Modeling and Rendering of Outdoor Scenes for Distributed Virtual Environments

Dieter Schmalstieg and Michael Gervautz
Vienna University of Technology
dieter|gervautz@cg.tuwien.ac.at

***Abstract**. We present an approach for modeling and real-time rendering of outdoor scenes, for use in virtual reality applications such as flight simulators and multi-user virtual environments. The models are based on a procedural representation using directed cyclic graphs. It allows to represent extremely complex scenes with little memory and modeling effort. Very large scale virtual environments are supported by a bandwidth-preserving networking approach that makes use of the compact representation and on-the-fly database amplification.*

1. Introduction

Many real-time graphics and virtual reality (VR) applications aim to immerse the user in an outdoor scenario composed to a large extent of natural phenomena a landscape, plants, trees, mountains and so on. Some of the most successful virtual reality applications are based on outdoor settings, among them flight simulators, tactical training systems, video games, and urban reconstruction projects. Outdoor environments are typically flat and sparsely occluded, so the area that can be observed by the user is rather large. Another desired characteristic is that the user should be able to move freely over a large area without reaching an artificial border too fast. The environment should contain plenty of detail (e. g. leaves on trees) even at close inspection to obtain a realistic impression. The successful simulation of a large virtual environment represented with a high degree of fidelity requires construction, run-time management, rendering, and network transmission of very large geometric databases.

Traditionally, research has focused on the problem of real-time rendering of very large geometric databases. Powerful rendering hardware for polygonal models has been developed for that purpose [Akel93]. In combination with texture mapping [Fole90], level of detail (LOD) rendering [Funk93], and scene culling [Funk95], even large scene databases can be rendered in real time.

Yet despite the power of state of the art graphics technology, the craving for even more realism often defeats the purpose, because the large scene databases are difficult to handle. In particular, we see three areas where improvement is needed:

1. **Modeling**: The construction of a large number of detailed models is extremely labor-intensive. While models of artificial structures such as machines or buildings are relatively easily obtained from CAD sources, this is not true for plants and other natural phenomena. The use of texture maps (e. g. photographs) reduces modeling costs, but this shortcut becomes painfully obvious when inspecting models at close-up. Instancing (i. e. using the same model multiple times) is also easily detected and destroys the user's believe in the virtual world.
2. **Storage requirements**: A very large geometric database requires lots of storage. Today's typical workstations have enough memory to store scene databases that by far exceed the capacity of the image generator. However, if only a small portion of an extensive virtual environment is visible at any time, and the application allows the user is to cover large distances, the actual database can easily exceed memory capacity. Loading data from disk in real-time has its own set of problems [Funk95], so a compact representation that allows to hold all or a large portion of the scene database is highly preferred.
3. **Networking**: If the geometry database is to be distributed over a network, a compact representation is even more essential. The rapid growth of Internet-based VR applications that suffer from notoriously low and unpredictable bandwidth drives the desire for compact geometric representations that can be transmitted in shorter time [Deer95].

A solution to these problems lies in the use of procedural modeling and fractal geometry. Procedural models allows the concise description of objects whose structure can be formulated as a program, and is especially suitable for plants and trees. A very powerful class for that purpose are Parametric Lindenmayer systems [Prus90a]. The algorithmic description is usually very compact, and can easily be extended to yield a large number of different objects instead of a single one, making the instancing of large populations effective. Creating a large scene from a very small data set is called database amplification in [Smit84].

Numerous methods for modeling and rendering of plants have been presented in the past, e. g. [Aono84, Gerv95, Max96, Neyr96, Prus90a, Prus90b, Reev85, Smit84, Trax97, Webe95], but most are aimed at photo realism and do not produce images in real-time. Most methods create an explicit geometric model from the procedural model as a preprocessing step to rendering. Such a geometric model can be used for virtual reality applications, but does no longer address the requirements regarding storage and networking. Some methods produce images without the use of explicit geometric primitives [Reev85, Neyr96, Max96], but they cannot make use of polygonal rendering hardware. Special support for real-time applications with level of detail rendering is presented in [Webe95], but the approach is also not storage preserving.

2. Overview of our approach

Our approach is based on the work by Gervautz and Traxler [Gerv95], who used directed cyclic graphs for raytracing of natural phenomena without an intermediate representation. This approach is equivalent to Parametric Lindenmayer systems [Prus90a]. A VR rendering system that employs directed cyclic graphs for virtual reality applications uniquely addresses the problems mentioned in the introduction:

- **Direct rendering of procedural models**. Unlike other procedural modeling approaches for interactive rendering, our models can directly be rendered, thereby creating geometry on the fly. There is not need for an intermediate polygonal representation.
- **Unified rendering** of procedural and non-procedural models is possible.
- **Good memory utilization**: Direct rendering of the procedural model supplants the use of explicit detailed geometry, and yields vast savings in storage, in particular if large populations are instanced. Database amplification can further be enhanced through the use of statistical distributions and random numbers.
- **Network bandwidth savings**: The compact representation is also very suitable for network transmission.
- **High quality rendering**: The problem of artifacts when viewing textures at close-up is solved by providing actual geometric detail, but without the penalty of elevated memory requirements.

The remainder of this chapter discusses the details of our approach: Section 3 gives background on the rendering of directed cyclic graphs. Section 4 discusses the application of directed cyclic graphs in distributed systems. Section 5 pays attention to the issue of efficient rendering. The discussion is complemented by details about a sample implementation using Open Inventor (section 6), followed by examples and results (section 0).

3. Background: Rendering Directed Cyclic Graphs

In this section, we aim to give the reader an introduction to the formalism of PL-systems, and its equivalent, directed cyclic graphs, as developed by Gervautz and Traxler. We also review the implications of modeling and rendering directed cyclic graphs for interactive applications.

3.1 A brief introduction to PL-systems

PL-systems are commonly written as a grammar called a *rewriting system*, consisting of an alphabet of modules (a symbol plus a set of parameters), a set of productions for every module that specify how to derive valid expressions, and an axiom. Starting with the axiom, productions are concurrently applied to the modules of an expression (hence the term *parallel rewriting system*) to derive new expressions. Associated with each parameter is an arithmetic expression that is evaluated upon application of a production, the result of the evaluation controls the selection of the production (if there is more than one production for a particular module). Images are generated by interpreting an expression geometrically, usually with a construction tools called turtle. Figure 1 shows a simple PL-system and the resulting model.

Instead of deriving an explicit geometric model, we use a representation equivalent to rewriting systems based on graphs. This approach is enabled by a simple modification to conventional modeling: extending a *directed acyclic graph* (DAG) to a *directed cyclic graph* (DCG). DAGs are the standard approach for modeling geometry databases for interactive applications: A hierarchical structure (tree) allows efficient definition and manipulation of properties such as material for arbitrary parts of objects or scenes. For example, transformation nodes modify the object space transformation matrix for all objects traversed after the transformation, allowing the construction of articulated figures.

axiom: **Worm**(4)
productions for Worm(c):
if(c=0): **Worm** → **Cone**
if(c>0): **Worm** → **Sphere** **Translation**(1,0,0) **Worm**(c-1)

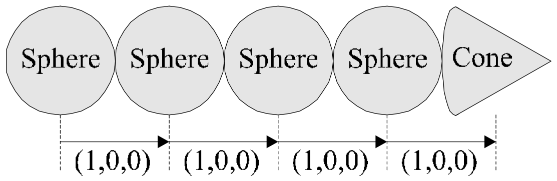

Figure 1: A very simple recursive model

Actions such as rendering are applied to such a data structure by graph traversal. Allowing multiple references to a subgraph enables instancing and turns a tree into a DAG. To represent recursive structures, we allow cyclic references in the graph structure, thus creating a DCG.

3.2 Translating a rewriting system into a DCG

An expression-based PL-system can easily be translated into an equivalent DCG using the process outlined in this section. We consider a form of rewriting system where symbols are divided into non-terminals and terminals. Only non-terminals can be substituted, and the productions for every variable require that there is at least one substitution that consists only of terminals. Only terminals have a geometric representation. To get an expression that consists only of terminals (and can hence be rendered), any remaining non-terminal is substituted according to the production that generates only terminals.

The right hand side of every production is interpreted as a subgraph. Concatenated modules are represented as children of a *group node*, that traverses all its children. For every non-terminal module of the alphabet, exactly one *selection node* is created. Upon traversal, the selection node traverses only one child as indicated by a parameter. The children of the selection node are the subgraphs constructed from the right hand sides of the productions for that particular module. Consequently, any non-terminal module in such a subgraph becomes a link to a selection node. Recursive productions (of the form A → … A …) thereby create cycles in the graph; indirect recursion is possible as well. The selection node for the axiom becomes the root.

The arithmetic expressions passed as parameters to the modules in the productions are translated into separate nodes, the *calculation nodes*. In these nodes, the old value of a parameter is saved and a new value for the parameter is computed from the given arithmetic expression, emulating the behavior of a call by value parameter in a recursive procedure. The initialization of parameters at the root of the graph is also done with calculation nodes. Calculation nodes evaluate the associated functions only when they are visited upon traversal, so their behavior can be characterized as lazy evaluation in terms of compiler technology. The example from Figure 1 is transformed into the graph in Figure 2.

3.3 Traversal of the DCG

An important step in the rendering of graph-based models is the graph traversal. The order of traversal is depth first and left to right (i. e. children of a group node are visited from left to right). For every node, the appropriate behavior is called; for example, a rendering traversal will render primitive nodes such as polygons or spheres, and for a group node simply traverse all its children.

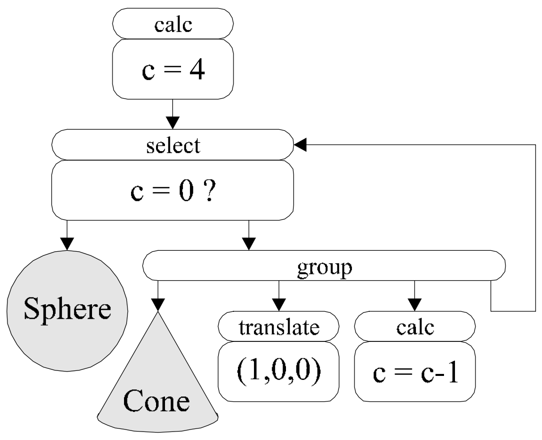

Figure 2: Simple recursive graph for the model shown in Figure 1

An important concept is the accumulation of state during the traversal, for example, transformations must be multiplied as they are encountered during the traversal. While the propagation of accumulated state is usually desired while traversing deeper into the graph, it should not affect other branches of the graph that are traversed later. Therefore, state is saved before performing depth traversal, and restored when the traversal returns from the subgraph.

Such a graph traversal works well for DAGs, but the cycles contained in DCGs would lead to infinite looping without special measures. Therefore, recursive models use a parameter-dependent selection node to branch into a terminating (i. e. cycle-free) subgraph after the desired number of recursions. The selected child is functionally dependent on one or more parameters, that are modified and evaluated during traversal. An obvious construction is to use one parameter as a counter of recursion depth, and terminate when it reaches a specific value.

3.4 Database amplification with parameterized models

Using model represented with DCGs, database amplification is very easily possible. Parameterized models allow the creation of a large and diverse population from a single model. A DCG can be though of as a genotype of a species, with the initial settings of the parameters responsible for the appearance of the phenotype. For example, a model of a fractal tree can be varied in the height of the tree, the number of branches, the color and so forth. Position in the scene is just another parameter affecting a translation node.

A population can be generated with very little effort in computation and memory consumption by creating the desired number of instance nodes that store initial values for the parameters of the model and reference the model. This can be achieved even more efficiently by an enumerator node that stores references and initialization data in arrays. For example, a forest can be created from a single model as in Figure 3.

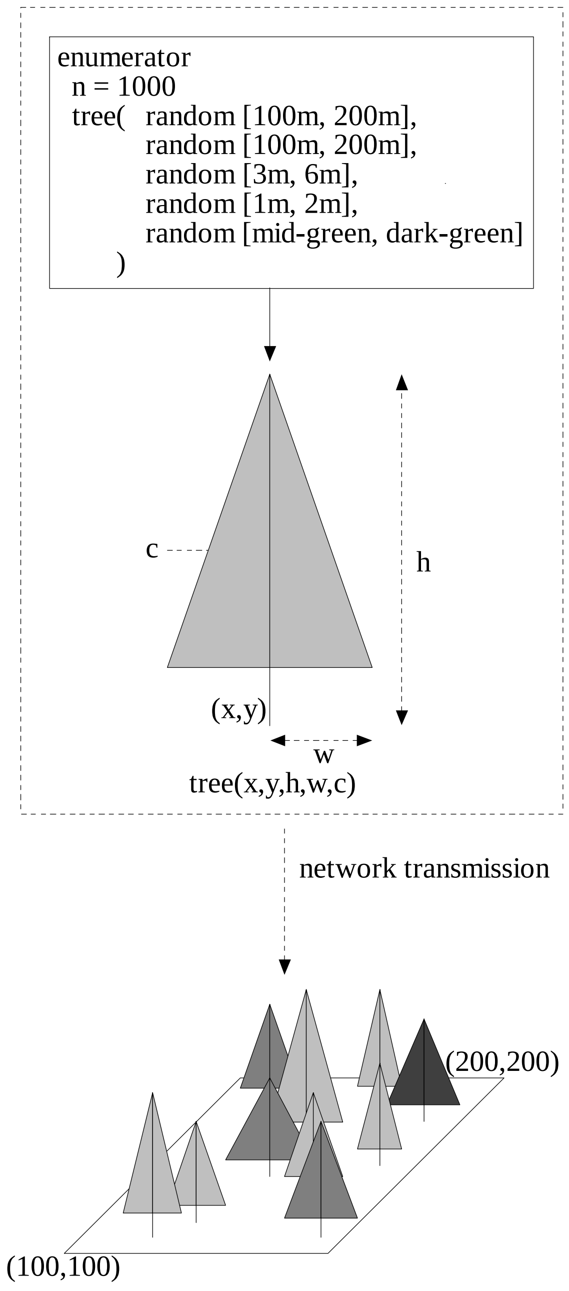

Figure 3: Creating a forest by varying a tree

Parameters can either be user-specified (for better control over the scene), or generated from random numbers. In that case it is only necessary to associate a unique seed value for the random number generator with each model, so that the random numbers used for a particular model can be re-created every time the model is traversed. Note that the random number generator must work cross-platform, so that models have the same appearance on every platform.

A combination of user-defined and random values is often desirable: For example, the user may wish to specify only the general appearance (height of a tree, peaks and valleys of a terrain model), and leave the details to a statistical distribution of random numbers (crease angle of individual branches, number of leaves on a twig, small variations in terrain height).

For distributed virtual environments, the demand-driven geometry protocol is adapted to work with DCG models. Since the same basic rendering algorithm is used, this is straight forward. As models are potentially instanced many times, the model's actual geometric description must only be transmitted when the first instance of a species is encountered, later instances can be specified by parameters only.

4. Exploiting directed cyclic graphs for distributed virtual environments

Database amplification as described in the last section is very useful for constructing and managing very large-scale distributed virtual environments by combining DCG models and *demand-driven geometry transmission* [Schm96]: A geometry database is maintained by a server, while users invoke individual clients to interact with the environment (Figure 4). The server stores the data for a very large virtual environment, composed of objects that are arranged spatially. The client allows the user to display and navigate this VE database. For this purpose, the client needs only those data items, that are actually being displayed. Consequently, there is no need to transmit the whole database from the server and store it at the client. It is sufficient if the client has the data for those objects available that are contained in its *area of interest* (AOI). By restricting the geometry transmission to the data that is actually required for display and making sure that data is delivered "just in time" for display, we can gain significant savings in network bandwidth and local memory requirements, allowing to handle more complex, more interesting data sets.

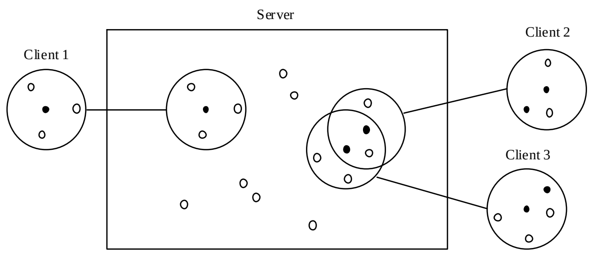

Figure 4. A distributed geometry database: A server stores geographically dispersed objects (small white circles). Each client's view is limited to an AOI (large circles).

The major benefit for this application is that very little storage is needed for the procedural models compared to conventional modeling, which benefits the size of the area of interest and the responsiveness of network transmission (e. g., when the user is moving fast). Furthermore, as models are potentially instanced many times, the model's actual geometric description must only be transmitted when the first instance of a species is encountered, later instances can be specified by parameters only.

5. Level of detail rendering

Especially when large and complex scenes are to be rendered, support for levels of detail is essential to assure that the image generator is not overloaded. Some models (e. g. terrain) exhibit plain self-similarity and can easily be displayed at multiple levels of detail by simply reducing recursion depth.

However, more complex DAG models use the recursion level to represent different parts of the object. A typical example is that early recursion levels create the stem and branches of a tree, and late recursion levels create the leaves. Consequently, recursion cannot simply be pruned, but instead sub-DAGS must be replaced by single primitives of appropriate shape and color. For example, the complex twig of a conifer tree can be replaced by a flat green cone. An example is given in Figure 7. While selection of such levels of detail is done automatically, the creation of levels of detail cannot easily be automated. In general, levels of detail must always be hand-crafted by substituting simple primitives for complex sub-graphs, which can be a labor-intensive process.

6. Implementation

A number of basic elements is required for the modeling/rendering system for DCGs, independent of implementation language, platform, or even rendering method:

- data structures for nodes, including geometric primitives (polygon sets, simple solid bodies such as spheres), materials, transformations; plus group, selection, and computation nodes
- a traversal mechanism that walks through a graph and calls appropriate functions for every node encountered during the traversal to perform performing rendering, bounding box computation etc.
- global variables accessible by the nodes, that can be used as the parameters of the PL-system
- a stack for parameters to simulate local scope of recursive function calls
- support for evaluation of functional expressions in the calculation nodes
- a text-based file format for easy specification of models, so that models can be created without using a compiler

All these features are not specific to fractal modeling, but rather are standard features of advanced modeling toolkits. So it is not surprising that the Open Inventor toolkit from Silicon Graphics [Stra92] comes with all the elements listed above, and is well suited for our needs. Using a commercial toolkit as a foundation also has the advantage that all the additional features combined in the toolkit are readily available. We decided to stick to the rule that no feature that was already available in Inventor should be re-implemented, so most of the work went into tweaking Inventor's features to work under circumstances not originally intended by the designers. The software resulting from this effort is called RECURSIV.

Inventor is an object-oriented toolkit composed of a large body of C++ classes, providing an abundance of nodes to construct scenes from. All the required nodes listed above are available. The exception are calculation nodes and group nodes supporting cyclic links (see below). New node types can be derived as subclasses of existing nodes, which helps to save implementation work.

The traversal mechanism in Inventor is based on action classes, that traverse a given scene graph, and call nodes' methods as appropriate. If a selection node (SoSwitch) is coupled to a counter that makes traversal branch into a terminal subgraph at some point, traversal of graphs containing cycles works as expected without additional measures. A more subtle issue, however, is the automatic *notification* of events (e. g. attribute changes) happening somewhere in the graph, that are propagated *upwards*. Notification can be seen as a kind of inverse traversal, and consequently does not get caught by the selection nodes. We therefore had to block notification after the first cycle by introducing a special kind of parent-child link in the form of a special group node called RecursionSeparator.

Inventor scene graphs store data in fields contained in nodes, for example, the SoMaterial node has a field called transparency. Additionally, global fields are supported, that are not part of any node and can be shared by many nodes. We use global fields to store the parameters of the PL-system.

Elements are Inventor's concept for holding state that changes during the traversal of a scene graph. Unlike global fields, that store user-defined data, elements are special purpose data items (for example, the current object space transformation is used by the rendering library). Separators are special group nodes that save the current state on a stack, and restore the old value when traversal returns from the subgraph of the separator. This is exactly the behavior we need for the parameters of the PL-system modified by calculation nodes. Since standard Inventor behavior does not allow to save global fields on the element stack, we created a new element, the RecursionElement, that registers the global fields declared as parameters, and stores/retrieves them from the stack every time a RecursionSeparator is traversed.

Mathematical expressions as required for the computation performed by the calculation nodes is provided in the form of calculator engines. Engines are Inventor's concept for introducing functional dependencies among fields in the scene graph (either triggered from outside, such as by the system clock, or to connect fields inside the scene graph). Engines also provide an expression parser, so we constructed a calculation node (RecursionCalculator) containing a single field that is made functionally dependent on the parameter global fields using a calculator engine. Upon traversal, the functionally dependent variable is evaluated (lazy evaluation), and the results is written back to the parameter global field associated with the calculation node (the left hand side of the assignment). Note that computing new values for parameters (performed by RecursionCalculator) is separated from saving parameter values on the stack (performed by RecursionSeparators), so multiple parameters can be computed in series for better efficiency.

```
RecursionCalculator { expression “c=4” }
DEF Worm RecursionSwitch {
  expression “c>0”     #child to
traverse
  Cone {}                 #terminal
child
  Separator {           #recursive
branch
    Sphere {}
    Translation { translation 1 0 0 }
    RecursionCalculator {    #dec
counter
                     expression “c=c-1”
}
    RecursionSeparator { USE Worm }
  }
}
```

Inventor has a well readable text-based file format. Writing new nodes automatically extends the file format, so that that an extension of the parser is not necessary. However, the file format syntax for engines is rather arcane, so we decided to build a small parser extension that simplifies specification, and also adds transformations that are functionally dependent on parameter global fields, a feature frequently required when modeling recursive artifacts. The example shows the Inventor file for the model from Figure 1.

We also added two Inventor actions that initialize and disassemble a recursive scene graph. The RecursionInitAction must be applied after reading in a scene graph from a file to transform expressions into engines, to create parameter global fields and register them with the RecursionElement for proper stacking behavior, and maintain reference counters for the parameters (one parameter can be shared by multiple objects). The RecursionDoneAction allows proper deletion of a scene graph: Inventor uses reference counting for nodes, which requires that cycles must be cut open before the graph can be properly deleted; furthermore, reference counters of parameter global fields must be updated.

While the extended Inventor file format is a very efficient tool for precise modeling of RECURSIV models, it lacks intuitivity. To overcome this restriction and make the tool more useful for casual users, an interactive graphical editor was developed that allows convenient specification of RECURSIV models with interactive previewing and tweaking of parameters. A screen shot is given in Figure 5.

7. Results

While modeling DCGs takes a little practice, we found that it is possible to achieve very appealing results (examples are shown in the color section). To support our claims of improved memory usage and network utilization, in the following we list a comparison of the sizes of some procedural models (uncompressed ASCII RECURSIV files) and their conventional counterparts where every detail is explicitly stored. We did only consider geometry, not color (color was fixed in both variants). In the binary file cones and cylinders were taken into account as 7 floats (3 bottom, 3 top, 1 radius), individual vertices of triangles with 3 floats (x, y, z).

- The tree model (Figure 8) consists of 16884 cones, 13776 cylinders and 603 leaves (triangle strips of length 4). The RECURSIV file uses 7556 bytes, while the binary file consumes 74076 bytes.
- The conifer tree, variant 1 (Figure 7) consists of 211 cylinders and 15600 cones. The RECURSIV file uses 7418 bytes, while the binary file consumes 442708 bytes.
- Terrain (Figure 7) can be set to arbitrary resolution. A pure fractal implementation in RECURSIV (only the edges of the terrain tile and the fractal dimension are explicitly specified) takes 3832 byte. A height field at resolution 1024x1024 with 1 float per height value takes 4MB.

It is easy to see how memory and network transmission time can be saved by using the procedural models instead of their conventional counterparts. Note that these are only for one instance of a given model. For example, the tree model is specified using only three parameters (height, average branch length, branching frequency), all other detail are generated from random numbers.

As far as rendering performance is concerned, the traversal of a DCG is only slightly more costly than the traversal of a DAG (the overhead coming mostly from the necessary calculations). Our implementation in Open Inventor shows that interactive frame rates (10 frames and above on an Indigo2) are easily achieved for scenes of moderate complexity (e.g. a small forest); the limits of the method mostly come from the absolute number of primitives (triangles) that the image generator hardware is capable of (note that the hierarchical structure of the model interferes with the use of high-performance triangle strips). The key lies in well designed models with efficient levels of detail.

8. Conclusions

We have presented a simple extension to DAG based rendering toolkits for interactive rendering. With the addition of cycles, PL-systems can be directly modeled and rendered as directed cyclic graphs. Interactive design of natural phenomena and efficient representation of outdoor scenes, especially for distributed virtual environments, are made possible by this approach. Future work will involve extending the number of models developed with this system to cover a larger range of botanical species, and developing larger-scale landscape models with realistic vegetation.

While detailed geometry is important at close range, image-based simplifications can be very efficient at medium and far range. The use of dynamic imposters [Scha96] is ideal for that purpose: The rendering cost for a large number of complex objects is reduced to rendering individual textured polygons, with the exception of infrequent refreshes of the imposters. Future work will involve improving the performance of rendering by impostors.

9. Acknowledgments

This work was sponsored by the Austrian Science Foundation (FWF) under contract number P-11392-MAT.

For further information on the recursIV project please refer to:

http://www.cg.tuwien.ac.at/research/vr/recursIV/

10. References

[Akel93] K. Akeley: Reality Engine Graphics. Proceedings of SIGGRAPH'93, pp. 109-116 (1993)

[Aono84] M. Aono, T. Kunii: Botanical Image Tree Generation. IEEE Computer Graphics and Applications, Vol. 4, No. 5, pp. 10-34 (1984)

[Deer95] M. Deering: Geometry Compression. Proceedings of SIGGRAPH'95, pp. 13-20 (1995)

[DeRe88] P. DeReffye, C. Edelin, J. Francon, M. Jaeger, C. Puech: Plant models faithful to botanical structure and development. Proceedings of SIGGRAPH'88, pp. 151-158 (1988)

[Fole90] J. Foley, A. van Dam, S. Feiner, J. Hughes: Computer Graphics: Principles and Practice. Addison-Wesley Publishing Co., ISBN 0-201-12110-7 (1990)

[Funk93] T. Funkhouser, C. Sequin: Adaptive Display Algorithm for Interactive Frame Rates During Visualisation of Complex Virtual Environments. Proceedings of SIGGRAPH'93, pp. 247-254 (1993)

[Funk95] T. Funkhouser, S. Teller, C. Sequin, D. Khorramabadi: The UC Berkeley System for Interactive Visualization of Large Architectural Models. Presence, Vol. 5, No. 1, pp. 13-44 (1995)

[Gerv95] M. Gervautz , C. Traxler: Representation and Realistic Rendering of Natural Phenomena with Cyclic CSG Graphs. Visual Computer, Vol. 12, No. 1, pp. 62-74 (1995)

[Max96] N. Max: Hierarchical Rendering of Trees from Precomputed Multi-Layer Z-Buffers. Proc. EUROGRAPHICS Workshop on Rendering Techniques'96 (eds. Pueyo, Schröder), Springer Vienna-New York, pp. 165-174 (1996)

[Neyr96] F. Neyret: Synthesizing Verdant Landscapes using Volumetric Textures. Proc. EUROGRAPHICS Workshop on Rendering Techniques'96 (eds. Pueyo, Schröder), Springer Vienna-New York, pp. 215-224 (1996)

[Prus90a] P. Prusinkiewicz, A. Lindenmayer: The algorithmic beauty of plants. Springer, New York (1990)

[Prus90b] Prusinkiewicz P., Hanan J.: Visualization of botanical structures and processes using parametric L-systems. Scientific Visualization and Graphics Simulation,Wiley & Sons, pp. 183-201 (1990)

[Reev85] W. Reeves: Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems. Proceedings of SIGGRAPH'85, pp. 313 (1985)

[Scha96] G. Schaufler, W. Stürzlinger: A Three-Dimensional Image Cache for Virtual Reality. Proceedings of EUROGRAPHICS'96 (1996)

[Schm96] D. Schmalstieg, M. Gervautz: Demand-Driven Geometry Transmission for Distributed Virtual Environments. Proceedings of EUROGRAPHICS (1996)

[Smit84] A. Smith: Plants, fractals and formal languages. Computer Graphics (Proceedings SIGGRAPH), pp. 1-10 (1984)

[Stra92] P. Strauss, R. Carey: An Object Oriented 3D Graphics Toolkit. Computer Graphics (Proceedings SIGGRAPH), Vol. 26, No. 2, pp. 341 (1992)

[Trax97] C. Traxler, M. Gervautz: Efficient Raytracing of Complex Natural Scenes. Proceedings Fractals'97, World Scientific Publishers (1997)

[Webe95] J. Weber, J. Penn: Creation and Rendering of Realistic Trees. Computer Graphics (Proceedings SIGGRAPH), pp. 119-128 (1995)

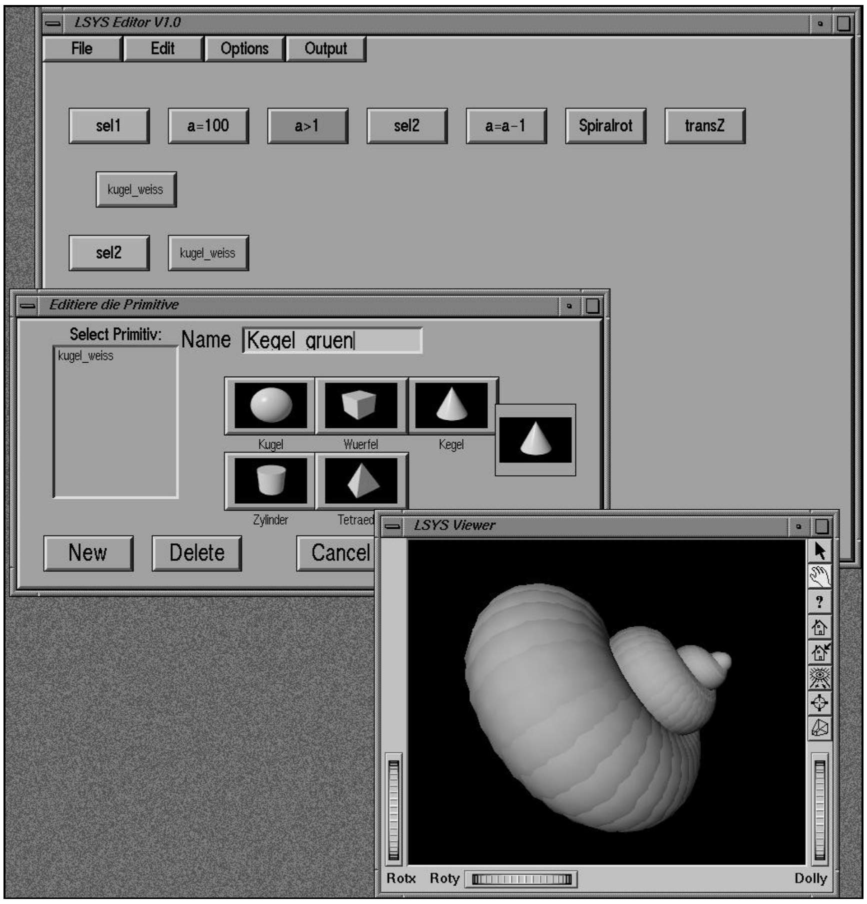

Figure 5: User interface of the interactive editor

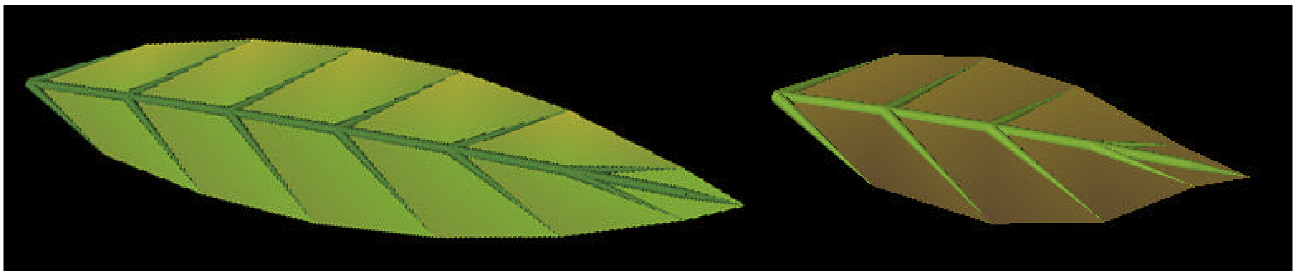

Figure 6: Parameters allow to model different leaves from one model

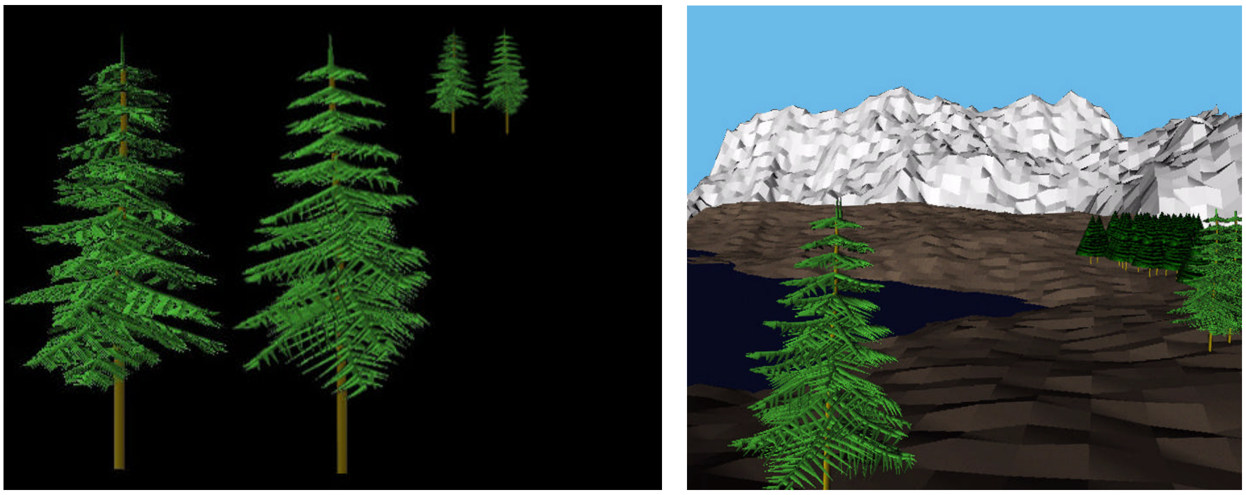

Figure 7: (left)Levels of detail for a tree are modeled by modifying sub-graphs; (right) a forrest from of conifer trees

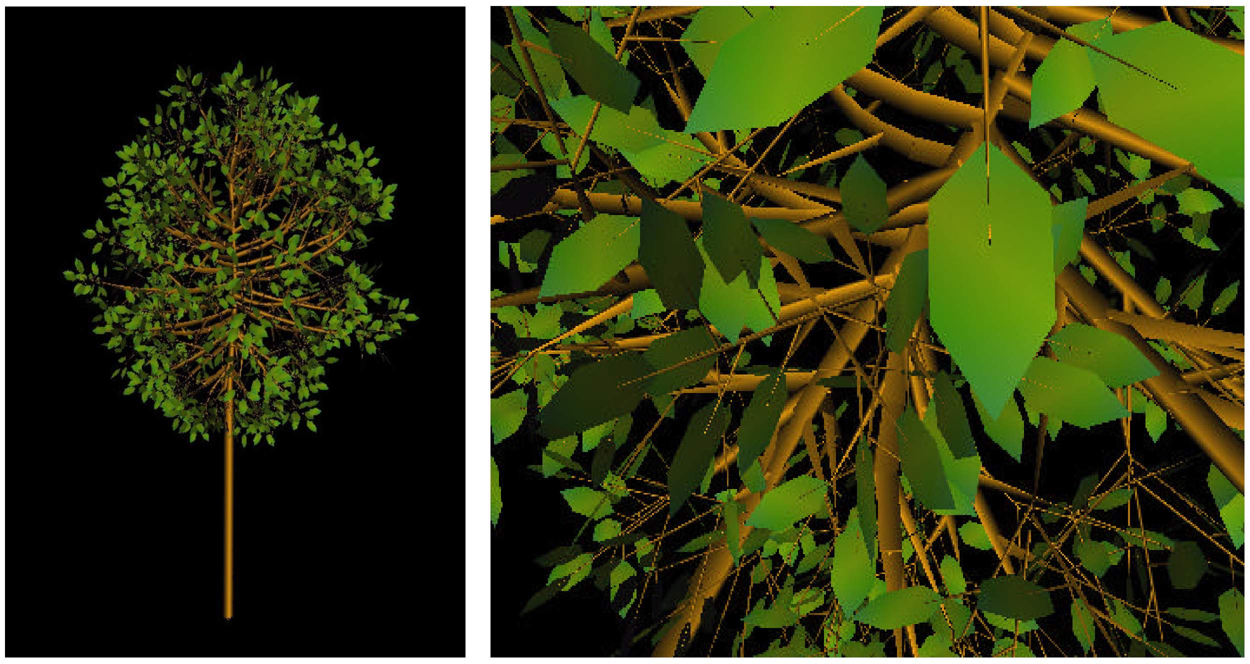

Figure 8: A monopodial tree at normal distance and a close-up view