Proceedings WAE'98, Saarbrücken, Germany, August 20-22, 1998
Ed. Kurt Mehlhorn, pp. 1-3

A Network Based Approach for Realtime Walkthrough of Massive Models*

Matthias Fischer, Tamás Lukovszki, and Martin Ziegler
Heinz Nixdorf Institute and Department of Computer Science, University of Paderborn
Fürstenallee 11, 33102 Paderborn, Germany
e-mail: {mafi, talu, ziegler}@uni-paderborn.de

ABSTRACT

New dynamic search data structures developed recently guarantee constant execution time per search and update, i.e., they fulfil the real-time requirements necessary for interactive walkthrough in large geometric scenes. Yet, superiority or even applicability of these new methods in practice was still an open question.

Their prototypical implementation presented in this work uses common libraries on standard workstations and thus represents a first strut to bridge this gap. Indeed our experimental results give an indication on the actual performance of these theoretical ideas on real machines and possible bottlenecks in future developments. By special algorithmic enhancements, we can even avoid the otherwise essential preprocessing step.

1. Introduction

Scientific visualization as well as computer tomography in medical diagnosis, computer aided design (CAD), and architectural construction are examples of applications which display large geometric data (*virtual scenes*) by interactive user control. If this control supports arbitrary changes of the virtual camera's position and orientation, this process is called *walkthrough*. In *dynamic* walkthrough, the user can insert and delete objects in the scene (modification).

Today three-dimensional (*massive*) objects are most commonly represented by their boundary, approximated and decomposed into surface polygons enclosing the object. These polygons (usually cut further into triangles) are stored as their vertices' and edges' coordinates together with a normal vector, color information, and texture data.

From this model representation, the computer generates a picture by performing the steps of projection, hidden-surface removal, and shading [6]. This time consuming process is called *rendering* and supported by hardware (e.g., z-buffer algorithm). The cost for rendering can be estimated by $\mathcal{O}(n + a)$ [9] and depends on two parameters: the number n of polygons and the sum a over all pixels and all polygons needed for drawing these polygons (without considering their visibility). For simplification, we will regard n as dominant.

In order to get a movie-like smooth sequence of images, as well as responsive navigation, a fixed *frame rate* of at least $20fps$ (frames per second) has to be computed. This strict real-time condition raises severe problems, since even high end graphic systems cannot guarantee such rates for very large scenes ($n \approx 1,000,000$).

*Partially supported by EU ESPRIT Long Term Research Project 20244 (ALCOM-IT), and DFG Grant Me872/7-1.

The walkthrough problem: Let $C(M)$ denote the complexity of scene M in the above cost measure (number of polygons), and let C_0 be the maximum complexity for which some specific rendering machine can still guarantee $20fps$. Then the problem is to compute a model M_r with a complexity $C(M_r) \leq C_0$ for every interval of time t. For $20fps$, time intervals are as short as $50ms$.

This problem is difficult to solve because from the visitor's position, model M_r should resemble scene M as much as possible. Additional severity arises for very large scenes which do not fit into to main memory (swapping) where access to secondary storage media can easily spoil the real-time requirements, if not performed carefully.

General approach: Most graphic systems pursue a similar approach to this problem (they differ in the kind of approximation and their rendering strategies):

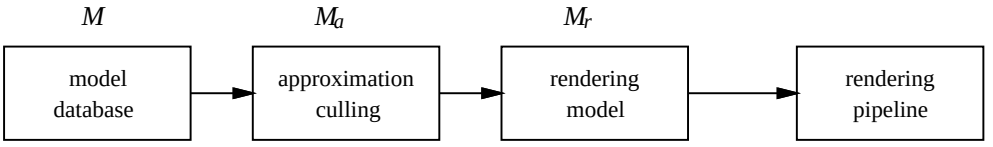

Figure 1: General approach

They compute a model M_a (See Fig. 1) including data structures for culling and polygonal model approximations of the scene M. Hence the complexity of M_a is larger than that of M. But during interactive walkthrough, the system needs only a part M_r of the scene M_a for an approximately correct view; e.g., large and very distant objects will be rendered using their approximations contained in M_a. Both approximations and culling information can be computed in a preprocessing phase (e.g., level of details models [7], visibility graph [12, 8]) or dynamically during walkthrough (e.g., repeated use of previously rendered objects via texture mapping [11] or visibility computations [3]).

The search problem: In this general approach, computation of the rendering poses the problem of deciding which objects are to be approximated (or to be rendered with full quality). For each frame, the system will have to search M_a for the sub-scene M_r presented to the visitor, choose – or even compute – approximations for all objects in M_r, and then render these approximations together with the rest of M_r.

This search problem is crucial for both visual impression and immediate responses to user interaction. In particular it should be really fast. Otherwise the costs for searching lets say some cluster of objects might consume most of the rendering time we intended to save by displaying approximations instead of the cluster itself.

In order to achieve high quality pictures, the processor should spend most of its time with rendering, not with searching.

[4] presents a very fast data structure for this goal. Other common approaches use a hierarchical structure based on ideas from Clark [2]. As an example consider the sequence of sub-scenes "world - country - town - house" organized in a tree. In existing systems, this concept is implemented with different data structure like BSP Trees (Binary Space Partitions) or Octrees.

Goals of this paper: This work is a major step towards a prototypic implementation of the ideas described in [5]. In particular, the authors' theoretical investigation on abstractly modeled dynamic walkthrough animation with several distributed visitors appears to yield a practically applicable system. In conjunction with methods from [4], this results in a network-based non-hierarchical data structure for solving the search problem in a scene.

Section 2 presents a brief description of this abstract animation system and its data structures. The current state of implementation is described in Section 3, and in Section 4 we describe an

evaluation of its performance. These experimental results are of importance to identify possible bottlenecks in future developments. In Section 5, we suggest an extension of the system considered in [5] to get rid of preprocessing by exploiting the advantage of locality.

2. A short survey of our system

Our model: Presume scene S consists of an arbitrary number of simple objects (e.g., balls) identified by their centers. The balls can be arbitrarily distributed over the scene, but they must not overlap.

Several visitors are sitting at their graphics workstations (rendering machines). They – or more precisely: their counterparts in this virtual world – can walk to arbitrary positions of the scene. For an approximately correct view, every visitor only needs the part $V_t(x) \subset S$ of the scene. The set $V_t(x) = \{b \in S : d(b,x) \leq t\}$ consists of all objects with Euclidean distance at most t from x. We call the set $V_t(x)$ the *t-environment of x*. The distance t is chosen so that the rendering complexity of the objects in $V_t(x)$ is at most C_0, i.e., the rendering machine can render the objects with a fixed frame rate (e.g., $20fps$). The scenes we have in mind are very large and have a great spatial extension so that most of the scene is stored on disks.

The scene can be modified by a modeler. Like the visitor he can walk to arbitrary positions of the scene. At any time he may insert or delete objects from the scene. These updates should occur in real-time and immediately affect the visitors' views.

Architecture: The very large scene description cannot be kept entirely in memory but must be fetched from secondary storage media. Since disk access is slow we introduce a two level access scheme for the rendering machine. Like a buffer the rendering machine will store a larger environment $V_T(y)$ for a position y and $T \gg t$ in the rendering machine's main memory so that the t-environment $V_t(x)$ can be extracted very fast.

To disburden the rendering processor, we assign this work to a third machine, the manager. This concept is introduced in [10], we adopt the approach here. The manager has access to the scene stored at disk. He computes the T-environment $V_T(y)$ for the rendering machine at fixed time intervals (steps). In every time interval $i+1$ the rendering machine will send the position x_i of its visitor to the manager who in turn computes $V_T(x_i)$ and sends this to the visitor.

The data sent should be a differential update of the form $V_T(x_i) \setminus V_T(x_{i-1})$ and $V_T(x_{i-1}) \setminus V_T(x_i)$ that allows easy computation of $V_T(x_i)$ out of $V_T(x_{i-1})$.

Let v be the maximal speed of the moving visitor and t_{int} the length of a time interval. If $T \leq 2vt_{\text{int}} + t$, then the visitor cannot in one step leave the t-environment $V_T(x)$, i.e., for an arbitrary successor position x' to x: $V_t(x') \subseteq V_T(x)$.

Data structures: For computing $V_T(x)$ or $V_t(x)$, we have to answer the following queries:

Given an arbitrary number m of balls. For position x in the scene (this position is known in data structure, too), SEARCH(x,T) reports all balls as a data structure $D(x)$ so that for each position y with $d(x,y) \leq T - t$ the t-environment $V_t(y)$ of y can be computed from $V_T(x)$ very fast. The last property is important since the rendering machine has to do this at least 20 times per second. More explicitly, we require SEARCH(x,T) to be of *output sensitive* running time in the sense that it has computational complexity $\mathcal{O}(1+k)$ where $k = |\text{SEARCH}(x,T)|$.

UPDATE(x,T,y) reports all balls in $V_T(x) \setminus V_T(y)$ and $V_T(y) \setminus V_T(x)$ such that – given $D(x)$ – the update $D(y)$ can be computed very fast.

INSERT(x) and DELETE(x) insert and delete a ball at the actual position x. Again, output sensitive running time is of high importance in order to respect the real-time requirements.

In [5], a data structure called γ-angle graph is presented that fulfils our requirements with the following additional properties: Let c be constant, m the number of balls, $l := |V_{T'}(x)|$, $l_1 :=$

$|V_{T'}(x) \backslash V_T(y)|$, and $l_2 := |V_{T'}(y) \backslash V_T(x)|$. Then

- the data structure needs space $\mathcal{O}(m)$
- SEARCH(x, T) can be done in time $\mathcal{O}(l + (\frac{T}{c})^2) = \mathcal{O}(T^2)$
- UPDATE(x, T, y) can be done in time $\mathcal{O}(l_1 + l_2 + \frac{T(r+c)}{c}) = \mathcal{O}(T \cdot (r + c))$, if $V_T(x)$ is given.
- DELETE(x) can be done in time $\mathcal{O}(c^2 \log(c))$ with high propability (w.h.p.)
- INSERT(x) can be done in time $\mathcal{O}(c^2)$ w.h.p.

For an exact description of the data structures, see [5]. A derandomized improvement with lower constants for both space and time requirements can be found in [4].

3. Implementation

Modeler/Visitor: Through the user interface the modeler/visitor can control three parts of the program: 'navigation', 'scene manipulation', 'measurements tools' (See Fig. 2).

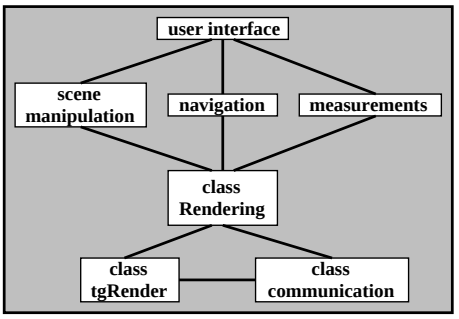

Figure 2: Implementation visitor

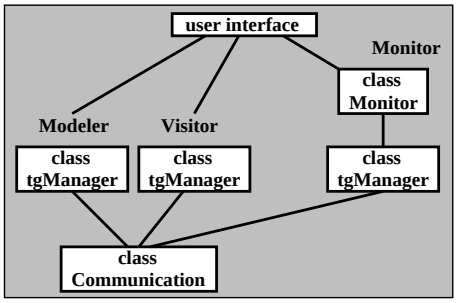

Figure 3: Implementation manager

The modeler/visitor controls navigation through the scene with the keyboard and mouse devices. She looks to the scene like a camera and can change the cameras' position and orientation arbitrarily. The modeler may insert and delete objects at every position.

In order to get reproducible experiments, we need some means to make the visitor move along the same path in the same scene several times. Therefore, we implemented a tool for recording, saving, and loading fixed camera positions and orientation. After loading these positions the computer will automatically move along them like the visitor did before.

Basis of the modeler/visitor program are the classes `Rendering`, `tgRender`, and `communication`. The data structure $V_t(x)$ for the γ-angle graph is controlled by the `tgRender` class, whereas class `communication` handles the communication with the manager and the `Rendering` class' task is to manage the polygonal scene description and to finally render the scene.

Manager: Like the modeler and visitor, the manager can be controlled via a graphical user interface. The basis for the manager is the class `tgManager`. The γ-angle graph is stored in this class as a static member. For every instance of a visitor/modeler that wants to walk through the scene, an object of the class `tgManager` is defined. In the example, we have a modeler and a visitor. Therefore, two objects are defined (cf. fig. 3).

For tests and evaluation of our implementation, a monitor is implemented which shows the γ-angle graph graph and the position and t-environment of every visitor/modeler (See Fig. 4).

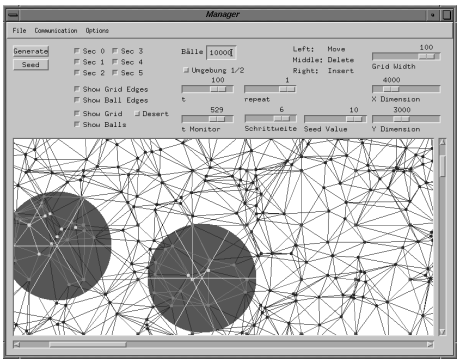

Figure 4: Screenshot Manager

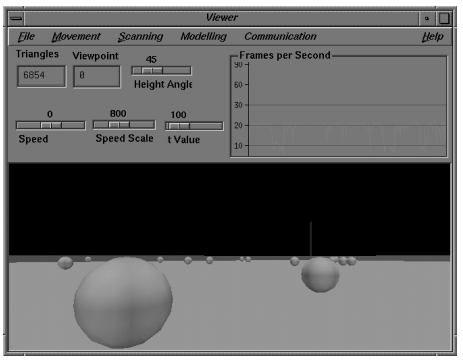

Figure 5: Screenshot Visitor/Modeler

For clipping the γ-angle graph in this window, we have a third instance of class `tgManager`. The t-environment of this object consist of the graph in the monitor window. The communication to the visitor and the manager is handled by an instance of the `communication` class.

Libraries and Communication Protocol: The implementation of the manager should run on a workstation without special graphics system (SUN Ultra Sparc, 200 MHz), therefore we used only standard libraries for Unix based systems. Our idea is to perform expensive computations on inexpensive computer systems. The special graphics system should be disburdened.

For the graphical user interface of the modeler, visitor, and manager, we use X11, XToolkit, and Motif. The implementation of the visitor and modeler runs on a special graphics workstation (SGI O2, 180Mhz). The rendering process of the scene is implemented with the standard libraries OpenGl and OpenInventor and for the graphical user interface Viewkit (See Fig. 6).

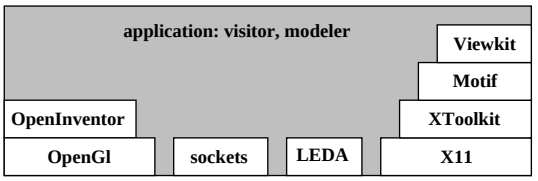

Figure 6: Libraries used

The communication between the modeler, visitor, and manager is based on the TCP/IP protocol (socket library). Therefore we can test our system on arbitrary systems, that are connected via Internet. Our aim is to test the system on different communication networks (e.g., ATM, Ethernet, etc.). A further advantage is the smaller communication overhead compared with other high level communication libraries (MPI, PVM, etc.). Some special data structures are implemented with LEDA [1].

4. Experimental Results

The objects of our scene are represented by unit size balls. A search data structure is responsible for reporting all balls within distance t. The scene will in general be very large, so it has to be stored on disk. The task of a further machine, the manager, is to access this disk. At fixed time intervals, the rendering machine receives updates from the manager to its locally stored part of the scene.

The goal of this work is to determine how fast the visitor can walk through the scene and how many balls can be rendered (number of balls). These parameters are crucial since they determine

practical applicability of our system. As it turns out, its performance is primarily limited by two factors: One bottleneck is the communication channel between manager and rendering machine. The second one arises from applying updates to the visitor's part of the scene. Therefore we explore the effect of these two constraints onto the speed of the visitor and the number of balls.

One surprising result of our research is that insertion of new objects to the scene graph reveals to be rather time consuming; much more expensive than deletion or moving. The problem occurs when, resulting from a visitor's movement, new balls of the t-environment $V_T(y)$ become part of the subscene handled by the rendering machine. Due to caching-like optimizations in the graphics library's internal data structures, the process of introducing new objects each time induces some kind of preprocessing or reinitialization. Emphasis lies on *insertion* of *new* objects: Deletion from the library data base is fast, and so is re-insertion of the same ball.

To the γ-angle graph , inserting and removing are symmetric and inexpensive operations. But the library's internal behavior is beyond our control. This has the following consequences for us: Updates of $V_T(y)$ of the rendering machine will be time consuming, but the update of $V_t(x)$ does not cause a great time demand. So we have to concentrate on the update of $V_T(y)$.

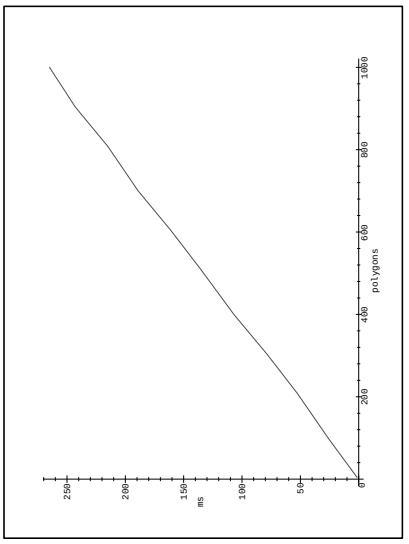

Figure 7: Running time for initialization in dependence of the ball complexity

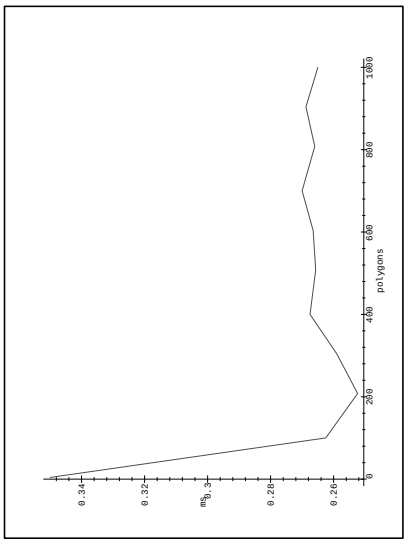

Figure 8: Running time for initialization per polygon in dependence of the ball complexity

The cost for the initialization of a ball depends of the number of triangles used for the representing its surface. Typically, in order to get a impression of a good approximation of a sphere, we need roughly 200 triangles. With the complexity of the ball we denote the number of triangles used for its representation. In Fig. 7 (figures are rotated), we have shown the initialization cost for balls of complexity from 4 to 1000 triangles. As we can see, the initialization time is nearly linear in the number of triangles. Since the curve did not cross the origin, the time cost per polygon of a ball is a little greater for very small balls. In Fig. 8, we can see this. We have drawn the initialization cost per triangle in dependence of balls with complexities from 4 to 1,000 triangles.

With this measured data we can compute how many balls we can initialize per second for balls of different complexities. In Table 1 we have shown the ratio balls/sec for balls of different complexity.

In order to get this values we need the rendering processor's full computational power (100%), i.e., it does not have any time for rendering.

triangles	4	100	208	304	400	508	604	700	808	904	1000
balls/sec	714.28	38.09	19.04	12.69	9.34	7.04	6.21	5.29	4.65	4.1	3.77

Table 1: Initialization ratio for balls of different complexity (triangles)

So at this point, we are confronted with the question how much time of the rendering processors computation power we will use for rendering computations and for initialization of the balls. This is a problem since the goal is not to disburden the rendering machine so that it can render the scene. A typically rendering capacity of our graphics workstation (SGI O2) can render up to 16,000 triangles (e.g., 80 balls each of 200 triangles), if the processor is not loaded with other work. In the following, we will describe that the practical measured values lead to satisfactory results. Furthermore we will show that there are two tradeoffs which are convenient for our model.

In Table 2, we have shown the maximum speed of the visitor for a scene consisting of balls having a complexity of 100 triangles.

percentage for initialization	size of t	maximum speed
10%	$100m$	1.32 $\frac{m}{s}$
10%	$500m$	6.61 $\frac{m}{s}$
20%	$100m$	2.97 $\frac{m}{s}$
20%	$500m$	14.86 $\frac{m}{s}$

Table 2: An example for scenes consisting of balls with a complexity of 100 triangles

If we allow 10% of the rendering machine's computation power for initialization, we get a maximum speed of 1.32 $\frac{m}{s}$ for a scene of $100m$ radius, and a maximum speed of 6.61 $\frac{m}{s}$ for a scene of $500m$ radius. In the other case, if we allow more time for initialization computation, e.g., 20% we get a maximum speed of 2.97 $\frac{m}{s}$ for a scene of $100m$ radius, and a maximum speed of 14.86 $\frac{m}{s}$ for a scene of $500m$ radius.

Here we have two tradeoffs: One between speed and size of the scene and the other between speed and the percentage for initialization computations. If we enlarge our scene, we can get a higher speed of the visitor. Future versions of our system will take advantage of this tradeoff, since in a small scene with a high density of balls the visitor walks slowly to see every part of the scene. Otherwise, in larger scenes with a lower density of balls, the visitor walks faster in order to reach the next ball.

We can get a higher speed if we enlarge the percentage for initialization computations. In this case, the remaining scene will have a less density, and so the visitor tries to reach the next object with high speed. We will exploit this in our implementation so that the rendering machine tries to initialize more balls if the visitor walks slowly. The absolute values of the maximum speed seem usable for a practical application.

For the communication bottleneck, we get similar satisfying results. We will describe them in our final report.

5. An extension of our architecture

At this point, our architecture will be extended in comparison to [5]. Our second problem induced by the management of large scenes is the expensive space requirement for the search of data structures of the objects. Because of locality, the visitor sees similar t-environments $V_t(x)$ of objects in consecutive steps of a few time intervals, i.e., she needs not a search data structure that manages all objects of the scene. We want to exploit this locality for saving memory of the manager and for saving time of the QUERY-operations of the rendering machine. Therefore, we use a two level search data structure of the manager. We distinguish a *coarse grained level* and a *fine grained level* (See Fig. 9).

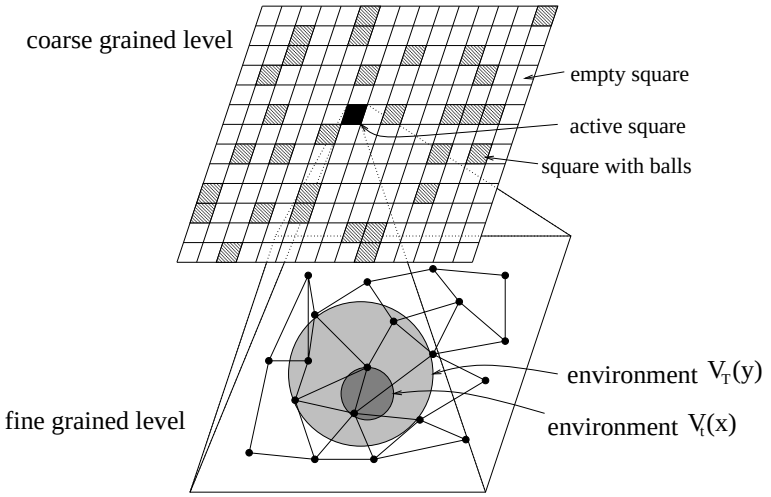

Figure 9: Two level access for the manager

At the coarse grained level, we divide the scene into *squares*. We denote squares with at least one ball as *full* and those without balls as *empty*. All balls of the same full square at the coarse level are stored on disk in a list. This list is represented by the center position of the full square and the unsorted list of elements. At each time, there is exactly one square that is *active* (See Fig. 9). This square contains the visitor's position. For this square, all balls from the disk are loaded into the main memory of the manager. These balls are contained in the so called *fine grained level*.

For each of the two levels, we have an (arbitrary) search data structure D_{COARSE} and D_{FINE}. Such a search data structure should have the operations QUERY and BUILD. BUILD builds the data structure for a given input from scratch and QUERY is query-operation (e.g next neighbor or range query) on the data structure. The input for D_{COARSE} consists of the positions of the full squares (we do not store the mesh explicitly) and the input for D_{FINE} of all balls of the active square. In our system, QUERY is one of the operations SEARCH, UPDATE, DELETE, and INSERT. The manager computes for balls of the active square the t-environment $V_T(y)$ as described above with the D_{FINE} data structure.

A walk of the visitor through the scene results in the following operations for this two level hierarchy: At every time we have a data structure D_{FINE} for the balls of the active square. The manager computes from this data structure the set $V_T(y)$ for the rendering machine. When the visitor leaves the active square, the manager has to search the next neighboring full square in the D_{COARSE} data structure (with QUERY operation of D_{COARSE}). If the directly neighboring square is an empty square there is nothing to do. Otherwise, the manager loads all balls of the new active square into the main memory and computes with the BUILD operation of D_{FINE} the data structure for the fine grained level from scratch. The data structure D_{FINE} and all balls of the previously active

square are removed from memory. Now the manager can compute the t-environments $V_T(y)$ with the QUERY operations of D_{FINE} and so on. The dimension of the mesh should be chosen in a way that it takes a lot of time for the visitor to cross a square. So the time for computing D_{FINE} from scratch and loading the balls will be expensive. Therefore it is recommended to hold permanently the data structures D_{FINE} for the at most 8 neighboring full squares of the active square. More than the 8 neighbors are not necessary since we assume the squares very large so it will go by a lot of time for moving across the active square.

We have a tradeoff between the dimension of the mesh and the number of balls of a square. The question is how is the optimum size of the mesh? Let m be the number of balls, s the distance of two consecutive moves of the visitor, d the dimension of the mesh ($d \times d$ mesh), and $c \cdot s$ be the size of the scene. Let $T^{\text{FINE}}_{\text{QUERY}}$ be the time for a query of D_{FINE} and let $T^{\text{COARSE}}_{\text{QUERY}}$ be the time for a query of D_{COARSE} (for BUILDanalogue). For a walk of the visitor of length $k \cdot s$, we get a total runtime T_{move} of

$$T_{move}(k, m, d, c) = kT^{\text{FINE}}_{\text{QUERY}}\left(\frac{m}{d^2}\right) + \frac{k}{c/d}\left(T^{\text{COARSE}}_{\text{BUILD}}\left(\frac{m}{d^2}\right) + T^{\text{COARSE}}_{\text{QUERY}}\left(d^2\right)\right).$$

For the following example, we assume the runtime for D_{COARSE} and D_{FINE} to be equal to $T_{\text{QUERY}} = q \cdot m\log(m)$ and $T_{\text{BUILD}} = p \cdot \log(m)$, and the balls are randomly distributed over the scene. Then we get for T_{move}

$$T_{move}(k,\, m,\, d,\, c,\, p,\, q) = k\,\left(q\log_2(\frac{m}{d^2}) + \frac{d}{c}\,\left(p\,\frac{m}{d^2}\log_2(\frac{m}{d^2}) + q\log_2(d^2)\right)\right).$$

The minimum of T_{move} depending on d is the solution the following equation for d

$$0 = -\frac{k\,(2\,q\,c\,d + m\,p\ln(\frac{m}{d^2}) - q\ln(d^2)\,d^2 + 2\,m\,p - 2\,q\,d^2)}{d^2\,c\ln(2)}.$$

If we solve this equation for, e.g., $m = 10^6$ balls, $k = 1{,}000$ steps of the visitor, $c = 1{,}000$ (area of the scene), and $p = q = 1$ (constants for runtime), then we will get a minimum of T_{move} for $d = 543$. Important is the question where T_{move} attains its minimum. If it does at the extreme points need not this two level structure. As we can see in some examples, the minimum depends strongly on constants p and q.

If a visitor walks more than twice through the entire scene, then our permanent process of computing and removing the fine grained level data structure will be more time consuming than computing the data structure for all objects of the scene once. But we have in mind that our scene has a large spatial extension, so we can save memory for the manager since he has a coarse data structure for the entire scene. A further advantage is that the data structure of the rendering machine for computing the t-environment $V_t(x)$ will have an $\mathcal{O}(m/d^2)$ input size instead of $\mathcal{O}(m)$. For our strict real time requirements, this constant factor is important for data structures with $\mathcal{O}(\log(n))$ QUERY-time (e.g., trees) since computing of the rendering machine is more critical as for the manager machine as we discussed above.

6. Ongoing Work

At the next step we will test search data structures in practice and compare them with other standard data structures. In a further step, we will implement the two level access for the manager and give an evaluation.

Acknowledgements

We would like to thank Friedhelm Meyer auf der Heide, Willy-Bernhard Strothmann, and Rolf Wanka for helpfull comments and suggestions.

References

[1] Christoph Burnikel, Jochen Könemann, Kurt Mehlhorn, Stefan Näher, and Stefan Schirra. Geometric computation in LEDA. In *Proceedings of the 11th Annual Symposium on Computational Geometry (SCG 95)*, pages C18–C19, 1995.

[2] James H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, 19(10):547 – 554, October 1976.

[3] S. Coorg and S. Teller. Real-Time Occlusion Culling for Models with Large Occluders. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics 1997*, pages 83 – 90, April 1997.

[4] Matthias Fischer, Tamás Lukovszki, and Martin Ziegler. Geometric Searching in Walkthrough Animations with Weak Spanners in Real Time. In *Proceedings of the Sixth Annual European Symposium on Algorithms*, 1998.

[5] Matthias Fischer, Friedhelm Meyer auf der Heide, and Willy-Bernhard Strothmann. Dynamic Data Structures for Realtime Management of Large Geometric Scenes. In *Proceedings of the Fifth Annual European Symposium on Algorithms*, pages 157–170, 1997.

[6] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice.* Addison Wesley, 1995.

[7] Thomas A. Funkhouser and Carlo H. Sequin. Adaptive Display Algorithm for Interactive Frame Rates During Visualisation of Complex Virtual Environments. In James T. Kajiya, editor, *Proceedings of the SIGGRAPH '93*, volume 27, pages 247 – 254, 1993.

[8] Thomas A. Funkhouser, Carlo H. Sequin, and Seth J. Teller. Management of Large Amounts of Data in Interactive Building Walkthroughs. In *Proceedings of the SIGGRAPH '91*, pages 11 – 20, 1992.

[9] P. S. Heckbert and M. Garland. Multiresolution Rendering Modeling for Fast Rendering. *Proceedings of the Graphics Interface '94*, pages 43–50, May 1994.

[10] Jonathan Mark Sewell. *Managing Complex Models for Computer Graphics.* PhD thesis, University of Cambridge, Queens' College, March 1996.

[11] Jonathan Shade, Dani Lischinski, David H. Salesin, Tony DeRose, and John Snyder. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. In *Proceedings of the SIGGRAPH '96*, pages 75 – 82, August 1996.

[12] Seth J. Teller and Carlo H. Sequin. Visibility Preprocessing For Interactive Walkthroughs. In *Proceedings of the SIGGRAPH '90*, pages 61 – 69, 1991.