Dissertation

THE REMOTE RENDERING PIPELINE

MANAGING GEOMETRY AND BANDWIDTH IN DISTRIBUTED VIRTUAL ENVIRONMENTS

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften

eingereicht an der Technischen Universität Wien
Technisch-Naturwissenschaftliche Fakultät

von

Dipl.-Ing. Dieter Schmalstieg

Abstract.

The contribution of this thesis is at the place where interactive 3-D computer graphics and distributed systems meet. Virtual Environments are concerned with the convincing simulation of a virtual world. One of the most promising aspects of this approach lies in its potential as a way of bringing people together, as a virtual meeting place.

To overcome current restrictions in network performance and bandwidth, techniques that have already been used for improving the rendering performance for virtual reality applications can be adopted and enhanced. In this context, we develop the concept of the Remote Rendering Pipeline, that extends the traditional rendering pipeline for interactive graphics to include the network transmission of geometry data. By optimizing the steps of the Remote Rendering Pipeline, and combining these improvements, a system that is better prepared to deal with complex and interesting virtual worlds emerges.

After a discussion of the state of the art in the fields of interactive 3-D graphics and distributed virtual environments, the Remote Rendering Pipeline is introduced, a conceptual model of rendering in distributed systems. Its elements are discussed in the following chapters: the demand-driven geometry transmission protocol, a strategy for managing partially replicated geometry databases for virtual environments; an octree-based level of detail generator; smooth levels of detail, a novel data structure for incremental encoding and transmission of polygonal objects; and a modeling and rendering toolkit for directed cyclic graphs allowing a compact representation of a large class of natural phenomena.

Kurzfassung.

Der Beitrag dieser Dissertation liegt dort, wo sich interaktive dreidimensionale Computergraphik und verteilte Systeme überschneiden. Virtual Environments beschäftigen sich mit der glaubhaften Simulation von virtuellen Welten. Der vielversprechendste Aspekt dieser Methode ist die Möglichkeit, Menschen in virtuellen Umgebungen zusammenzuführen.

Gegen die derzeitigen Beschränkungen in Netzwerkdurchsatz und -bandbreite können Algorithmen angepaßt und erweitert werden, die bereits für interaktive Graphik erfolgreich eingesetzt werden. In diesem Zusammenhang wird das Konzept der Remote Rendering Pipeline vorgestellt, einer Erweiterung der traditionellen Rendering Pipeline, die eine Netzwerkübertragung der Graphikdaten miteinschließt. Die Optimierung der Abschnitte dieser Pipeline ergibt ein Graphiksystem, das komplexe und interessante virtuelle Welten darstellen kann.

Nach einer Erörterung des Standes der Forschung in den Bereichen interaktive dreidimensionale Graphik und verteilte virtuelle Welten wird die Remote Rendering Pipeline vorgestellt, ein konzeptuelles Modell für verteilte graphische Systeme. Die Elemente dieser Pipeline werden in den folgenden Kapiteln beschrieben: das bedarfsgesteuerte Geometrie-Übertragungs-Protokoll, eine Strategie zur Verwaltung von teilweise replizierten geometrischen Datenbanken; ein Octree-basierter Detailstufengenerator; Smooth Levels of Detail, eine neue Datenstruktur für die schrittweise Codierung und Übertragung von polygonalen Objekten; und ein Modellierung- und Darstellungs-Softwarewerkzeug für gerichtete zyklische Graphen, die eine kompakte Darstellung einer wichtigen Klasse von natürlichen Phänomenen gestattet.

Acknowledgments.

I would like to thank all the people who helped me in finishing this thesis:

Michael Gervautz, my advisor, for keeping me struggling,

Werner Purgathofer, for giving me a chance to pursue my ideas,

Eduard Gröller, the master, who probably spent more time instructing me than his own PhD students,

Holda Schmalstieg, my mother, who gave me love, food, life, shelter, more love and moral support (though not necessarily in that order),

Peter Schmalstieg, my father, for his continuous support (in every respect),

Tomek Mazuryk, who shared my office and my thoughts for quite a while,

Zsolt Szalavari, for always listening,

Gernot Schaufler, with whom I share the credit for the work on "smooth levels of detail", and performed the best cooperation I've ever experienced,

Herbert Buchegger, *Gerd Hesina*, *Stephan Mantler*, *Rainhard Sainitzer*, and *Peter Wonka*, who implemented parts of the software described in this thesis,

and finally *Dr. Dide*, who over time became like a brother to me.

TABLE OF CONTENTS

1. Introduction 11

1.1 Virtual Reality, the Universe, and Everything 11
1.2 What this thesis is about 12
1.3 Individual publications about this work 14

2. Interactive 3-D Graphics 15

2.1 Motivation from human factors 15
2.2 The rendering pipeline 16
2.3 Performance of the rendering system 16
2.4 Visibility processing 17
2.5 Levels of detail 19
2.6 Image based rendering 23
2.7 Managing latency 24
2.8 Managing large geometric databases 25
2.9 Summary 26

3. Distributed Virtual Environments 27

3.1 Introduction 27
3.2 Fundamentals of Distributed Virtual Environments 28
3.3 Network design 30
3.4 Local simulation 33
3.5 Summary 33

4. An Overview of the Remote Rendering Pipeline 34

4.1 Introduction: What is Remote Rendering? 34
4.2 Geometry management with the Remote Rendering Pipeline 35
4.3 Exploiting task-related knowledge for optimization 36

5. Demand-Driven Geometry Transmission 38

5.1 Introduction 38
5.2 Data management 38
5.3 Geometry Data Structure 40
5.4 Strategy of the client 41
5.5 Protocol design 44
5.6 Implementation 45
5.7 Results 48
5.8 Summary 49

6. An Octree-Based Level of Detail Generator 50

6.1 Introduction and motivation 50
6.2 Octree quantization for levels of detail 50
6.3 Dealing with VRML specifics 54
6.4 Joining nodes 57

6.5 Implementation....60
6.6 Results....60
6.7 Summary....61

7. Smooth Levels of Detail....62

7.1 Introduction and motivation....62
7.2 Representing the model as a hierarchical cluster tree....63
7.3 Manipulation of the cluster tree....65
7.4 Transmission Protocol....67
7.5 Hierarchical precision encoding of vertices....69
7.6 Model reconstruction and rendering....70
7.7 Results and Comparison....71
7.8 Using smooth LODs with demand-driven geometry transmission....73
7.9 Summary....73

8. Interactive Rendering of Natural Phenomena Using Directed Cyclic Graphs....75

8.1 Introduction....75
8.2 Overview of our approach....76
8.3 Background: Rendering Directed Cyclic Graphs....77
8.4 Efficiency of rendering....80
8.5 Implementation....81
8.6 Results....82
8.7 Summary....82

9. Conclusions....83

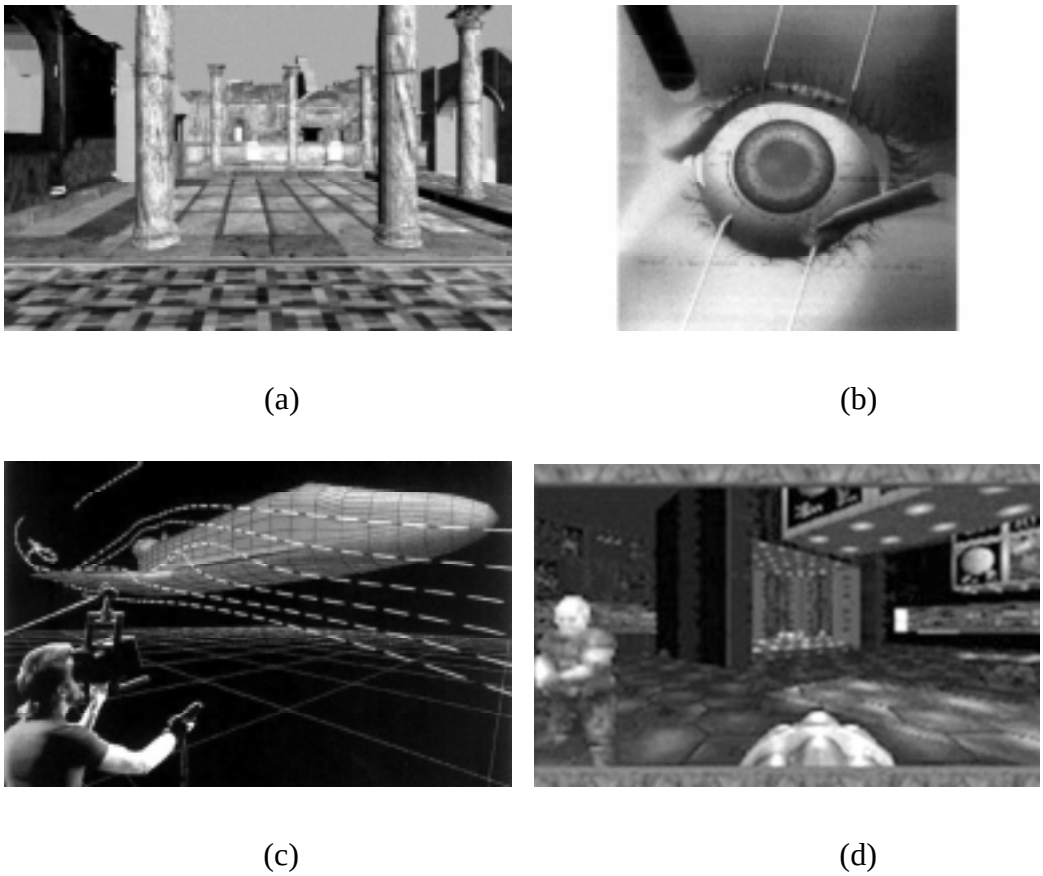
9.1 Critique of the Remote Rendering Pipeline....83
9.2 Future work....85

10. References....86

1. Introduction

1.1 Virtual Reality, the Universe, and Everything

Virtual Reality has been the darling of the media for a couple of years now, and has long replaced artificial intelligence as the most popular buzzword haunting the computer corners of press and TV networks. We see many interesting applications emerge, such as architectural walk-throughs, surgical planning, scientific visualization and computer games.



(a) (b)

(c) (d)

Figure 1: Examples for virtual reality applications: (a) architectural walk-through of a historic site, (b) planning system for eye surgery, (c) virtual wind tunnel, (d) educational software

While all the interest has helped to attract attention and gather funds for research in the field, overexpectations and hype have also seriously hindered common understanding of the technology, and led to a confusing and inconsistently used vocabulary.

Allow us therefore to start with a very brief tour of the field we are interested in, and introduce a few essential terms that will help us in explaining the work presented in this thesis, focused on Virtual Reality, Interactive 3-D Graphics, and Virtual Environments.

Virtual Reality is used - unfortunately - to describe everything from computer games to CAD packages. In our opinion, this term is strongly connected to the concept of immersion of a person into a simulated surrounding produced by a computer. The goal is to make the synthesized experience created by presenting artificial stimuli to the human senses convincing enough to achieve a „suspense of disbelief“, ultimately making the real and virtual surrounding indistinguishable.

Virtual Reality should really be considered a user interface technology. A lot of research has been devoted to hardware at the interface between humans and computer: Head-mounted displays, stereoscopic vision, spatial sound, position and orientation trackers and data gloves are only a few prominent examples. In this context, a lot of human factors must be considered. Some of the ergonomic issues involve the presentation of computer graphics, which builds the bridge to the second area.

Interactive 3-D Graphics is concerned with the generation of images by a computer in real time, which is anything between 10 and 50 frames per second, depending on the application. This requirement is doubled if a stereoscopic display has to be supported. An insufficient frame rate or high latency of response to user interaction destroys immersion and may even lead to simulator sickness, but fortunately many applications allow to trade frame rate for image quality.

Traditionally computer graphics has focused on the quality of images, culminating in photo-realistic image synthesis. The increasing power of computer systems has made it possible to create three-dimensional animated images in real-time, but with the available computing power grow the demands of applications. It is easy to see that raw hardware muscle will never satisfy the performance goals, so smart algorithms are needed that better manage the available resources and fundamentally reduce the effort of 3-D rendering.

Virtual Environments are concerned with the convincing simulation of a virtual world. This task does not only involve computer graphics, but also a lot of other complex issues including animation, modeling and authoring, interaction, manipulation and navigation, autonomous agents and artificial intelligence, and distributed systems providing support for a large number of human users. The complexity of bringing all this issues together into a functioning whole involves sophisticated methods of software engineering and systems design, which is the challenge of research in virtual environments.

1.2 What this thesis is about

The contribution of this thesis is at the place where interactive 3-D graphics and distributed systems meet. One of the most promising aspects of virtual environments lies in its potential as a way of bringing people together, as a virtual meeting place. Participation of multiple users requires the employment of a distributed system, which is inherently more complex than an application designed for a single user. A lot of work has been done investigating issues in *distributed virtual*

environments, focusing among others on animation, interaction, concurrency, and network topologies. However, we feel that a particular shortcoming of current systems has been rather neglected, namely the conflict between the ever-growing size of databases holding the geometric description of the 3-D objects populating the virtual world, and the painfully limited bandwidth available for distributed virtual environments.

Problem statement

Networked applications involving 3-D graphics, but especially virtual environments, require shared access to geometry databases describing the objects that must be displayed. Such databases representing virtual worlds can become very large if objects involve a lot of detail and the world is composed of a large number of objects. We may build large and powerful servers to manage these virtual worlds, but the typical desktop computer used for displaying the virtual world is not as powerful.

In order to generate images of the virtual world, the geometric models of the visible objects must be available locally at the computer that performs the rendering. Therefore most applications involve a distribution step where the geometry database is copied into the local domain (main memory, hard disk). Either this step is carried out off-line (e. g., CD-ROM distribution of a computer game), or the data is downloaded over a network. One associated problem is that the size of the virtual world may exceed the capacity of the computer's storage, so the scalability of virtual worlds is restricted. Even worse is the problem that network connections are notoriously slow, and the situation rapidly deteriorates with the growth of the Internet and the inadequate performance of network commodities such as modems. Slow connections and long download times mean that applications are not fit for interactive usage. Apart from being annoying to the user, the illusion of immersion into a virtual world is destroyed.

Solution that help to streamline this process of geometry database distribution allow for larger amounts of data to be handled: We can have more detailed objects, and we can handle more objects simultaneously, so that virtual worlds can be more realistic. Beyond the ability to simply handle three-dimensional data faster, a new type of application suddenly comes into reach: We can start thinking of a continuous 3-D Internet, a true *Cyberspace* [72].

Proposed solutions

To overcome the problems restricting the use of geometry data in networked applications, techniques that have already been used for improving the rendering performance for virtual reality applications can be adopted and extended to reduce bandwidth needs and improve network related performance. In this context, we develop the concept of the *Remote Rendering Pipeline*, that extends the traditional rendering pipeline for interactive graphics to include the network transmission of the data to be rendered. By optimizing the steps of the Remote Rendering Pipeline, and combining these improvements, a system that is better prepared to deal with complex and interesting virtual worlds emerges.

As a prerequisite to the work presented in this thesis, chapters 2 and 3 discuss the state of the art in interactive 3-D graphics and distributed virtual environments, respectively. Chapter 4 introduces the

Remote Rendering Pipeline, whose elements are discussed in detail the following chapters. Chapter 5 explains a strategy for managing partially replicated geometric databases for virtual environments called the *demand-driven geometry transmission protocol*. Chapter 6 discusses an *octree-based level of detail generator*. Chapter 7 deals with a novel data structure for incremental encoding and transmission of polygonal objects called *smooth levels of detail*, chapter 8 outlines a *modeling and rendering toolkit for directed cyclic graphs* allowing a compact representation of a large class of natural phenomena. Conclusions are drawn in chapter 9, and chapter 10 lists relevant references.

1.3 Individual publications about this work

Results of this work have been previously published by the author. The following papers describe the preliminary outcome of the work:

- D. Schmalstieg, M. Gervautz: *Towards a Virtual Environment for Interactive World Building*. Proceedings of the GI Workshop on Modeling, Virtual Worlds, Distributed Graphics (MVD'95), Bonn (Nov. 1995)
- D. Schmalstieg, M. Gervautz: *Implementing Gibsonian Virtual Environments*. Proceedings of the Thirteenth European Meeting on Cybernetics and Systems Research, Vienna, Austria, April 1996. Republished in: Cybernetics and Systems - An International Journal (ed. R. Trappl), Vol. 27, No. 6, pp. 527-540, Taylor & Francis, Washington DC (1996)
- D. Schmalstieg, M. Gervautz, P. Stieglecker: *Optimizing Communication in Distributed Virtual Environments by Specialized Protocols*. In: Virtual Environments and Scientific Visualization'95 (ed. M. Göbel), Springer Wien-New York (1995)
- D. Schmalstieg, M. Gervautz: *Demand-Driven Geometry Transmission for Distributed Virtual Environments*. Computer Graphics Forum (Proceedings EUROGRAPHICS), Vol. 15, No. 3, pp. 421-433 (1996)
- D. Schmalstieg, G. Schaufler: *Incremental Encoding of Polygonal Models*. Proceedings of the 30th Hawaii International Conference on System Sciences (HICSS-30), Maui, Hawaii, USA, Vol. V, pp. 638-645, Jan. 7-10 (1997)
- D. Schmalstieg: *An Octree-Based Level of Detail Generator for VRML*. Proceedings of the ACM SIGGRAPH 2nd Symposium on VRML, pp. 127-133, Monterey CA, Feb 24-27 (1997).
- D. Schmalstieg, G. Schaufler: *Smooth Levels of Detail*. Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS'97), pp. 12-19, Albuquerque, New Mexico, March 1-5, (1997)
- D. Schmalstieg, M. Gervautz: *Modeling and Rendering of Outdoor Scenes for Distributed Virtual Environments*. Proceedings of ACM Symposium on Virtual Reality Software and Technology 1997 (VRST'97), pp. 209-216, Lausanne, Switzerland, Sep. 15-17 (1997)

2. Interactive 3-D Graphics

2.1 Motivation from human factors

In producing the illusion of a virtual world, the most important contribution is certainly being made by the visual component. Ideally, we would like to be able to produce images that always match or exceed the limits of human visual perception in all aspects. Unfortunately, limitations in the performance of the image generators we employ for this task defeat this goal. The ever-increasing demands of applications make it unlikely that this situation will ever change significantly. Working solutions require us to trade off image fidelity for graphical complexity and performance of the application. In doing so, it is vital to understand those human factors that dominate the perception of interactive 3-D computer graphics [35].

Visual acuity denotes the degree to which visible features can be perceived, and is commonly measured as the angle subtended at the eye. On-axis resolution is around 1 arc minute; rendering graphical features that fall below the threshold of perception would be wasteful, and the effort could be used better elsewhere.

The human *field of view* is limited to about 180 degrees horizontally and 120 degrees vertically. These are the biological constraints on the visible portion of the environment, which are usually further restricted by display systems. Again, putting effort in displaying graphics outside the field of view is a waste of performance.

Latency is the time measured from the setting of an input until the corresponding output is manifested. Many factors contribute to latency: input devices, software architecture, rendering time, display scan-out. For the rendering portion of the system, latency is typically taken as the time after the eyepoint is set until the last pixel of the corresponding frame is scanned out by the display device. Excessive latency can lead to over-compensation and control oscillations induced by the user.

A sufficiently *high frame rate* is necessary to fool the human eye into seeing a continuous and smooth motion. Generally, most displays have refresh rates close to or above the flicker limit (60-80Hz) of the human visual system, so the display itself introduces no significant problems. However, the time it takes to generate an image by the image generator is bound to scene complexity. Low frame rates make motion choppy and are especially problematic when rapid motion is possible as with a head-mounted display. Furthermore, latency is increased since a low frame rate means a longer time until the next image can be presented.

Constant frame rates are desirable as variations in frame rate tend to distract the user from the task at hand. Variable frame rates also cause temporal inaccuracies because the change in frame rate

affects the latency. An unanticipated change in latency leads to a frame not being displayed at the time it was planned for, and results in jerky motion.

2.2 The rendering pipeline

On most current image generators, rendering is implemented as a pipeline (see Figure 2) that usually includes the following stages:

- *Database traversal* is typically done on the host CPU and can become a bottleneck if the traversal is unable to keep the graphics pipeline full.
- *Polygon processing* includes vertex transformations, lighting calculations and 2-D triangle setup. On very low end image generators, this stage is performed by the host CPU, but the trend goes towards geometry processors that are specialized for this job.
- *Pixel processing* involves operations such as depth buffer testing, anti-aliasing, texturing, and alpha blending. Most of these operations require access to memory, e.g. for looking up texture values, which can have great impact on the performance. Pixel fill rates can also depend on the size of the primitives being rendered because of the per-primitive setup effort.

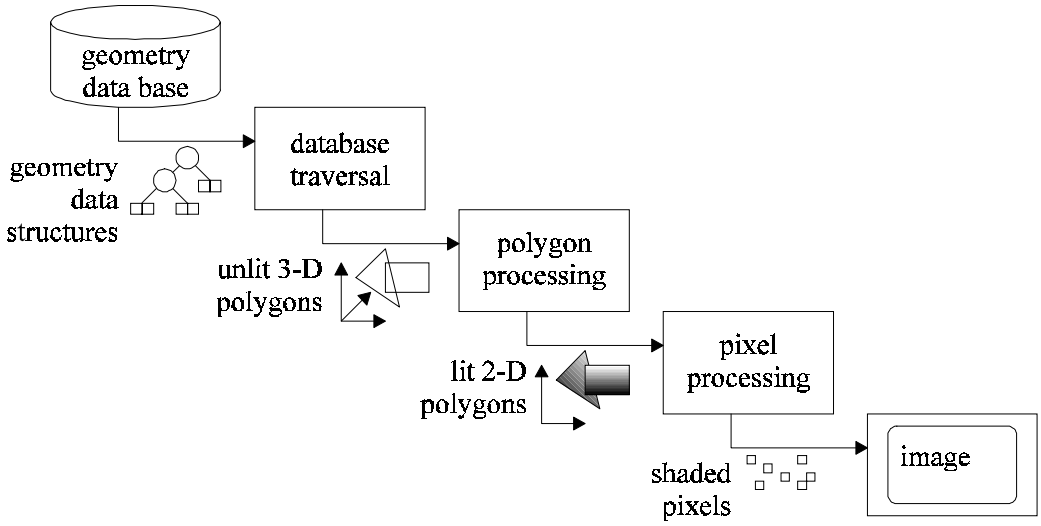


Figure 2: The rendering pipeline

2.3 Performance of the rendering system

Maximizing frame rate and image quality becomes a problem of making the best use of the each available stages and avoiding bottlenecks. The remainder of this chapter discusses methods and techniques that aim to achieve optimal performance, and also take into account the relevant human factors.

Reduction of geometric complexity

The usual measure for scene complexity is the number of geometric primitives of which the scene is composed. As there is a hard limit on this number imposed by the hardware, any *reduction of*

geometric complexity is desirable. We are primarily interested in such simplifications that cannot be perceived by the human user due to the biological limitations mentioned in section 2.1. If the simplification cannot be concealed, there usually is a trade-off in image fidelity versus frame rate that has to be resolved. The most important methods for reducing geometric complexity are *visibility processing* (section 2.4), *levels of detail* (section 2.5), and the relatively new field of *image based rendering* (section 2.6).

Optimizing runtime rendering

Besides reducing the load on the rendering system, it is also necessary to ensure efficient use of the available capacity. This involves tuning the geometric database to avoid suboptimal structures in the data. Some of the measures are relatively simple: Many pipelined image generators are optimized for triangle strips sharing adjacent triangles. Restructuring polygonal data in long strips helps to boost performance. As switching graphics mode and graphical attributes (such as color or shading) involves a performance penalty, the data should be presorted by type and mode if possible to avoid changes in the state of the image generator. Transformations that are important for the modeling process can hurt performance during rendering. Preprocessing allows all static transformation to be eliminated. Many animations do not depend on run-time parameters, so they can be precomputed and simply replayed at run-time, thus saving computation time.

Beyond these simple optimizations, there are performance issues that require more sophisticated techniques, in particular latency management (section 2.7) and management of large geometric databases (section 2.8).

2.4 Visibility processing

In the early days of raster graphics, a lot of attention has been paid to algorithms that resolve the problem of visibility of opaque surfaces. With the invention of the z-buffer, a relative brute force solution replaced all other methods in workstation-class image generators on the merit of its efficient hardware implementation. However, the virtual environments we would like to render today are composed of so many geometric primitives that resolving occlusion with the help of a z-buffer alone is infeasible. We rather have to compute as accurately as possible those portions of the scene that are actually visible, and let the hardware deal with this subset. Backface culling is a trivial example.

This task of visibility processing is carried out by the host CPU, and can be divided into two phases: *Visibility preprocessing* is an off-line task that computes data that can later be used in the *visibility culling* at run-time to quickly eliminate large portions of the scene that are certainly invisible.

The simplest way of visibility culling is *culling on the viewing frustum*. The problem here is to not let the rendering/clipping process do the work, but rather to rapidly discard the portion of the scene that lies outside the viewing frustum. A thoughtful spatial organization of the geometric primitives in the scene is necessary, which is usually done by grouping spatially coherent primitives into objects. Hierarchical bounding volumes can be used to sort out the invisible geometry. Naylor proposed to use a binary space partitioning (BSP) tree for the scene that can be efficiently intersected with a viewing frustum of any shape [53].

Virtual environments can broadly be categorized into sparse (i. e. most of the geometry within the viewing frustum can at least partly be seen, up to the virtual horizon), and densely occluded (such as building interiors where most of the geometry contained in the viewing frustum is hidden behind walls and other large occluders).

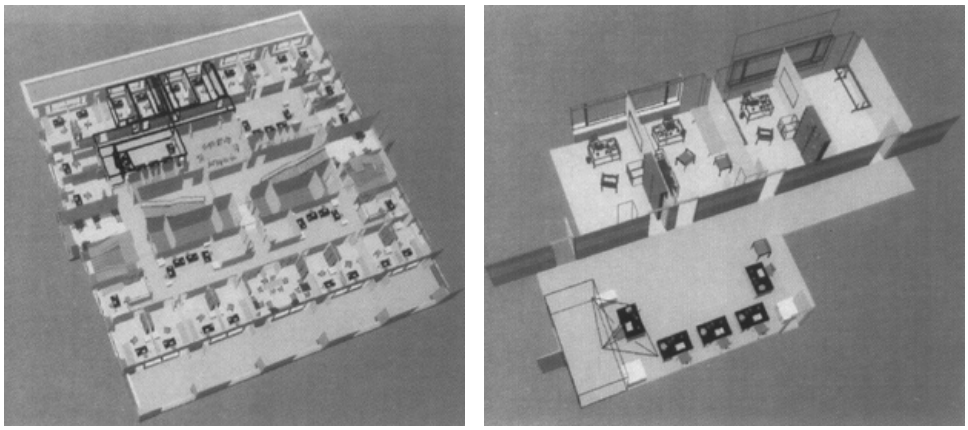


Figure 3: Visibility culling of a building interior from [22]. Left is the unpruned model, right shows the portion visible from the observer

Airey [1] first proposed to make use of the structure of densely occluded environments by employing *Potentially Visible Sets* (PVS). The environment is decomposed into cells. The criterion for the decomposition into cells is that the visible portion of the scene should be roughly the same from any viewpoint within a cell. This is usually approximately equivalent to the rooms of a building that are interconnected only by doors, windows, stairs, and hallways. Any such opening is referred to as a *portal*. The method precomputes a subdivision into cells, builds the cell adjacency graph, and associates with each cell a PVS, which is a conservative overestimate of the actual visible portion that can be efficiently computed. The geometry associated with the PVS is then rendered, and a standard z-buffer is used to resolve exact visibility. Airey used a shadow volume BSP buffer to estimate the PVS. Teller and Sequin [85] have taken this concept further and found an analytic solution to the portal-portal visibility problem. Using linear programming, they compute a complete set of cell-to-cell and cell-to-object visibilities (compare Figure 3). This approach is computationally intensive.

Therefore Luebke and Georges [45] propose a variant that works without visibility preprocessing: At runtime, they compute 2-D bounding boxes of the portals projected to screen space. During the traversal of the cell adjacency graph, as each successive portal is traversed, its bounding box is intersected with the aggregate culling box using only a few min-max comparisons. The content of each cell is tested for visibility through the current portal sequence by comparing the screen-space projection of each object's bounding box.

Greene, Kass & Miller [31] developed an algorithm called hierarchical z-buffer visibility that works well not only for densely occluded environments, but also for environments that have open spaces. The algorithm uses an octree in object space. The scene is rendered by recursively traversing the octree front to back. Before the geometry associated with a particular node in the octree is rendered, its visibility is estimated by testing the cube associated with the octree node for visibility. This test is

further accelerated by a z-pyramid maintained in image space. Using the octree and the z-pyramid as auxiliary data structures, the algorithm is able to exploit simultaneously object space and image space coherence, and temporal coherence for animated walk-throughs as well. However, today's hardware does not fully support the features required by the algorithm, and it has to rely on software rendering.

Coorg and Teller [14] present a spatially and temporally coherent visibility algorithm that exploits properties of the scene by distinguishing large occluders from occludees. The algorithm works entirely in object space, using fast conservative visibility tests. It uses an octree-based subdivision, and eliminates large portions of the model without touching most invisible polygons.

Another class of visibility algorithms deals with the reduced problem of 2 ½ D visibility, that is representative for all environments where the user's movement is constrained to a plane (such as walking on a single floor of a building). Yagel and Ray [89] present a visibility algorithm for cells based on a regular grid subdivision in 2-D. They classify the cells (grid elements) as open, occluded, or containing a wall, and compute approximated eye-to-cell visibility. Schmalstieg and Tobler [73] developed an algorithm that is based on a 2-D triangular mesh. The edges of the triangles can be elevated and interpreted as walls. They propose a recursive algorithm that can be rapidly executed at run-time and exploits spatial coherence by yielding not only the PVS but also exact visible portions of the walls. This algorithm lends itself especially for low-end platforms that do not have 3-D hardware acceleration.

Visibility computation is generally restricted to walk-throughs of static environments, because they include a heavy preprocessing stage that constructs an auxiliary spatial data structure. Sudarsky and Gotsman [83] make a fist attempt to include dynamically moving objects by introducing temporal bounding volumes that can be used for incremental updates of the spatial data structure.

2.5 Levels of detail

In very large virtual environments it is commonly the case that many objects are very small or distant. The size of many geometric features of these objects falls below the perception threshold or is smaller than a pixel on the screen. To better use the effort put into rendering such features, an object should be represented at multiple levels of detail (LODs). Simpler representation of an object can be used to improve the frame rates and memory utilization during interactive rendering. This technique was first described by Clark already in 1976 [13], and has been an active area of research ever since. The important questions for the application of levels of detail are: What strategy to use for selecting an appropriate level of detail for each object? How to best stage the transition between two successive levels of detail? And how to create good levels of detail for an original high-fidelity geometric object?

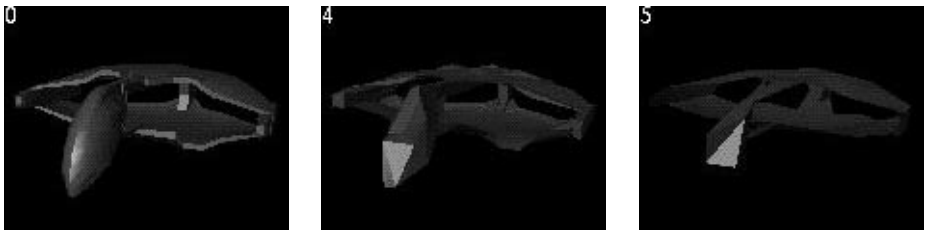


Figure 4: Three levels of detail for a "Romulan warbird" spaceship

Level of detail selection

There is no unique way to characterize the best selection of levels of detail for a group of objects comprising a scene, since human perception and aesthetics are hard to catch in a single formula. Instead, heuristics are used. Simple heuristics use the distance of the object from the observer or the size projected to the screen as a measure for the LOD. Unfortunately, these static heuristics do not adapt to variable load on the graphics pipeline: If too many complex objects are close to the observer, an overload can neither be detected nor avoided, and if rendering load is low, the image generator may be idle. Therefore, reactive level of detail selection is employed by flight simulators and real-time rendering toolkits such as Performer [61] with adaptive level of detail selection according to the time required by the last frames. However, such a strategy still fails to guarantee bounded frame rates, since sudden changes in the rendering load can be underestimated. Funkhouser and Sequin attacked this problem with a predictive selection algorithm [22] formulated as a cost/benefit optimization problem: What selection of levels of detail for each objects produces the best image while the accumulated cost for rendering each objects stays below the maximum capacity of the image generator at the desired frame rate? They use a cost heuristic based on the polygon and pixel capacity of the image generator, and a benefit heuristic constructed as a weighted average of factors such as the object's size, accuracy, and importance. The optimization problem is a variant of the well-known knapsack problem and can be incrementally and approximately solved with tractable computation effort for every frame.

Level of detail switching

For the use of levels of details, one may not neglect the issue how to stage the transition between two successive representations. The simplest way to do the transition is hard switching: At some point, the simpler model replaces the more complex model. This meets the performance goal, but can cause visual popping which may be disturbing. Instead of simply switching the models, for a short transition period they can be drawn blended together. This substantially reduces the popping effect, but temporarily increases the rendering load while both models are being drawn. Yet superior to blending is geometric morphing of one object into another. While this approach has certainly the best visual and performance effect, it works only for levels of detail with well-defined geometric correspondences.

Level of detail generation

The principal challenge of level of detail generation is to develop a way that takes a detailed model as input and automatically simplifies its geometry, while preserving appearance as good as possible. While in principle level of detail generation is relevant for complex models composed of any type of

geometric primitive, in practice almost exclusively polygonal models are used [15], so research has focused on this class.

Certain aspects are important in the classification of an algorithm that performs the task of polygonal simplification:

- *Local vs. global*: Local techniques operate on individual primitives such as vertices, adjacent edge segments or use some polygon characteristics. They are more apt to fulfill requirements related to small features such as a local preservation of shape. Global techniques attempt to optimize the polygon mesh based upon more general, high-level features of the model.
- *Error bounds*: Some of the methods guarantee user-controlled error bounds on the quality of the simplified objects according to some metric (such as the maximum distance between the surface of the original and the simplified model).

Besides these fundamental consideration, level of detail generation algorithms can be distinguished by algorithmic principles:

Mesh simplification algorithms work with polygonal meshes, often triangle meshes. Local operations on the surface of the object are performed, with a stress on the preservation of important visual features such as shape and topology. As an input, these algorithms expect topologically sound, manifold meshes. Unfortunately, this criterion is often not met by models generated with CAD packages, which leads to all sorts of practical problems. These algorithm also put more weight on feature preservation, for example, the simplification ratio is bound by the requirement of not reducing the genus of the object. The best results are achieved for smooth, organic objects that are over-tessellated, such as models obtained from a 3-D scanner.

The decimation algorithm by Schroeder, Zarge & Lorensen [74] analyses the vertices of the original model for possible removal based upon a distance criterion. A local re-triangulation scheme is then used to fill the hole resulting from the removed vertex.

Turk's re-tiling method [87] optimizes a triangle mesh by introducing new vertices to form a new representation. The new vertices are uniformly distributed on the surface of the original object. The original vertices are then iteratively removed, and the surface is locally re-triangulated to best match the local connectivity of the surface.

Hoppe et al. [36] present a triangular mesh simplification process which was based upon their surface reconstruction work. This technique introduced the concept of an energy function to model the opposing factors of polygon reduction and similarity to the original geometry. The energy function, used to provide a measure of the deviation between the original and the simplified mesh, is minimized to find an optimal distribution of the vertices.

Vertex clustering algorithms ignore topology in both input and output data. As a result, the algorithms perform robustly for degenerate input data, and can achieve arbitrary high compression for any kind of geometry. On the downside, the generated artifacts are much more severe, and local features are not preserved so well.

The fundamental idea of vertex clustering is to reduce the number of vertices of a polygonal model (usually a triangle mesh). Due to perspective distortion individual vertices of an object move closer

together on the screen as the distance to the observer increases until they finally fall into one pixel. By creating a cluster of such close vertices and replacing all cluster members by a representative vertex, the number of vertices is reduced. The set of triangles is modified to include only the vertices in the new set. In the course of that process, triangles will degenerate to lines or points and can be removed. Therefore the set of triangles is reduced as well, and any such intermediate data set can be used as an individual LOD.

Several selection criteria have been presented to choose the vertices that are to be clustered. Rossignac and Borrel [64] propose a simple, yet efficient uniform quantization in 3-D. Schaufler and Stürzlinger [66] use a hierarchical clustering method.

Reconstruction algorithms do not try to simplify the original object, but rather aim to build a new object from scratch that is gradually refined to better approximate the original object.

DeHaemer and Zyda [18] combine two approaches for approximation of quadrilateral meshes topologically equivalent to regular grids. One approach tries to fit polygons to the original mesh by recursively subdividing them. The other approach starts with a polygon of the original mesh and tries to grow it by merging it with its neighbors until a fitting threshold is exceeded.

He et al. [34] propose to sample and low-pass filter the object into multi-resolution volume buffers and apply the marching cubes algorithm to obtain a triangular mesh. This method is very robust and has the advantage that it can also deal with non-polygonal (e.g. CSG) input models. However, the resulting meshes are still over-triangulated as a result of the marching cubes algorithm.

Progressive representations take the idea of reconstruction algorithms a step further by representing the original object by a series of approximations that allow a near-continuous reconstruction and can be encoded incrementally in a very compact way.

Lounsbery et al. [44] transform polygonal objects a multi-resolution data set of wavelet coefficients derived from a triangular mesh with subdivision connectivity. Levels of detail can easily be constructed by omitting higher order detail coefficients in the reconstruction process. [19] presents a method to transform an arbitrary mesh into an equivalent one with the required subdivision connectivity. This work is taken further in [10] to include colored meshes and support progressive reconstruction of the model.

The progressive meshes introduced by Hoppe [37], based on edge collapse operation, yields a lossless, continuous-resolution representation for triangular meshes. The representation is generated as a sequence of repeated edge collapses, and is simply inverted in the progressive reconstruction process. The order of applied operation is determined by adopting the mesh simplification method from [36]. A similar approach that is also based on edge collapse operations is presented in [63].

Terrain

A specialized problem worth discussing is the representation and simplification of terrain. Digital terrain is generally represented using an elevation model or height field of sample points, effectively a two-dimensional discrete function. Often the sample points are arranged in a regular grid. Such data is easily obtained from sources such as satellite range images and exhibits excessive detail. For speedy rendering, a triangulation of the sample points with a low polygon count must be obtained.

Such triangulation schemes can roughly be categorized into regular subdivisions and triangular irregular network (TIN) models. For a survey, see [17]. Interactive rendering is best achieved by using multi-resolution subdivisions, so that levels of detail can be selected at run-time individually for different regions of the terrain. [20] outlines how real-time management of multi-level terrain data can be achieved. Recent work by Lindstrom et al. [43] proposes a scheme for computing continuous levels of detail for a regular subdivision height field that can be computed incrementally at run-time and supports a user specified error threshold.

2.6 Image based rendering

The relatively new field of *image based rendering* tries to take advantage of the observation that while the complexity of geometry in a scene is potentially unbound, the complexity of images (of a given resolution) is finite and can easily be estimated in advance for guaranteed rendering performance. Display algorithms typically require modest computational effort and are apt for low-cost and entertainment devices. Furthermore, the source of images can be computer models or digitized photographs, with the option of mixing the two together.

The most established technique that falls in that area is *texture mapping*, which simulates detail by mapping images (often defined using bitmaps) onto flat surfaces. Partially transparent textures can be used to simulate geometry with complex outlines. The widely available hardware-support makes texture mapping the most popular choice for visually rich virtual environments. However, the disadvantage of artifacts stemming in the finite resolution of texture maps cannot fully be overcome by sampling strategies such as mip-mapping [21]. Texture mapping is complemented by *environment mapping*, to capture the light entering a scene from outside in a special texture map, a technique which is also available in hardware now.

A relative straight extension of texture mapping for the purposes of virtual environments are *billboards* [61]. Radially or spherically symmetric objects such as trees can be approximated by a single texture-mapped polygon, which is always oriented to face the observer.

Maciel and Shirley [48] introduced the concept of an impostor: An image of an object in the approximate direction of the observer is presented in place of the object itself by rendering it as a texture map onto single polygon. Schaufler extended this concept to the dynamic generation of impostors at run-time [67], rather than as a preprocessing step: The impostor is generated by finding a screen-aligned rectangle surrounding the object and rendering the object into a corresponding rectangular frame buffer using graphics hardware. The resulting image is read from this buffer and used to define the texture on the impostor rectangle.

Subsequently, Schaufler and Stürzlinger [69] and Shade et al. [75] concurrently developed a hierarchical image cache that uses the concept of impostors to accelerate the rendering of very large polygonal scenes up to an order of magnitude: The scene is decomposed into cells by a hierarchical spatial data structure such as an octree or a BSP tree. This data structure is traversed depending on the projected size of the cells, and a cache of impostors for each node is created and updated as required by an error metric on the validity of the impostor.

The first proposal to build hardware that supports this idea is Talisman [86]. Aimed at the low-end image generator market, this architecture discards the concept of a frame buffer in favor of small image layers that at are composed on the fly at full rendering speed. During the composition process, a full affine transformation is applied to the layers to allow translation, rotation and scaling to simulate 3-D motion. Temporal image coherence is exploiting by re-using the image layers in a way similar to impostors.

Image based rendering for rendering polygonal scenes as outlined above are probably the most sophisticated acceleration tools suitable for the class of scenes categorized as sparse earlier. Beyond polygonal scenes, some work has recently been published that attempts purely image based rendering, so that the notion of geometric complexity is completely abandoned.

Chen and Williams [11] proposed to synthesize a dynamic view of the environment from a set of environment maps that are composed by image warps. Regan and Post [60] developed a hardware featuring multiple frame buffers that are combined by evaluating depth values. Re-rendering of objects can be delayed until the an object's view becomes too erroneous.

Quicktime VR [12] is an attempt to use cylindrical or spherical image maps that are warped in real time to simulate camera panning and zooming. The method works very efficiently on low-performance platforms and is now successfully used in entertainment products. A similar approach based on plenoptic modeling is presented in [51].

A new approach to model the appearance of objects without the use of explicit geometry was simultaneously introduced by Levoy and Hanrahan [42] as the light field and Gortler et al. [30] as the lumigraph. The object's appearance is represented a 4-D function, which is a subset of the plenoptic function describing the flow of light from all directions in all directions. This function is sampled to synthesize an image from any given viewpoint.

2.7 Managing latency

All effort to reduce rendering complexity to fit the maximum capacity of the image generator at the target frame rate can easily be defeated by a suboptimal utilization of the hardware. In particular, the rendering process is constantly facing deadlines in the form of refresh cycles for the display device. If the rendering is not completed before the scan-out of the new frame starts, a whole frame's time is lost, so it is essential to complete rendering timely in order to keep the graphics system occupied.

One way to achieve this is to make use of both host CPU and graphics processor. A typical setup of the pipeline for the CPU includes an application task (simulation, modification of the scene graph), a database task (scene graph traversal, visibility culling, level of detail selection) and a draw task (transferring data to the graphics subsystem). As with any pipeline, these tasks should take even time slices to achieve a maximum utilization, and should furthermore be tuned to meet the refresh cycle deadlines. This goal can be defeated by dynamic shifts in the load of individual stages. However, if any stage always picks up the latest available instance of the pipelines data, the pipeline is at least kept from stalling.

One limiting factor in this setup are the fixed time slices enforced by the refresh cycle of the display. This factor was criticized by Bishop et al. in [4]. They proposed frameless rendering, where individual random pixel are updated rather than frames, and always use the latest viewpoint information, which is particularly suitable for immersive systems employing head-tracking. However, current image generators do not support this approach. Mazuryk, Schmalstieg and Gervautz [50] proposed a simple scheme that trades traditional double buffering in favor of a copy/zoom operation supported by 2-D bit block transfer hardware, which is a standard component of today's image generators. New frames can be presented independently of the refresh cycle, and the zoom operation can be used to compensate pixel-dominated rendering overload for low-cost image generators. This approach can be combined with 2-D image deflection to reduce dynamic viewing errors in head-tracked displays. This approach was further refined by Schaufler, Mazuryk and Schmalstieg in [68] by replacing 2-D image deflection with the more capable 3-D image deflection on the basis of dynamic impostors.

2.8 Managing large geometric databases

For very large scenes, interactive rendering also involves a number of issues in database management, for the sheer size of the involved data sets. Conservative use of main memory can enable handling of larger geometric databases. While memory is gradually moving away from being the most limiting factor in large-scale applications, this constraint is readily replaced by the limited bandwidth of network connections such as the Internet, that are essential in distributed virtual environments. Slow network transmission of large geometric data sets interferes with the responsiveness requirements of interactive 3-D applications.

An important aspect is the paging if geometry and texture from the secondary storage into memory. An implementation should take care of:

- Achieving full I/O bandwidth by arranging the data so it can be fetched from the secondary storage in large chunks rather than small items.
- Minimizing impact on frame rate by either dividing the I/O task over multiple frames, or running it as an asynchronous process that does not interfere with the main rendering task.
- Accurate prediction and timing to avoid situations where the required data is unavailable, and rendering must be stalled.

Funkhouser, Sequin and Teller [22] present an application capable of presenting an interactive walk-through of a database much larger than memory. The data is pre-loaded by predicting the users movements and reducing the required data based on visibility considerations for building interiors.

For networks, progressive level of detail representations as discussed above allow to make instant use of partially transmitted data. A related topic is geometry compression: Data sets can be transmitted in a compressed form, then expanded at the receiver for more rapid rendering. The first step in this direction was made by Deering [16], who proposed a compression scheme for triangular meshes based on generalized triangle strips, including normals and colors. Taubin and Rossignac [84] introduce a procedure they call topological surgery, which transforms a triangular mesh into an alternate representation based on spanning trees for triangle strips and vertices, that can be encoded

in an extremely compact form and yields high compression rates. Levoy [41] proposed to combine JPEG compression of animated sequences with simple polygonal rendering data to yield higher compression ratios for digital movies.

2.9 Summary

Interactive 3-D graphics is a rapidly evolving field inside computer graphics. While traditionally, computer graphics has focused on the quality of images alone, the real-time requirements of interactive applications make it necessary to employ all sorts of trade-offs to maintain frame rates and satisfy ergonomic needs. Techniques such as visibility processing, levels of detail, image based rendering, and the management of latency and large geometric databases make it necessary to adopt methods not only from computer graphics, but also from real-time systems, databases, networking and many other domains. Despite the many shortcomings that current implementations have, the importance of interactive 3-D graphics is certain to grow at an enormous pace.

3. Distributed Virtual Environments

3.1 Introduction

Traditional symbolic user interfaces devices limit the amount of information exchange between user and machine. Since we interact with the real world through highly developed skills such as our visual system, providing an interface that uses human skills - rather than relying on artificially created interaction techniques that suit computers better than the humans - has the potential of dramatically increasing the usability of the medium computer.

Advances in cost and performance of certain key technologies such as graphics accelerators and networks have established virtual environments as an increasingly popular new medium. As can be seen from Figure 5, there is much more to virtual environments than fast graphics: Components such as simulation, interaction or animation each involve a large body of knowledge of their own. The software engineering effort of assembling these components into a working whole makes virtual environment system architectures a fruitful area of investigation.

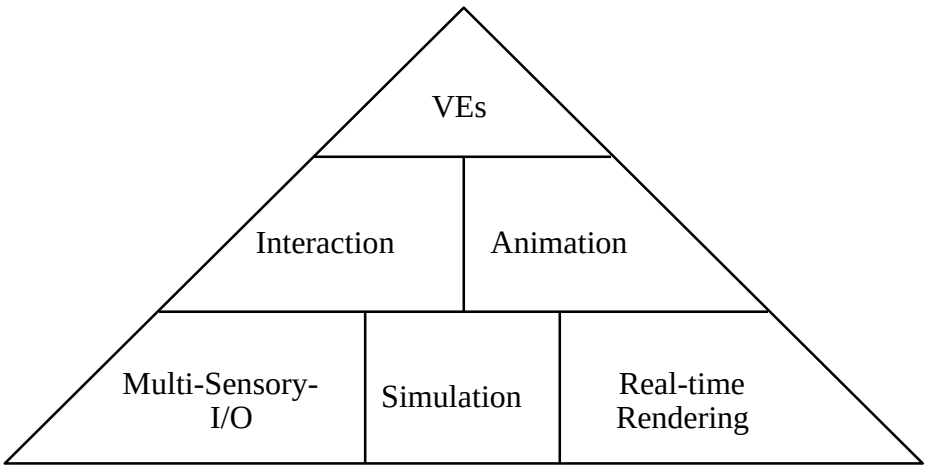


Figure 5: Virtual environment building blocks

If we are building virtual environments, we have to consider several important design issues:

- *Applications*: To provide for concurrent access to a diverse range of activities, virtual environments should allow multiple, concurrent applications. Applications should rather be considered tools that are available in the virtual world, and it should be possible to bring new tools into the environment.

- *Worlds*: Different task should be supported in the virtual environment, and clearly an environment cannot be optimal for very diverse needs (e.g. an adventure game and an architectural design application). Consequently, multiple, concurrently available virtual worlds should be supported, although this certainly complicates the design. If both multiple applications and worlds are possible, the assignment of features to applications and/or worlds becomes an issue of itself. In general, one can state that more flexibility is better.
- *Users*: A question must be found to the issue how the user is represented in the virtual environment. The visual representation of the user in the world as seen by other concurrent users is often called *avatar*. Snowdon and West [81] state that there are no essential, logical differences between users and applications, since both are external to the system itself, but have an active manifestation within the virtual world.

We can conclude that the execution of a virtual environment must support the simulation of a large number of independently executing units (avatars, agents, tools, etc.), so an object-oriented database is a natural choice. Because of their active behavior, we call the entities in this database *actors*. Managing a database of actors can be very different to maintaining a traditional database with predominately reactive objects.

3.2 Fundamentals of Distributed Virtual Environments

In this chapter, we focus on *distributed virtual environments*, since the contribution of this thesis falls into that area. A distributed virtual environment is one that executes concurrently on multiple computers (nodes) connected over a network. While the integration of the necessary ingredients for a virtual environment is by itself a challenge, adding the aspect of distribution and concurrency obviously further complicates the task. However, good solutions require distribution for two reasons:

1. Reaching the necessary performance is usually impossible without the use of concurrently executing units. Should a single unit of hardware (e.g. a CPU) perform inadequately, more hardware can be devoted to the task. Probably more important still is the notion of concurrently and independently executing units that function as an ensemble [26]. In the context of virtual environments, this has been called the *decoupled simulation model* [76], It allows independent execution of communicating tasks such as simulation and rendering. A good example of such an approach is the Performer library [61] that distributes execution to multiple tasks and occupies multiple CPUs if available.
2. While distributed execution is optional for single user systems, it becomes mandatory as soon as multiple users are to be supported. Each user has to use his or her own console and I/O hardware, so the distribution is inherent. Multi-user applications have a much larger potential ranging from games to computer supported cooperative work, and also better fit the idea of virtual environments as a new electronic medium: People communicate with each other via a virtual environment.

Network properties

The actors in the simulation must communicate to carry on the simulation. Problems arise when two communicating actors are located at separate nodes on the network executing the distributed simulation. The network is intrinsically slow and the cause of severe problems: *Latency* introduced by sending messages over the network is at conflict with the need for concurrent execution. The *limited and unpredictable bandwidth* places limits on the number of messages that can be exchanged and affects the scalability in the size of the participating network nodes and actors. Most of the work discussed in this chapter is concerned with the optimization of the network aspect of virtual environments.

Furthermore, in any distributed system there is often the need to have replicated information. *Consistency* of these items must be addressed despite the shortcomings of the network connection.

Persistence of the actors in the simulation has been neglected in early virtual environments. Still most systems are initialized from a fixed configuration, and there is no means to have an environment that continues to exist and evolve over time (much like today's networked information systems like the World Wide Web).

Little effort has been made to achieve *geometric or geographic continuity*. If multiple virtual environments coexist in the same network, it either impossible to migrate between the environments or the only option are portals that „beam" the user to the destination such as in VRML [56] or DOOM [38]. Few systems support geometric continuity, most notably RING [24] and NPSNET [47].

Network data characteristics

It is important to understand the characteristics of the data that is transmitted over the network. We distinguish two varieties with fundamentally different properties:

Simulation data appears in small units (typically a few 10 or 100 bytes), but in large numbers. An example are position updates that are sent to remote nodes to inform other users that an actor has moved. A simple protocol would only require an actor identifier plus a tuple of coordinate values per packet, but if N actors move through a shared virtual environment simultaneously, and simulation fidelity requires M updates per second, then N*M packets are generated per second.

Model data consists of the geometric model of the actor (e.g. a polygon mesh), and a description of the actors behavior. The latter is optional: some virtual environments do not support active actors other than the users' avatars; other virtual environments may only know a fixed set of actors whose behavior is built into the application, so an identifier of the actor's class suffices to determine its behavior. More powerful virtual environments allow to formulate arbitrary behavior using a scripting language, for example the behavior of actors in VRML 2.0 [3] can be written in Java [39]. Both geometry and scripts can take up considerable space. The transmission characteristics for model data is different from simulation data: The actor takes substantial time to transmit even on fast networks, but transmission is relatively infrequent compared to simulation updates as the model data is used for longer periods of time. Simple systems even require that all model data is available before the virtual

environment starts executing, so model data never gets transmitted under time-critical conditions (for example, in DOOM [38] and earlier instances of SIMNET [8].

Different strategies give best results for these two types of data, but the constrained nature of the network dictates one objective: „Avoid communication that is not strictly necessary.“ [81]. Work that attempts to meet this goal centers around the issues of replication and network topology, which are discussed in the following section.

3.3 Network design

Replication means maintaining local copies of remote actors. The requirement to render what is currently visible from the user's viewpoint makes it necessary to replicate at least those actors contained in the viewing frustum. The need for a local copy of any currently visible actor also places the fundamental limit on scalability of the virtual environment: Each node must have the capacity to store the actors visible to the user and keep them up to date. If local density exceeds these capacities, the simulation goals cannot be met [25]. Fortunately, environments that involve a very large actor population typically also simulate a vast space, so for most applications this never happens [47].

Network topology

Otherwise, scalability is mostly affected by the network topology. In the following, we give a short review of the relevant approaches:

- *Unicast*. Reality Built For Two [5], VEOS [7], and MR Toolkit [77] are based on unicast peer-to-peer designs. A separate unicast message is sent to all other nodes for every state change. The $O(N^2)$ complexity of message exchange quickly saturates the network, and a large number of nodes is impossible.

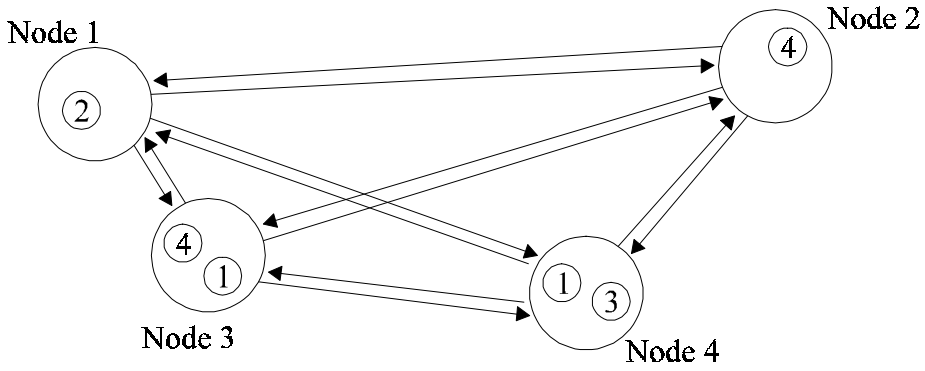


Figure 6: Peer-to-peer unicast network topology. In this example, large circles represent network nodes, each one occupied by one user. The small circles with numbers represent the avatars of other users (nodes) that must be simulated and displayed locally. Arrows indicate the flow of update information according to the individual needs for local simulation.

- *Broadcast*. SIMNET [8] and VERN [6] use broadcast messages to send updates to all other nodes at once. In that way, only O(N) messages are sent over the network, but every local node must constantly process all messages from all other nodes, so the size of the environment is severely limited by the capabilities of the least powerful node.

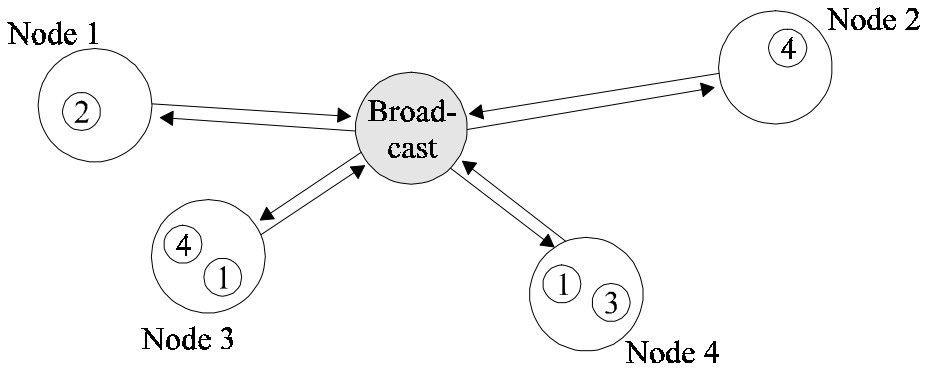


Figure 7: Peer-to-peer broadcast network topology

- *Multicast*. NPSNET [47] and DIVE [9] employ multicast to send update messages to a subset of participating workstations. The general idea is to map entity properties to multicast groups, and send entity updates only to relevant groups. For example, NPSNET partitions the world into 2D hexagonal cells each of which is represented by a multicast group. Actors send updates only to the multicast group representing the cell in which they are located, and listen only to multicast groups representing cells that can be seen from the current viewpoint.

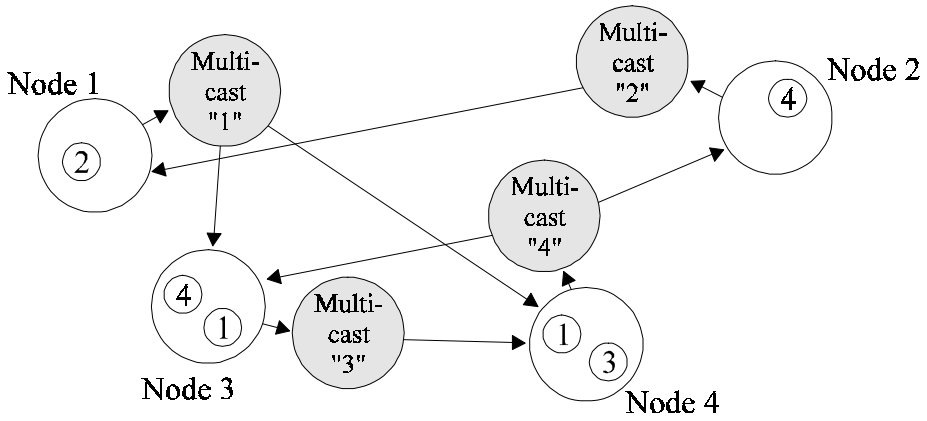


Figure 8: Peer-to-peer multicast network topology (assuming that a separate multicast group exists for each node)

- *Client-Server*. RING [24], BrickNet [78], DVS [29] and WAVES [40] are client-server systems. Users invoke clients connected to message servers. Clients do not send messages directly to other clients, but instead send them to servers that forward them to other clients and servers participating in the same distributed simulation. A key feature in client-server design is that servers can process messages before propagating them to other clients, culling, augmenting, or altering the messages.

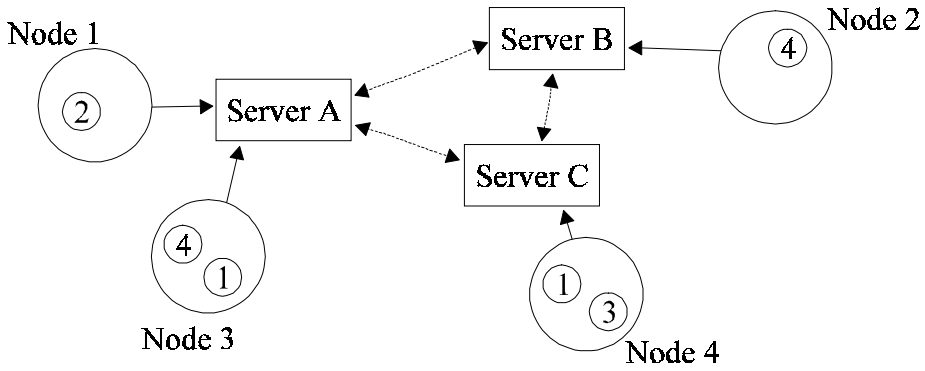


Figure 9: Client-server network topology

Discussion

There has been a long-lasting battle between the advocates of client-server and peer-to-peer network topologies for the purpose of distributed virtual environments. While a central server obviously makes issues such as consistency much easier, it also easily becomes a bottleneck. However, only a server enables persistence. For example, the geometric description of actors must come from somewhere in the first place, and assuming a priori distribution is obviously not a viable option. Even systems that claim to be strictly peer-based have hidden server components (e. g., NPSNET, that refer to Internet servers to download model data [47]). Furthermore, strictly peer-based systems cannot guarantee a continuous evolution of a virtual world: when the last participating user leaves the virtual environment, it ceases to exist.

While peer-based design avoid the potential bottleneck of a central resource, naive networking schemes such as broadcasting fail because they overload the network. This inefficiency can be partially overcome by the use of multicast, but there are problems that remain: Not all networks (e.g. modem connections) support multicast. Besides, there is a trade-off in the choice of the size of multicast groups that affects scalability and performance: Large multicast groups mean that too many actors are associated with the multicast group, and the involved overhead in message processing by the nodes becomes intractable. Small multicast groups mean that actors often change the relevant multicast group by crossing the borders of the associated region, and the overhead of joining and leaving multicast groups becomes intractable.

The best compromise appears to have a hybrid system design like the one proposed by Funkhouser [25]: Users invoke a client to connect to a server that manages the region in which the user resides. The servers communicate in a peer-style via multicast. As servers do not move, multicast membership does not change. Client-server communication can be connection-less using datagrams, so that users frequently changing servers does not affect performance. The only remaining problem is that of local density of actors for which a sort of load balancing would be necessary [81]. Our own virtual environment that serves as a testbed for the Remote Rendering Pipeline is based on such a hybrid design [70].

3.4 Local simulation

The frequency of exchanging simulation updates can be further reduced by local simulation of actors in order to lower the load on the network. If the behavior of a remote actor is known or partially known, it can be computed, and simulation updates from the remote node can be delayed until the actor's real state deviates from them simulated state. The interval of the updates can be extended by allowing a small difference in the state of local and remote actor.

This idea has been successfully used in the simulation of vehicles in NPSNET [46]: Dead reckoning means extrapolation of a vehicle's position according to current position, speed, and acceleration. Only if a vehicles position (controlled by a human or artificial driver) deviates significantly from the simulated extrapolation, an update is sent. The authors report a large reduction in network utilization. The idea has been further improved by Singhal and Sheridon [79], who use a more sophisticated extrapolation scheme based on position history, and apply their protocol to actors other than vehicles.

Roehl [61] argues that the idea of local simulation should be extended beyond simulation of position. The problem here is to identify behaviors that are easily formalized into simple algorithm apt to local simulation, and that apply to a large number of tasks so that their use is beneficial. We believe that this will be an important field for future work.

3.5 Summary

We have elaborated on the design and implementation of distributed virtual environments. Virtual environments are complex software systems that require a flexible design to support applications, worlds, and users. These requirements - in particular support for multiple users - lead to the implementation of a virtual environment as a distributed system. Issues that must be addressed include persistence, consistency, continuity, performance and scalability. A discussion of popular system designs and network topologies is given, along with the most relevant techniques for improving performance and optimal utilization of the network.

4. An Overview of the Remote Rendering Pipeline

4.1 Introduction: What is Remote Rendering?

Networked multi-user virtual environments require that users share a common scene over a network [29, 47, 70]. Examples include networked walkthroughs of large information spaces (buildings, databases, ultimately a 3-D Internet?) and interactive applications such as immersive cinema, networked games and computer supported cooperative work.

A virtual environment requires efficient rendering of the three-dimensional objects forming the simulated world. In a distributed virtual environment, the work is divided between processes. One process will maintain the actor database and run the simulation, whereas another is responsible for rendering. These processes will often run on separate CPUs or workstations, which creates the need for *Remote Rendering*. In multi-user systems, this is always required, independent of network topology.

Given today's typical hardware setup with high-speed CPUs, fast system buses and comparatively slow network transmission, it is very reasonable to assume that the network is the most constrained resource of the whole system. We therefore have to develop new strategies for the visualization of distributed geometry databases, with the overall goal of minimizing bandwidth consumption on the network, and we can afford to devote substantial computational resources to the task.

Three distinct models for distributed graphics are in use today:

1. Image-based: Rendering is performed by the sender, and the resulting stream of pixels is sent over the net (e.g. digital TV, X pixmaps).
2. Immediate-mode drawing: The low-level drawing commands used by drawing APIs are issued by the application performing the rendering, but not immediately executed, but sent over the network as a kind of remote procedure call. The actual rendering is then performed by the remote CPU (e.g. distributed GL [54], PEX in immediate mode [65]).
3. Geometry replication: A copy of the geometric database is stored locally for access by the rendering process. The database can either be available before application start (kept on local harddisk, such as seen in computer games like DOOM [38] and networked simulations such as NPSNET [46]), or downloaded just before usage, such as current VRML browsers do [33].

Variations of geometry replications are now commonly used for networked VR applications. However, several severe problems constrain the usability of the method: Low network throughput and large database sizes are responsible for long download times. As the data has to be shipped to the user at some point, this problem is always present. Making the user wait for more than a couple

of seconds destroys immersion and makes many interactive applications completely useless. Furthermore, extended waiting periods mean that a download process cannot be invoked frequently, so exploratory behavior of 3-D data spaces becomes impossible. This prohibits the exploration of large, continuous virtual worlds.

4.2 Geometry management with the Remote Rendering Pipeline

As an improved conceptual model for Remote Rendering, we introduce the *Remote Rendering Pipeline*. A rendering pipeline describes the way that geometric data takes from modeling to the final image. We generalize this idea to include additional stages required by a distributed system (Figure 10).

In this task model, the local data held in the main memory of the remote site doing the actual rendering becomes a geometry cache, while the full geometry database is held at another site, the master site. In the following, we discuss the components of the Remote Rendering Pipeline in more detail.

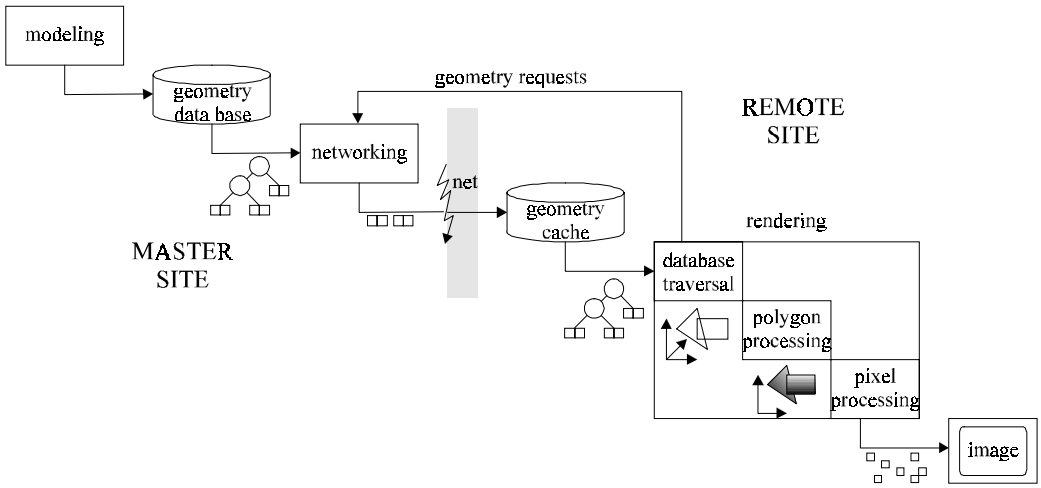


Figure 10: The stages of the Remote Rendering Pipeline

Modeling stage: This stage is performed off-line and strictly speaking not part of the pipeline executed at run-time. It has been included to provide a more complete picture of the process. The output of the modeling stage are the geometric models whose images are finally displayed at the end of the pipeline. Note that the modeling stage involves more than just creating the artistic input for the virtual environment. For example, levels of detail must be precomputed to be able to perform level of detail rendering in later stages.

*Geometry database***:** The geometry database is the collection of the geometric models of all actors in the virtual environment. This database is maintained at the master site, and is usually never fully passed on to later stages of the pipeline. Therefore it can be potentially very large, unaffected by the capacities at the remote site.

Networking stage. The networking stage is a process executed at the master site. Its job is to transmit the required data to the remote site on demand. Note that there is a loop back from the

remote site for the purpose of issuing request for particular pieces of geometric data as they become necessary. Because of this loop, the model is not strictly a pipeline, but these requests are just steering information negligible in size compared to the geometry data stream moving downstream.

Geometry cache. The geometry cache holds local copies of geometric data from the geometry database for immediate rendering. The goal is to keep its content at all times equivalent to the portion of the virtual environment visible to the observer. All the techniques described in this thesis aim at improving performance so that this goal can always be met.

*Rendering stage***:** This last stage is equivalent to the conventional rendering pipeline described in chapter 2. It starts with a rendering preprocess stage, sometimes also called the database traversal stage. During this traversal, geometric primitives that must be rendered are sent to polygon processing stage, followed by the pixel stage that generates the final image. Other important tasks performed in rendering preprocess are visibility culling and level of detail selection. During the rendering preprocess, it is also checked whether the content of the geometry cache may lack visible items or items that are likely to become visible in the near future, and request for these items are issued. Care is taken that the right requests are made, so that necessary items are always available timely, but usually no useless requests place stress on the network.

4.3 Exploiting task-related knowledge for optimization

A fundamental idea of this thesis was to optimize the stages of the Remote Rendering Pipeline by exploiting all sorts of knowledge about the virtual environment and the relevant tasks. Loss of information generally makes tasks less efficient, either because the information must be reconstructed or approximated, or because algorithms must be brute-force compared to what would be possible using more information. More sophisticated solutions can be found by identifying unused information in the task. A good example would be the use of coherence in computer graphics [32].

The contribution made in this thesis consist of the theoretical concept of the Remote Rendering Pipeline, and of practical measures to implement it. The latter can be assigned to three layers of increasingly high level information:

1. *Data layer*. This is the simplest layer on which we operate. We assume that the data we operate comes in the form of polygonal models. This fact is used to compute levels of detail in two varieties: standard (discrete) levels of detail (chapter 6), and smooth levels of detail (chapter 7). The data is then transferred partially, and in case of smooth levels of detail, also incrementally over the network, so that higher degree of parallelism between rendering and networking is realized, and the pipelined architecture is better exploited.
2. *Environment layer*. We make use of the nature of interaction of multiple users in a large virtual environments by introducing the demand-driven geometry transmission protocol (chapter 5), to make efficient use of the geometry cache introduced in the previous section.
3. *Modeling layer*. When modeling a geometric object, the designer uses a lot of information that is not represented in the resulting model. We try to preserve some of this information for a special class of objects with fractal structure by modeling and rendering them as directed cyclic graphs

rather than as collection of primitives (chapter 8). While this is certainly a special case, the natural phenomena that can be modeled in that way are useful for many outdoor environments.

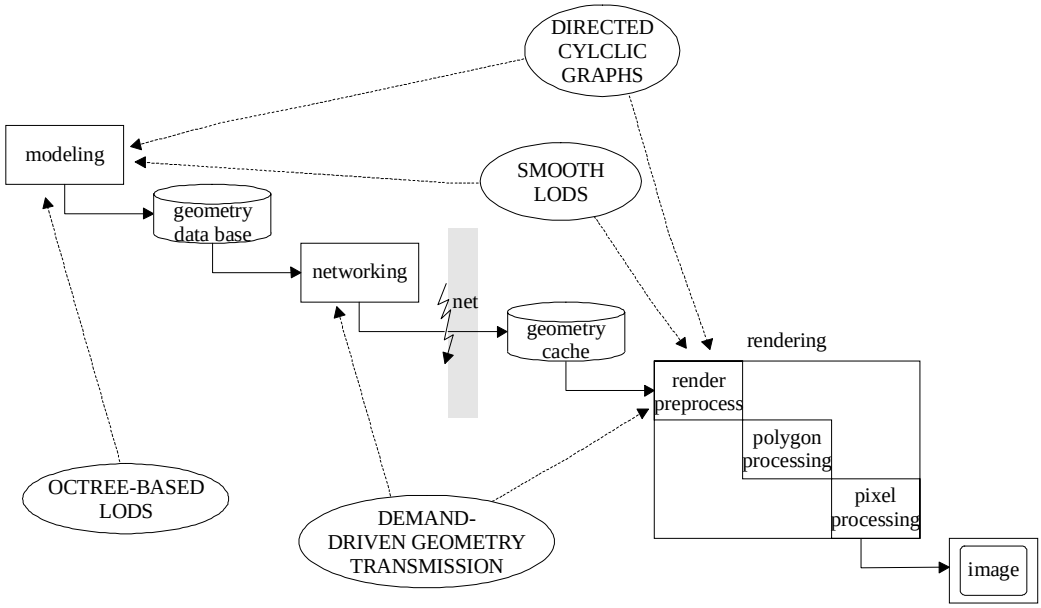


Figure 11: Relation of the projects to the stages of the pipeline

Figure 11 shows the relation of the contributions outlined in this thesis to the stages of the Remote Rendering Pipeline. The next four chapters give the details of the research work that was carried out for the purpose of this thesis.

5. Demand-Driven Geometry Transmission

5.1 Introduction

For the purpose of demand-driven geometry transmission, we attempt to optimize transmission of geometry data in a client-server system. As detailed in chapter 4, the transmission of geometry data for a scene as a whole has several drawbacks. We therefore aim at the development of a method for more fine-grained network transmission, that works incrementally. If the required data is delivered over the network „just in time“ for the rendering process, both intractable setup times and elevated storage requirements can be significantly reduced.

We also consider the typical use of levels of detail for the rendering of objects. As only one level of detail of a given object can be displayed at any time, we can further improve performance by considering single levels of detail as the unit of transmission instead of complete objects.

5.2 Data management

In our system, the server stores the data for a 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 (Figure 12). 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). We have decided to use spherical AOIs rather than the viewport itself, because rapid head movement as possible with head-mounted displays cannot change the set of objects in the AOI so rapidly.

Thus by restricting the geometry transmission to the data that is actually required for display, we can gain significant savings in network bandwidth and local memory requirements, allowing to handle more complex, more interesting data sets. Note that the visible data set is dependent on the viewpoint of the observer, which changes over time, and on it the visible data set. If the system is able to deliver the data just in time for display, there are no visible differences over a non-distributed virtual environment that has all data available locally.

The set of objects contained in the AOI changes as either objects or the client itself move. To keep the area of interest up to date regarding the objects contained within, the client can request data from the server. The selection of the data is up to the client, so various strategies for data management are possible. To perform the task of requesting data, the client has to know about the objects that are

contained within its AOI. Again, we do not want to require the client to know about all objects in the environment, for tracking the objects in the AOI is sufficient.

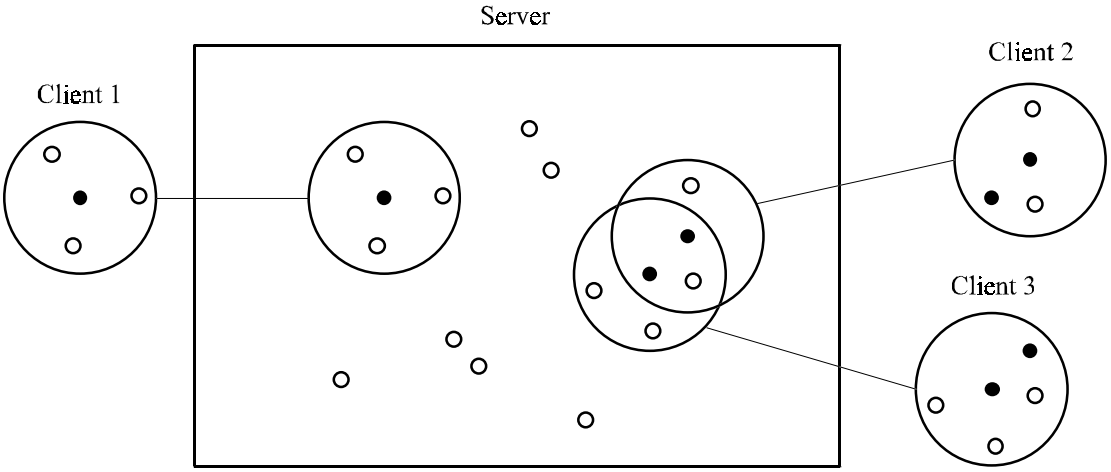


Figure 12: A distributed geometry database: A server stores geographically dispersed objects (small white circles). Each client's view is limited to an AOI (large circles). If AOIs overlap, clients can see each other (small black circles)

The server monitors the AOI for every connected client and periodically sends updates regarding the activity of contained objects as appropriate. If an object moves into the AOI or the user moves the AOI near an object, the object has to be newly introduced to the client (send object info), that cannot know about this object otherwise. From then on, it is sufficient to send updates if the position of the objects changes. Figure 13 shows some cases.

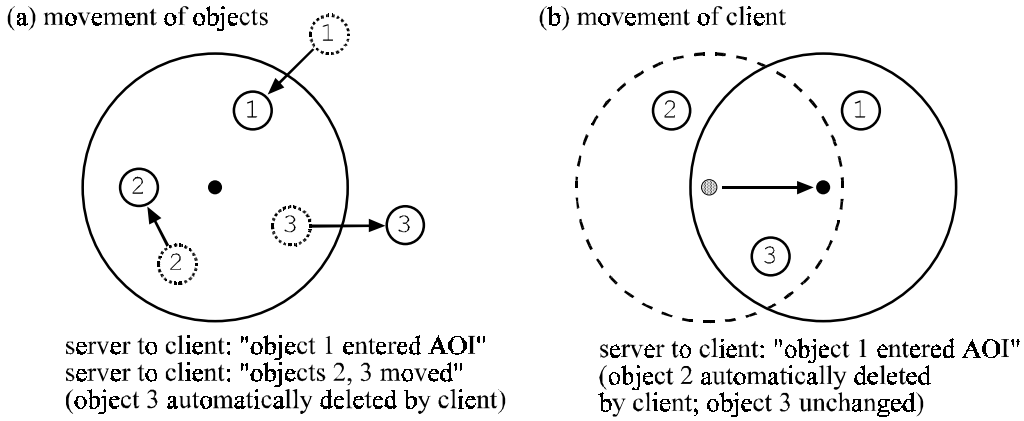


Figure 13: The server updates the client on activity within its AOI, if objects (a) or the client (b) move. To reduce network traffic, only those messages are sent that cannot be deduced by the client independently.

This scheme requires the server to remember the set of object infos that have been transmitted to the client. The task is not too complicated for the server because the set changes incrementally.

The distributed nature of the rendering process should be transparent to the user. In particular, the image presented to the user should be smoothly animated and updated at a sufficiently high and constant frame rate. This goal is defeated by variations in scene complexity (many objects, complex objects) and in network throughput. Rendering complexity is managed by a rendering engine capable

of displaying LODs. With the help of the LOD datastructure, a strategy was developed that also compensates for the shortcomings of network transmission.

The fundamental idea is to consider these LODs instead of complete object as the unit of network transmission. Objects can be displayed even if not all of their (LOD-)data is available. Only those LODs that are needed for rendering must be available locally at the client. As the user moves his viewpoint, or simulated objects (e.g. vehicles) change their position, the selection of LODs changes.

Transmission of even a coarse LOD takes time, and this delay must be compensated for, or the data will not be available when needed. Therefore *prefetching* is adopted. When the LOD selection algorithm decides to switch to a certain LOD, the prefetching module requests the next finer level.

In cases when prefetching fails, an available coarser LOD can be displayed instead (*graceful degradation*), trading a continued constant frame rate for a decrease in image fidelity. Such a degraded image can be progressively improved in phases of reduced activity by downloading the missing LODs (*progressive refinement*).

The time-critical part of acquiring the data is most important, but storage requirements may not be neglected. To keep the size of the client's database *cache* (i.e. memory available for geometry data) from overflow, we must dispose unneeded objects. If an object leaves the AOI, it is deleted and memory is freed.

For a particular client, the representations of other users (avatars) appear just like ordinary objects. If a client changes position, it has to transmit the new position to the server, which not only has to know the client's current position to monitor the client's AOI, but also has to update the other clients on the new position.

A more detailed discussion of the client's strategy is given in section 5.4.

5.3 Geometry Data Structure

In order to make use of the hardware support for interactive rendering, we must model the virtual environment object database as a collection of polygonal datasets. VR applications have two additional important requirements:

- Interactive rendering of large scenes in real time requires that the objects of which the scene is composed are modeled with levels of detail (LOD). At runtime, the fidelity of each object can be chosen independently from the available LODs, so that the polygon budget for a frame is not exceeded [23]. In the next section it will be shown that the LODs can also be used to optimize network usage.
- A flat datastructure (e.g., a simple array of triangles) is not sufficient. For efficient manipulation and high-level animation of a dynamic environments, a directed acyclic graph (DAG) is well suited. Its hierarchical structure allows flexible manipulation of the data as needed by VR applications [82].

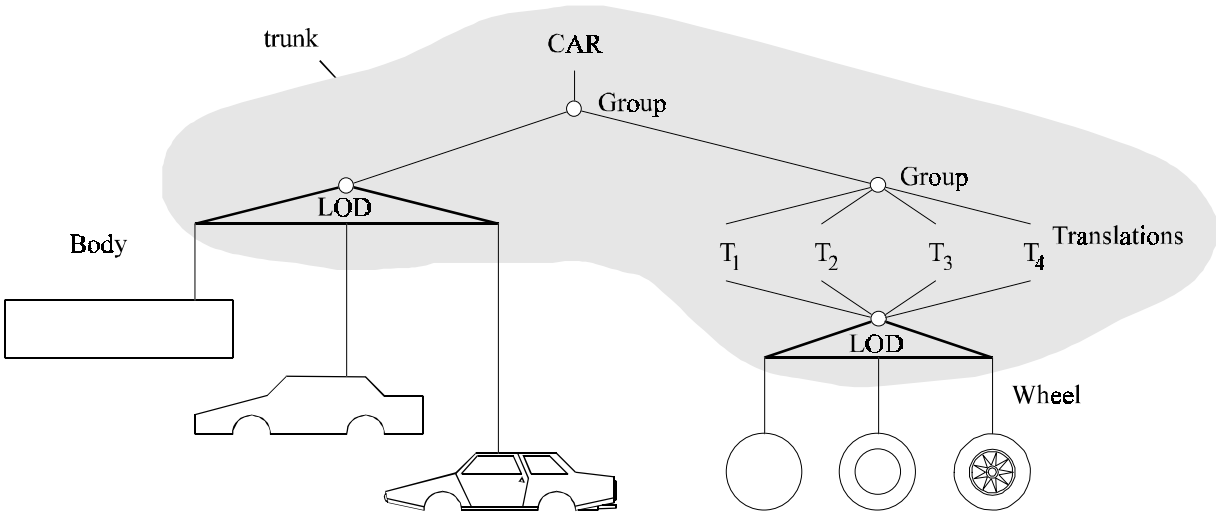


Figure 14: Car modeled with levels of detail. The model is divided in LOD geometry and a "trunk" (shown in gray). Most of the data is hidden in the LODs, while the trunk is generally only a control structure.

We divide this datastructure into two parts: the subgraphs below a level-of-detail node (LODs) and the trunk (i.e. the graph from the root down to and including all LOD nodes, but not below). Figure 14 shows an example. For network transmission of such geometry graphs, the separation allows to transmit the trunk before the LODs, and then "mount" the LODs on the trunk. Note that not all LODs have to be available in order to display the object. The trunk plus a single LOD is sufficient to display the object, although not in every desired fidelity. Note that in most cases, the trunk will only be a very small data structure composed predominantly of "organizing" nodes such as groups and transformations, while the LODs contain the bulk of the data (polygons, color, ...).

The database of the server is a flat collection of objects. Each object is composed of a geometry representation (trunk plus LODs), and a matrix defining the object's position and orientation. In the next section we will show how a "view" (subset) of this database is kept at the client.

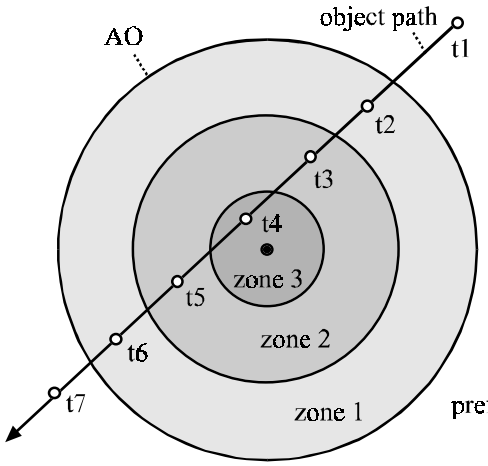
5.4 Strategy of the client

In this section, we give details on the strategy for management of geometry data, as exercised by the client.

Prefetching

To compensate the delay introduced by the network transmission, we use prefetching to anticipate the requirements of the renderer. The already available level of detail algorithm can be used for this purpose, only with a different parameter - a finer LOD is already selected "earlier" than needed for rendering (when still relatively far away). Objects that are approached will be displayed with increasing resolution, so the next finer LOD is a good guess for prefetching (Figure 15).

This scheme is not unfailable: as the strategy assumes high frame to frame coherence in the data set being displayed, violations of this assumption lead to the failure of the prefetching efforts. If either the user moves too fast, or objects move too fast, the data needed for display at the appropriate resolution cannot be made available in time.



time	request	display
t1	LOD1	-
t2	LOD2	LOD1
t3	LOD3	LOD2
t4	-	LOD3
t5	-	LOD2
t6	-	LOD1
t7	-	-

prefetching for a moving object with 3 LODs

Figure 15: A prefetching strategy. As an object moves through the AOI (see arrow), it traverses multiple zones indicating which LOD of the object is displayed. The next finer LOD is always requested one zone in advance to compensate for network delay.

Even worse are applications that allow sudden changes in the visible set: objects that appear out of nothing or change their representation into a more complex shape. The same applies to abrupt changes of the user's viewport and position, including "teleport" functions and the initialization phase when no data at all has been transmitted to the client. Furthermore, if the network itself works unexpectedly slow, it may simply not allow to transmit as much data as estimated.

Graceful degradation

If timely delivery of the required LOD data cannot be achieved for one of the reasons mentioned above, the client can use a coarser version of the object instead. For immersive applications, displaying a degraded image (even bounding boxes may suffice in some cases!) is far better than stalling display update. A coarser LOD of the object under consideration should usually be available. The most frequently used metric is distance between object and observer or projected size of the object on the screen. Both metrics change slowly for typical applications, so that the display gradually switches from coarser to finer LODs, generally giving the server enough time to transmit the next LOD. If time is not sufficient to do so, the next coarser LOD can be displayed a little longer, even though it may not have the adequate resolution according to the heuristics.

Note that graceful degradation can fail if there is a total network overload or breakdown, that makes it impossible to send even the coarsest LODs in time, but this should rarely happen.

Progressive refinement

In some situations the client may be able to store and display a scene at high resolution, but network is so slow that the data cannot be transmitted in time. To deal with this problem, we make use of the fact that as the observer is approaching an object, the objects representation is updated with consecutively finer LODs. We can therefore issue requests so that the LODs of an object are always transmitted in order from coarser to finer. (If – optimally – the data from a coarser LOD can at least partially be reused in a finer LOD, the total amount of data for an object is decreased.)

If a degraded version of an object is being displayed after some coherence-destroying activity, and the situation improves (user standing still, not a lot of movement), the time can be used to complement the missing data and gradually switch to a higher resolution of LODs for the objects in question.

However, if high activity is continuing, it may be more necessary to get coarse approximations of new objects or objects with only very coarse representation first. By executing data transmission out of order (using a priority queue), the more important data items can get expedited transmission. Priority is computed from the difference between available and desired LOD, if this value is the same for multiple pending transmission, the one with the lower level of detail is selected.

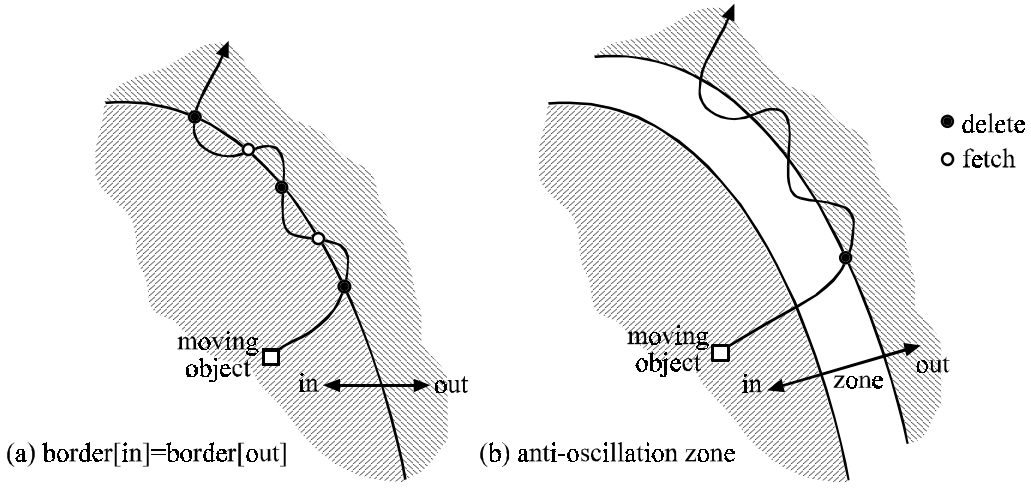


Figure 16: Suppressing oscillation. Fetching and discarding object data can lead to unwanted oscillating activity (a) that can be suppressed by a "safety" zone (b)

Cache size and replacement

For a conventional rendering LOD algorithm, the upper bound on the number of primitives that are selected is given by the maximum number of primitives that the rendering system can process. If we combine this method with networking, another restriction has to be taken into consideration: the available memory works like a cache.

Naturally, the cache size is limited, so the active data set must exceed neither the renderer's processing capacity nor the cache size. As a typical workstation has ample memory, the graphics processing power is usually the more restrictive factor. However, for video game consoles and set top boxes this may be different. With a different setup, it may be necessary to modify the strategy so that the memory is filled with more objects, but in coarser representation.

The cache replacement strategy is governed by the LOD algorithm. It determines what items of the data set are currently needed. One LOD of a particular object is a unit data item of the cache algorithm. There are two possible units that can be selected for deletion from the cache: One LOD (of one object), or one object (with all LODs of that object that are present). While replacement of individual LODs allows a more fine-grained data management, handling objects as a whole is simpler and therefore faster.

We opted for discarding objects completely if they are not needed anymore, mainly for simplicity. An alternative is to discard single LODs in the opposite order of acquisition (finest resolution – consuming most memory – first), so to keep memory footprint as small as possible.

Discarding objects when they leave the AOI may lead to unwanted oscillating effects if the object continues to move near the border of the AOI. As a countermeasure, the AOI region has to be chosen sufficiently large so that geometry is not immediately discarded when the object becomes invisible. Furthermore, the AOI used for determining leaving objects is slightly larger than for entering objects, so to increase the distance an object must travel to "come back" once it has left (Figure 16).

5.5 Protocol design

The design of the network protocol not only determines performance of the network module, but also the capabilities and semantics of the application. Our aim was to design an application-layer protocol that interoperates with the VR application.

The best way to discuss the resulting protocol is to examine the protocol units that the protocol is composed of. An overview is given in table 1. We can separate these into three groups: connection management, avatar control, and geometry transmission [71]. We can further distinguish whether the message is sent by client or server ("o" for "origin" column in Table).

Message	O.	Parameters	Comments
Connection management			
init connection	c→s	client id position orientation AOI data avatar data connect info	Building up the connection: client registers with a unique client ID; states his initial position/orientation and size of AOI; uploads user's geometric description (avatar). Other connection-management information are not of interest for this discussion.
kill connection	c→s	client id	disconnect
Avatar control			
update client pos	c→s	position orientation	client tells its new position to server, so server can compute set of obj. in client's AOI
update object pos	s→c	obj id position orientation	server updates client on new position of a moving (animated) object to allow correct selection of LODs
Geometry management			
request geometry	c→s	obj id lod no	client decides a specific LOD is needed, and requests it by specifying object ID and LOD
transmit geometry	s→c	obj id lod no data	server answers request of client for data and sends geometry data, identified by object ID and LOD
transmit objectinfo	s→c	new obj id data	server updates client on object set contained in client's AOI (without request!) by informing on a new object and associated info (number of LODs, size, ...)
kill object	s→c	obj id	delete obj. that has disappeared, e.g. destroyed
update AOI	c→s	AOI data	change AOI data (size), e.g,. if the client becomes overloaded

Table 1: Protocol units

5.6 Implementation

We have implemented a prototype of the system as outlined in the previous chapters to obtain experimental data on how the algorithm and protocol perform. The implementation was done on SGI workstations using C++ and Open Inventor. A discussion on some of the design decisions that have been made is given in this section.

Graph traversal

When a request for a particular LOD reaches the server, the referenced object's geometry graph is traversed and packaged (Figure 17): The traversal is performed in preorder/depth first (multiple referenced nodes in a DAG are only visited once).

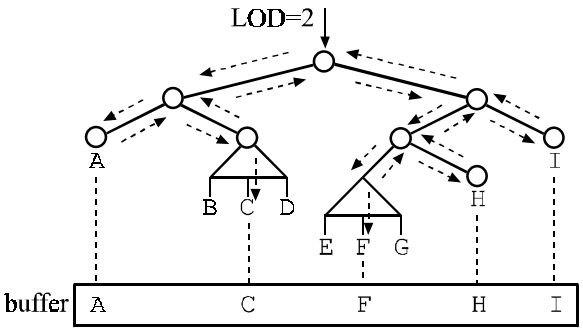


Figure 17: Traversal and packaging of a LOD request. The graph is traversed, but for every LOD node only one child is selected. The resulting data is collected and packaged for network transmission

When a LOD node is reached, the required child is determined. This child subgraph is linearized and added to a buffer. Upon completion of the traversal, the buffer contains a list of items that together make up the requested data. The content of the buffer is transmitted to the client, where it is unpacked. The client traverses the trunk of the graph (remember that the trunk is always transmitted before the first LOD), and unpacks one item of the buffer for every LOD node encountered. As the order of traversal is well-defined, the data is automatically put in the right place.

Level of detail node

The datastructure for a level of detail node must provide some measure on the subjective quality of the individual LODs to select the right LOD.

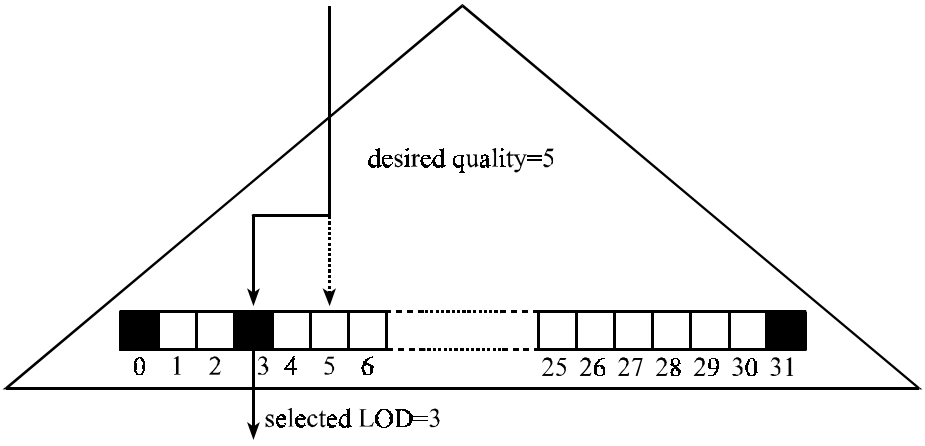


Figure 18: LOD node traversal. If a LOD with the desired quality cannot be found, the next coarser one is used.

A simple solution is to use the index of the LOD node's children correspond to quality. However, this requires that the quality "distance" between successive LODs is at least roughly the same.

We project the quality measure onto a finite and fixed length scale of (for example) 32 possible grades, which can be used as indices for accessing each of the children. Thus not all 32 entries have to accommodate a geometry subgraph For example, Figure 18 shows a LOD with children 0, 3, and 31 only.

The graph must be traversed for rendering and transmission. The desired quality measure is passed to the traversal algorithm as a parameter. When the traversal reaches a LOD node, this measure is compared with the indices of the available children, and if the desired child is not available, the next available child with lower quality is chosen (Figure 18). We are able to handle a model containing multiple LOD nodes with different numbers of children. Thus a reasonable combination of multiple LOD nodes can be found independently of the model.

Software architecture

Our software architecture is divided into server and client programs (Figure 19) The server runs two important software modules: the connection manager and the geometry database manager. Currently we run both managers within a single UNIX process, but as their relationship is well-defined, performance may be enhanced by building a decoupled system configuration in the style of MR [77] running on a multi-processor machine.

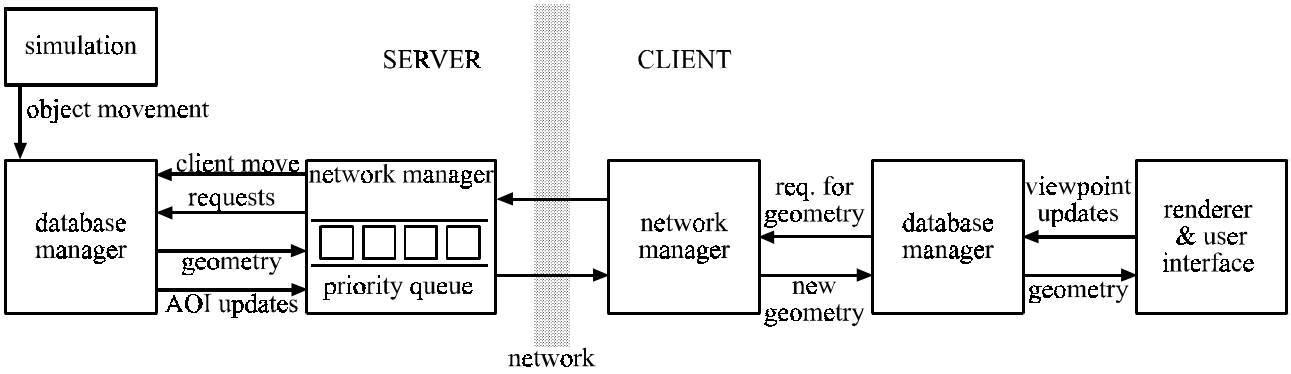


Figure 19: Software architecture of client and server programs

The connection manager is responsible for dealing with the network. We use UDP sockets for handling the network connections between client and server, which are bi-directional. As the name suggest, this module implements connection management as shown in table 1.

The database manager stores a collection of objects forming the virtual environment. Objects can either be controlled by a simulation, or they represent a user's avatar, and are controlled directly by the user. The database manager also keeps track of the client's AOI and sends updates on objects within the AOI as necessary.

The client has a network and database manager, very similar to the one of the server. However, the actions performed by the client are quite different. Before every frame, a LOD oracle is invoked to find an appropriate level of detail for rendering. The oracle is also used to find out if any LOD must be requested from the server.

For rendering, each object graph stored by the database manager is traversed and appropriate rendering actions are performed for every node. The case where rendering traverses a LOD node, but the prefetching has failed to provide the desired graph data, is automatically handled by the traversal: the next coarser available level is used instead. The users actions, in particular viewpoint changes, are passed to the database manager, that reacts appropriately.

5.7 Results

The protocol presented in this paper was implemented for test purposes. Picture 1 (see appendix) shows a screen shot of the virtual environment with demand-driven geometry transmission system. The window in the upper middle shows a bird's eye view of the server. Two clients with their respective areas of interest can be seen. The upper left and lower left windows give the 2-D and 3-D view of the first client, respectively. Note the replicated geometry in the 2-D view and the levels of details for objects in the 3-D view. The upper right and lower right window show the corresponding view for the second client. Note that user's can see each other, so the avatars are displayed in the 3-D views (humanoid figures).

We constructed a virtual environment for test purposes by randomly placing objects on a plane. The objects were procedurally constructed to contain multiple levels of detail with progressively less primitives. Distant objects appear smaller because of perspective projection, so we decided to reduce the number of primitives per LOD corresponding to a 1/x function where x is the LOD number. The objects have 6 LODs. LOD selection was done based on the distance, where the radius of the AOI was divided into uniform intervals. For our tests, we used a prerecorded walk-through sequence of 500 frames.

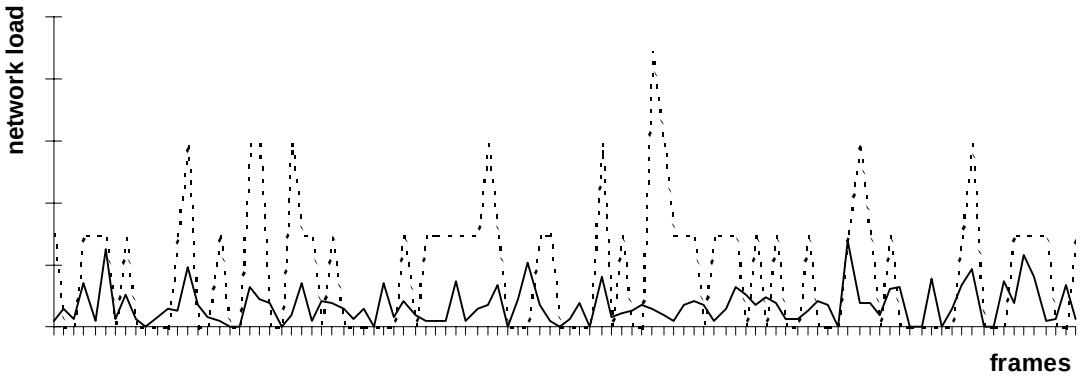


Figure 20: Network load of demand-driven transmission with (solid) and without (dashed) LOD management

Comparing the total size of the virtual environment database to the transmitted data is not fair, since the client's area of interest contains a roughly constant number of objects (if we assume a uniform distribution of objects), but the complete virtual environment can be made arbitrarily large. We were rather interested in the comparison of demand-driven transmission with LOD management and without (i.e. objects are always transmitted completely, including all LODs).

Figure 20 shows the network load of the first 100 frames of the walk-through. The solid curve shows the network load with and the dashed line without LOD management. Downloading complete objects gives stronger load variations (when an object enters the AOI, all its data has to be transmitted at once), while transmitting individual LODs tends to better distribute the effort. Furthermore, high-resolution LODs of objects that never come close to the observer are not downloaded, which explains why the load for transmission with LOD management is generally lower than without.

We also wanted to see how the number of LODs would influence the performance for a fixed AOI. Therefore we measured the total number of transmitted bytes for the walk-through sequence without LOD management, and also with LOD management for an object with 3-20 LODs. (Note that objects with more than 10 LODs are never used in practice.) We computed the total transmitted data with LOD management as a fraction of the number without LOD management. The results show that more LODs reduce the fraction of transmission, but the improvement drops significantly as more LODs are added. For a realistic setting with 6-8 LODs for an object, the total reduction in network traffic can be up to a factor of 3 compared to no LOD management (Table 2).

# LODs	3	4	6	8	10	20
load fraction(%)	53.8	48.3	39.6	36.8	34.4	27.6

Table 2: Savings of network load of demand-driven transmission with LOD management over always transmitting complete objects. Results show remaining percentage of transmitted data using LOD management.

5.8 Summary

This chapter presents a strategy for managing network transmission of geometry data in distributed virtual environments. In our client-server based approach, the client request geometry from the server based on individual levels of detail instead of downloading complete objects or even entire scenes. The approach is based on a limited area of interest that must be kept up to date. Load variations are handled by prefetching, graceful degradation and progressive refinement. Results show a significantly improved network performance.

6. An Octree-Based Level of Detail Generator

6.1 Introduction and motivation

A rendering system that heavily relies on the use levels of detail to improve performance, such as the one presented in chapter 5, naturally raises the question how the levels of detail for the objects are generated. While today's modeling software often supports control of the polygon count, relying on external tools is often impractical when assembling virtual environments for a number of reasons (such as the need to use models from sources that cannot provide levels of detail, such as simulated or measured data, or simply models "grabbed" from free sources on the Internet). Besides, the hierarchical structure of geometry models as described in chapter 5 is not generally supported.

Therefore, we decided to implement a level of detail generator we called LODESTAR to support the modeling and assembly stage of our virtual environment system. VRML 1.0 [56] was chosen as the input/output format, because it supports the desired hierarchical geometry structure, is compatible with the Open Inventor toolkit used to implement the rendering portion of the system, and also allows to make use of the momentum generated by the VRML effort, and the rapidly growing number of resources (models, software etc.).

The contributions made by implementing the tool described in this chapter are both in the scientific domain (a successful experiment with a new clustering algorithm as described below), and in the software engineering domain (a practical guide how to deal with the problems encountered when working with real geometric data from various sources rather than with idealized models often used to evaluate level of detail generation algorithms.

6.2 Octree quantization for levels of detail

Real-world applications almost always involve ill-behaved data, and for very large scenes and slow connections, it should be possible to produce very coarse approximations as well as moderately coarse ones. The best choice under these circumstances are LOD generation methods that ignore the topology of objects and force a reduction of the data set. This can be achieved by clustering multiple vertices of the polygonal object that are close in object space into one, and remove all triangles that degenerate or collapse in the process. Such an algorithm does not allow fine-tuned control of details, but can robustly deal with any type of input data, and produce arbitrarily high compression. Previous attempts at vertex clustering have been done with uniform quantization [64] or a binary tree [66].

We propose to use octree quantization [27] for vertex clustering. Octree quantization is superior in quality to uniform quantization and in speed to binary trees. The three-dimensional spatial structure

represented by an octree allows simple clustering operations on three-dimensional samples in linear time. The method works well for colors (the three dimensions being the R, G, and B component), and has been adapted for (x, y, z) vertex coordinate tuples in this work. In the following, we outline the quantization method. We start by explaining how to create the octree data structure, and proceed with details on how to identify clusters, and how to select the representative for each identified cluster. Finally, we describe how to obtain the simplified model from the original model using the octree as an auxiliary datastructure.

Building the octree

Octree quantization was originally developed to select the entries for a color lookup table that optimally represent a given image. Instead of color pixels, we enter the vertices of the model into an octree. Intermediate nodes of the octree represent subdivisions of the object space along the x, y, and z direction. The goal is to place exactly one vertex in each subvolume. The octree is successively refined by further subdivision of leaf nodes when entering new vertices until this criterion is satisfied. Theoretically, this can generate arbitrarily deep octrees, but in practice a certain octree depth is never exceeded as the input data comes in finite precision floating point numbers.

When entering a new vertex, the octree is recursively traversed by comparing the coordinates of the new vertex against the coordinates stored in the octree node, and traversing the link to the appropriate child node until a leaf or a nil pointer is encountered. Three cases must be distinguished:

Case 1: The selected link is a nil pointer, so the corresponding subvolume is empty, and we can simply create a new leaf node and store the vertex in that node (Figure 21).

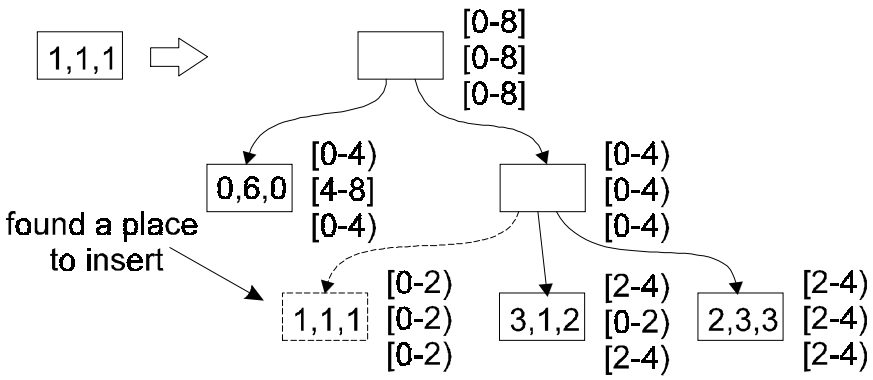


Figure 21: Inserting a vertex into an empty subvolume

Case 2: The link points to a leaf node, and the new vertex is equal to the vertex stored in the leaf: No new node is created, but the vertex counter of the existing node is simply incremented. Note that this automatically sorts out doublets in the vertices, which are a major defect of many VRML models found today, because only one copy of each vertex is finally output (Figure 22).

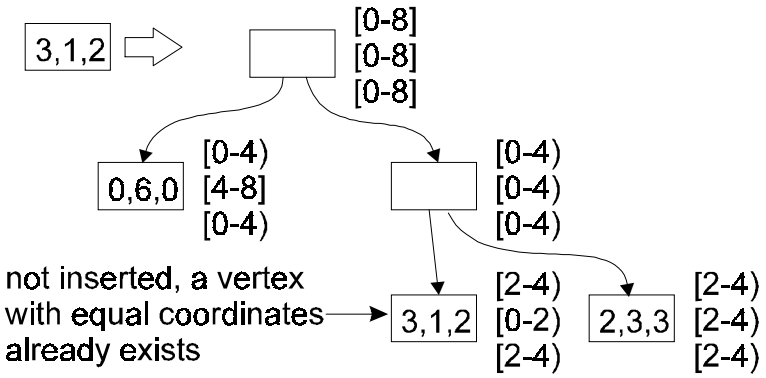


Figure 22: Inserting an already existing vertex

Case 3: The link points to a leaf node, but the new vertex is not equal to the vertex stored in the leaf: The leaf vertex and the new vertex fall into the same subvolume, so the octree must be subdivided in that location. A new intermediate node is created, and the old leaf node and a new node containing the new vertex are inserted as children of the new intermediate node (Figure 23).

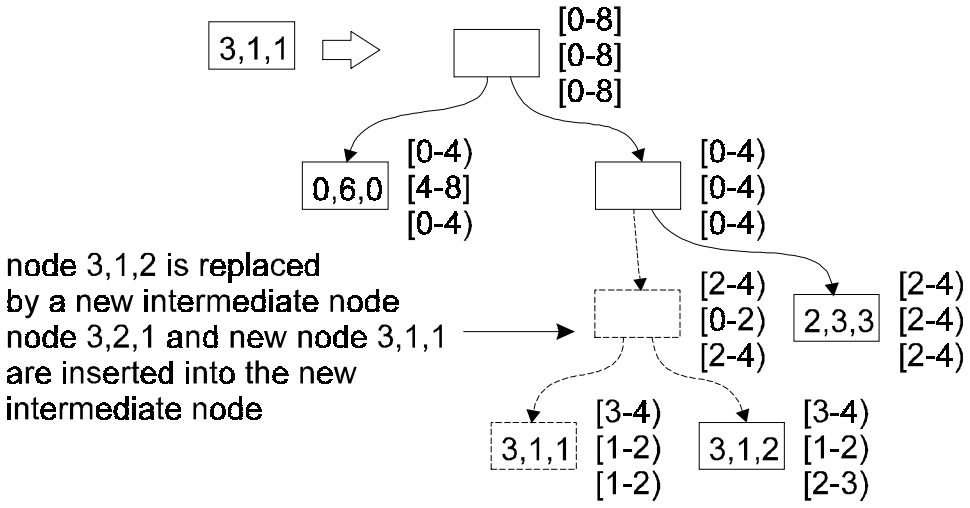


Figure 23: Inserting a vertex into an occupied subvolume

Vertex clustering

The number of vertices is reduced by combining multiple close vertices into one cluster. For such a cluster, a representative is chosen from the set of vertices represented by that cluster. This has the advantage that no new vertex must be synthesized, and the original set of vertices can be kept unchanged.

Vertex clustering is done by replacing leaf nodes that share an intermediate node as common parent with that parent, setting the vertex count of the parent to the sum of the vertex counts of its children. In selecting the cluster, the following criteria are relevant:

- From all clusters, select the one whose nodes have the largest depth within the octree, for they represent vertices that lie closest together.
- If there is more than one such cluster, additional criteria can be used for the selection according to the user's preferences: Selecting the cluster that represents the fewest vertices will keep the error

sum small. Selecting the cluster that represents the most vertices will tend to generate coarser representations of finely tessellated areas with fewer vertices, but preserve small distinctive features instead. Experiments show that this latter strategy usually produces better results.

Selecting the cluster representative

The remaining problem is which strategy to use to select the representative vertex from the vertices in the cluster (Figure 24). To do so, we use three different heuristics with user defined weight. Let the *involved triangles* be those triangles that have at least one vertex in the cluster.

Error area is an attempt to measure the change in the extent of the object's surface: If a cluster of vertices is replaced by a representative, the areas of most involved triangles change. The error area is defined as the difference in the summed area of the involved polygons before and after the clustering. The vertex that produces the smallest error area is chosen.

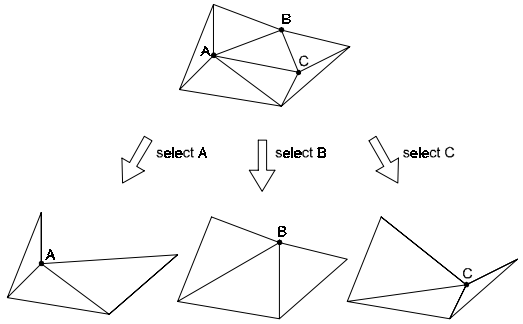


Figure 24: Different choices of the representative influence the area of the resulting triangle mesh

Error volume is an attempt to measure the object's change in volume: For every involved triangle, we construct the tetrahedron from the three original vertices and the potential representative. The volume of such a tetrahedron is zero if one of the vertices is elected the representative. The summed volume of all such tetrahedrons is taken as the error volume, and the vertex with the smallest error volume is elected. One disadvantage of this approach is that all volumes are zero if the vertices lie in a plane, so it is only useful in combination with another heuristic.

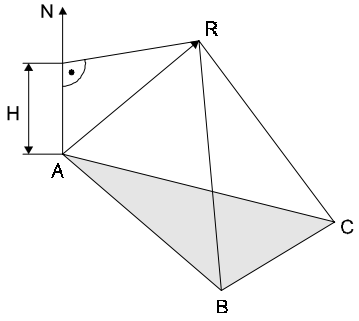


Figure 25: The error volume is computed from the tetrahedron with the original triangle ABC as a base and the chosen representative R as the top

Weighted mean is an attempt to find the vertex that "best" represents the other vertices: An average vertex is synthesized from the cluster as a weighted mean, where the weights are the vertex counts of

the nodes in the cluster (remember that leaf nodes represent a single vertex, intermediate nodes represent all leaves in their subtree). The vertex that is closest to the mean is chosen. Unfortunately, this does not take into account any surface properties, and our experiments show that results using this heuristic are visually not as appealing as the other two heuristics. Weighted mean was kept for the sole purpose of handling indexed line sets (see below).

Computing the reduced triangle set

After the number of vertices has been reduced by the desired amount, the set of triangles associated with the reduced vertex set must be reconstructed. For every triangle, its vertices are replaced by the representative chosen for that vertex. This process may produce doublets (triangles with identical vertices) for which only one instance is kept. Triangles may also collapse into lines, most of which are identical to the edge of another triangle and can be discarded. The remaining lines are usually important for the appearance of the model and are thus saved. Sometimes triangles collapse into points, which are removed from the model.

6.3 Dealing with VRML specifics

Up to now we have silently assumed that the geometric model consists of an unstructured set of triangles, and we have neglected in the discussion a variety of properties specific to VRML models.

Non-polygonal nodes

VRML models do not only consist of triangles or polygons, but also of other primitives like spheres or text, and of context-defining nodes such as transformations. However, the essential structure of VRML scenes is the IndexedFaceSet and its helper nodes Coordinate3, Normal, TextureCoordinate2, and Material. Large amounts of geometric primitives are almost exclusively specified using IndexedFaceSets, and therefore it is reasonable to concentrate on this node for level of detail generation. Level of detail generation dealing with VRML geometry other than IndexedFaceSets may become an interesting area for future research, but this is beyond the scope of this work.

Scene graph structure and output format

VRML models and scenes are not “flat”, but are rather arranged in a hierarchical scene graph, so an algorithm dealing with a single set of polygons is not sufficient. The simple yet effective solution that was used in LODESTAR is to traverse the VRML model and apply the LOD generation to every IndexedFaceSet individually, producing for each a new LOD node (details on how to deal with multiple IndexedFaceSets are given in section 6.4).

LODESTAR replaces every IndexedFaceSet in the original file with a subtree containing the computed LODs. This subtree contains a single LOD node. If the structure of the file requires that additional nodes (such as bindings) are output, the whole structure is wrapped in an additional Separator. The children of the LOD node are the IndexedFaceSets containing the computed levels of detail.

If any triangles collapsed to lines are produced as a result of the clustering process, an additional IndexedLineSet is generated to complement the IndexedFaceSet, and the resulting structure is wrapped in a Separator.

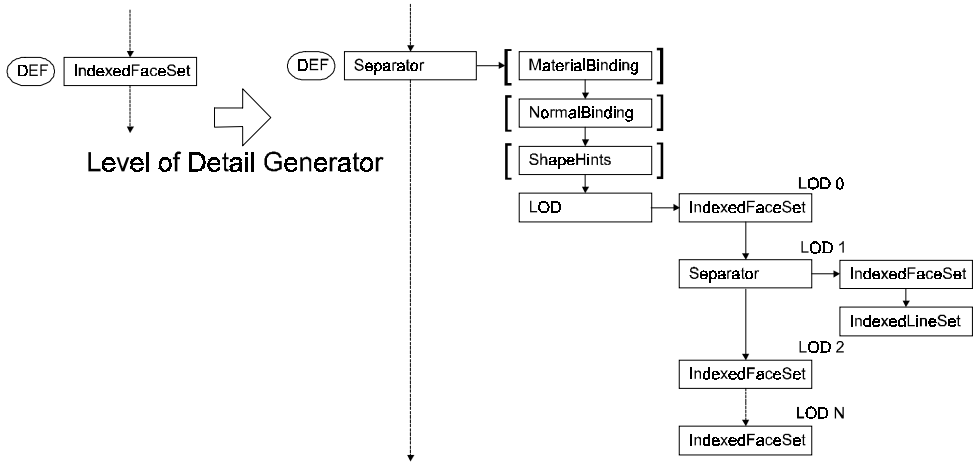


Figure 26: An indexed face set is converted into a subtree with a single LOD node

Triangulation

As already mentioned, most level of detail algorithms including LODESTAR can only deal with triangles as an input. The triangulation is necessary because after a vertex clustering operation, any n-sided triangle (with n>3) almost certainly becomes non-planar. Therefore all n-sided polygons are triangulated first by using the algorithm from [52]. As a side effect, all concave polygons are removed from the model, which allows the use of algorithms that are simpler, more robust and faster.

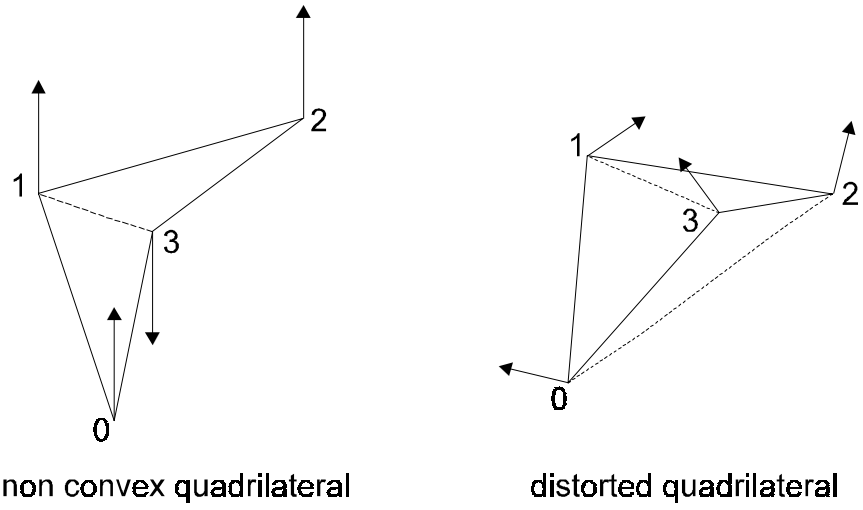


Figure 27: Degenerated quadrilaterals must be split

The exception are quadrilaterals, for which the error is often small and hence tolerable (i.e. non-visible). It is necessary, though, to check any quadrilaterals for validity after a modification of its vertices. Concave quadrilaterals or quadrilaterals which are distorted in space more than a user-

specified threshold (measured as the maximum angle between the normals at the vertices) are split into two triangles (Figure 27).

Triangulation increases the number of polygons and can involve a performance penalty. However, most 3-D rendering engines triangulate all geometry internally [54], so with the use of triangle strips, a performance penalty can be avoided. This consideration of course assumes that the renderer detects and uses triangle strips, which unfortunately cannot be influenced from within a VRML file.

Lines

IndexedLineSets can be treated almost like IndexedFaceSets: Vertices are clustered with octree quantization, and a new IndexedLineSet is constructed from the reduced vertex set for every level of detail. The output is equivalent to the structure depicted in Figure 26, except that IndexedFaceSets are replaced by IndexedLineSets. However, for IndexedLineSets the only applicable heuristic for representative selection is weighted mean.

Bindings

Non-indexed bindings impose a one-to-one relationship between entries in the IndexedFaceSet fields and the corresponding helper nodes. They cannot be maintained if multiple levels of detail are to share the same materials, normals etc. Therefore non-indexed bindings are transformed into the corresponding indexed bindings, and an index will be synthesized. In this case, an additional MaterialBinding or NormalBinding is generated.

Range values

The selection of a LOD in VRML is performed by comparing ranges. A viewer switches to the next LOD if the distance of the object to the viewpoint is greater than or equal to a specified value. For satisfactory performance, the LOD generator has to compute reasonable range values.

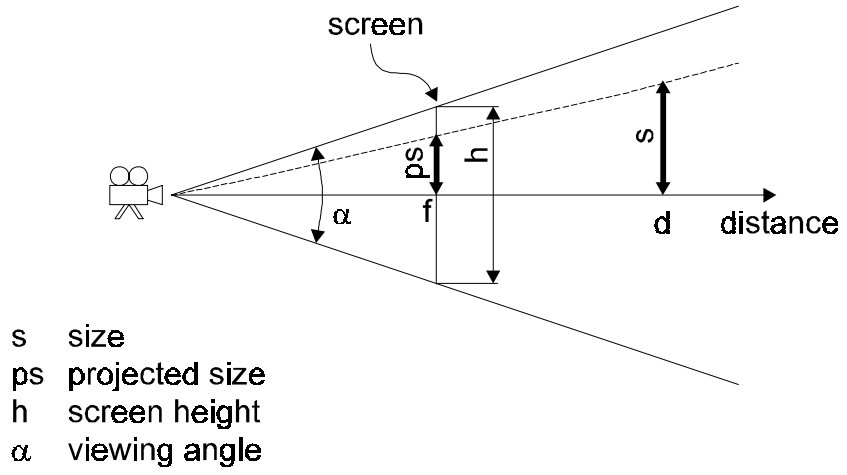


Figure 28: A heuristic is used to compute the range values required for the LOD node

The next level of detail is computed by moving vertices from a cluster to a selected representative. The maximum visible error introduced by this operation is equal to the maximum distance a vertex can move in screen space due to a clustering operation (deviation). The goal is to compute LOD

switching ranges in such a way that this maximum visible error does not exceed a user defined threshold, that is specified as a percentage of the screen height.

The viewer must switch LODs if the maximum deviation *s* projected onto the screen is greater than error range specified as a fraction of the height of the screen. Let *rootsize* be the extent of the cube associated with the octree root (note that the actual size is computed from the local coordinates in the octree modified by the current scale factor from preceding Scale or Transform nodes!) and *depth* be the level of the octree corresponding to level of detail being computed:

$$s = \text{rootsize} \cdot 3½ / 2\text{depth-1}$$

The height of the screen *h* is computed from the camera height angle *alpha* and the focal length *f*:

$$h = 2 \cdot \tan(\alpha/2) \cdot f$$

Given the desired *errorrange* (in percent), we can compute the maximum projected deferral *ps* as:

$$ps = \text{errorrange}/100 \cdot h$$

Finally, from the relation *d/s = 1/ps*, we can compute the range *d* as

$$d = s / (\text{errorrange}/100 \cdot 2 \cdot \tan(\alpha/2))$$

To take into account the extent of the cluster, for the actual range one has to add the radius of the bounding sphere of the cluster to *d*.

6.4 Joining nodes

Often VRML files are produced with primitive converters that generate many IndexedFaceSets in sequence, each containing very few polygons. Computing levels of details for every IndexedFaceSet of such a model has a tendency of ripping apart the model and produces useless LODs (see Figure 29), a problem also reported by [55].

Fortunately, most of these degeneracies can be cured with a very simple algorithm that joins sequential IndexedFaceSets into one if possible. This algorithm does not even require knowledge of the involved geometry but can operate in purely syntactical way on the VRML file. It does not work in every case (this would require a deep analysis of both model structure and geometry), but it cures most of the degeneracies that we have encountered so far, and even more importantly, it works very fast.

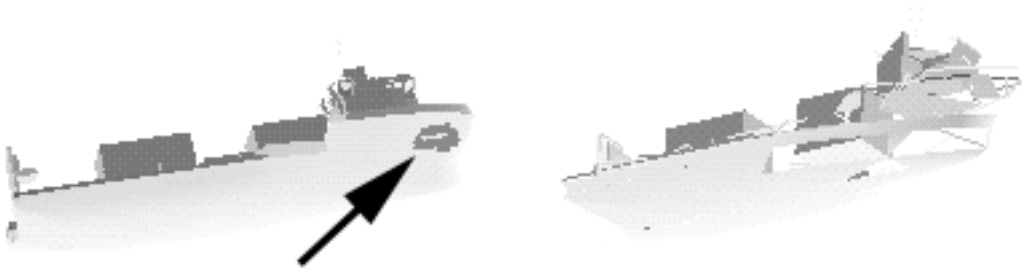


Figure 29: The hull of the ship model was represented by many small IndexedFaceSets (one shown in upper image). When computing LODs, the hull parts are modified individually, and undesirable holes appear. This can be suppressed by joining nodes.

For the components of a boundary representation (IndexedFaceSet, IndexedLineSet Material, Normal, TextureCoordinate2, Coordinate3) and Separators, Groups and Bindings, subsequent nodes of the same type are joined unless the second is tagged with DEF.

In case of multiple Separator or Group nodes, the sub-groups can be joined. In this process, the components of a boundary representation that span multiple sub-groups are joined into one node of that type, so a single IndexedFaceSet can be synthesized.

Example 1: Joining two Separator nodes with different Material sub nodes and possibly different IndexedFaceSet sub nodes. Note that it is necessary to insert a new MaterialBinding so that the synthesized Material node is put in correct relation to the synthesized IndexedFaceSet.

```
Separator {
  Material { diffuse 0.5 0.6 0.4 }
  IndexedFaceSet { coordIndex
                   [1,2,3,-1] }
}
Separator {
  Material { diffuse 0.3 0.3 0.3 }
  IndexedFaceSet { coordIndex
                   [4,5,6,-1] }
}
```

becomes

```
Separator {
  Material {
    diffuseColor [0.5 0.6 0.4,
                  0.3 0.3 0.3]
  }
  MaterialBinding { value
                   PER FACE INDEXED }
  IndexedFaceSet {
      coordIndex [1,2,3,-1,4,5,6,-1]
      materialIndex [0,1]
  }
}
```

Example 2: Joining two Separator nodes with possibly different Coordinate3 sub nodes and possibly different IndexedFaceSet sub nodes: The algorithm also works with Coordinate3, Normal and TextureCoordinate2 nodes:

```
Separator {
  Coordinate3 { point
   [ 10 11 12, 13 14 15, 16 17 18] }
  IndexedFaceSet { coordIndex
                   [0,1,2,-1] }
}
Separator {
Coordinate3 { point [ 20 21 22,
             23 24 25, 26 27 28] }
IndexedFaceSet { coordIndex
                 [0,1,2,-1]}
}
```

becomes

```
Separator {
  Coordinate3 { point
    [10 11 12, 13 14 15, 16 17 18,
    20 21 22, 23 24 25, 26 27 28] }
  IndexedFaceSet { coordIndex
              [ 0,1,2,-1,3,4,5,-1] }
}
```

Trailing Separators

The joining algorithms works by traversing the scene graph bottom-up from the leaves, so that joinability can be propagated upwards. To improve chances of joinability, trailing Separators are removed (a Separator node on the end of a list is not necessary).

```
Separator {
  Separator { IndexedFaceSet {
              coordIndex [0,1,2,-1] }
  }
}
```

becomes

```
Separator {
  IndexedFaceSet { coordIndex
                   [0,1,2,-1] }
}
```

Limitations

The joining algorithm is a heuristic that was developed after studying the kind of degeneracies that are commonly found. It only works for relatively simple cases involving direct relations between the components of an IndexedFaceSet. Care must be taken that no other node such as a Transform is present that forbids the joining.

6.5 Implementation

The algorithm described in this work has been implemented under C++ and ported to a variety of platforms, including multiple flavors of Unix, OS/2, and DOS. Because of the design decisions outlined earlier, it runs very fast. It works reasonably robust on input files that do not exactly comply to the VRML specification (such as some Inventor files). Furthermore, the software can be used as a „cleanup“ filter for VRML files: As explained in section 6.2, the process removed doublets in the vertices, colors etc., so invoking the program with the „no levels of detail“ option cleans up redundant models. See the appendix for some sample results.

6.6 Results

The LODESTAR code was tested with a large number of models downloaded via the Internet. Here we present a few quantitative results and images to give an impression of the performance of the implementation. The „Enterprise“ model (10 LODs) was computed in 5.4 seconds and the „Galleon“ model (8 LODs) was computed in 7.8 seconds on an SGI Indy R4400/150 workstation.

LOD	Triangles	Range	LOD	Triangles	Range
0	6343	36	5	1083	193
1	5020	42	6	553	355
2	3999	52	7	167	679
3	3182	72	8	51	1325
4	1960	112	9	16	2615

Table 3: „Enterprise“ model statistics

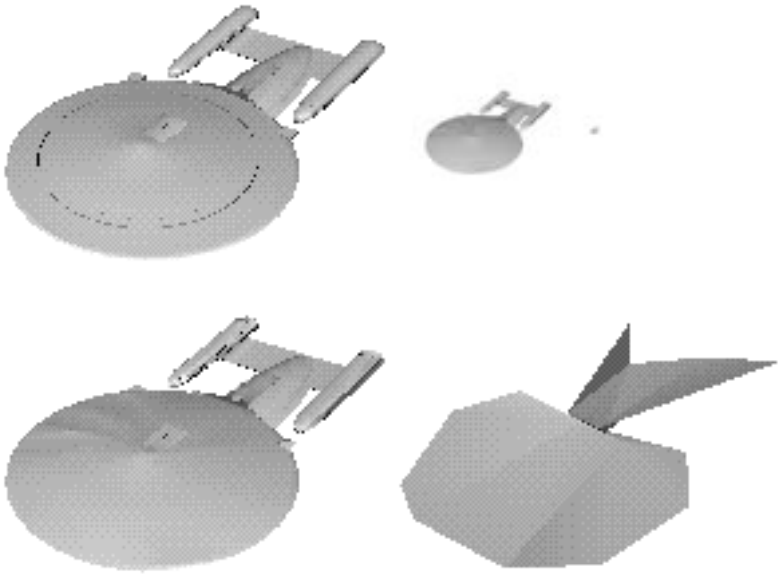


Figure 30: Three LODs of the enterprise mode (LOD # 0, 4, 8). If displayed in a size corresponding to the computed ranges, the quality degradation is no longer visible

LOD	Triangles	Line	Range	LOD	Triangles	Line	Range
0	4698	0	1962	4	1478	8	12505
1	4142	0	2664	5	1478	8	12505
2	3686	0	4070	6	108	4	46122
3	2981	9	6882	7	24	0	90932

Table 4: „Galleon“ model statistics

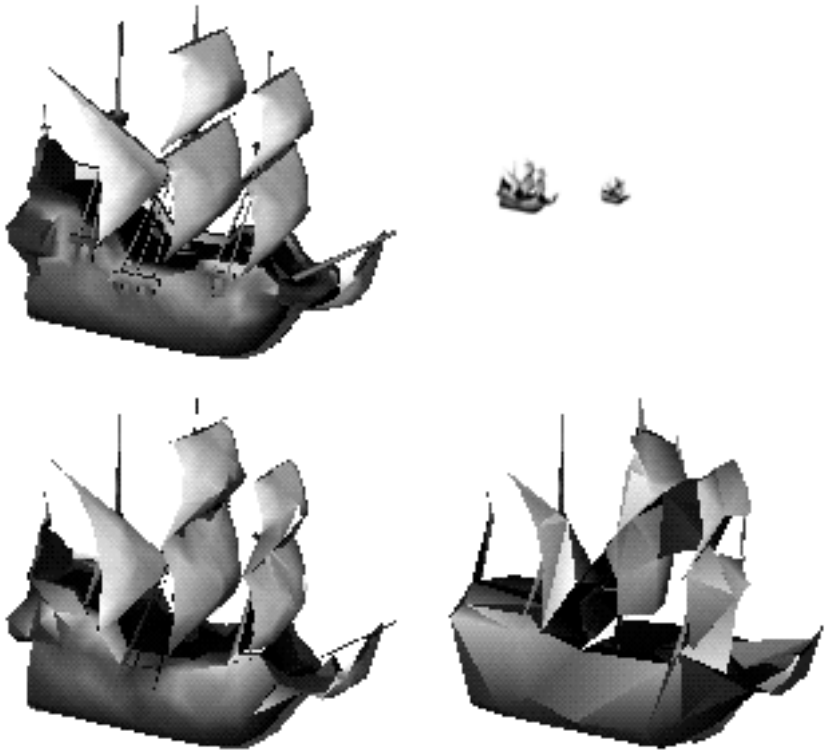


Figure 31: Three LODs from the galleon model (LOD # 0, 4, 5)

6.7 Summary

We have presented an algorithm that produces levels of detail for polygonal objects. This algorithms reads and writes VRML files and takes care of the particular needs of the hierarchical structure and other advanced features of this format. It uses octree quantization to cluster vertices and thereby reduces the number of vertices and faces in the model. Results show that the algorithm works robustly and efficiently for a large class of models. It has successfully been used to assemble virtual worlds for the system described in this thesis.

7. Smooth Levels of Detail

7.1 Introduction and motivation

Frequently polygonal models are very large, exceeding rendering capacity and network throughput. Adding levels of detail partly addresses the rendering problem, but makes overall model size even larger. The reason for this problem is that the standard approach of representing polygonal data as lists of vertices and triangles is not powerful enough. In this chapter, we present a data structure that does not suffer from the mentioned shortcoming and fulfills the following requirements:

1. *Smooth LODs*. The model data structure should represent many levels of details (not only 3-6, but hundreds or thousands of LODs), so that a continuous (or almost continuous) refinement of the model is possible by repeatedly adding small amounts of local detail to the model.
2. *Incremental decoding*. Decoding of the smooth LODs should be incremental, i. e. the next finer LOD should be represented as the difference to the current LOD. By reusing all the data from the coarser LODs, model size can be kept small despite the large number of LODs.
3. *Interactive LOD selection*. The smooth LODs data structure should support selection and rendering of any specific LOD in real-time allowing to vary the level of detail (both coarser and finer) at interactive speeds (during rendering).
4. *Incremental transmission*. It should be possible to incrementally transmit the model over the network, starting from the coarsest approximation and progressing to the original model. In particular, rendering should be able to make immediate use of all the data received up to a certain moment, and render a model not yet fully transmitted. This is important for progressive refinement of large models that take an extended period to transmit, and allows continuous operation in case of network failures.
5. *Compact representation*. It is preferred if the smooth LODs data structure introduces no overhead in model size compared to the original, uncompressed polygonal model. Ideally, the introduction of smooth LODs should yield compression instead of increasing the model size.
6. *Variable resolution within the model*. If the many LODs within the data structure have only local influence on the appearance of the model, the corresponding details can be selected individually, resulting in variable resolution within a single model. This is particularly useful for models with a large extent (e.g. a ship model observed from its deck), where parts close to the observer should have high fidelity, whereas distant parts can be represented by a coarse approximation while avoiding cracks in the area of transition from one LOD to another.

Our data structure is based on a binary tree that is created by hierarchically clustering vertices of the original model, thereby constructing a *cluster tree*. Every clustering operation simplifies the model, and therefore every node of the cluster tree represents a single level of detail. A linearization of the tree in the inverse order of the clustering process yields a sequential representation of the data structure that is suitable for network transmission. It also incrementally encodes the model, and therefore fulfills our requirements 1, 2, and 4. The next section discusses the creation of this data structure in detail.

7.2 Representing the model as a hierarchical cluster tree

Hierarchical clustering for LOD generation, as presented in [66], is based on the idea that groups of vertices which project onto a sufficiently small area in the image can be replaced by a single representative: a many-to-one mapping of vertices. As a consequence, the number of triangles is reduced when the triangles' vertices are replaced by their representatives from the reduced vertex set, and collapsed triangles are filtered out. Repeated application of the clustering operation yields a sequence of progressive simplifications (LODs). If exactly two clusters are combined in every step, the result is a binary tree, the *cluster tree*.

Construction of the cluster tree

The cluster tree is built by successively finding the two closest cluster in the model and combining them into one. The combined cluster is stored in a new node which has the two joined clusters as its children. The process is repeated until only one cluster containing all the vertices remains, which is the root of the cluster tree.

For each new cluster, a *representative* is chosen from the set of vertices in the cluster. More precisely, we chose the representative to be one of the two representatives of the child clusters. The distance of two clusters (used to find the closest clusters) is computed as the Euclidean distance of the two childrens' representatives. This value is also stored as the *cluster size* in the new cluster's node for further use. The process works as follows:

- *Initialization*: Form a cluster for each vertex, with the vertex serving as the representative
- *Step 1*: Find the two clusters with the closest representatives
- *Step 2*: Replace the two clusters identified in step 1 by a joint cluster, select a new representative
- *Step 3*: If more than one cluster remains, go to step 1

The cluster tree contains instructions for a continuous simplification of the model, and therefore can be used to construct a sequence of smooth levels of detail. However, in its form described above, it only stores the vertices of the model, but not the triangles. To use the cluster tree as an alternate representation of the original polygonal model, the triangles must also be encoded and stored in the cluster tree in a way so that the original model (or any desired level of detail) can be reconstructed from the extended cluster tree alone.

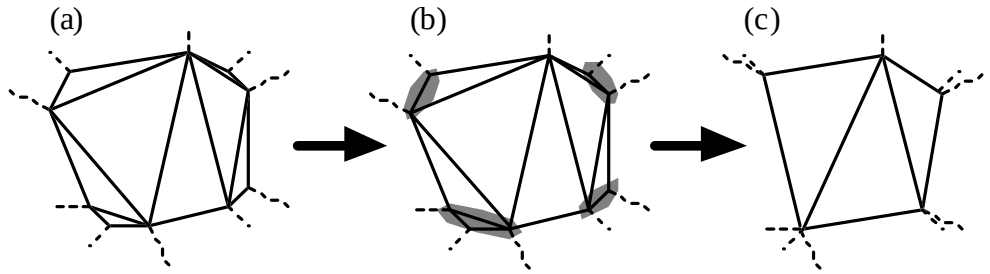


Figure 32: The clustering process: A mesh (a) is mapped onto a vertex cluster tree, which is used to group vertices (b). From the reduced vertex set, a simplified model (c) is computed.

This is done by recording the events (changes) in the triangle database when two clusters are joined (and consequently one representative vertex is eliminated). The inverse operation of these events can be used to reconstruct the triangle database by reconstructing the cluster tree node by node. If the events are appropriately recorded, the smooth LODs can be generated by a simple traversal of the cluster tree in the inverse order of the clustering process with appropriate output.

Triangle event recording during clustering

When the clustering stage combines two clusters into one, those triangles which have at least one vertex in the new cluster must be changed accordingly. For each such triangle, three cases can be distinguished:

1. The triangle has one vertex in the new cluster, and this vertex is elected the new cluster representative. Therefore, no change is made to the triangle at all, and the event need not be recorded.
2. The triangle has one vertex in the new cluster, but this vertex is *not* elected the new cluster representative. This vertex must be changed to the new cluster representative. A list (the *update list*) of all such triangles is kept in the cluster node (Figure 33a).
3. The triangle has two vertices in the new cluster. Therefore it collapses to a line which is discarded from the triangle set. A list (the *collapsed list*) of all collapsed triangles is kept in the cluster node (Figure 33b).

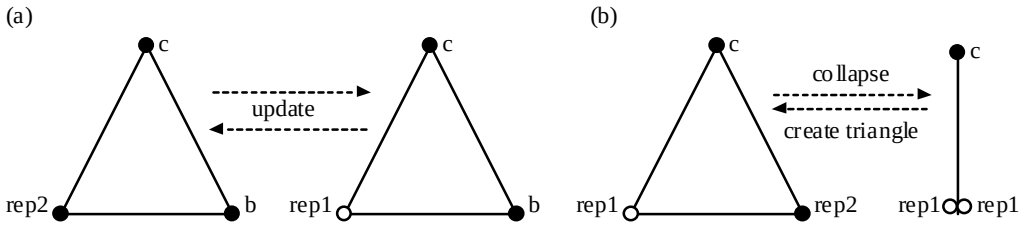


Figure 33: Two events in the triangle database during clustering are of interest for the reconstruction of the original triangles: Collapsing triangles (a), and triangles whose vertices are updated (b).

The lists kept for events of type 2 and 3 make it efficient to perform the construction of the new triangle list for each generated level of detail. Stepping from one LOD to the next is done by adding only one vertex (adding one cluster, see Figure 34). The involved changes are small, so coherence

between LODs is exploited by storing only the changes in the update list and collapsed list at each node.

A cluster tree containing the cluster representatives and the information on triangles (update list and collapsed list) completely encodes the information contained in the original model, plus instructions how to create all intermediate levels of detail. In the next section, we describe basic operations on the cluster tree.

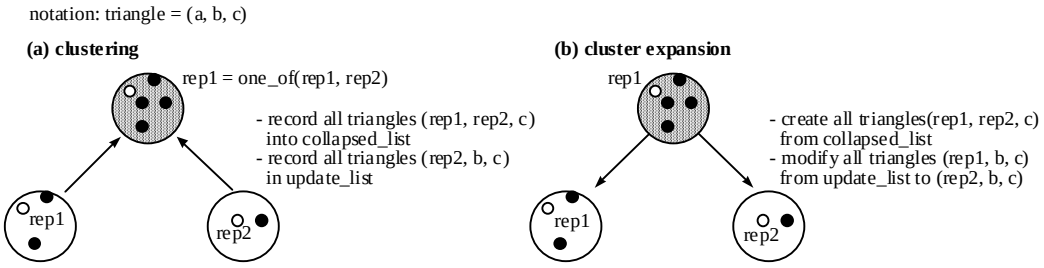


Figure 34: During the clustering, two vertex clusters are joined into one, and the effect on the triangles is recorded (a). The inverse operation, cluster expansion, uses the recorded data to reconstruct the triangles (b).

7.3 Manipulation of the cluster tree

While the cluster tree has the desired property of efficiently representing the original model plus all its levels of detail, it is not directly usable. For rendering, it is still necessary to reconstruct a vertex list and triangle list (either for the original model or for a level of detail). A tree is also not suitable for network transmission, it must be linearized first. Furthermore, a simple method for selecting an arbitrary level of detail is required. Therefore, we define a number of basic operations on the cluster tree, from which the required functions (linearization, model reconstruction, LOD selection, and rendering) can easily be constructed.

Traversal of the cluster tree

During the hierarchical clustering process, the nodes of the cluster tree were generated in the order of increasing cluster size. Traversal of the cluster tree is done in the exact inverse order of its creation. A set of active nodes is maintained to reflect the current status of the traversal. Starting with the root of the cluster tree, the algorithm processes the cluster tree node by node, in the order of increasing cluster size. Every visited interior node is replaced by its two children. The following pseudo-code sketches the algorithm:

```
activeNodeSet = root
while not empty(activeNodeSet)
  current = get node from activeNodeSet with biggest
            cluster size
  process current
  if(not isLeaf(current->left))
    add current->left to activeNodeSet
  if(not isLeaf(current->right))
    add current->right to activeNodeSet
endwhile
```

Reconstruction of the polygonal model

The original polygonal model, consisting of a vertex list and a triangle list, can be reconstructed using the cluster tree traversal function. The root introduces the first vertex. With every visited node, one new vertex is introduced and added to the vertex list (the other child inherits the parent's representative). At the same time the triangle list is reconstructed by processing each visited node's collapsed list and update list. Every entry in the collapsed list introduces a new triangle into the triangle list (reversing the process by which this triangle was collapsed and removed). Every triangle in the update list contains the parent cluster's representative, which must be replaced by the new vertex mentioned above. When all nodes have been visited by the traversal, the original model has been completely restored.

Selection of a LOD

The original model is only the most detailed version of a large number of LOD approximations. A convenient way to select any desired LOD from the available range is to terminate when all nodes belonging to a particular LOD have been visited. The desired LOD is specified as a threshold that is compared to the cluster size contained in every node. A modified traversal algorithm no longer continues until the active node set is empty, but terminates if the biggest cluster size of any such node is smaller than the given threshold. The reconstructed triangle and vertex lists up to that point represent the desired level of detail and can directly be used for rendering.

Refinement

For refinement of the model, the fundamental operation is to switch from a given level of detail to the next finer one. A particular LOD is defined by a list of active node in the cluster tree, and the corresponding vertex and triangle lists. This is achieved by expanding the node which is selected for refinement into its two successors, and using the information contained in that node to extend the triangle list and vertex list. This is an incremental operation that typically requires only a small amount of processing and can be carried out at interactive speed. Selection of a LOD as previously mentioned is nothing else than the repeated application of refinement, starting with a single vertex and an empty triangle list.

Simplification

The inverse operation to refinement is simplification, which is used to switch from a given level of detail to the next coarser one. Two nodes are clustered into their common parent node. One vertex is removed from the vertex list, and references to that vertex in the triangle list are removed. Collapsed triangles are filtered out, which simplifies the model.

Linearization

The traversal can not only be used to reconstruct the model for rendering, but also to generate a sequential version of the cluster tree suitable for network transmission. Nodes are visited in the same order as for LOD selection, but instead of reconstructing the original model, the information contained in the node is output into a sequential data stream. During that process, triangles and vertices are automatically renumbered in the order in which they are visited, so that references always point back to available valid indices and incremental decoding becomes possible.

7.4 Transmission Protocol

Using the linearization operation introduced in the last section, it is very simple to create the stream of packets required for network transmission. No redundant information is stored in the network packages, so the requirement of compactness is satisfied by the network protocol, which actually represents the smooth LODs model in less bytes than the original model (see section 7.7 for results). Effectively, the protocol can be used as a compression method.

Recall that the following information must be encoded for every node in the cluster tree:

- the new vertex introduced by the refinement operation
- the update list encoding which triangles must be modified to contain the new vertex
- the collapsed list encoding which new triangles must be created when the new vertex is introduced.

The goal of the protocol was to encode the required information with as little data as possible. Our protocol currently deals with vertices, triangles and surface materials and consists of four packets types: VERTEX, TRIANGLE, MULTI, MATERIAL.

- **VERTEX**

  Format: VERTEX(parent, x, y, z, update list):

A new vertex is introduced. One node of the cluster tree is replaced by its two children. The coordinates of the representative of one of the new clusters are encoded in this package.

*Parent cluster***.** The parent field indicates the cluster that is being split in two. Indices can only point to already existing clusters, so they can have variable length: As the number of clusters increases, more bits are needed to encode the index. This variable length encoding of indices saves 50% of the bits needed for indices.

Vertex coordinates. The (x,y,z) tuple gives the coordinates of the new vertex. Details on the encoding of the vertices are given in the next section.

Update list. VERTEX also encodes the update list associated with the parent node. Already encoded triangles which contain the parent cluster's representative can either continue to use that representative or from now on use the new vertex. This information must be encoded to allow updating of the triangles correctly. The update is simply the replacement of the parent cluster's representative with the new vertex within the triangle. One bit is sufficient to indicate for each candidate triangle containing the parent cluster's representative whether or not the update should take place. These bits are compactly stored as a bit list.

A variable length bit list is used to encode these updates. Since the number of candidate triangles as well as the order of the triangles given by their position in the global triangle list is known to both sender and receiver, the update process is well defined.

- **TRIANGLE**

  Format: TRIANGLE(vertex id, orientation)

As the reconstruction of the object from the network data steam is the inverse operation of the clustering stage, for every new vertex encoded by VERTEX, the triangles stored in the parent node's collapsed list must be re-introduced as *new* triangles. This is done by a sequence of TRIANGLE packets. The triangle in question collapsed because new vertex and the parent's representative were clustered, so two of the original vertices are already known. The missing third vertex is encoded in the packet as an index into the array of vertices. Like cluster indices, vertex indices can have variable length.

The new triangle has either the orientation (new vertex, parent rep, vertex id) or (parent rep, new vertex, vertex id), which is distinguished by the orientation bit.

- **MULTI-TRIANGLE**
  Format: MULTI(duplicate flag, vertex id)

The clustering process may produce identical triangles that are not collapsed and consequently not removed. These doublets were intentionally left in the data, because removing them would greatly complicate the coding and decoding process. Instead, the MULTI package can introduce either 2 or 4 related triangles at once, which efficiently covers the most frequent cases produced by the clustering algorithm. If the duplicate flag is zero, 2 triangles with either orientation (new vertex, parent rep, vertex id) and (parent rep, new vertex, vertex id) are created. If the duplicate flag is one, 2 triangles of either orientation are created.

- **MATERIAL**
  Format: MATERIAL(index)

While polygonal models always contain geometry, they may or may not contain materials or colors. Our models consist of a small set of fixed materials, that can be encoded in an 8 bit index. A MATERIAL packet sets the current material of the following geometry to the new value until another material package is encountered. As our models use only a few different materials, such packets are relatively infrequent, and no further optimization efforts were taken. Material definitions are distributed once to all participating sites. If required, material definitions can be given in the header of the model. A more sophisticated shading support may include vertex colors for pre-shaded (e.g., radiosity) models.

Packet format. Packets are headed by a variable length tag according to the frequency of the individual packet types and their expected length. Table 5 summarizes the packets including their parameters (field sizes in bits are given in parenthesis).

Packet	Tag	Fields		
vertex	0	parent (variable)	coordinates (s. below)	update list (variable)
triangle	10	vertex id (variable)	orientation (1 bit)	
multi	110	duplicate flag (1 bit)	vertex id (variable)	
material	111	material id (8 bit)		

Table 5: Protocol packets with parameters and sizes in bit

7.5 Hierarchical precision encoding of vertices

About half the size of the model is due to the vertex coordinates. These are not effected by the algorithms and therefore are not yet compressed. Deering argues that while coordinate data is usually represented using floating point arithmetic, the finite extent of geometric models allows representation using fixed point arithmetic [16]. To minimize errors resulting from lossy compression via quantization, we have developed a hierarchical precision encoding scheme for the coordinate data. Our method still yields compression ratios of 1:2 to 1:3.

For every ordinate, a neighborhood is chosen by defining a fraction of the object diameter. If the new ordinate lies within the neighborhood of the corresponding ordinate of the parent cluster's representative, the ordinate is encoded with a relative offset to it. This offset is stored as a fixed point value ("relative encoding"). As the new vertex is expected to be in the vicinity of the parent's representative, most of the ordinates can be encoded relatively, thus saving storage. If the ordinate is not in the neighborhood, it is stored as an absolute (32 bit) single precision float ("absolute" encoding).

Typically we define the neighborhood to be a quarter of the extent of the model (computed separately for every axis), and consequently can bound the error to $(1/4) * 1/(2^{16}) = 0.000004\%$ of the model extent. At this precision, we use either 8 or 16 bit values (many relative values are small, and consequently 8 bit or less are sufficient).

Another method further reduces storage consumption: A special bit code indicates if the difference to the parent's ordinate is zero. In this case the specification of the 16 bit delta value can be omitted ("null" encoding). Often CAD models have edges aligned to the axises of the coordinate system, so this is frequently the case.

Note that while the use of fixed precision for relative encoding makes the compression scheme lossy, the inaccuracies introduced can be controlled by the user by selecting the fraction of the model extent which is to be considered as the neighborhood of parent vertices.

The distinction between the encoding variants is made by variable length tags. Table 5 gives an overview of coordinate encoding.

Coordinate	Tag	Fields
relative16	0	16 bit fixed
relative8	10	8 bit fixed
null	110	(none)
absolute	111	32 bit float

Table 6: Protocol for encoding of coordinates

7.6 Model reconstruction and rendering

Model reconstruction

At the receiver's side, the geometric model must be reconstructed from the data stream. The cluster tree is de-linearized by successive refinement operations (i. e. node expanding) operations, which create child nodes from the data fields of the network packages. At the same time, a vertex list and triangle list can be reconstructed. The reconstruction process is incremental and fast, which allows to perform decoding and rendering in parallel, always displaying the best approximation possible with the data received so far.

Rendering

The representation of the model as a cluster tree allows more than one way of rendering. The more conventional approach is to take "snapshots" of the reconstructed triangle list after a certain amount of data has arrived, thereby obtaining conventional, discrete levels of detail. In that case, the reconstruction of the cluster tree can be omitted. The advantage is that the resulting LODs can be used by an existing LOD renderer without any modification.

Interactive selection of smooth LODs

During initialization, the cluster tree makes it possible to select smooth levels of detail on the fly during rendering, which is a more powerful method than simply creating a small set of LODs. In an initial step, the LOD selection operation is used to create a triangle list for display.

For every successive frame, a new threshold is chosen according to the new viewpoint, and the corresponding smooth LOD is selected. Depending on whether the new LOD is finer or coarser than the previous one, refinement or simplification operations are used to modify the active node set, the vertex and triangle lists. Usually only few manipulations are necessary, so the incremental LOD selection runs at interactive speed.

The selection of smooth LODs from the cluster tree also works if the transmission is still incomplete, because every partially created tree is consistent in the set of vertices, triangles and clusters. As soon as new data arrives and is inserted into the tree, the model can be refined to incorporate the new data, if desired.

Variable resolution within one object

The comparison of the cluster size against the threshold can also be made by estimating the cluster's projected screen size. This allows to make a different selection for every node, depending on the distance of the cluster to the observer. The displayed model allows non-uniform simplification and automatically adapts to the user's position. Those parts of the object that are further away from the observer will be displayed coarser than those that are near. Consequently, the polygon budget is exploited more efficiently. However, neither cluster size nor update list can be precomputed any more, but the incurred performance penalty can be kept within tolerable limits. This area is subject to further work.

7.7 Results and Comparison

Comparison of model sizes

Table 7 compares the sizes of models encoded as a smooth LOD packet stream as detailed in section 7.4 to the original models (vertex list and triangle list) with and without levels of detail (see Figure 37 for images). Every model is listed with its vertex and triangle count, the original object size, computed from 12 byte per vertex and 6 byte per triangle, assuming 16 bit indices for vertex references in triangles).

Model name	# of vertices	# of triangles	object size	LOD size	smooth LOD size	% of obj.size	% of LOD size
lamp	584	1352	13968	17712	6106	43.7	34.5
tree	718	1092	15168	20460	7288	48.0	35.6
shelf	1239	2600	30228	37188	12635	41.8	34.0
plant	8228	13576	179352	200154	89921	50.1	44.9
stool	1024	1600	21864	30528	8406	38.4	27.5
tub	3422	5404	73488	84906	26993	36.7	31.8
sink	2952	4464	62208	81558	23743	38.2	29.1
ball	1232	2288	28512	39420	14099	49.4	35.8
curtain	4648	8606	107412	109770	44334	41.3	40.4

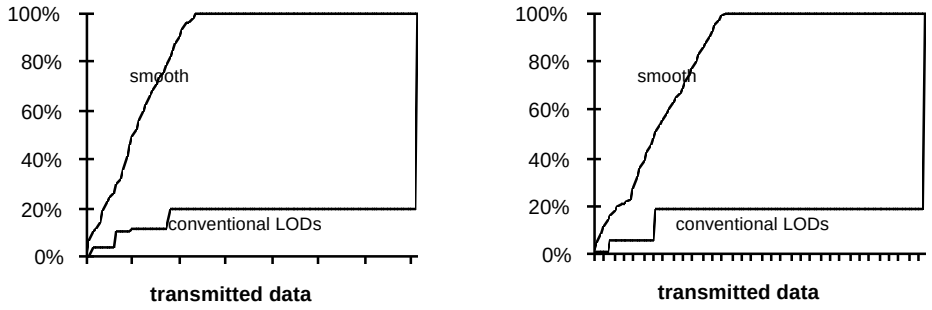
Table 7: Comparison of model sizes - smooth LODs against conventional models (sizes given in bytes)

The next column (*LOD size*) lists the size of the model with 5 conventional LODs including the original object (additional LODs only increase the triangle count, vertices are reused from the original model with the approach described in section 7.3!). These values should be compared to the size of the corresponding smooth LOD model (*smooth LOD size*), stored in the format given in section 7.5. The size of the smooth LOD model is also given as a percentage of the original model (*% of obj. size*) and level of detail model (*% of LOD size*).

Note that the smooth LOD model is always not only significantly smaller than the level of detail model, but also smaller than the original model. As far as model size is concerned, smooth LODs come for free!

Comparison of the visual effect

Our experience shows that the refinement of a model with smooth LODs is superior to the coarse-grained switching between a few (typically 3-6) conventional LODs. However, such a subjective statement is hard to prove formally. If we assume that image quality is roughly proportional to the number of triangles used for display, we can compare smooth to conventional LODs by plotting triangles available for rendering as a function of transmitted bytes for both methods. Figure 35 shows two such examples.



(a) shelf (b) plant

Figure 35: Comparison of visual effect of smooth vs. conventional LODs. We measured the quality as the number of triangles available for a certain amount of data (1 notch on the x-axis ≈ 5 KB)

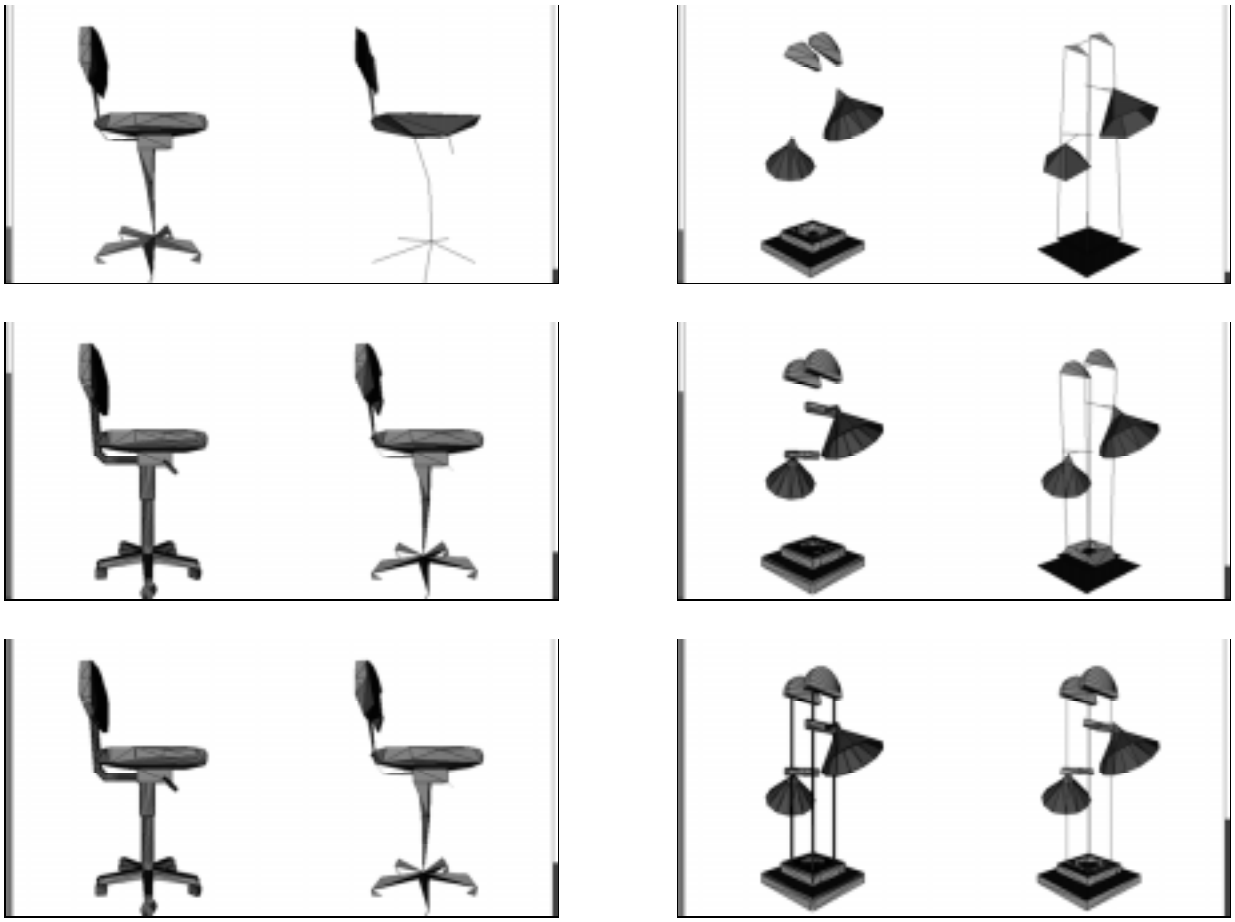


Figure 36: Comparison of the development stages of sample objects. The half of each image shows smooth LODs, the right half conventional LODs for corresponding amounts of data. Black bars on the sides of the images indicate the amount of triangles received and displayed.

The maximum triangle count is reached much earlier using the smooth LODs than using conventional LODs because of the smooth LODs' more compact representation (see the *% of obj. size* column in Table 7). This difference is also obvious when comparing the obtained images. (compare Figure 36).

Note that the roughly linear correspondence between transmitted data (x-axis) and available triangles (y-axis) is very suitable for networked virtual environments, where an object is approached at constant velocity, while its geometric representation is still being transmitted over a network of constant bandwidth.

7.8 Using smooth LODs with demand-driven geometry transmission

Smooth LODs have a number of properties that makes them extremely well suited for use in distributed virtual environments. In particular, they can easily be integrated with demand-driven geometry transmission as presented in chapter 5. Instead of downloading separate and discrete models for each level of detail of an object, the demand-driven geometry transmission algorithm simply processes a smooth level of detail stream. There is only one minor modification required: Level of detail selection algorithm are designed to pick one from a few - not more than 10 - discrete representations. Smooth levels of detail have a large number (often in the thousands) of possible levels of detail. While any selection algorithm principally works with such a large number of distinct representations as well, the associated overhead of issuing requests for small items of geometry data is too large. The advantage of a continuous stream would be destroyed by dividing the transmission into small chunks. Therefore, it is better to treat the smooth LODs model like a conventional LOD model by selecting a few of the possible models from the continuous stream and compute the selected LOD in terms of their cost/benefit ratio. Geometry requests are only issued for these selected LODs. The purpose of the continuous stream is not defeated in this way, since progressive transmission and decoding can proceed as usual, it only stops when the desired quality is reached. The only disadvantage is that a slightly better or worse than optimal LOD (according to the cost/benefit heuristic) may be chosen, which - if done properly - cannot be perceived by the observer.

7.9 Summary

We have presented a new polygonal model representation called smooth LODs designed for interactive rendering and transmission in networked systems. A hierarchical clustering method which has been used to compute conventional simplifications of triangle meshes is extended to yield a continuous stream of approximations of the original model. A very large, practically continuous number of levels of detail is computed. The result can be represented in an extremely compact way by relative encoding. The resulting data set is smaller than the original models without levels of detail. If the data set is transmitted over a network, a useful representation is available at any stage of the data transmission. The data set can be used to compute conventional levels of detail, or the underlying hierarchical structure can be exploited to generate and incrementally update any desired approximation for rendering at runtime.

When running real world applications on low cost systems, the constraint of using a coarse LOD only if the difference to the high fidelity model is not noticeable is regularly violated because of insufficient rendering performance (see Figure 36). Slow network connections such as Internet downloads make the user wait for completion of transmission while the model is already displayed at

full screen resolution. In these situations, our approach is clearly superior, because it makes new data immediately visible (compare Figure 35) and finishes earlier due to its compact representation.

Figure 37: Models used for evaluation

8. Interactive Rendering of Natural Phenomena Using Directed Cyclic Graphs

8.1 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 [2]. In combination with texture mapping [21], level of detail (LOD) rendering [23], and scene culling [24], 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 [24], 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 [16].

A solution to these problems lies in the use of procedural modeling and fractal geometry [49]. 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 [57]. 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 [80].

Numerous methods for modeling and rendering of plants have been presented in the past, e. g. [28, 57, 58, 80, 88], but most are aimed at photorealism 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 [59], but they cannot make use of polygonal rendering hardware. Special support for real-time applications with level of detail rendering is presented in [88], but the approach is also not storage preserving.

8.2 Overview of our approach

In this work, we present a method for interactive rendering of natural phenomena modeled using directed cyclic graphs. This method is an adaptation of the raytracing work by Gervautz and Traxler [28]. In the domain of interactive rendering, we can make use of some unique properties if directed cyclic graphs:

- *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.
- *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 instantiated. 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 at close-up view is solved by providing actual geometric detail, but without the penalty of elevated memory requirements. Inefficient rendering can be prevented by still using levels of detail and impostors at medium and far ranges, where quality degradation cannot be perceived.

The remainder of this chapter discusses the details of our approach: Section 8.3 gives background on the rendering of directed cyclic graphs. Section 8.4 pays attention to the issue of efficient rendering. The discussion is complemented by details about a sample implementation using Open Inventor (section 8.5), followed by examples and results (section 8.6).

8.3 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.

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 38 shows a very 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)

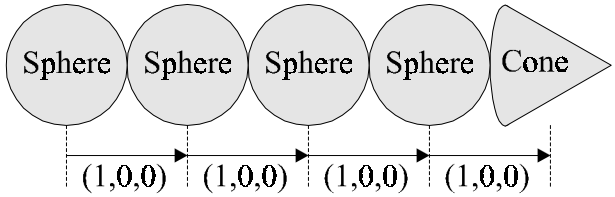


Figure 38: A very simple recursive model

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.

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 $A \rightarrow \ldots A \ldots$) 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 38 is transformed into the graph in Figure 39.

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. 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.

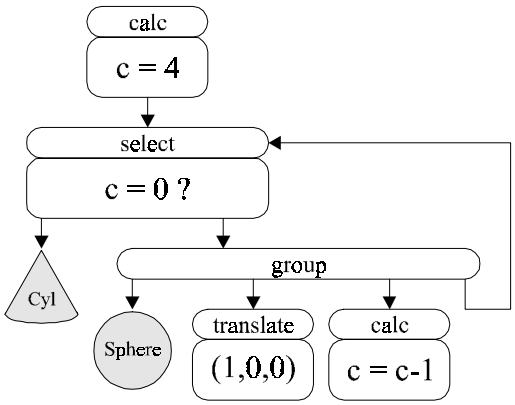


Figure 39: Simple recursive graph for the model shown in Figure 38

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 (Picture 2). 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.

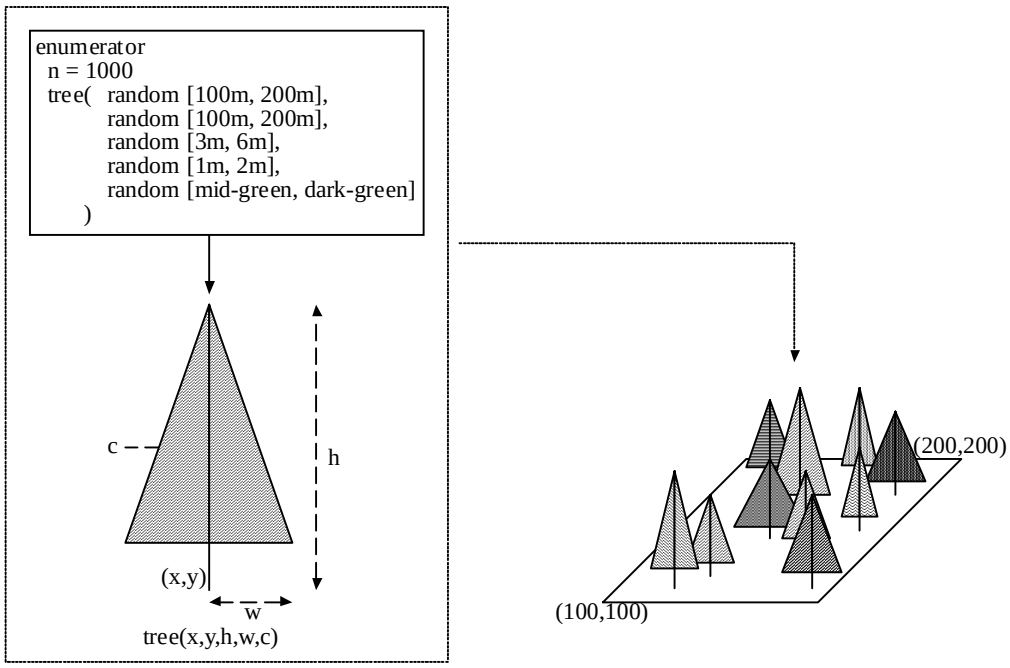


Figure 40: Creating a forest by varying a tree

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 40 (a more realistic scenario would use a small set of different tree species).

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 instantiated 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.

8.4 Efficiency of rendering

Levels of Detail

DCGs are particularly suited for level of detail rendering because of their self-similarity: Reducing recursion depth yields an object that resembles the original object with fewer detail. This strategy requires only the modification of a single initialization parameter instead of a complicated simplification process, and only one model is required for multiple levels of detail. For example, Picture 4 shows a simple sympodial branching structure. Note that the number of geometric primitives increases exponentially. However, one should note that more sophisticated structures do not show plain self-similarity that allows such a trivial level of detail configuration. 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.

Dynamic Impostors

While detailed geometry is important at close range, image-based simplifications can be very efficient at medium and far ranges. The use of dynamic impostors 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 impostors. When an objects comes into close range, the rendering automatically switches to rendering detailed geometry from the original procedural model, so that no degradation in quality can be perceived. Since typically only few objects are close to the observer, performance goals can be met.

8.5 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 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 [82] 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 has a well readable text-based file format. This file format was extended to accommodate the extensions for modeling directed cyclic graphs, in particular the possibility to perform computations on the variables (RecursionCalculator). Inventor's reference mechanism (DEF/USE) is employed for the cyclic references. The following is the RECURSIV file for the example from Figure 38.

```
RecursionCalculator { expression "c=4" } #init counter
DEF Worm RecursionSwitch {   #label called recursively
  expression "c>0"     #decide which child to traverse
  Cone {}                              #terminal child
  Separator {                        #recursive branch
    Sphere {}
    Translation { translation 1 0 0 }
    RecursionCalculator { expression "c=c-1" }
    RecursionSeparator { USE Worm }     #recursive call
  }
}
```

8.6 Results

While modeling DCGs takes a little practice, we found that it is possible to achieve very appealing results. Some examples are shown in the color section at the end of the text, including trees, terrain and linear fractals (Picture 3).

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 (Picture 5) 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 2 (Picture 6, second from left) consists of 211 cylinders and 15600 cones. The RECURSIV file uses 7418 bytes, while the binary file consumes 442708 bytes.
- Terrain (Picture 7d) 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 (such as seen in the color) plate 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 details are generated from random numbers.

8.7 Summary

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.

9. Conclusions

9.1 Critique of the Remote Rendering Pipeline

This thesis has presented a framework, called the remote rendering pipeline, aimed at the improvement of performance of larger scale distributed virtual environments. The work introduces a number of methods, that taken together allow more efficient management of geometric data in a distributed virtual environment. These building blocks are

- a strategy for distributing data over the network on demand (demand-driven geometry transmission), thus saving the effort for unneeded data;
- a tool for the generation of conventional levels of detail from the popular VRML file format, to help the integration of standard Internet techniques with the virtual environment;
- a data structure for smooth levels of detail that allow to discontinue geometric object transmission and reconstruction at any point in the data stream at the expense of quality, and also yields a significant compression ratio over raw data;
- a modeling and rendering toolkit based on directed cyclic graphs that integrates with conventional modeling approaches and allows extremely compact rule-based descriptions of natural phenomena such as trees or mountains.

A central concept is demand-driven geometry transmission. Data of geometric objects is sent to the users only as necessary, assuming a constant area of interest containing all the objects that the user can see. Such a strategy makes virtual worlds scaleable to almost arbitrary size, since the limiting factor is no longer the storage and rendering capacity of the client, but rather the data management capability of the server from which the data is requested.

Since the server by itself can be a distributed system, free scalability can be achieved. The demand-driven geometry transmission protocol itself is simple and readily implemented. It also makes the clients relatively independent from the virtual world, since the protocol allows the client to decide on the strategy for selecting the geometric objects, so a client can adjust the data rate according to its own capacities.

Limitations of this approach include the assumption that the user’s behavior is relatively static (user does not move faster than a given threshold, which is small compared to the size of the virtual world), such as often found in vehicle simulation. Given such a behavior, not only demand-driven geometry transmission but also dead reckoning methods can be used. Demand-driven geometry transmission also works best in open spaces, such as outdoor scenarios. Although indoor worlds work with the approach, performance could be improved more easily by using visibility computation.

Furthermore, the method requires that the world is modeled as a collection of discrete objects. This makes sense if one also wants to attribute objects with behaviors. However, often scenes or objects are only available as unstructured collections of polygons. Very large and extended objects such as buildings and roads that stretch through a large portion of the area of interest also break the elegance of the algorithm. Finally, the mediation of all communication by a server introduces some latency into the system.

Overall, demand-driven geometry transmission has proven to be a viable concept, and is certainly a useful technique for future large-scale virtual environments, as acknowledged by recent commercial trends.

The level of detail generator Lodestar was implemented to help the integration of various pieces of the system. It demonstrates some important lessons learned when attempting to create such a tool for use with real world data rather than the somewhat artificial examples often used as test cases for new algorithms. Its worth and novelty lie in the use of octree quantization for vertex clustering, and in the robust handling of all sorts of input data. It is used in the organization of data for the virtual environment.

Smooth levels of detail have turned out to be a surprisingly simple, yet very effective method to generate progressive approximations of polygonal objects. Among the benefits of this approach are the simple implementation, robustness against degenerate data, high compression ratio and of course all the virtues of progressive encoding, transmission, reconstruction, and rendering. Similar strategies were published concurrently, and while they may be superior in certain qualitative domains (such as guaranteed error bounds), no rival algorithm matches the strategy in simplicity and effectiveness. Smooth levels of detail are also easily integrated with demand-driven geometry transmission into a powerful system for the management of graphical database state.

Modeling and rendering of natural phenomena with directed cyclic graph is maybe the most controversial of the proposed techniques. While the fictious „compression ratio“ of a model represented as a DCG compared over its conventional counterpart is overwhelming, one may also not overlook the shortcomings of the approach. On the side of the benefits, we can list that the approach allows for a very broad class of procedural models to be represented, and it also appears to be the only totally unified modeling framework for procedural and non-procedural models. However, modeling of DCGs takes a lot of effort and requires understanding of the underlying model. The full use of the approach only becomes apparent in the presence of excessive detail, but then the performance penalty may be significant. Methods based on levels of detail and impostors can help to control that, but the question remains if systems trimmed to the highest performance will ever be able to tolerate the overhead involved in evaluating procedural models in real time. However, DCGs are only one way of procedural computer graphics models. The more fundamental idea of using procedural models to transmit instructions how to generate geometry rather than the geometry itself is beneficial in any case, in particular with the rising popularity of interpreted cross-platform languages such as Java.

9.2 Future work

The virtual environment used as a testbed for the ideas that have been presented in this thesis is continually evolving, but a lot of fine-tuning is still required for optimal performance.

Future work will extend the system to support a network of servers, in order to demonstrate the feasibility of a very large virtual environment and move closer towards the vision of building Cyberspace.

10. References

[1] AIREY J., J. ROHLF, F. BROOKS Jr., Towards Image Realism with Interactive Update Rates in Complex Virtual Building Enviroments, Computer Graphics, Vol. 24, No. 2, pp. 41 (1990).

[2] AKELEY K., RealityEngine Graphics, Computer Graphics (Proceedings SIGGRAPH), pp. 109-116 (1993).

[3] BELL G., R. CARRY, C. MARRIN, VRML 2.0 final specification, http://vag.vrml.org/VRML2.0/FINAL/ (1996).

[4] BISHOP G., H. FUCHS, MCMILLAN L., SCHER ZAGIER E., Frameless Rendering: Double Buffering Considered Harmful, Proceedings of SIGGRAPH'94, pp. 175-176 (1994).

[5] BLANCHARD C., S. BURGESS, Y. HARVILL, J. LANIER, A. LASKO, M. OBERMAN, M. TEITEL, Reality Built for Two: A Virtual Reality Tool, SIGGRAPH Symposium on 3D Interactive Graphics, pp. 35-38 (1990).

[6] BLAU B., HUGHES C. E., MOSHELL J. M., LISLE C., Networked virtual environments, SIGGRAPH Symposium on Interactive 3D Graphics, pp. 157-160 (1992).

[7] BRICKEN W., G. COCO, The VEOS Project, Presence, Vol. 3, No. 2, pp. 111-129 (1994).

[8] CALVIN J., A. DICKEN, B. GAINES, P. METZGER, D. MILLER, D. OWEN, The SIMNET virtual world architecture. Proceedings of Virtual Reality Annual International (VRAIS'93), pp. 450-455 (1993).

[9] CARLSSON C., O. HAGSAND, DIVE - A platform for multi-user virtual environments, Computers and Graphics, Vol. 17, No. 6, pp. 663-669 (1993).

[10] CERTAIN A., J. POPOVIC, T. DEROSE, T. DUCHAMP, D. SALESIN, W. STUERZLE, Interactive Multiresolution Surface Viewing, Proceedings of SIGGRAPH'96, pp. 91-98 (1996).

[11] CHEN S., L. WILLIAMS, View interpolation for Image Synthesis, Computer Graphics (Proceedings SIGGRAPH), pp. 279-288 (1993).

[12] CHEN S., Quick Time VR - An Image-Based Approach to Virtual Environment Navigation, Computer Graphics (Proceedings SIGGRAPH), pp. 29-38 (1995).

[13] CLARK J., Hierarchical Geometric Models for Visible Surface Algorithms, Communications of the ACM, Vol. 19, No. 10, pp. 547-554 (1976).

[14] COORG S., S. TELLER, A Spatially and Temporally Coherent Object Space Visibility Algorithm, Technical Report TM-546, Laboratory for Computer Science, MIT (1996).

[15] DEERING M., Data Complexity for Virtual Reality: Where do all the Triangles Go? Proceedings of of Virtual Reality Annual International Symposium (VRAIS'93), pp. 357-363 (1993).

[16] DEERING M., Geometry Compression, Computer Graphics (Proceedings SIGGRAPH), pp. 13-20 (1995).

[17] DE FLORIANI L., P. MARZANO, E. PUPPO, Multiresolution Models for Topographic Surface Description, The Visual Computer, Vol.12, No.7, Springer International, pp.317-345 (August 1996).

[18] DEHAEMER M., M. ZYDA, Simplification of Objects Rendered by Polygonal Approximations, Computers & Graphics, Vol. 15, No. 2, pp. 175-184 (1991).

[19] ECK M., T. DEROSE, T. DUCHAMP, H. HOPPE, M. LOUNSBERY, W. STUETZLE, Multiresolution Analysis of Arbitrary Meshes, Computer Graphics (Proceedings SIGGRAPH), pp. 173-182 (1995).

[20] FALBY J., M. ZYDA, D. PRATT, L. MACKEY, NPSNET, Hierarchical data structures for realtime 3-dimensional visual simulation, Computers & Graphics, Vol. 17, No. 1, pp. 65 (1993).

[21] FOLEY J., A. VAN DAM, S. FEINER, J. HUGHES, Computer Graphics, Principles and Practice, Addison-Wesley Publishing Co. (1990).

[22] FUNKHOUSER T., C. SEQUIN, S. TELLER, Management of Large Amounts of Data in Interactive Building Walkthroughs, SIGGRAPH Symposium on Interactive 3D Graphics, pp. 11-20 (1992).

[23] FUNKHOUSER T., C. SEQUIN, Adaptive Display Algorithm for Interactive Frame Rates During Visualisation of Complex Virtual Environments, Computer Graphics (Proceedings SIGGRAPH), pp. 247-254 (1993).

[24] FUNKHOUSER T., RING - A Client-Server System for Multi-User Virtual Environments, SIGGRAPH Symposium on Interactive 3D Graphics, pp. 85-92 (1995).

[25] FUNKHOUSER T., Network Topologies for Scalable Multi-User Virtual Environments, Proceedings of VRAIS'96, Santa Clara CA, pp. 222-229 (1996).

[26] GELERNTER D., Mirror Worlds, Oxford University Press (1992).

[27] GERVAUTZ M., W. PURGATHOFER, A simple method for color quantization: octree quantization, New Trends in Computer Graphics, Proceedings of Computer Graphics International'88, p. 219-231, Springer, Geneva 1988.

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

[29] GHEE S., J. NAUGHTON-GREEN, Programming Virtual Worlds, SIGGRAPH'94 Course Notes, No. 17 (1994).

[30] GORTLER S., R. GRZESZCZUK, The Lumigraph, Computer Graphics (Proceedings SIGGRAPH), pp. 43-54 (1996).

[31] GREENE N., M. KASS, G. MILLER, Hierarchical Z-Buffer Visibility, Computer Graphics (Proceedings SIGGRAPH), pp. 231-237 (1993).

[32] GRÖLLER E., Coherence in Computer Graphics, PhD dissertation, Vienna University of Technology (1993).

[33] HARDENBERG J., G. BELL, M. PESCE, VRML: Using 3D to surf the Web, SIGGRAPH'95 Course Notes, No. 12 (1995).

[34] HE T., L. HONG, A. KAUFMAN, A. VARSHNEY, S. WANG, Voxel Based Object Simplification, Proc. SIGGRAPH Symposium on Interactive 3D Graphics, pp. 296-303 (195).

[35] HELMAN J., Designing VR Systems to Meet Psysio- and Psychological Requirements, SIGGRAPH Course Notes, No. 23 (1993).

[36] HOPPE H., T. DEROSE, T. DUCHAMP, J. MCDONALD, W. STUETZLE, Mesh Optimization, Computer Graphics (Proceedings SIGGRAPH), pp. 19-26 (1993).

[37] HOPPE H., Progressive meshes, Proceedings of SIGGRAPH '96, pp. 99-108 (1996).

[38] ID SOFTWARE, DOOM, Computer game (1994).

[39] SUN MICROSYSTEMS, Java homepage. http://www.javasoft.com/ (1996).

[40] KAZMAN R., Making WAVES: On the design of architectures for low-end distributed virtual environments, Proceedings Virtual Reality Annual International Symposium (VRAIS'93), pp. 443-449 (1993).

[41] LEVOY M., Polygon-Assisted JPEG and MPEG Compression of Synthetic Images, Computer Graphics (Proceedings SIGGRAPH), pp. 21-25 (1995).

[42] LEVOY M., P. HANRAHAN, Light Field Rendering, Computer Graphics (Proceedings SIGGRAPH), pp. 31-42 (1996).

[43] LINDSTROM P., D. KOLLER, W. RIBARSKY, L. HODGES, N. FAUST, G. TURNER, Real-Time Continuous Level of Detail Rendering of Height Fields, Computer Graphics (Proceedings SIGGRAPH), pp. 109-118 (1996).

[44] LOUNSBERY M., T. DEROSE, J. WARREN, Multiresolution analysis for surfaces of arbitrary topological type, ACM Transactions on Graphics, Vol. 16, No. 1, pp. 34-73 (Jan. 1997).

[45] LUEBKE D., C. GEORGES, Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets, Proceedings SIGGRAPH Symposium on Interactive 3D Graphics, pp. 105-106 (1995).

[46] MACEDONIA M., M. ZYDA, D. PRATT, P. BARHAM, S. ZESWITZ, NPSNET: A Network Software Architecture for Large-Scale Virtual Environment, Presence, Vol. 3, No. 4, pp. 265-287 (1994).

[47] MACEDONIA M., M. ZYDA, D. PRATT, D. BRUTZMAN, P. BARHAM, Exploiting Reality with Multicast Groups: A Network Architecture for Large-scale Virtual Environments, Proceedings of of Virtual Reality Annual International Symposium (VRAIS'95) (1995).

[48] MACIEL P., P. SHIRLEY, Visual Navigation of Large Environments Using Textured Clusters, SIGGRAPH Symposium on Interactive 3-D Graphics, pp. 95-102 (1995).

[49] MANDELBROT B., The fractal geometry of nature, Freeman & Co. (1982).

[50] MAZURYK T., D. SCHMALSTIEG, M. GERVAUTZ, Zoom Rendering: Improving 3-D Rendering with 2-D Operations, The International Journal of Virtual Reality, Vol 2, No. 2 (1997).

[51] MCMILLAN L., G. BISHOP, Plenoptic Modeling: An Image-Based Rendering System, Computer Graphics (Proceedings SIGGRAPH), pp. 39-46 (1995).

[52] NARKHEDE A., D. MANOCHA, Fast Polygon Triangulation Based On Seidel's Algorithm, Graphics Gems V, Academic Press, pp. 394-397 (1995).

[53] NAYLOR B., Interactive playing with large synthetic environments, SIGGRAPH Symposium on Interactive 3D Graphics (1995).

[54] NEIDER J., DAVIS T., WOO M., OpenGL Programming Guide - The Official Guide to Learning OpenGL, Addison-Wesley Publishing Company (1993).

[55] PAJON J.-L., Y. COLLENOT, X. LHOMME, N. TSINGOS, F. SILLION, P. GUILOTEAU, P. VUYSLTEKER, G. GRILLON, D. DAVID, Building and Exploiting Levels of Detail, An Overview and Some VRML Experiments, Proceedings of VRML'95, San Diego CA, pp. 117-122 (1995).

[56] PESCE M., VRML - Browsing & Building Cyberspace, New Riders, Indianapolis (1995).

[57] PRUSINKIEWICZ P., A. LINDENMAYER, The algorithmic beauty of plants, Springer, New York (1990).

[58] PRUSINKIEWICZ P., JAMES M., MÉCH R., Synthetic Topiary, Computer Graphics (Proceedings SIGGRAPH) (1994).

[59] REEVES W., Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems, Computer Graphics (Proceedings SIGGRAPH), pp. 313 (1985).

[60] REGAN M., R. POST, Priority Rendering with a Virtual Reality Address Recalculation Pipeline, Computer Graphics (Proceedings SIGGRAPH),pp. 155-162 (1994).

[61] ROEHL B., Distributed Virtual Reality - An Overview, Proceedings of SIGGRAPH VRML'95 Symposium (1995).

[62] ROHLF J., J. HELMAN, IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics, Proceedings of SIGGRAPH'94, pp. 381 (1994).

[63] RONFARD R., J. ROSSIGNAC, Full-range Approximation of Triangulated Polyhedra, Proceedings of EUROGRAPHICS'96, pp. 67-76 (1996).

[64] ROSSIGNAC J., P. BORREL, Multi-Resolution 3D Approximation for Rendering Complex Scenes, IFIP TC 5.WG 5.10 II Conference on Geometric Modeling in Computer Graphics (1993).

[65] ROST, J. FRIEDBERG, P. NISHIMOTO, PEX: A Network-Transparent 3D Graphics System, Computer Graphics & Applications, Vol. 9, No. 4, pp. 14-26 (1989).

[66] SCHAUFLER G., W. STÜRZLINGER, Generating Multiple Levels of Detail from Polygonal Geometry Models, Virtual Environments'95 (ed. M. Göbel), Springer Wien-New York, pp. 33-41 (1995).

[67] SCHAUFLER G., Exploiting Frame to Frame Coherence in a Virtual Reality System, Proceedings of Virtual Reality Annual International Symposium (VRAIS'96), pp. 95-102 (1996).

[68] SCHAUFLER G., T. MAZURYK, D. SCHMALSTIEG, High Fidelity for Immersive Displays, Short paper, ACM SIGCHI'96 conference companion. Also technical report TR-186-2-96-02, Vienna University of Technology, Austria (1996).

[69] SCHAUFLER G., W. STÜRZLINGER, A Three-Dimensional Image Cache for Virtual Reality, Proceedings of EUROGRAPHICS'96, pp. 227-236 (1996).

[70] SCHMALSTIEG D., M. GERVAUTZ, Towards a Virtual Environment for Interactive World Building, Proceedings of the GI Workshop on Modeling, Virtual Worlds, Distributed Graphics (MVD'95), Bonn, pp. 49-56 (Nov. 1995).

[71] SCHMALSTIEG D., M. GERVAUTZ, P. STIEGLECKER, Optimizing Communication in Distributed Virtual Environments by Specialized Protocols, Virtual Environments and Scientific Visualization'95 (ed. M. Göbel), Springer Wien-New York, pp. 1-10 (1995).

[72] SCHMALSTIEG D., M. GERVAUTZ, Implementing Gibsonian Virtual Environments, Proceedings of the Thirteenth European Meeting on Cybernetics and Systems Research, Vienna, Austria, April 1996. Also appeared in: Cybernetics and Systems - An International Journal (ed. R. Trappl), Vol. 27, No. 6, pp. 527-540, Taylor & Francis, Washington DC (1996).

[73] SCHMALSTIEG D., R. TOBLER, Exploiting Coherence in 2 1/2 D Visibility Computation, Computers & Graphics, Vol. 21, No. 1, pp. 121-123 (1997).

[74] SCHROEDER W., J. ZARGE, W. LORENSEN, Decimation of Triangle Meshes, Proceedings of SIGGRAPH'92, pp. 65-70 (1992).

[75] SHADE J., D. LISCHINSKI, D. SALESIN, T. DEROSE, J. SNYDER, Hierarchical Image Caching for Accelerated Walkthruoghs of Complex Environments, Computer Graphics (Proceedings SIGGRAPH), pp. 75-82 (1996).

[76] SHAW C., J. LIANG, M. GREEN, Y. SUN, The Decoupled Simulation Model for VR Systems, Proceedings of SIGCHI, pp. 321-328 (1992).

[77] SHAW C., M. GREEN, J. LIANG, Y. SUN, Decoupled simulation in virtual reality with the MR toolkit, ACM Transactions on Information Systems, Vol. 11, No. 3, pp. 287-317 (1993).

[78] SINGH G., L. SERRA, W. PNG, Hern NG, BrickNet: A Software Toolkit for Network-Based Virtual Worlds, Presence, Vol. 3, No. 1, pp. 19-34 (1994).

[79] SINGHAL S., D. CHERITON, Exploiting Position History for Efficient Remote Rendering in Networked Virtual Reality, Presence, Vol. 4, No. 2, pp. 169-194 (1995).

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

[81] SNOWDON D., A. WEST, AVIARY: Design Issues for Future Large-Scale Virtual Environments, Presence, Vol. 3, No. 4, pp.288-308 (1994).

[82] STRAUSS P., R. CAREY, An Object Oriented 3D Graphics Toolkit, Computer Graphics (Proceedings SIGGRAPH), Vol. 26, No. 2, pp. 341 (1992).

[83] SUDARSKY O., C. GOTSMAN, Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality, Proceedings of EUROGRAPHICS'96, pp. 249-258 (1996).

[84] TAUBIN G., J. ROSSIGNAC, Geometric Compression Through Topological Surgery, Technical Report RC-10340, IBM T. J. Watson Research Center (1996).

[85] TELLER S., C. SÉQUIN, Visibility Preprocessing For Interactive Walktroughs, Computer Graphics (Proceedings SIGGRAPH), Vol. 25, No. 4, pp. 61-69 (1991).

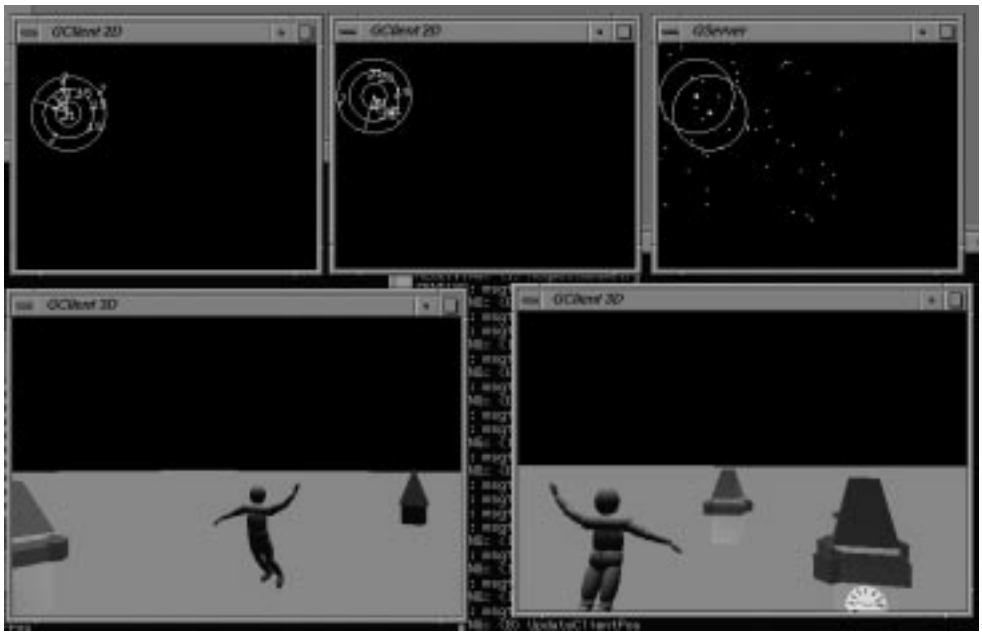
[86] TORBORG J., J. KAJIYA, Talisman: Commodity Realtime 3D Graphics for the PC, Computer Graphics (Proceedings SIGGRAPH), pp. 353-364 (1996).

[87] TURK G., Re-Tiling Polygon Surfaces, Computer Graphics (Proceedings SIGGRAPH), pp. 55-64 (1992).

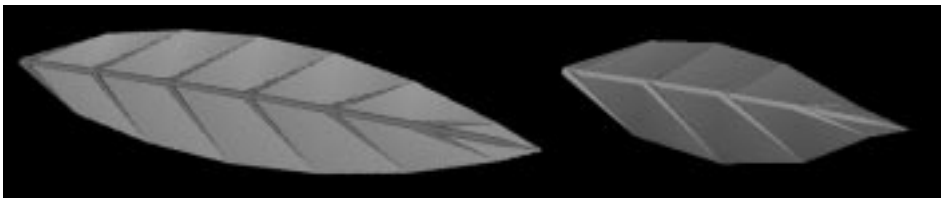
[88] WEBER J., J. PENN, Creation and Rendering of Realistic Trees, Computer Graphics (Proceedings SIGGRAPH), pp. 119-128 (1995).

[89] YAGEL R., W. RAY, Visibility Computation for Efficient Walkthrough of Complex Environments, Presence, Vol. 5, No. 1, pp. 45-60 (1995).

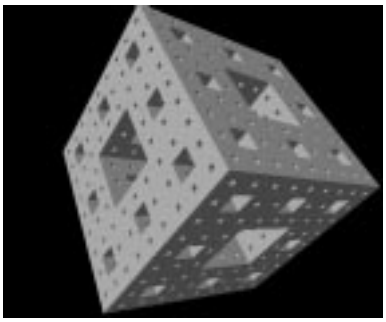
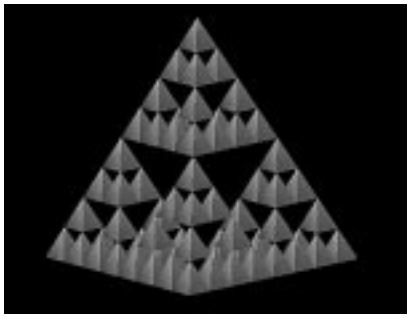
Appendix: Pictures



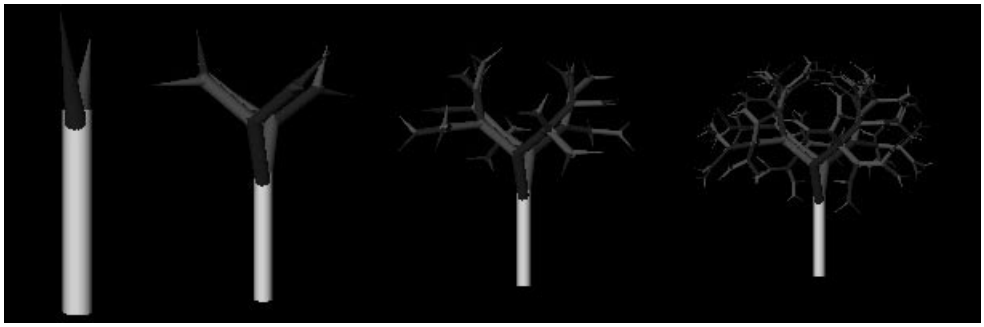
Picture 1: Demand-driven geometry transmission system view (server and two clients)



Picture 2: Parameters allow to model very different leaves from one model

(a)  (b) 

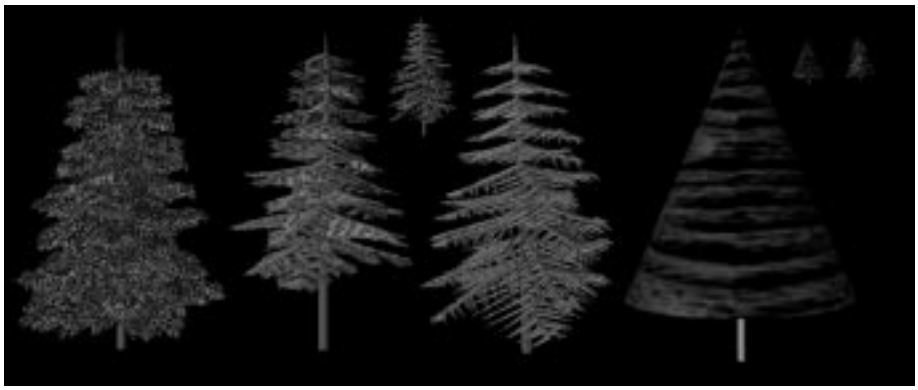
Picture 3: Linear fractals can easily be modeled as directed cyclic graphs



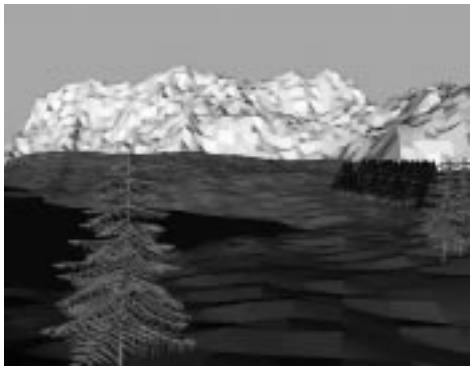
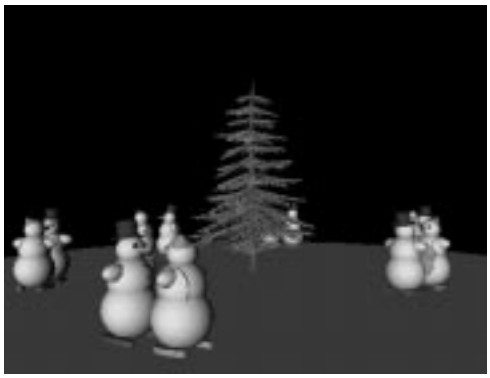
Picture 4: Recursion depth controls the development of a sympodial branching structure

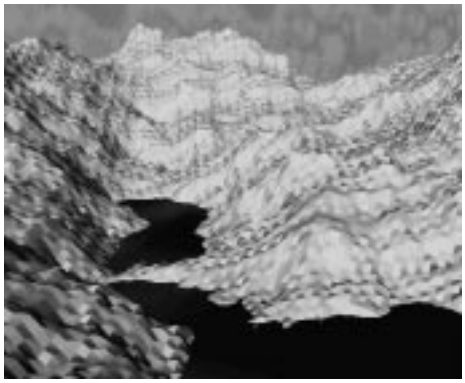
(a)  (b) 

Picture 5: A monopodial tree at normal distance and a close-up view



Picture 6: Levels of detail for a tree are modeled by modifying sub-graphs

(a)  (b) 

(c)  (d) 

Picture 7: Detailed and diverse scenes can be modeled using directed cyclic graphs