A Client-Server-Scenegraph for the Visualization of Large and Dynamic 3D Scenes

Jörg Sahm
Fraunhofer IGD
Fraunhoferstr. 5
64283 Darmstadt, Germany
sahm@igd.fhg.de

Ingo Soetebier
Fraunhofer IGD
Fraunhoferstr. 5
64283 Darmstadt, Germany
ingo@igd.fhg.de

ABSTRACT

With the increasing capabilities of hardware for 3D graphics and network, 3D multi-user environments get more and more interesting for e-business, entertainment and cooperative work. This aspect concerns not only high-end devices, like caves or graphic workstation, also small mobile devices like laptop computers or PDAs become more and more suitable for the visualization of 3D graphics. In order to visualize a large and dynamic 3D scene on multiple clients with different capabilities, an appropriate scene representation is required. In this paper a client-server-scenegraph is introduced, which addresses critical problems in the area of distributed 3D graphics such as the handling of dynamic 3D scenes, which exceed the capacity of the clients or even of the server. Another point is the selection of the data, which has to be transmitted from the server to the clients, although the users and the scene elements are moving. Since many clients lack the memory, they only hold the currently most relevant scene information. For that reason there has to be an efficient matching between the server's and the clients' data.

Keywords

scene representation, distributed/network graphics, out-of-core rendering

1. INTRODUCTION

In the last few years several client-server frameworks have been developed, which transmit only parts of a large 3D scene from the server to the clients. Size and quality of the transmitted data depend on the capabilities of the client, the client's position and view direction, and the available transmission bandwidth. Since clients typically lack the memory and the computing power in comparison to the server, the clients only hold a subset of the server's scene. If the scene is interactive and dynamic, the clients' data is changing rapidly: While the server has to transmit the elements, which are currently in the client's area of interest, the client discards the information outside of its area of interest in order to save memory and running power. One problem is the definition of such an area of interest. Another problem is to determine the areas of interest for multiple clients and 3D scenes with thousands of elements on the server side in real-time. Since the elements may change their position, server and client are in need of a scene representation, which can handle even dynamic scenes. If an element is transmitted, discarded, or transformed, then server and client have to communicate about the element. Since this kind of communication occurs frequently and the scenes may contain a large number of elements, there is the problem to identify a specific element on the server and its corresponding partner on the client very fast. Because of the resulting data effort of large 3D scenes, another problem is to handle scenes, which exceed even the server's memory capacity. In this paper a client-server-scenegraph is introduced, which encapsulates several data structures and methods for the solution of the listed problems.

Journal of WSCG, Vol.12, No.1-3, ISSN 1213-6972
WSCG'2004, February 2-6, 2004, Plzen, Czech Republic.

Related Work

A lot of computer graphics libraries depend on scene graph data structures. There are modern programming libraries like SGI's OpenGL Optimizer [Sgi98a] or the OpenSG graphics library [Rei02a], which are using scenegraph data structures as a representation for the 3D scene. One use of the

scenegraph data structure is the visibility determination of a certain viewpoint in the scene. A detailed overview on visibility algorithms is presented by Cohen-Or et al [Coh00a].

Another area is the transmission of 3D scenes over a network. The previously mentioned visibility algorithms can also be used to choose the 3D object, which should be transferred over the network. Funkhouser et al [Fun93a] presented a heuristic approach using a definition for cost and benefit of objects in the scene. These properties are optimized to choose an appropriate LOD of an object of the scene. In the year 1995 Funkhouser [Fun95a] presented a client-server-architecture in order to reduce the message traffic between multiple participants in Virtual Reality (VR) scenes. Virtual Reality was the first field of application, in which the concept of client-server-rendering was used. In Mann et al [Man97a] use a high-end workstation as server in order to send large sets of textures to the clients. While the clients interpolate the rendered images from their local information, the server renders difference images between the clients' and the original view. The missing textures are sent by demand to the clients. Other work in this area includes the approach described by Teler et al [Tel01a]. Similar to the approach of Funkhouser et al [Fun93a], he defines cost and benefit for transmission, which is then optimized for a certain viewpoint. Schneider et al [Sch99a] present a complex performance model where an optimal transmission method is chosen depending on network performance, available resources and user preferences. The approach of Hesina et al [Hes98a] maintains an area of interest around the user's viewpoint. There is a prefetch algorithm for transmitting the graphical objects over the network belonging to this area of interest. Contrary to the presented approach, these papers do not mention a scenegraph representation of the 3D scene.

There are also papers addressing the problem of managing very large scenes. Those scenes may even exceed the main memory of a standard workstation and thus they can be only rendered out-of-core. Varadhan et al [Var02a] presented a concurrent approach using a prioritized prefetching strategy for loading graphical objects from disk. Another approach was presented by Klein et al [Kle02a]. They are using a data structure to create an approximate image of the scene with a special kind of polygon sampling.

2. CONCEPT

In this section the concept of the client-sever-scenegraph is introduced. At first an algorithm for the rearrangement of a dynamic space partition tree is introduced. After this a new concept for the definition of an area of interest is explained. This concept can be used for the transmission of scene elements as well as for the later discussed out-of-core rendering. Finally, the concept presents a solution for the identification problem of corresponding elements on server and clients.

The Dynamic Space Partition Tree

Each 3D scene is represented by a dynamic space partition tree (SPT) on server and client as well. Although the concept of space partition trees is not new (e.g. see [Fuc80a, Sch69a, Gre93a]), there are not efficient approaches for large and dynamic 3D scenes in real-time. In the presented SPT, the nodes are divided into inner nodes and leaf nodes. Inner nodes do not have a visual appearance but represent an axis aligned parallelepiped region of the scene. Leaf nodes represent the 3D objects of the scene and are classified into group nodes and element nodes. An element node encapsulates not only the visual appearance of a 3D object, but also its specific behavior. While an element node represents only a single object of the scene (e.g. the door of a car), group nodes contain several element nodes, which are connected to an animation hierarchy. The approach differentiates between static, passive, and active leaf nodes. While the state of static nodes never changes, active nodes are able to perform simulation specific actions. Passive nodes normally remain static, but can be forced to action because of some kind of interaction.

2.1.1 The Basic Rearrangement Operations

The rearrangement of the SPT becomes necessary because of scene manipulations, which are either caused by simulation and animation instructions or by user interactions. These manipulations consist of the insertion of elements into the scene, the removal of elements from the scene, and the transformation of elements inside of the scene. In order to rearrange the SPT, the approach makes use of two basic operations, namely the divide operation and the reunite operation. Leaf Nodes completely inside of the region of an inner node *I* are added as children to *I*. If the number of leaf nodes inside of *I* exceeds a specific threshold, then *I* is divided into several subregions. These subregions are represented by other inner nodes, which are also added to *I*. The number and the size of the subregions depend on the chosen SPT implementation (e.g. an Octree or a KD-Tree). The presented approach is not in need of a specific implementation, but requires, that the axis aligned bounding box (AABB) of a child node is completely inside of the parent node's AABB. The AABB of an inner node is identical to the represented region. While the AABB of an element

node is defined by its geometric information, the AABB of a group node is defined by the AABBs of all its element nodes. If a leaf node intersects several inner nodes, it is added to the parent inner node, which completely contains the leaf node. Greene et al [Gre93a] propose some alternatives, but, if applied to dynamic scenes, they are too expensive concerning memory and running time. If the number of leaf nodes inside of an inner node's subtree is below the specific threshold, then the leaf nodes are added to the inner node, and the subtree is deleted. In the following this process is denoted as "the inner node is reunited".

Typically, the elements of an animation hierarchy are close to each other and inherit the transformation of their parents. In many applications this aspect results in similar move directions of combined elements, e.g. if the user drives a virtual car, then all the car's doors, windows and wheels are moved in the same direction. If the car leaves an inner node's AABB performing a continuous translation, then each element of the car's animation hierarchy could cause a rearrangement of the SPT. For that reason the element nodes inside of a group node are not handled separately by the basic operations, but only the group node is added to an inner node.

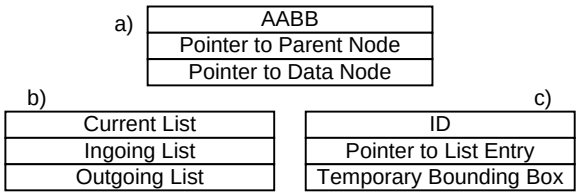

Figure 1. Figure a illustrates the basic structure of all node types, including a pointer to a data node (see section 2.3). Figure b shows the inner node structure, and figure c the leaf node structure.

As a further optimization leaf nodes may define a temporary bounding box (TBB) [Sud96a] (see figure 1). This TBB guaranties, that the leaf node will not leave the TBB for a certain time interval. If the TBB is set, the leaf node is sorted into the SPT by its TBB and not by its AABB. During the time interval of the TBB, the leaf node does not cause changes of the tree. But since the TBB typically contains the leaf node's AABB, the leaf node is sorted into a lower level of the tree, what can result in lesser performance of applications such as occlusion culling or collision detection.

2.1.2 The Rearrangement Data Structures

Each inner node contains three lists, namely the current list, the incoming list, and the outgoing list (see figure 1). The current list represents the actual state s_{t_c} of the animation or simulation at the time t_c, and contains references to all the leaf nodes, which are currently inside of the inner node's AABB. Unlike the current list, the incoming list and the outgoing list represent the next time step t_{c+1} of the animation or simulation. If a leaf node will leave the inner node in the time step t_{c+1}, then a reference to this leaf node is added to the inner node's outgoing list. If a leaf node will enter an inner node in the time step t_{c+1}, then a reference to this leaf node is added to the incoming list of the target inner node. If the function *size(L)* returns the entry count of a list *L* then the value $n = size(current) + size(incoming) - size(outgoing)$ represents the inner node's number of leaf nodes at the time t_{c+1}. This approach has two advantages: The first advantage is, that the current list never changes, until t_{c+1} becomes the current time step. So this list is always consistent and readable for parallel algorithms (e.g. visualization, collision detection, searching). Furthermore the state $s_{t_{c+1}}$ can be determined simultaneously to these algorithms. The second advantage is, that the necessary rearrangement of the SPT at the time t_{c+1} has not to consider each element manipulation separately, but only the effective sum of all manipulations, which is computed by $r = n(t_{c+1}) - n(t_c)$. For that reason the algorithm introduced in section 2.1.3 can take advantage of compensating operations: If a leaf node L_1 leaves an inner node I_A and enters an inner node I_B, while a leaf node L_2 leaves I_B and enters I_A, then I_A and I_B have neither to be divided nor reunited. Because *size(current)* only returns the local leaf node count of an inner node (i.e. the number of leaf nodes, which could not be added to a subregion of the inner node), each inner node provides the global leaf node count *g*, which returns the number of all leaf nodes inside of the inner node's subtree.

In order to collect all inner nodes, which are affected by an element manipulation in the next time step t_{c+1}, the SPT provides a modified list for each level of the SPT. The first time a leaf node leaves or enters an inner node *I* in the time step t_{c+1}, a reference to *I* is added to the modified list of *I*'s level. So the modified list does not contain double references to *I*.

2.1.3 The Rearrangement Algorithm

The rearrangement algorithm is started each time a new time step is entered. During the rearrangement the current lists of the inner nodes are locked. The algorithm processes the inner nodes bottom-up, i.e. it starts processing the inner nodes referenced by the affected modified list of the highest level and then steps to the next lower level (the SPT's root has level 0). The following pseudo code illustrates the algorithm (for the values *r*, *g*, *n* see section 2.1.2):

```
<set level to highest affected level>
<while level greater 0)
  <set A to first inner node of modified list
   of current level>
  <while A not equals end of modified list>
    <compute n of A>
    <remove all references of A's outgoing
     list from A's current list>
    <add all references of A's incoming
     to A's current list>
    <clear A's incoming and outgoing list>
    <compute r of A>
    <if r unequal 0>
      <add r to g of A>
      <add r to g of parent P of A>;
      <if P not in modified list with level - 1
       then add P to modified list with level - 1>
      <if g equals 0 then delete A>
      <else>
        <if A not divided and g of A greater threshold
         then divide A>
        <if A divided and g of A lesser threshold>
         then reunite A>
    <set A to next node of current modified list>
  <clear modified list at current level>;
  <decrement level by 1>
```

Removing the references of the inner node's outgoing list from its current list seems to be an expensive operation, because searching in an unsorted list requires the complexity O(n). This can be easily changed to O(1), if each leaf node contains a reference to the entry inside of its parent's current list. This works well, because each leaf node can not be addressed by several current lists at the same time.

The Area of Interest Computation

An important task of the server is the determination of the scene elements, which have to be transmitted to the clients. For this the server is in need of a proper client representation, which is called the "client's area of interest" in the following. In the presented approach the area of interest of a client is defined by four nested AABBs, which contain the client's view frustum (see figure 2). Leaf nodes inside of the smallest AABB get the highest priority, while leaf nodes outside of the largest AABB get the lowest priority. The server provides a specific priority renderer for each of the four AABBs. Renderer do not only visualize the scene, but traverse the SPT in order to produce some kind of output. Since the server supports multiple clients, the first renderer begins with the largest AABB of the first client:

1. If the inner node is outside of the AABB, then mark the inner node as Outside. Set the node's priority to 0.
2. If the inner node is inside of the AABB, then mark the inner node as Inside. Set the node's priority to 1.
3. If the inner node intersects the AABB, then mark the inner node as Partial. All leaf nodes inside of the inner node have to be tested against the AABB. Partial leaf nodes are marked as Inside. After that, the renderer steps down to the children of the inner node recursively.

Similar to visibility culling algorithms the renderer makes use of spatial coherence. If an inner node is marked as Inside (Outside), then all its children are Inside (Outside). If the first renderer steps to the second client, it only has to traverse the Partial and Outside inner nodes. The renderer of the second ABBB only traverses inner nodes, which have been marked as Inside or Partial by the first renderer, etc. If there are many clients, this algorithm becomes very expensive, when applied in each simulation step. So the algorithm works asynchronously. In each cycle of the priority algorithm the first renderer processes one client, the second renderer two clients, the third renderer four clients, and the fourth renderer eight clients. Since a renderer R_n, n = 2..4 only traverses a subset of the renderer R_{n-1}, R_n's running time is significantly shorter in comparison to R_{n-1}. The basic concept is, that changes inside of the smaller AABBs have to be considered very fast, while changes outside of the largest AABB can be ignored for a specific time interval. With an increasing number of clients, either the frequency of the priority algorithm or the size of the AABBs has to be increased.

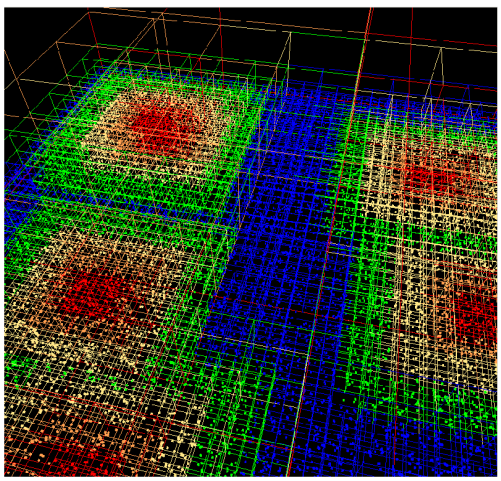

Figure 2. Each client is represented by its area of interest on the server side, which consists of four nested AABBs. Elements inside of the smallest AABB have the highest priority and are colored red in the illustrated server test environment. Elements colored blue are out of the areas.

The Out-of-Core Rendering

The areas of interest do not only determine the leaf nodes, which may have to be transmitted to the clients, but also the nodes, which could be swapped out to the file system, if the server or the clients lack the memory (on the client side the algorithm described in section 2.1.3 processes only the four AABB's of the client's area of interest in order to provide the out-of-core rendering). For example leaf nodes with the priority 0 should be swapped out at first, because they are outside of all areas of interest and are so currently not accessed. But the data could not be selected only by the nodes' priority. For example static leaf nodes do not cause rearrangements of the SPT, so they have to be swapped out before dynamic leaf nodes, even if their priority is higher. Swapping out inner nodes with low level is critical, since the renderer traverse these nodes very often. Another point is to divide the data of leaf nodes and inner nodes into "often accessed" and "rarely accessed" information (e.g. the AABB of a node is an often accessed data). Rarely accessed information is encapsulated by a separated data node (see figure 1), which is referenced by the corresponding leaf node or inner node. So if the server or the client requires memory, these data nodes are swapped out to the file system before their main nodes.

The basic concept of the swapping strategy is very simple: If the memory usage is below a specific threshold, then check for swapped out data and swap it into memory. If the SG's memory usage exceeds the threshold, then check for data in memory and swap it to the file system. Since the swapping of large data sets would be too expensive, the strategy processes only a specific amount of data in one step. The strategy swaps the data from the memory into the file system in the following order:

1. Data nodes of static leaf nodes
2. Static leaf nodes
3. Data nodes of dynamic leaf nodes
4. Dynamic leaf nodes
5. Data nodes of inner nodes
6. Inner nodes

If swapping data from IOC into memory, then the order is vice versa. For each of the enumerated data types, the strategy provides a specific I/O renderer. The data with lower priority and higher level is swapped out to the file system at first, while the data with higher priority and lower level is swapped to memory at first.

Identification

Since the clients usually don't have the server's capacity, they only hold a subset of the server's scene. If a leaf node is in the client's area of interest, it is transmitted to the client, where it is added to the client's SPT. Sometimes a leaf node causes further communication between server and client, e.g. because of user interactions. For that reason, there has to be an identification between the corresponding leaf nodes of server and client. Since a scene may contain thousands of elements, the search for a specific leaf node could be very expensive.

As illustrated in figure 1 each leaf node contains an ID. If a leaf node is added to the server's SPT, then a reference to this leaf node is inserted into a key-value-map. The map returns a key, which is set as the leaf node's ID. If the leaf node is transmitted to a client, then the client inserts the node into its own map. So the client's leaf node has a different ID than the server's node. For that reason, the server provides a lookup-map for each client, which maps the client's ID to the ID of the server (see figure 3). Additionally, the leaf nodes on the server side provide a vector structure, which contains the leaf node's IDs for all clients. If the server requests an information about a specific leaf node from the client, then the server transforms the leaf node's ID into the according client's ID with the help of the node's vector. Furthermore the vector identifies all clients, to which the leaf node was transmitted.

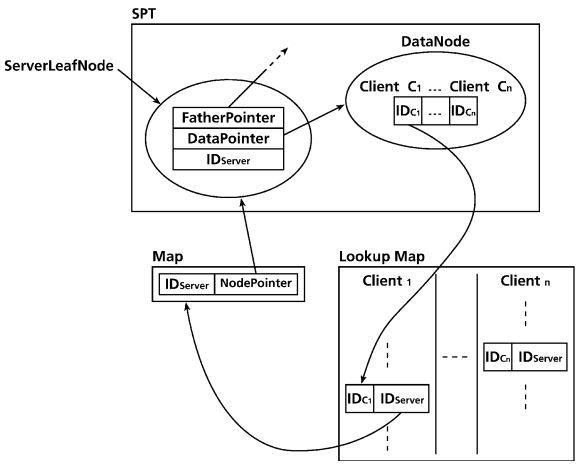

Figure 3. The server provides several maps in order to identify corresponding leaf nodes on server and client. All pointers are of type *NodePointer*, which can point to an address inside the memory or a file. The *ServerLeafNode* is the server specific implementation of the leaf node (see section 3). The *DataPointer* addresses the separated data node with the rarely accessed information.

Why are the ID's of clients and server not identical? The reason is, that searching for an information inside of a map should only require a constant amount of time. The map of the presented approach

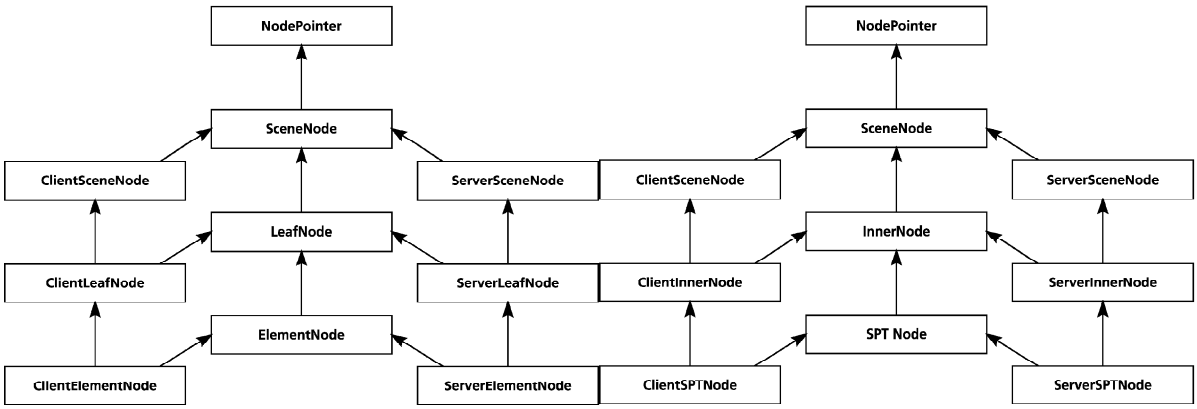

Figure 4. The inheritance hierarchy.

is defined by a tree with a constant number of levels. While the map's data is stored in the leaves, the inner nodes provide one array and one heap: The array contains pointers to other nodes (i.e. to the children). The heap contains partial sorted indices of pointers inside the array, which lay upon the path to a leaf with free entries. If a new information has to be added, the algorithm traverses the tree, until a leaf with free entries is reached. The returned key represents the path from the root to the leaf:

<set *key* to 0>
<set *level* to 0>
<set *maxlevel* to maximum level>
<set *arraysize* to size of array>
<set current *node* to root>
<while *node* is not leaf>
  <get the first index *i* from the first heap>
  <add $i * arraysize^{(maxlevel - level)}$ to *key*>
  <get the pointer P_i from the array>
  <if P_i is invalid>
    <then create a new node *N* and set P_i to *N*>
  <set *node* to P_i>
  <increase *level* by 1>
<get the first index *i* from the first heap>
<add $i * arraysize^{(maxlevel - level)}$ to *key*>
<get the pointer P_i from the array>
<if P_i is invalid>
  <then create a new node *N* and set P_i to *N*>
<set P_i to the data>
<for all visited inner nodes>
  <if heap of visited child is empty>
    <then remove used *i* from heap>

The size of the array and the maximum level depend on the size of the returned key. Since the system should support very large scenes, a 64 bit key is used. So the map contains 8 levels and each node has an array with 256 entries. Because the number of the levels is independent from the number of the map entries, the map has always a search complexity of O(1). Since even this map could exceed the memory capacity of a standard PC, the map provides, in combination with a last recently used (LRU) approach, an ideal structure for a disk paging algorithm (the inner nodes manage 256 entries and have a constant size). Because of the heap the information is always added to the free entry, which represents the smallest key.

3. IMPLEMENTATION

All components of server and client are implemented in C++. The GUI uses Qt for the graphical user interface and an OpenGL based renderer for visualization. All multi-threaded aspects are implemented with help of the ACE library. The SPT is implemented as an Octree. In order to realize the out-of-core rendering, all references are implemented as a special pointer class (*NodePointer*). This pointer class contains several flags and a 64 bit pointer to an address in the system's memory or file system. One of the flags indicates, whether the addressed node is stored in the memory or in a file.

In order to get a maximum of flexibility, the inheritance hierarchy illustrated in figure 4 has been used for the different node types. If substituting Element with Group, figure 4 shows the inheritance hierarchy of the group nodes. *SPTNodes* represent the specific implementation of the SPT. Since each node should know its own state and address, all nodes inherit from *NodePointer*. Some of the *NodePointer*'s flags represent the type of a node. Possible types are the leaves of the hierarchy. While the nodes in the middle of the inheritance hierarchy provide fields and methods, which are not client or server specific, the nodes of the left side implement client specific aspects and the nodes of the right side server specific aspects.

4. RESULTS

Testing was performed with random generated scenes. Because the main intention of the approach is the management of large, dynamic, and distributed data sets, the appearance trees of the leaf nodes only described cubes. If the scenes contained dynamic leaf

nodes, these nodes flew in arbitrary direction to the end of scene and returned on the opposite side. All results were measured on an AMD Athlon 2000+ MHz with 512 MB memory and a 123.5GB IDE IBM hard disk.

As mentioned in section 2.1.1 the SPT differentiates between the three element manipulations insert, remove and transform. Table 1 illustrates the number of operations, which were performed in one second. The threshold Θ represents the maximum amount of leaf nodes, before an inner node has to be divided. The insert and remove operations include the list insertion, the map insertion, and the complete rearrangement of the SPT. The transform operation includes not only the rearrangement of the SPT, but also the computation of the leaf nodes' animation. An increased Θ implies a decreased complexity of the SPT, so performance is getting better. A further increase of Θ has to be adapted to other applications, such as visualization or collision detection. Taking the results of the insert operation, the client could insert about 40000 received leaf nodes per second to the scenegraph.

Θ	Insert	Remove	Transform
1	32000	35000	130000
10	38000	42000	265000
20	40000	44000	270000
30	41000	46000	275000
40	43000	48000	276000
50	44000	49000	278000

Table 1. The results of the element manipulations insert, remove, and transform.

Table 2 shows the isolated results of the server's identification map. The table presents the number of operations, which were performed in one second. While the search operation is independent of the number of elements inside of the map, the add and remove operation have to process the partial sorted heaps. The variation of these operations is not caused by the maximum amount of elements, but by the current tree constellation of the map.

Add	Remove	Search
1845000 - 1850000	1998000 - 2000000	7700000

Table 2. The maximum number of map operations in one second.

Table 3 presents the results of the SPT's rearrangement. Testing was performed with a scene of 50000 dynamic leaf nodes. The number of the SPT's inner nodes (second column) varies because of the changing SPT structure. As it can be seen from the last two columns, the number of divide and reunite operations is very small in comparison to the total number of inner nodes.

Θ	Inner Nodes	Divide	Reunite
1	38400 - 39500	0 - 1150	0 - 960
10	14500 - 15300	0 - 161	0 - 160
20	4750 - 5350	0 - 13	0 - 15
30	4700 - 5300	0 - 13	0 - 11
40	4700 - 5300	0 - 9	0 - 9
50	4650 - 5150	0 - 14	0 - 7

Table 3. Very dynamic scenes are compensated by the rearrangement.

Because the presented I/O strategy swaps data from the memory into the file system and vice versa, table 4 illustrates the results of the "Write" and "Read" operations in seconds. While "Write" represents the swapping from memory to files, "Read" means the swapping from files to memory. Testing was performed with a SPT subtree of 50000 leaf nodes. Between each test, the hard disk's I/O cache was overwritten with other data. Column two contains the number of the subtree's inner nodes. The algorithm processed all structures of the subtree, which are enumerated in section 2.3. The algorithm's running time depends on the number of inner nodes, so running time becomes almost constant in the lower rows.

Θ	Inner Nodes	Write	Read
1	39000	3.4s	2.9s
10	15000	2.9s	2.4s
20	5050	2.6s	2.2s
30	4950	2.4s	2.2s
40	4900	2.3s	2.2s
50	4800	2.3s	2.2s

Table 4. The results of the I/O strategy.

Table 5 presents the total effort of all priority renderer. Testing was performed with a scene of 100000 static leaf nodes and 100000 dynamic leaf nodes. Column two shows the average number of inner nodes. The last three columns contain the number of visited inner nodes/leaf nodes, which depend on the number of clients. That means, if the four priority renderer would traverse all the nodes of a scene for a specific client, the algorithm would visit more than 200000 leaf nodes. As it can be seen in the last column, this value is not reached even by 30 clients.

Θ	Nodes	10 Clients	20 Clients	30 Clients
10	43400 200000	3600 23300	6150 42300	10900 70700
20	20900 200000	2700 25000	4650 44350	8200 74250
30	20650 200000	2650 25150	4600 44500	8100 74500
40	17000 200000	2400 26200	4200 46300	7400 77400
50	8650 200000	1750 29400	3050 52000	5350 88000

Table 5. The priority renderer traverse only a small percentage of the scene.

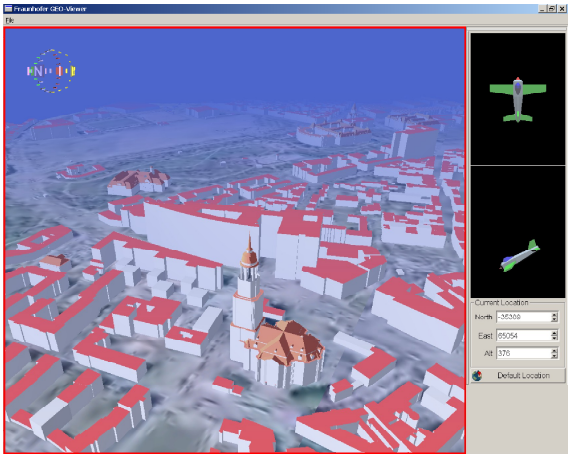

Figure 5. The visualization of the city of Hamburg, presented at the InterGeo 2003, was realized with the server's architecture (© Fraunhofer IGD, GIS-tec, and the city of Hamburg).

5. CONCLUSION

In this paper a client-server-scenegraph for the distributed visualization of large and dynamic 3D scenes was introduced. Although the concept of SPTs is well known, a new approach for the fast rearrangement of dynamic SPTs was explained. The approach takes advantage of compensating element transformations and allows the parallel processing of the scene's data. Furthermore a new area of interest concept was illustrated in combination with an algorithm for the fast computation of these areas. The area of interest concept can not only be used for the transmission of the elements, but also for a swapping strategy in order to realize out-of-core rendering. Finally, a solution for the identification problem was given, which bases on key-value-maps with a constant search time. The system has been tested with randomly generated dynamic scenes as well as with a terrain/city visualization application (see figure 5).

6. ACKNOWLEDGMENTS

This work was funded by the Heinz-Nixdorf-Foundation.

7. REFERENCES

[Coh00a] Cohen-Or, D., Chrysanthou, Y., and Silva, C. A survey of visibility for walkthrough applications. EUROGRAPHICS 2000, course notes, 2000.

[Fuc80a] Fuchs, H., Kedem, Z.M., and Naylor, B.F. On Visible Surface Generation by A Priori Tree Structures. ACM Computer Graphics (Proc. of SIGGRAPH '80), pp.124-133, 1980.

[Fun95a] Funkhouser, T. RING: A client-server system for multi-user virtual environments. Symposium of Interactive 3D Graphics, ACM SIGGRAPH, pp. 85-92, 1995.

[Fun93a] Funkhouser, T., and Sequin, C. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. Proc. of SIGGRAPH '93, 1993.

[Gre93a] Greene, N., and Kass, M. Hierarchical Z-buffer visibility. In Computer Graphics Proc., Annual Conference Series, 1993, pp. 231-240, 1993.

[Hes98a] Hessina, G., and Schmalstieg, D. A Network Architecture for Remote Rendering. Proc. of 2nd International Workshop on Distributed Interactive Simulation and Real Time Applications (DIS-RT'98), Montreal, Canada, 1998.

[Kle02a] Klein, J., Krokowski, J., Fischer, M., Wand, M., Wanka, R., and Meyer auf der Heide, F. The Randomized Sample Tree: A Data Structure for Interactive Walkthroughs in Externally Stored Virtual Environments. VRST '02, Hong Kong, 2002.

[Man97a] Mann, Y., and Cohen-Or, D. Selective Pixel Transmission for Navigation in Remote Virtual Environments. Computer Graphics Forum, Vol. 16, No. 3, pp. 201-206, 1997.

[Rei02a] Reiners, R. OpenSG: A Scene Graph System for Flexible and Efficient Rendering for Virtual and Augmented Reality Applications. Darmstadt, Technical University, PhD Thesis, 2002.

[Sch99a] Schneider, B.O., and Martin, I.M. An adaptive framework for 3D graphics over networks. Computer & Graphics 23, 1999.

[Sch69a] Schumacker, R., Brand, B., Gilliland, M., and Sharp, W. Study for Applying Computer-Generated Images to Visual Simulation. Technical Report AFHRL-TR-69-14, NTIS AD700735, U.S. Air Force Human Resources Lab., Air Force Systems Command, Brooks AFB, TX, 1969.

[Sgi98a] Silicon Graphics Inc. SGI OpenGL Optimizer Whitepaper. http://www.sgi.com/software/optimizer/whitepaper.pdf, 1998.

[Sud96a] Sudarsky, O., Gotsman, C. Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality. Proc. of EUROGRAPHICS 1996, 1996.

[Tel01a] Teler, E., and Lischinski, D. Streaming of Complex 3D Scenes for Remote Walkthroughs. Proc. of EUROGRAPHICS 2001, 2001.

[Var02a] Varadhan, G., and Manocha, D. Out-of-Core Rendering of Massive Geometric Environments. IEEE Visualization 2002, 2002.