T1 ∪
                    calcT1_3 move(0,0.5,0) scale(0.5,0.5,1) T1 ∪
                        calcT1_4 move(0.5,0.5,0) scale(0.5,0.5,1) T2
T2      if(cnt > 0, 2, 1)                                         // triangle on the lower left part of the xy-plane
1:T2 →      calc_TRI2 move(0,0,z1) zshear(z1-z2,z1-z3) tri2       // terminating rule for triangle T2
2:T2 →      calcT2_1 scale(0.5,0.5,1) T2 ∪                        // subdivision of triangle T2
                calcT2_2 move(-0.5,0,0) scale(0.5,0.5,1) T2 ∪
                    calcT2_3 move(0,-0.5,0) scale(0.5,0.5,1) T2 ∪
                        calcT2_4 move(-0.5,-0.5,0) scale(0.5,0.5,1) T1
```

Fig. 8: A PCSG-system for a fractal terrain.

As introduced so far CSG graphs are a compact representation for ray tracing of complex recursive objects. But the method is very time consuming, because a ray has to pass through all possible cycles of a graph to make sure that no intersections are missed. For high order approximations of fractals or graftals this is impractical. We need a mechanism to decide whether a ray intersects anything during the current cycle or not. In the last case the cycle can be terminated. However, the application of conventional optimization techniques on CSG graphs leads to problems, which are discussed in the next chapter.

5 Improvement techniques

Bounding volumes are used for CSG trees to optimize ray tracing. The most common type are axis aligned boxes or spheres. Both the Boolean combinations of two boxes and spheres, and the intersection calculation with a ray are quite simple. Nevertheless they are harder to calculate for CSG graphs than for CSG-trees.

As we have seen in the last chapter almost every node in the graph represents more than one part of the final object. Beside that, these different objects are defined in different coordinate frames. Therefore the bounding volume of such a node has to enclose all these objects. One solution would be to store different bounding volumes with different sizes and locations for each use of the node. But this would need a large amount of memory, almost as much as a conversion of the graph into a large cycleless tree. To avoid storing a huge amount of bounding boxes, and loosing the advantage of low memory usage, we store just one box that we call hyper-bounding box. The hyper-bounding box encloses all objects represented by this one node, and can be used in every cycle to discard non intersecting rays.

There are no problems with hyper-bounding boxes if the transformations of a CSG graph form an IFS-code, i.e. if they are contractive. This is true for the graph that generates the Sierpinski tetrahedron (Figure 3). It consists of four parts, which are again scaled and translated versions of a Sierpinski tetrahedron. In this case the hyper-bounding box encloses each part as tight as the whole object. For non contractive systems the hyper-bounding boxes will only fit to the root node of the graph, for some sub-graphs it will be too large. However, due to the hierarchical behavior of the structure a bounding box for deeper levels of recursion can be seen as the intersection of all the

bounding boxes lying on the way from the root-node to that specific node. As long as there are any transformations involved -- and no transformation would not make any sense -- this intersection is smaller for each recursion and therefore these bounding boxes are useful not only for the root. On the other hand, we found out that in practice almost every system of natural phenomena can be expressed as contractive system.

Hyper-bounding boxes can be calculated the following way in a preprocessing step before rendering: The graph is traversed in reverse order, i.e. from the leaf-nodes (the primitives) back to the root. For primitive nodes an adequate bounding box is calculated out of the geometry of the primitive, which is also used as hyper-bounding box for those nodes. For each non primitive node the following steps are taken:

- Calculate a new bounding volume out of the bounding volumes (not the hyper-bounding volumes!) of the successor(s). An operator node combines the volumes of its successors and calculates a new volume that encloses the result. Similarly a T-node maps the bounding volumes of its successor and calculates a new one, that encloses the transformed one. For C- and S-nodes no calculation has to be done.
- Join this bounding volumes with the hyper-bounding volume to generate a new one.

After these steps every node contains an accumulation of all bounding boxes in its hyper-bounding box.

The simple geometry of the conventional bounding volumes is another problem when using multiple transformations. Imagine a CSG graph that iteratively rotates a cylinder about 45•. Figure 9 shows the successive bounding boxes. The rotation about 45• violates the geometric properties of the boxes, because their edges are no longer parallel to the axes of the world coordinate system. The T-node has to calculate a new box with the requested properties that encloses the rotated one. The following sequence demonstrates how fast these boxes grow. To avoid this problem the bounding volume of a T-node has to be calculated directly out of the bounding boxes of the primitives defined in the primitive's coordinate frame and the transformation that is performed on those primitives in the T-node's coordinate frame.

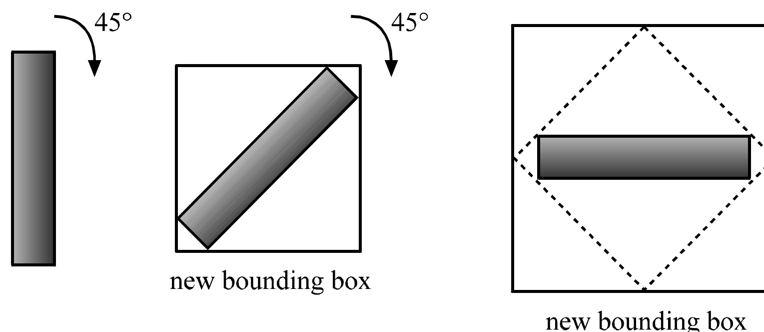


Fig. 9: Growing bounding boxes during rotational transformation

In this way, bounding boxes are calculated, which can be used to remove rays during the intersection calculation. Ray tracing is done in the described way. Because in T-nodes a ray is transformed into a local coordinate frame and bounding-boxes are defined relative to this coordinates, there is a new chance to remove the ray with the same bounding box in each cycle.

Table 1 gives some statistics on the pictures. These are the average rays per second, the number of primitive objects, that are necessary for an expansion of the CSG graph into a CSG tree, and the rendering time. All pictures were rendered on an INDY-Workstation with a R4000SC. Antialiasing was achieved by adaptive oversampling.

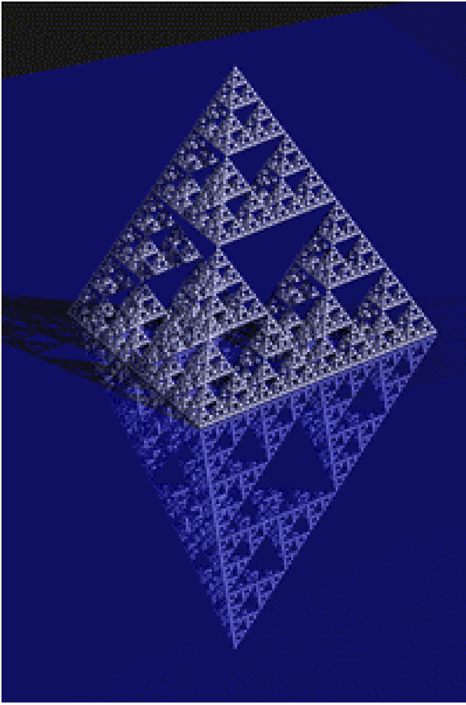
Table 1	rays per second	primitives equivalence	rendering time
picture 1	526.67	4097	02:04:44
picture 2	7.46	131072	52:41:17
picture 4	48.5	6542	07:56:45
picture 5	235.64	2050	00:44:00
picture 6	90.98	70027	04:18:15
picture 7	91.54	45057	05:49:43

6 Conclusion and future work

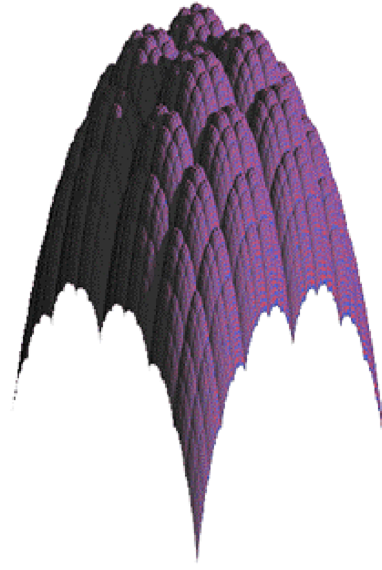
PCSG-systems and their translation into cyclic CSG graphs leads to a compact data structure for ray-tracing recursive objects, that allows a very high degree of complexity, because the scene is not built up explicitly. This method is capable of visualizing linear and stochastic fractals, plants as well as fractal terrain described by the method of Carpenter [CARP80]. The concept of hyper-bounding boxes enables the use of bounding-boxes for CSG-graphs, too.

A higher degree of optimization should be achievable by the additional use of a 3d-grid. Both a regular grid, called SEAD in [FUJI86], or an octree should be suitable. Each voxel of the grid is either empty or represents a state of a CSG graph. Such a state is defined by a transformation matrix, a set of values for the parameters and a reference to a node of the graph. The voxels hit by a ray can be calculated by 3DDDA as described in [FUJI86]. The CSG graph is set to the state associated with the foremost voxel, that is not empty and the current ray is passed to the corresponding node. In the best case this node is a primitive object, i.e. a terminal node of the graph. Usually this node will be an internal node so that cycles are necessary. But we are convinced that even a coarse grid is sufficient to minimize the amount of cycles that a ray has to pass until it can hit a primitive. Nevertheless we additionally need bounding volumes, which enclose the objects very tight and allow an early termination of cycles as mentioned in chapter 4. Under consideration are ellipsoids and parallel planes to approximate the convex hull of an object as described in [KAY86].

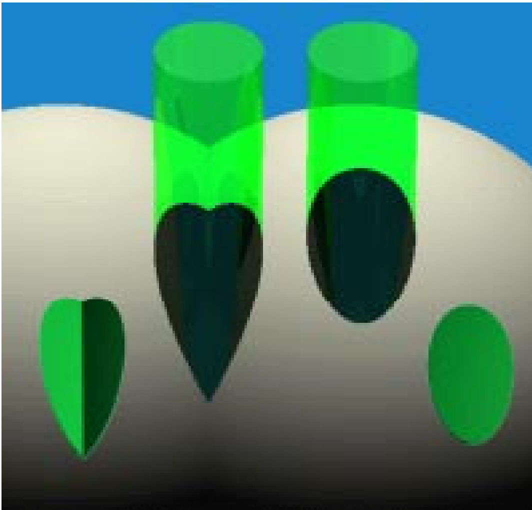
Another topic of our research is the animation of recursive objects. Of primary interest hereby is the animation of plant development. For this purpose a CSG graph is controlled by discrete clock ticks. Both the topological and the geometrical shape of a plant is a function of time. Topological development is achieved by switching rules at certain ticks, and geometrical development means that transformations depend on time. In fact this is done by setting parameter values according to time dependent functions. Animation of evolved plants is easier to realize. The motion of branches in wind is only a modification of the geometry. The branching angels change depending on the direction and speed of the wind and the weight of the limbs.



Picture 1



Picture 2

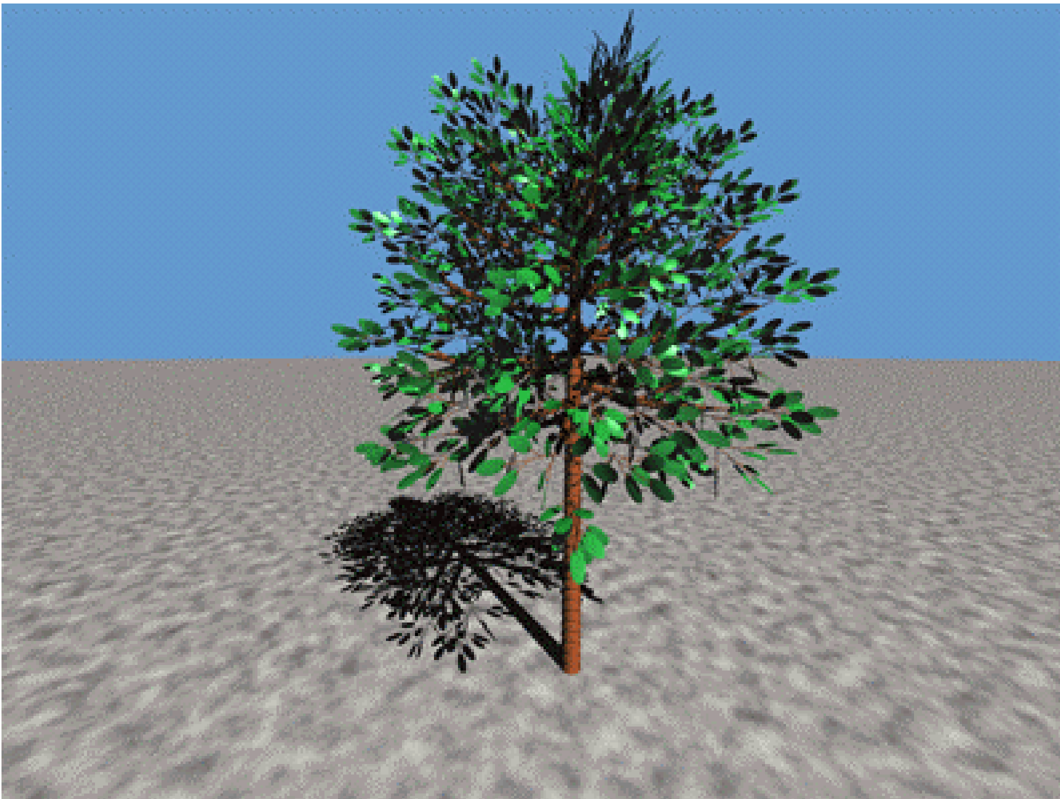


Picture 3

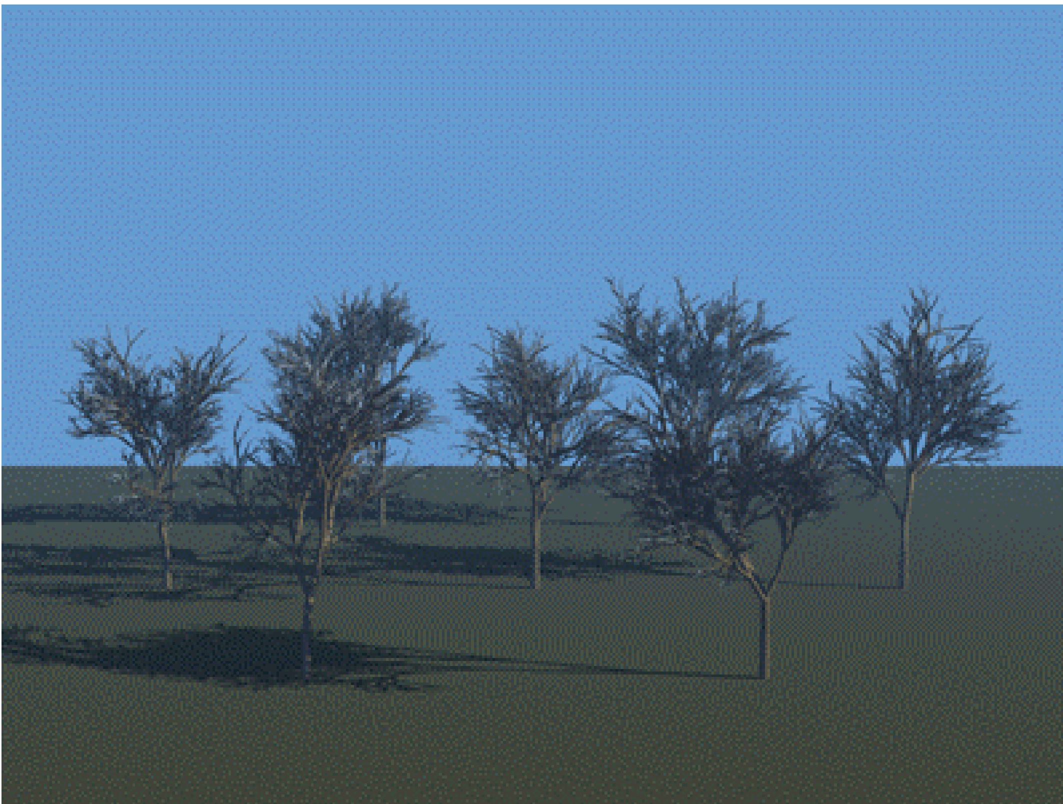
Picture 4

Picture 5

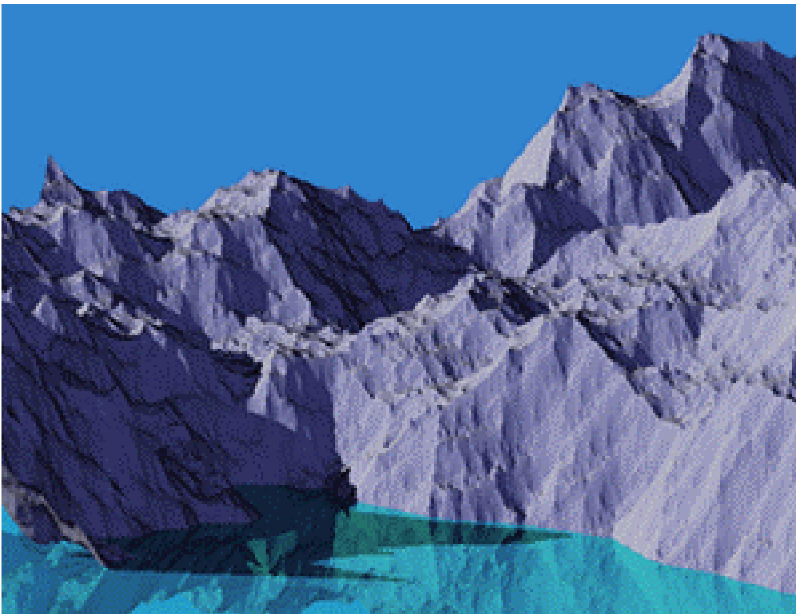
Picture 6



Picture 7



Picture 8



Picture 9

Acknowledgments

This work is supported by the “Fond zur Förderung der wissenschaftlichen Forschung (FWF)”, Austria, (project number: P09818). We would like to thank Robert F. Tobler and Wolfgang Stürzlinger for the implementation of the ray tracing system, we built upon and for helpful discussions.

References

[BARN88] Barnsley M.F (1988) Fractal modeling of real world images. In Peitgen H.O, Saupe D (ed):The Science of Fractal Images, Springer Verlag, New York, pp. 219-242

[BOUV85] Bouville C (1985) Bounding ellipsoids for ray-fractal intersection. ACM Computer Graphics SIGGRAPH Proc., Vol. 19(3), pp. 45-52

[CARP80] Carpenter L.C (1980) Computer rendering of fractal curves and surfaces. ACM Computer Graphics SIGGRAPH Proc. Suppl.

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

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

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

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

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

[MAND82] Mandelbrot B. (1982) The fractal geometry of nature. W.H.Freeman & Co.

[PRUS90] Prusinkiewicz P, Hanan J (1990) Visualization of botanical structures and processes using parametric L-systems. Scientific Visualization and Graphics Simulation,Wiley & Sons, pp. 183-201

[REFF88] DeReffye P, Edelin C, Francon J, Jaeger M, Puech C (1988) Plant models faithful to botanical structure and development. ACM Computer Graphics SIGGRAPH Proc., Vol. 22(4), pp. 151-158

[ROTH82] Roth S.D (1982) Ray Casting for Modeling Solids. Computer Graphics and Image Processing, Vol. 18, pp. 109-144

[RUBI80] Rubin S.M, Whitted T. (1980) A 3-dimensional representation for fast rendering of complex scenes. ACM Computer Graphics SIGGRAPH Proc., Vol. 14(3), pp. 110-116

[SMIT84] Smith A.R (1984) Plants, fractals and formal languages. ACM Computer Graphics SIGGRAPH Proc., Vol. 18(3), pp. 1-10