Efficient ray tracing of complex natural scenes*

Christoph Traxler **Michael Gervautz**

Institute of Computer Graphics
Technical University of Vienna
Karlsplatz 13/186-2
1040 WIEN
AUSTRIA
email: traxler@cg.tuwien.ac.at gervautz@cg.tuwien.ac.at
url: http://www.cg.tuwien.ac.at/research/csg-graphs/index.xhtml

Abstract

In this paper we present a method for the consistent modelling and efficient ray tracing of complex natural scenes. Both plants and terrains are modelled and represented in the same way to allow mutual influences of their appearance and interdependencies of their geometry. Plants are generated together with a fractal terrain, so that they directly grow on it. This allows an accurate calculation of reflections and the cast of shadows. The scenes are modeled with a special kind of PL-Systems and are represented by cyclic object-instancing graphs. This is a very compact representation for ray tracing, which avoids restrictions to the complexity of the scenes. To significantly increase the efficiency of ray tracing with this representation an adaptation of conventional optimization techniques to cyclic graphs is necessary. In this paper we introduce methods for the calculation of a bounding box hierarchy and the use of a regular 3d-grid for cyclic graphs.

Key Words: natural phenomena, PL-systems, cyclic object-instancing graphs, ray tracing

1 Introduction

Realistic rendering of natural scenes is still a challenging task. Such scenes consist of different classes of geometrical shapes and usually each class demands a particular modelling and visualization technique. The necessary combination of these techniques results in problems which affect the quality of the resulting image. A common method is to render single objects of a scene with the appropriate method and compose the images with an α-channel. Reeves used particle systems and probabilistic algorithms to render complex natural scenes with an α-channel related technology called frame-buffer [REEV85]. Complex reflections and the casting of shadows, like those of trees onto terrain or themselves, can only be accomplished approximatively in this way. For the purpose of a consistent rendering method, where all objects are visualized in a uniform way and can mutually influence their appearance, it is necessary to find a general modelling technique and an efficient representation for all objects. The common boundary representation is not suited, because of the very huge data sets needed for natural scenes.

Among all the different techniques for modelling natural phenomena we found parametric L-Systems (PL-Systems) [PRUS90] to be the most powerful and general one. They are an important

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

(project number: P09818)

extension of the classical L-Systems, which were introduced by the biologist Lindenmayr [LIND68] to simulate the development of the topology of plants. Prusinkiewicz incorporated geometric aspects by attaching parameters to the symbols of these parallel rewriting systems, making it possible that the geometry of the plants also evolves out of the process. PL-Systems are mainly used to simulate the growth of botanical organisms and are fully described in [PRUS90]. A recent extension are environmentally sensitive PL-Systems [PRUS94], where branches are pruned against the boundary of a predefined volume. The generality of PL-Systems has not been explored so far. They have scarcely been used to generate other objects than plants, though they are capable to generate all objects with a more or less repetitive structure. Smith showed that even conventional L-Systems (without parametric control) are suitable for the modelling of fractal terrains and linear fractals [SMIT84]. In our previous work [GERV96] we described how to translate the midpoint displacement method for the generation of terrains into the notation of PL-systems. The benefits of stochastic L-Systems, which are important to create different individuals of a plant species, can also be achieved with PL-Systems. The combination of these systems makes it possible that plants grow on a fractal terrain. This allows a unique and consistent decsription of very complex natural scenes.

Another major advantage is the unique representation of PL-Systems, which allows a consistent rendering of all objects in the scene. PL-Systems can be translated into cyclic object-instancing graphs, which are a memory-efficient object representation for ray tracing and other visualization techniques. Thus the complexity of the scene is less restricted, like in the usual approach, where the whole scene is built up explicitly in memory. A similar idea was introduced by Kajiya [KAJI83] for ray tracing fractal terrains. The triangles of the mesh are only subdivided if a ray hits their bounding volumes, which are prisms. The height of each prism is estimated from stochastic properties of the midpoint displacement algorithm. In [BOUV85] this method was improved by using bounding elipsoids instead of prisms. In [HART91] an object-instancing graph was used for the ray tracing of linear fractals defined by Iterated Function Systems (IFS). All these techniques are restricted to a specific class of objects, namely stochastic and linear fractals. With PL-Systems we can generate objects of these classes as well as plants and other objects with a repetitive structure. Furthermore we have a unique modelling tool that allows a mutual influence between objects of different classes within the same scene.

In our previous work [GERV96] we introduced CSG-PL-Systems, an adaption to the CSG-concept, and their representation as cyclic CSG-graphs. In the next chapter we summarize this idea in an abstract way. The CSG paradigm has been droped, because we only use the union-operator. The concepts of object-instancing and the procedural representation of primitives has been preserved. Thus CSG-expressions are reduced to binary expressions and CSG-graphs have turned into cyclic object-instancing graphs. In chapter 3 we describe how to use a hierarchy of bounding boxes for these graphs. Chapter 4 shows how a 3d-grid yields an additional increase of efficiency. Finally we present in chapter 5 how PL-Systems and their representation as cyclic graphs are used for complex landscape modelling and rendering.

2. Ray Tracing with cyclic object-instancing graphs

To obtain a cyclic graph we have to represent each component of a PL-System by an appropriate node. In accordance to the CSG-PL-Systems [GERV96], we use PL-Systems that generate binary expressions as formal languages to describe a scene. Primitive objects or subexpressions enclosed in brackets are combined by the binary operator ´+´. This operator is equivalent with the union-operator of CSG. The other CSG-operators are not very meaningful within a feedback process and have been omitted at all. In addition to that we incorporate transformations as unary operators, which consist of a set of affine mappings. Global parameters are modified by a calculation component, that consists of assignment statements and can be conceived as unary operator as well. Within this kind of PL-System a distinction between grammar symbols and terminating symbols is necessary. Productions can only be applied to grammar symbols. In addition to a couple of so called *generating productions*, which specify the feedback system, *terminating production* must exist, that replaces all remaining grammar symbols by a string of terminal symbols to form a valid binary expression at the end of the derivation. The selection of a certain production for a grammar symbol during the derivation is done by a specific selection statement. Thus the components of these PL-Systems can be summarized to binary combination operators, primitive objects, transformations, calculations and grammar symbols with their productions. There are no restrictions for these kind of systems in contrast to the classical PL-Systems for turtle interpretation [PRUS90].

To translate a PL-System into a cyclic graph all components are represented by specific nodes. Grammar symbols are represented by *Selection-nodes* (*S-nodes*), which join all productions for the grammar symbol and are associated with their selection statement. Each right hand side of each production can be translated into a binary tree and attached as successor to the corresonding S-node. To form a cyclic graph, each occurence of a grammar symbol in the right hand side is finally replaced by an edge to the corresponding S-node. Combination operators are represented by +-nodes, which have two successors. Transformations and calculations are represented by *Transformation-nodes* (*T-nodes*) and *Calculation-nodes* (*C-nodes*) respectively, which have only one successor.

To visualize the object with ray tracing the cyclic graph has to be traversed by the rays. A selection node evaluates its selection statement and passes the ray to the appropriate successor. T-nodes map the rays from one coordinate frame into another. It is much more efficient to map the rays than primitive objects. Thus the rays are mapped from the world coordinate system into the object coordinate system by several intermediate steps during the recursive traversal. If the ray reaches a C-node the assignment expressions are evaluated to modify the parameters, which affect flow control and transformation arguments. A +-node simply passes the ray to both of its successors. When the ray reaches a primitive object the intersection calculation is done. To map the normal vector from an intersection with a primitive, T-nodes buildt up a transformation chain during the traversal which is linked to the intersection points. This avoids matrix multiplication and guarantees that only the normal vector of the foremost hit is transformed back to the world coordinate system. A more detailed description on ray tracing with cyclic graphs is given in our previous paper [GERV96].

We close this chapter with an example of a PL-System for binary expressions, that generates a simple branching structure. For simplicity only an excerpt of the definitions of transformations and calculations are printed in Figure 2.1. The object is created with a single grammar symbol and three productions. The selection statement depends on two counter values. One generating production builds up the trunk, whereas the other is responsible for the crown of the tree like object. Figure 2.2 shows the corresponding cyclic object-instancing graph and Figure 2.3 the rendered image.

```
Selection     Tree                    // Specification of the PL-system
{
     C_initialize Tree;               // Assignment to the C-node

     if (cntTrunk > 0)                     // Generate trunk
          Tree -> T_scaleCyl cylinder + C_Trunk T_shiftTrunk Tree;
     else if (cntTree > 0)                 // Generate rest of tree
          Tree -> T_scaleCyl cylinder + C_Tree
                             (T_attach_Branch1 Tree + T_attach_Branch2 Tree);
          else
               Tree -> T_scaleCyl cylinder;// Terminating production
}

Calculation C_initialize              // C-node for initialization
{
     cntTree       =    10;           // age (order) of the plant
     cntTrunk      =    2;            // length of trunk
     betaBr        =    60.0;         // branching angle
}
Calculation C_Trunk                   // C-node for trunk recursion
{
     --cntTrunk;                      // decrement counter for trunk generation
}
Calculation C_Tree               // C-node for tree recursion
{
     --cntTree;                       // decrement counter for crown generation
     betaBr -= 4.0;              // decrement branching angle
}
Transformation T_attach_Branch1       // T-node for 1st branche
{
     trans     0.0  0.0  1.99;
     roty betaBr;
     scale     0.8  0.8  0.8;
}
```

Figure 2.1: A PL-system for a simple tree-like branching structure.

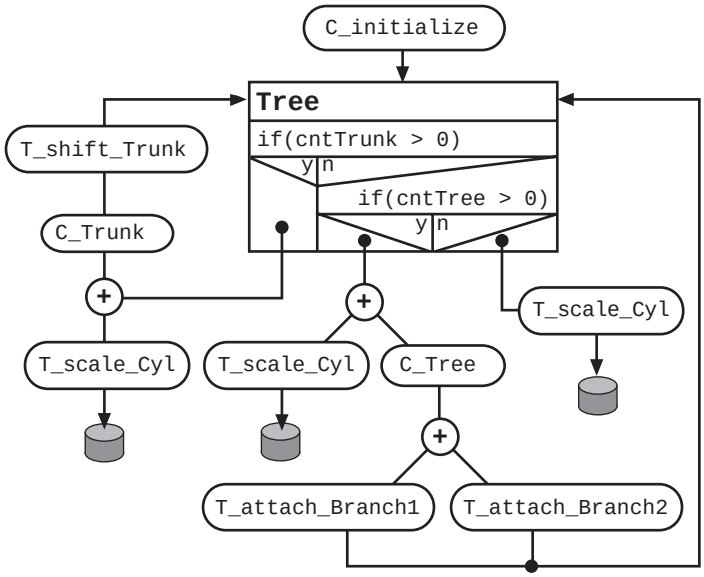


Figure 2.2: The cyclic graph for the tree like structure

Figure 2.3: The rendered image

3. Calculation of bounding boxes for cyclic graphs

A common method to accelerate ray tracing is to enclose the scene by a hierarchy of axis aligned bounding boxes. The parts of the scene, which can be hit by a ray are determined fastly and the traversal is terminated as soon as the ray misses a bounding box. The bounding box hierarchy can easily be calculated for the binary tree representation in a preprocessing step but is more difficult for cyclic graphs. For a binary tree the hierarchy is obtained in a bottom up manner. A bounding box is created for each leaf node, which refer to properly transformed primitive objects. The combining nodes combine the bounding boxes of their left and right successors and pass it to their predecessors. If a cyclic graph is unfolded into a binary tree it is obvious that almost each node of the graph corresponds to more than one node of the tree. Instances of these nodes represent different parts of the scene and thus get different bounding boxes. Maintaining a list of bounding boxes needs a vast amount of memory, comparable to the representation as huge binary tree and destroys the advantage of a memory-efficient data structure. Therefore we only calculate the bounding boxes for the root level instance of each node. As explained later this is sufficient and not a severe problem, though they do not enclose the different parts of the scene very tightly in the most cases.

If all the transformations used in a cyclic graph are contractive then all bounding boxes are perfectly tight and fit for each instance of the corresponding node. In this case they form an IFS and the contraction mapping principle holds [BARN88]. Thus the combination of the bounding boxes in an arbitrary recursion level perfectly fits into the hyper bounding box of a lower recursion level. This is because the objects are assembled by smaller copies of themselves. For that case it is not necessary to calculate the bounding boxes out of the geometry of the primitive objects and the transformations applied to them. As explained by Hart and DeFanti [HART91] we can use the Collage Theorem [BARN88] to calculate a tight bounding box for each node of the graph. Unfortunately a restriction to contractive transformations destroys the generality of PL-Systems. Their main purpose is to modell objects that evolve out of a growth process, which is contradictory to contraction mappings. The most natural looking plant modells are obtained by simulating a plant's development, which demands growing structures [PRUS90, chapter 8.2].

Calculating tight bounding boxes for cyclic graphs

To obtain correct bounding boxes for a cyclic graph that is not equivalent to an IFS, the simple passing of bounding boxes from a successor to a predecessor node must be avoided. The bounding box for the root level instance of each node is directly calculated out of the geometry of the primitive objects, whereby all transformations have to be taken into account, that are encountered on the cyclic paths to the primitives. An instance of a T-node maps an object from a coordinate frame C_k into another coordinate frame C_{k+1}. C_0 is the object coordinate system of a certain primitive object and C_n indicates the world coordinate system if *n* transformations are necessary to map the primitive into it.

A cyclic object instancing graph defines a plenty of cyclic paths from the root to a primitive object. Along these paths there are many instances of one or more T-nodes. Let us first ignore the binary operators and extract such a path as transformation chain $C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow ... \rightarrow C_{n-1} \rightarrow C_n$. We now have to calculate a bounding box for each occurring chain directly out of the geometry of the primitive object defined in C_0. The geometry of an object is defined by a set of points that approximate its convex hull sufficiently. An axis aligned bounding box is simply derived by finding the minimal and maximal coordinates of the point set. Let T_i denote the transformation of that T-node instance, which defines the mapping from C_{i-1} into C_i. To calculate a bounding volume for C_n we just have to map the point set from C_0 into C_n by applying the transformations $T_n \times T_{n-1} \times .. \times T_1$.

The graph is traversed and transformations are pushed onto a stack until a primitive object is reached and a path is completed. Each instance of an +-node joins the bounding boxes originating from its left and its right path. In this way all nodes of the graph obtain a root level bounding box. The bounding box of the S-node encloses the whole object and the other nodes the corresponding parts of the object. Figure.3.1 shows the root level bounding boxes for the simple branching sructure in a 2d-projection. For very large scenes it is more efficient to estimate the bounding boxes of some subgraphs from stochastic properties. These are subgraphs, which generate a huge number of relatively tiny objects, like pins of a conifer tree for example. Furthermore the bounding box for

each subgraph should be calculated only once and not any time, when it is instanced by another graph. A subgraph generating a twig of a conifer tree puts a lot of pins around the twig, but a bounding box enclosing all possible sizes of pins is only calculated once for the corresponding subgraph.

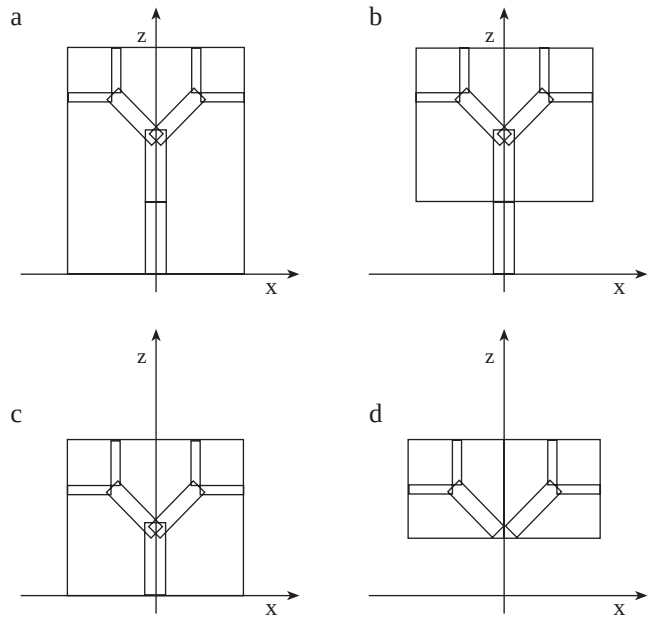


Figure 3.1: Some root level bounding boxes for the cyclic graph, that generates the branching structure: a) bounding box of the S-node and the +-node in the cycle that builds up the trunk, b) bounding box of the T-node *T_shift_Trunk*, c) bounding box of the first +-node in the cycle that builds up the crown, d) bounding boxes of the T-nodes *T_attach_Branch1* and *T_attach_Branch2.*

Analysing the optimization effect

The algorithm described in the last section calculates tight bounding boxes for the root level instance of each node of a cyclic graph. For lower level instances they are too large in the most cases. Fortunately this problem is diminished by an essential aspect of cyclic graphs, - the iterative transformations. The rays are transformed from one coordinate frame into another. Thus the geometric relation of a ray to one and the same bounding box changes with every mapping. Since this relation is symmetric, we can also say, that constant rays are tested against iterativeley transformed bounding boxes. If a ray hits the same bounding box in two succeeding coordinate frames C_i and C_{i+1}, than we can view this as hits of two different bounding boxes B_i and B_{i+1}, defined relatively to C_i and C_{i+1}. This means that the ray hits the intersection of both bounding boxes $B_i \cap B_{i+1}$. Thus the hit of a bbounding box at a certain recursion level *k* can be seen as the hit of the intersection of all bounding boxes encountered on the path from the root level to that recursion level, i.e. $B_0 \cap B_1 \cap ... \cap B_{k-1} \cap B_k$. This intersection usually gets smaller with increasing recursion depth, so that a ray has a new chance to miss one and the same bounding box after each transformation into a new coordinate frame. In these cases the root level bounding boxes are usefull for deeper levels of recursion as well. In the worst case the intersection scarcely shrinks, so that the optimization factor decreases. The root level bounding box appears almost the same in each coordinate frame and once the ray hits it, the traversal proceeds until the deepest level of recursion is reached. This happens for example if objects are scattered randomly among the scene. Only translations are used in such a graph and the intersection of the bounding boxes does not get smaller.

Picture 1 shows the bounding box hierarchy of the simple branching structure. When the current ray misses a bounding box the background is colored according to the recursion depth. The brighter the color the deeper is the level of recursion at which the corresponding ray misses the bounding box. The difference between red and blue has just the purpose to enhance the contrast. The picture shows that there are rays which miss a bounding box at deeper recursion levels. Thus root level bounding boxes are meaningful for deeper levels of recursion as well.

4. Using a regular grid as additional optimization

Preprocessing

An additional optimization technique that can be applied to cyclic graphs is a regular 3d-grid, as described by Fujimoto et. al. [FUJI86]. The grid is filled in a top down manner by traversing the graph with a copy of the root level bounding box of the S-node. T-nodes map the bounding box in a local coordinate frame and +-nodes pass it to their left and right successors. The traversal is terminated at the S-node if the bounding box is small enough with respect to the size of a voxel or if a certain recursion depth is reached. It is also possible that the traversal stops at a primitive object. In all cases the bounding box determines which voxels have to be occupied. The voxels are associated with a *state* of the graph, which consists of a reference to a node of the graph, a transformation matrix and a set of parameter values. A state represents a certain level of recursion, which corresponds to a subset of the scene. The matrix is determined by the accumulation of all transformations that have been applied to the bounding box on its cyclic way to this level of recursion. The set of parameter values corresponds to the modifications of all instances of C-nodes that occure on the path down to this recursion level.

If there are voxels that have been already filled during the traversal of other paths, the primitive is inserted as second object indicating that this ambiguity has to be resolved by the proper instance of an +-node. This instance is the junction of the paths on which both primitives can be reached. The corresponding +-node now resolves the ambiguity by deleting both entries and inserts the appropriate state, that refers to itself. If a primitive object is inserted afterwards as new second object, that was reached during further traversal of a path originating from a higher level instance of an +-node, then this ambiguity is again resolved by this node in the same way. Since only binary combinations are admissible there are never more than two entries within one voxel. Figure 4.1 illustrates in 2d-space how the grid is used for the simple branching structure.

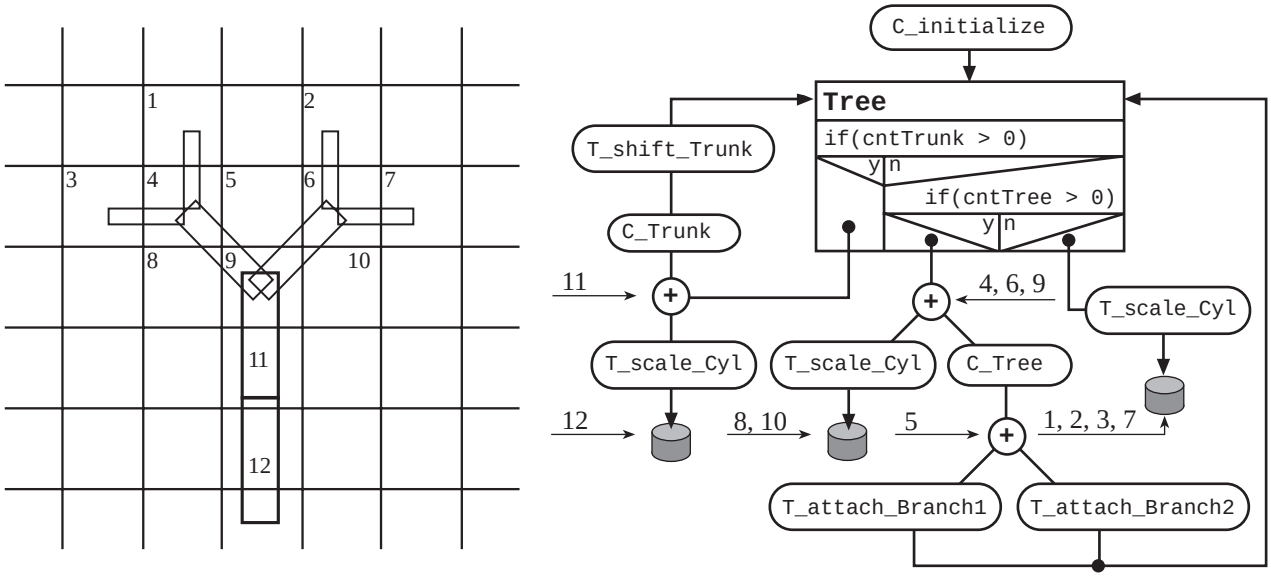


Figure 4.1: Voxels that are occupied by the branching structure are numbered to indicate to which node of the cyclic graph they point to.

Ray traversal

The voxels that are passed by the current ray are quickly determined with a 3d-DDA [FUJI86]. The grid subdivides the scene into regions of spatial coherence and thereby reduces the number of ray-objects intersection calculations. For a cyclic graph coherent regions are defined by a state of the graph, which corresponds to a recursion level. All the cyclic traversals a ray has to perform to reach this level are now avoided by setting the graph to the state, associated with the foremost voxel that was intersected by the ray. This is done by transforming the ray into the coordinate frame defined by the matrix of the state, setting the parameters to the values stored within the state and passing the ray to the node the state referes to. In the simple case the ray is transformed into the local coordinate frame of the primitive object and directly intersected with it. In all other cases the ray is passed to

an internal node and starts to traverse the few remaining levels of recursion until it reaches a primitive.

Analyzation

Using a grid is a tradeoff between the compact representation of the scene by a cyclic graph and an explicit representation as large binary tree. Thus the method can be adapted to the memory resources of the user. If enough memory is available the resolution of the grid can be increased to a number, where the most non empty voxels contain primitive objects. In this case the main advantage of the cyclic graph is to efficiently fill up the grid in the preprocessing step and avoid to build up the whole scene for that purpose. If the memory resources are low the main advantages of the graph is the compact representation for ray tracing. But even a coarse resolution of the grid yields a speedup for the most scenes. This speedup is especially significant if the primitive objects are scattered over the scene. In all cases where the root level bounding box does not change much along the hierarchy of coordinate frames, so that their intersection is large, the grid is a meaningfull additional optimization. The bounding box of a T-node, that is responsible for two objects, which are far away from each other, has to enclose both of them, whereas they belong to different voxels of the grid. Another increase of efficiency can be observed for PL-Systems, where expensive calculations and a lot of transformations are necessary, since the results of calculations and the accumulated transformations are preprocessed and stored within the voxels.

In those cases where the primitives are combined to form a compact object, like linear fractals, the grid can be omitted, because the root level bounding boxes are sufficient to discard non intersecting rays. This applies to scenes that can be created with IFS as well. In this case the cyclic graphs works very similar to the cyclic object-instancing di-graph presented by Hart and DeFanti [HART91]. For complex natural scenes, where many plants are placed onto a terrain and expensive calculations are necessary, the grid allows to scale ray tracing time with respect to memory resources.

5. Modelling and realistic rendering of large landscapes

The optimization techniques presented in the last two chapters allow the efficient rendering of complex landscapes with ray tracing. The number of primitives for an explicit representation of such scenes is huge, but the compact representation by cyclic object-instancing graphs relieves the user from memory considerations. In this chapter we describe how these benefits can be facilitated to modell and render large landscapes. An artificial landscape basically consists of a fractal terrain and plants growing on it. So we combine a PL-System for terrains with those that generate plants. In this way the terrain influences the appearance of each plant, and allows to fulfill natural constraints. The decision whether a plant is generated on a certain location of the terrain or not can depend on a variety of geometric aspects. In our scenes the height and the inclination of the location are the most important factors. Plants do not grow in water or above the timber-line nor if the place is too steep. An additional factor that can be used is the direction of the normal vector at the location, to make the growth on south slopes more likely than on other places. These constraints are checked by some simple conditions within the PL-System indicating whether a plant should be generated at a certain subdivision step of the terrain or not.

Terrain generation

Our initial approach was to generate the terrain by midpoint displacement as described by Carpenter [CARP80]. In our previous paper we described the implementation of Carpenter's method as (CSG-)PL-System in detail [GERV96]. Unfortunately the shape of the terrain is unpretictable and only some properties, like its roughness can be controlled. To avoid this restriction and allow some design of the landscape, we use a height field. It can originate from arbitrary data, like a stereo satellite scan for example. In our scenes we used scattered data interpolation to design the landscape by a couple of points, which indicate the location of valleys and peaks. We blended the resulting smooth surface with a fractal terrain of high roughness by α-channeling to add a rock-like microstructure to it. This method has also the advantage, that we can use the spectral synthesis method to generate a fractal terrain as described by Saupe [SAUP88] which does not suffer from artefacts like midpoint displacement. The height field is directly accessed by C-nodes to set the height of each triangle vertex during the subdivision process. That means that the subdivison schema of midpoint displacement is preserved but the calculation of the displacement values has been replaced by a simple access to the height field.

Placing plants

At a certain subdivision level the PL-System checks for each triangle if one or more plants should be placed onto it befor it is further subdivided. This level controlls the relative distance between plants and their frequency, because the size of triangles decreases and their number increases with each subdivision step. A random translation within the triangle avoids that the plants are arranged like in a plantation. More irregularities are achieved by a probability for plant growth, so that for some triangles no plants are created although all other conditions are valid. It is important that all plants look different. Thus most parameters are initalized with random values, like branching angles, scaling factors or the age of the plant.

An example

The productions of a PL-System, that generates an atoll with palms is printed in Figure 5.1. The PL-System is specified in C-like style. To keep the example short all definitions of C- and T-nodes have been omitted. The non-recursive production for *Terrain* joins two triangles to a rectangle and scales it with the T-node *T_mnt*. The triangles are subdivided by the productions for *Triangle1* and *Triangle2* respectively. The first production in the selection statement of *Triangle1* is a kind of loop, where a palm is generated. The level where plants can be generated is determined by the parameter *CONST_cpalm*, which is compared with the counter *cntTri*. The probability for palm growth should be 80%. This is achieved by comparing a random value from the interval [0.1] with the parameter *CONST_cpalm*, which is initialized with 0,8. Finally palms should not grow too close to the shore but in the middle of each island. Therefore the height *h1* of the triangle under consideration must be slightly above sea level. The inclination is not checked, because the entire terrain is rather flat. The palms are generated by another PL-System, which is not printed in the example. Its parameters are initalized with random values by the C-node *C_initPalm*, which also calcualtes the position of the palm on the triangle. The T-node *T_palm2beach* moves the palm to the precalculated location on the terrain and compensates the scaling along the x- and y-axis that accumulates during preceding subdivisions. The triangle is further subdivided by the second production as long as the counter is greater than zero. The height field is accessed by the C-nodes *C_T1_1* to *C_T1_4*. Finally the feedback process terminates with a properly transformed triangle.

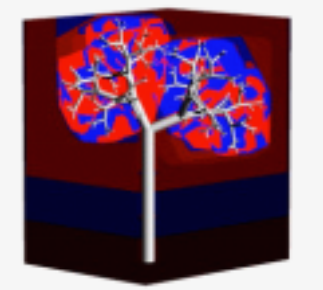
```
Selection Terrain {
     Terrain -> T_mnt (Triangle1 + C_init_Tri2 T_moveTri2 Triangle2);
}
Selection Triangle1 {
     if((cntTri==level) && (rand(0)<=probability) && (h1>=sea_level + 0.24))
                                                  // put a palm onto the triangle
          Triangle1 -> C_initPalm T_palm2beach Palm + Triangle1;
     else if (cntTri>0)                    // subdivide triangle
          Triangle1 -> C_T1_1 T_move1_1 Triangle1 + C_T1_2 T_move1_2 Triangle1 +
                       C_T1_3 T_move1_3 Triangle1 + C_T1_4 T_move1_4 Triangle2;
          else                             // terminating production
               Triangle1 -> C_TRI1 T_triangle triangle;
}
// Selection Triangle2 is specified analogously
```

Figure 5.1: The productions that generates a terrain with plants

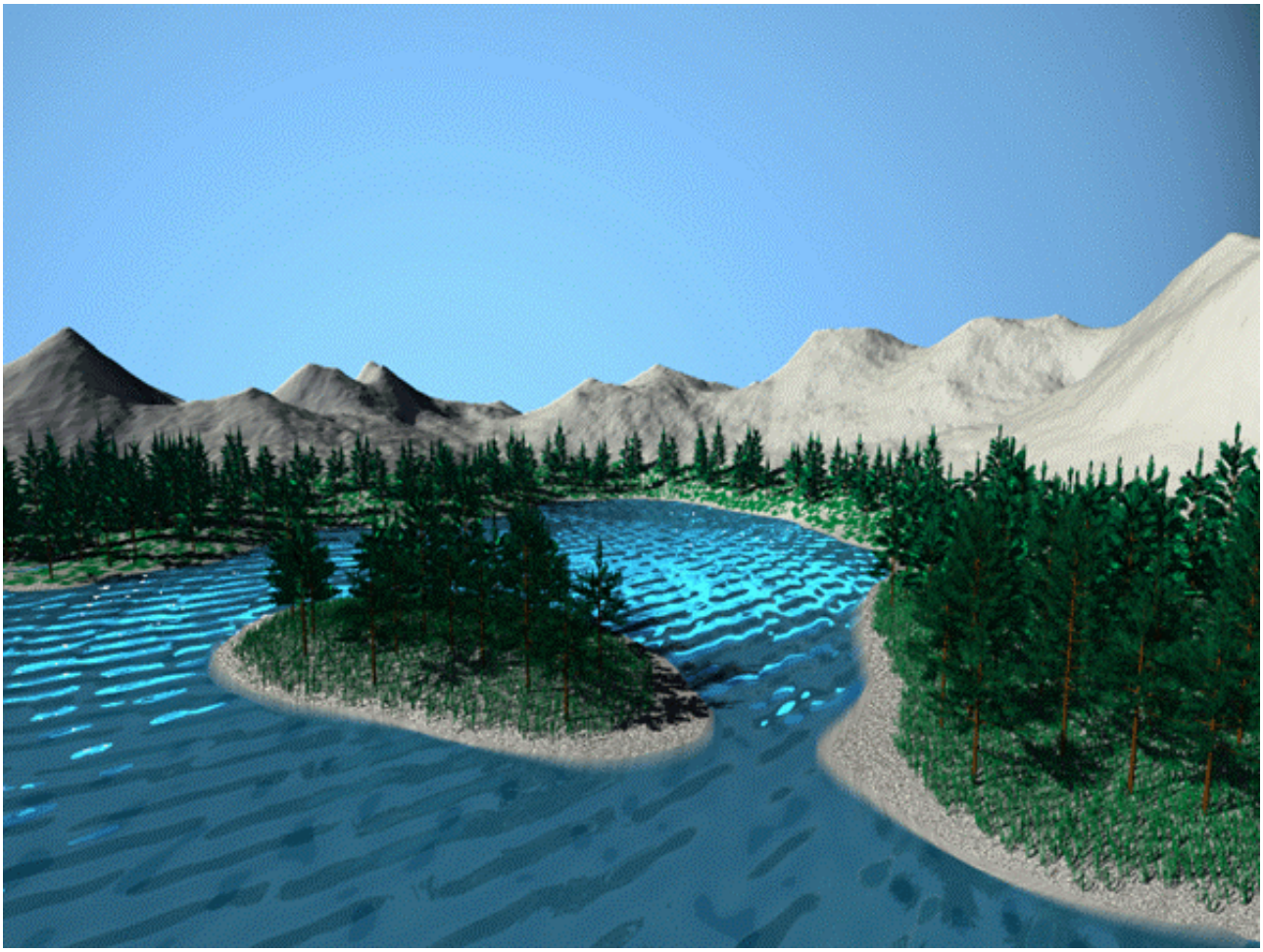
Results

Picture 3 shows the resulting image of the atoll. Picture 2 shows a Canadian National Park scene. Conifer trees are spread among the designed landscape. A more accurate model demands expensive calculations in the preprocessing step, a collision detection mechanism and last but not least an interactive modelling tool. For both scenes we used a 16x16x8-grid. The average number of rays per second for the scene with the atoll is 15,5 on a SGI-Indy with a R4400 processor and is 2,3 times faster than rendering without grid. More than 1 Mio. primitives would be necessary to build up the scene explicitly. No statistics are available yet for the National Park scene except the number of primitive needed for an explicit representation, which is greater than 100 Mio. Picture 4 illustrates that our method is also suitable for the modelling and ray tracing of linear fractals. The picture is called “Fractals in the Hausdorff Room” and shows a couple of objects, that are usually defined by IFS. Menger's Sponge, von Koch's Dodecahedron, and Sierpinski's Tetrahedron stand on

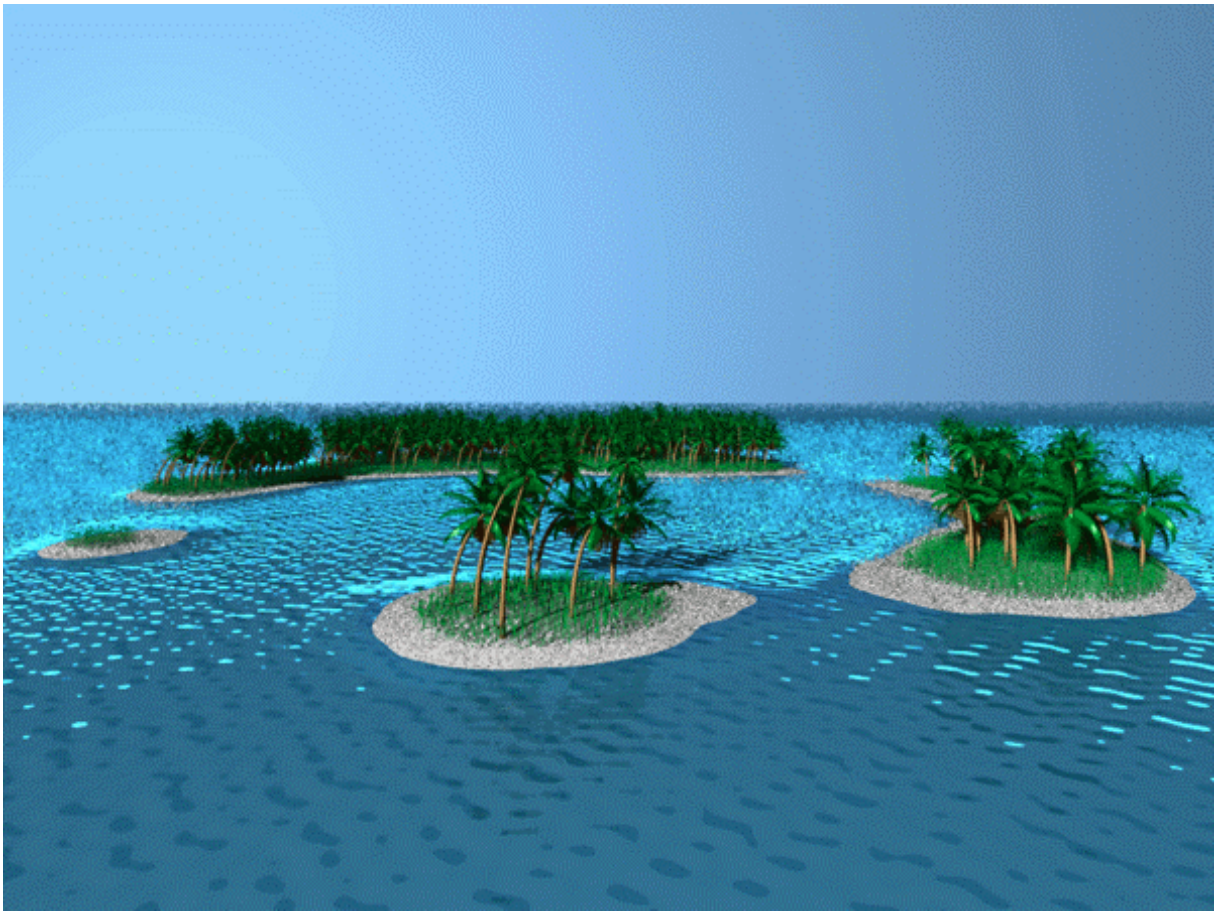
the table. The floor is paved with Menger's Carpet, which is also used for the relief on the walls and the struts connecting the table legs. Outside in the desert we placed two Sierpinsky pyramids. The vase in the right hand corner contains a PL-System version of Barnsley's Fern. This scene was rendered without grid and the average number of rays are 212,4. About 1 Mio. primitives would be necessary for an explicit representation. Antialiasing is achieved by adaptive oversampling for all pictures.



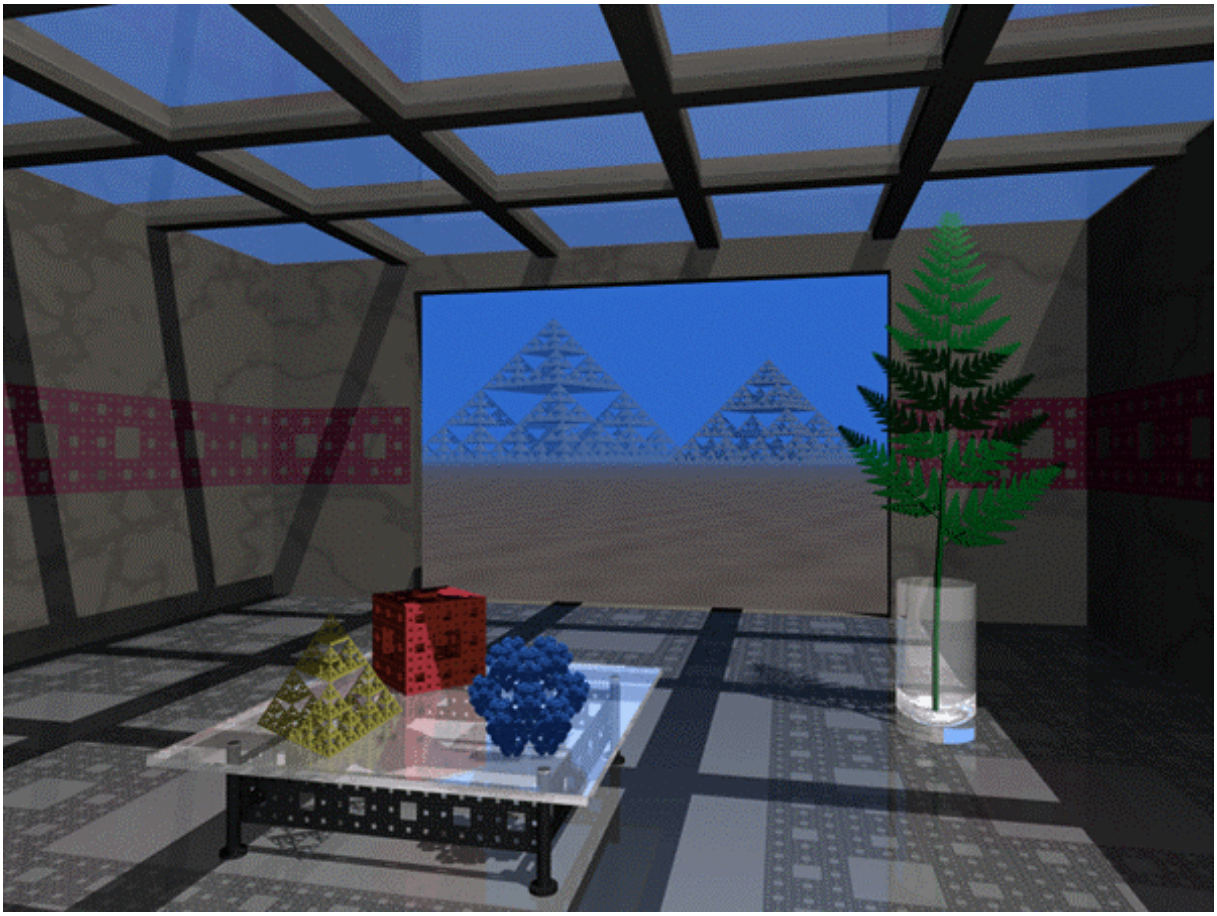
Picture 1



Picture 2



Picture 3



Picture 4

6. Conclusion

PL-Systems and their representation as cyclic object-instancing graphs are a very powerful technique for modelling and realistically rendering very complex scenes, like landscapes. They are capable of generate other objects with a repetitive structure as well, like linear fractals or buildings. The major aspect of our method is the consistent representation of all parts of the scene, which allows mutual influences of their visual appearance and interdependencies of their geometry. Furthermore this representation is very compact, so that the complexity of the scene and the approximation accuracy of objects are not restricted. Since conventional optimization techniques, namely the bounding box hierarchy and the 3d-grid have been adopted successfully to cyclic graphs, this data structure is also efficient for ray tracing.

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 for reviewing this paper and Fritz Heigl for helping converting this paper to PDF.

References

[BARN88] Barnsley M.F, Fractals Everywhere, Academic Press, Inc., 1988

[BOUV85] Bouville C, Bounding ellipsoids for ray-fractal intersection, ACM Computer Graphics SIGGRAPH Proc., Vol. 19, No. 3, pp. 45, 1985

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

[FUJI86] Fujimoto A, Tanaka T, Iwata K, ARTS: Accelerated ray-tracing system, IEEE Computer Graphics & Applications, Vol. 6, No. 4, pp. 16, 1986

[GERV96] Gervautz M., Traxler C., Representation and realistic rendering of natural phenomena with cyclic CSG graphs, The Visual Computer, Springer Verlag, Vol. 12, No. 2, pp. 62, 1996

[HART91] Hart J.C, DeFanti T.A, Efficient antialiased rendering of 3d linear fractals, ACM Computer Graphics SIGGRAPH Proc., Vol. 25, No. 4, pp. 91, 1991

[KAJI83] Kajiya J.T, New techniques for ray tracing procedurally defined objects, ACM Transaction on Graphics, Vol. 2, No. 3, pp. 161, 1983

[LIND68] Lindenmayr A., Mathematical models for cellular interaction in development, Parts I and II, Journal of Theoretical Biology, Vol. 18, pp. 280, 1968

[PRUS90] Prusinkiewicz P., Lindenmayer A., The algorithmic beauty of plants, Springer Verlag, New York, 1990

[PRUS94] Prusinkiewicz P., James M., Méch R., Synthetic Topiary, ACM Computer Graphics SIGGRAPH Proc. 1994, pp. 351, 1994

[REEV85] Reeves W.T., Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems, ACM Computer Graphics SIGGRAPH Proc. 1985, Vol. 19, No. 3, pp. 313, 1985

[SAUP88] Saupe D., Algorithms for random fractals, In Peitgen H.O. and Saupe D. ed.: The Science of Fractal Images, Springer Verlag, pp. 71, 1988

[SMIT84] Smith A.R, Plants, fractals and formal languages, ACM Computer Graphics SIGGRAPH Proc., Vol. 18, No. 3, pp. 1, 1984

[WERN94] Wernecke J., The Inventor Mentor, Addison Wesley Publishing Company, 1994