

# Centralized Spatial Data Structures for Interactive Environments

Joachim Günther, Florian Mannuß\*, André Hinkenjann†  
Bonn-Rhein-Sieg University of Applied Sciences

## ABSTRACT

Ray Tracing, accurate physical simulations with collision detection, particle systems and spatial audio rendering are only a few components that become more and more interesting for Virtual Environments due to the steadily increasing computing power. Many components use geometric queries for their calculations. To speed up those queries spatial data structures are used. These data structures are mostly implemented for every problem individually resulting in many individually maintained parts, unnecessary memory consumption and waste of computing power to maintain all the individual data structures. We propose a design for a centralized spatial data structure that can be used everywhere within the system.

**Index Terms:** D.2.11 [Software Engineering]: Software Architectures—Domain-specific architectures I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual reality

## 1 INTRODUCTION

Modern VE-Frameworks are evolving to modular and individually configurable component based systems. Every component is a self-contained system entity, a plug-in in our terminology, that can be loaded at run-time. Within the system every plug-in has its dedicated task, like rendering the scene, physical simulation or maintaining input/output devices.

Increasing computing power enables more complex calculations. Many of these calculations issue geometric search queries. Some examples are:

- Culling techniques like back face culling, view frustum culling or occlusion culling. One example of the use of data structures in these areas is hierarchical view frustum culling. Another example is hierarchical z-buffering [1], where an octree of the scene is used for occlusion culling.
- Collision detection queries and distance queries usually employ bounding volume hierarchies (compare [9])
- Visual rendering, especially ray tracing, is accelerated using hierarchical data structures, like bounding volume hierarchies, kd-trees or hybrid solutions [10]

The above mentioned search queries use spatial data structures to sort their data. Traditionally, every plug-in implements its own data structure. Consequences of this strategy are that implementations have to be repeated. This encompasses several tasks, like creating and updating the data structure, implementing geometric algorithms and interfacing to the VE-Framework. For every additional data structure stored in an individual plug-in all scene data has to be replicated leading to a waste of memory. Creating and updating the structures is also done for every data structure instance resulting in a waste of (computing) power.

---

\*e-mail: Florian.Mannuss@h-brs.de

†e-mail: Andre.Hinkenjann@h-brs.de

If the interface to the scene data (maintained by the VE framework) changes, every plug-in that uses it has to be re-adapted to these changes.

In addition, since the spatial data structure is usually an integral part of the corresponding plug-in, it cannot be replaced easily with an alternative. From a software engineering point of view this is an error-prone and time consuming solution.

We propose a solution where the spatial data structure(s) is implemented as a plug-in itself. All plug-ins/applications use a standardized interface to access it. All error-prone work blocks are moved to one central point and only one interface for querying the scene data from the VE-Framework has to be implemented. Possible implementation flaws need to be fixed at only one central position. Enhancements done here influence all plug-ins immediately.

The proposed component should offer the flexibility to switch the utilized spatial data structure at will, enabling us to keep track with new developments in the research on spatial data structures and the related algorithms. In Virtual Environments spatial data structures have to be interactive not only for querying the structure but also for updating it. Making the restructuring process interactive is a current research topic as in [11], [8] or [4]

A positive effect when using a data structure component is, that scientists can focus on solving their complex problems without the need to implement highly optimized spatial acceleration structures and the appropriate queries themselves. For example the developer of an assembly simulation needs to know the colliding objects of the assembled parts. For the collision detection an acceleration structure is needed and a centralized acceleration structure can be easily used. This separation has the additional advantage that the acceleration structure can be optimized on special purpose hardware with the result that the collision detection is computed for example on the GPU while the application is running on the CPU. Heterogeneous systems can be build using such component based system.

In this paper we present a methodology to integrate a multi purpose spatial data structure as an individual component into component based VE-Frameworks. As reference system for our implementation we use our VE-Framework *basho*[6], [7]. First we present related work to this topic and afterwards our design and first results.

## 2 RELATED WORK

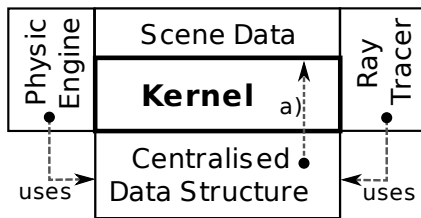
Due to the structure of traditional VE frameworks, until now central spatial data structures are not an area of active research. MAVERIK [3] implements a "spatial management structure" (SMS) as an integral part of its framework. In contrast, we propose to separate the data structures and their interfaces from the kernel. This way, the kernel stays compact and does not change with new features the developers demand from the data structures. Also, we propose to add appropriate query algorithms to the central data structure plug-in, allowing to implement e.g. efficient collision or ray tracing queries in the plug-in.

### 3 DESIGN

Our design uses a small kernel only responsible for management tasks. All functionality, like rendering, engines for scene manipulation, I/O devices are known by the kernel only through abstract interfaces. Implementations are loaded at run-time as individual plug-ins. Once the kernel is implemented there is hardly a need to modify it. Only the plug-ins, implementing the functionality should be modified and extended.

Every application should be independent from a certain plug-in implementation. Applications specify all needed plug-ins in an configuration file. This ensures the exchangeability of plug-ins within one application. However, there can be dependencies between plug-ins. Data changes are propagated via messages through the whole system and plug-ins or applications can register themselves for those messages. There are system messages, like object created, altered or deleted and every plug-in or application can define its own messages, if needed.

The design of the centralized spatial data structure must fit into these demands. Figure 1 shows a rough sketch how the Centralized Data Structure (CDS) will be embedded into the system.



**Figure 1:** Idea how the centralized spatial data structure is used. The physics engine and the ray tracer query the CDS and the data structure itself queries the scene data (a) via the kernel in order to update their data.

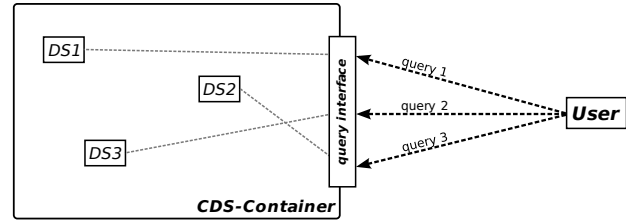
Most VE-Frameworks embed a scene graph as data structure storing the graphic primitives. In our case the scene graph and therefore the graphic primitives are stored in a separate plug-in in order to be independent from the scene graph implementation. Furtheron, all scene data is encapsulated in *virtual objects*. These objects are part of the kernel and represent entities like a pen or tea pot, not simply a transformation or a geometry node. Manipulating the scene is done only through these virtual objects.

Our aim with integrating a centralised spatial data structure in our system is not to replace the scene graph. Instead it is a separate acceleration structure as sketched in figure 1 that can get utilised by other plug-ins or application code. Manipulating the scene using the virtual objects leads to changes in the scene graph. After objects are changed the centralised spatial data structure queries the changes via the kernel.

An optimal central spatial data structure would be one that can process every imaginable geometric search query in an optimal way. Unfortunately, such a spatial data structure does not exist. On one hand there are queries that can only be processed by special data structures and on the other hand a data structure that can process different queries cannot process all queries as fast as a special designed data structure can do. Nevertheless certain queries can be grouped and processed by a single data structure. The task is to realize a superordinate structure that can maintain arbitrary queries for different data structures.

Accessing the centralized data structure while developing an application or plug-in for the VE-Framework has to be as simple as possible. Ideally, as sketched in figure 2 only the query has to

be specified and passed over to the CDS without the need to worry about how the operation will be processed in order to receive the result.



**Figure 2:** The cds routes user queries to the respective data structures

In order to be as flexible as possible the design has to meet certain demands. First, queries and data structures need to be separate entities and it must be possible to assign a certain query operation to the best suited data structure through configuration. In addition, replacing a data structure should be a matter of changing the configuration leading to the ability to extend the pool of supported data structures and queries supported through the CDS.

In order to provide a maximum flexibility for configuration we implement a container called *CDS-Container*, sketched in figure 2, as storage and management facility. This CDS-Container is implemented as a single plug-in. Every implemented data structure is a separate plug-in, too (called *CDS plug-in* in the remainder of the paper.) Besides the implementation of the data structures the implementation of all supported queries is part of this plug-in.

The CDS-Container can be compared with the VE kernel as both are only managers and once implemented there should be no need for many changes.

The query interface in figure 2 is part of the CDS-Container and is the central access-point. It is used to request the desired query object. A query object is used to forward the request to the chosen data structure. Every query type has its special parameter set and return values. For a ray-object intersection we pass the ray as a parameter to the query and get the nearest hit object or all hit objects with the intersection point and its normal as return value. In case of a collision query we ask if two objects collide or request a list of all collision pairs. This list can be arbitrarily extended and as a consequence there can't be one interface to describe all queries. Instead we define an abstract interface class hierarchy. Every query type has to define an interface as part of the class hierarchy. These interfaces are then returned by the query interface and they have to be implemented by the data structures.

```
class cdsCollisionQuery : public cdsQuery {
    virtual cdsQuery* createQueryObject() = 0;

    // Detects all colliding object pairs
    // and returns a list of those
    virtual void getCollisionPairs( CollPairList &cpl ) = 0;

    // Detects whether a collision between
    // the two objects consists or not.
    virtual bool collide( virtualObject *a,
                        virtualObject *b ) = 0;
    ...
};
```

**Figure 3:** Part of the interface for the collision detection query. *cdsQuery* is the base class of the query class hierarchy.

Figure 3 shows an example fragment of the collision detection query interface. This query interface does not belong to any CDS plug. The CDS-Container returns only the base class `cdsQuery` and it is the users task to down-cast into the right interface type. Due to their pure virtual nature they are an arbitrary collection of header-files. It is important that every data structure offering for example collision detection implements this interface in order to keep the data structures exchangeable.

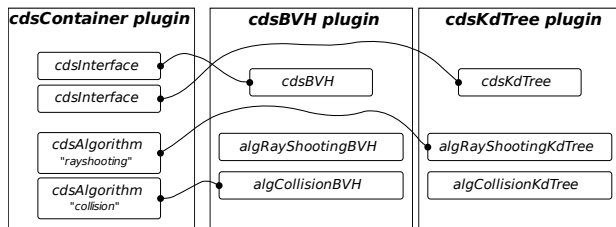
All data structures that have to be loaded and the utilized queries are specified through configuration. Information that has to be provided are the plug-in names of the data structures and the assignment of the queries to the data structures that are responsible for processing the request.

While the CDS plug-in can be used in many other software frameworks, we have used it in our own VE framework *basho*. In our case the CDS-Container plug-in registers itself as receiver of plug-ins and instructs the kernel to load the specified CDS plug-ins. All loaded CDS plug-ins are passed over to the CDS-Container.

```
Extension cdsContainer gds
DataStructures VecString 2 cdsBVH cdsKdTree
Queries VecString 4
  RayShooting cdsKdTree
  Collision cdsBVH
!Extension
```

**Figure 4:** Example configuration of the CDS-Container plug-in as it is done in the *basho* VE-Framework.

Figure 4 shows one possible configuration of the CDS-Container plug-in. After the `DataStructures` keyword all CDS plug-ins that need to be loaded are enumerated. These plug-in names are used in the `Queries` section to bind the queries to the desired data structure. In the example we use the kd-tree plug-in for `RayShooting` and the bounding volume hierarchy for `Collision`. The same query names (e.g. *RayShooting*) are used to request a query object via the query interface.



**Figure 5:** The *cds* routes user queries to data structures

Using the configuration from figure 4 the resulting plug-in structure is shown in figure 5. The CDS-Container stores references to all loaded data structures and queries. As the central control instance the CDS-Container has two major tasks. First, processing the user queries via the query interface and second to control the creation and update process of the loaded data structures.

After a CDS plug-in has been loaded it registers all its query objects using the plug-in and query name as key tuple using the prototype factory inside the CDS-Container. If a query has to be processed the container uses the configuration to find the data structure that is configured to process the query and builds the key tuple. This tuple is used by the factory to find the right prototype. A new object is created by calling `createQueryObject()` (see figure 3) from the prototype object and is then returned to the user.

In *basho* plug-ins are passive components. They are either called from the kernel or they have to register themselves for all messages they are interested in. The CDS-Container responsible for managing the creation/update process of all loaded data structures registers itself for *insert*, *update* and *delete* messages for the scene objects. After receiving a message the container queries the corresponding object for its changes and calls the appropriate method of all loaded data structures. All interface methods a data structure has to implement for creation and update are shown in figure 6.

```
class cdsInterface : public extensionInterface {
  // Add the object to the data structures
  virtual bool add( virtualObject *vo ) = 0;

  // The transformation of the object has been changed.
  virtual bool update( virtualObject *vo ) = 0;

  // The geometry of the object has been changed.
  virtual bool rebuild( virtualObject *vo ) = 0;

  // Remove object
  virtual bool remove( virtualObject *vo ) = 0;

  // Called at the end of the frame to indicate
  // that all changes of the actual frame appeared.
  virtual bool build() = 0;
};
```

**Figure 6:** Interface methods a data structure has to implement for the creation and update process.

The `update()` method will be called if only the transformation of the object has changed and `rebuild()` if the geometry of an object has changed. Depending on the implementation of the data structure in the `update()` case only the new transformation has to be set, but in the `rebuild()` case the data structure has to be rebuild, making such a distinction useful. The data structures are only passive receivers of `add()`, `update()`, `rebuild()` and `remove()` calls with the task to reorganize their structure. However, it is not useful to start this restructuring process every time one of those methods is called since it is a time consuming process. It is much more efficient to buffer requests and do the restructuring at the end of every frame. The `build()` method is used for this task. At the end of every frame the last method the container calls is the `build()` call.

Using the naming schema and the described interfaces for the queries and data structures enables the design decision to implement the container once. By adding new algorithms and data structures changes are rarely needed.

## 4 IMPLEMENTATION AND RESULTS

Our example implementation of the centralised data structure includes a bounding volume hierarchy (BVH) using the update strategy as described in [5]. Supported queries are collision detection and ray shooting. The other data structure is a kd-tree with SAH construction heuristic as in [2]. Here, ray shooting is implemented as a query example.

Both data structures are using a two level hierarchy. Every object of the scene is added into its own data structure in local coordinates. All local data structures are then added into one top level structure using the transformation matrices of the objects to transform the local structures to world space. In case of an `update()` operation only the matrix of the local structure and the top level structure, storing only the bounding boxes of the local structures, have to be

updated. Our VE-Framework handles scene object as entities, like an table, chair or telephone thus making this two level structure a natural choice.

In order to test the CDS-container we implemented a ray tracer as a rendering plug-in and a collision detection engine plug-in that colours colliding pairs yellow. Using the CDS-container these plug-ins do not need to implement ray intersection or object collision detection strategies. The developer can choose between a kd-tree or the BVH implementation. Switching to the kd-Tree is done by simply replacing *cdsBVH* with *cdsKdTree* after *RayShooting* in the configuration shown in figure 4. No changes are needed in the source code.

## 5 CONCLUSION AND FUTURE WORK

We proposed the design of an easily extendable centralised data structure for interactive environments. This data structure is composed of a management container storing the queries and different data structures. Which data structures are loaded and which data structure is responsible to process a certain query is free configurable. The container and the data structures with their supported queries are stored in independent plug-ins. This way, only those components are loaded that are needed. If no centralised data structure is needed, there is then no need to load the plug-ins.

Currently the example implementation is running only single threaded. With an increasing number of CPU-cores it is important to support parallel data structure construction and updates as well as parallel execution of search queries.

Another feature yet to be implemented is run-time configuration. This includes the ability to add or remove data structures and to be able to switch the assignment of one algorithm to a certain data structure at run-time. Such a feature is useful in order to find out the optimal query/data structure combination for one problem by interactive testing or some automatic optimisation process.

## ACKNOWLEDGEMENTS

This work was partially sponsored by the German Federal Ministry of Education and Research (BMBF) under grant no 1762X07

## REFERENCES

- [1] N. Green, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 231–239. ACM, ACM Press / ACM SIGGRAPH, 1993.
- [2] V. Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.
- [3] R. Hubbard, J. Cook, M. Keates, S. Gibson, T. Howard, A. Murta, A. West, and S. Pettifer. Gnu/Maverik: A Microkernel for Large-Scale Virtual Environments. *Presence: Teleoper. Virtual Environ.*, 10(1):22–34, 2001.
- [4] T. Ize, I. Wald, and S. G. Parker. Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization*.
- [5] C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha. Rt-deform: Interactive ray tracing of dynamic scenes using bvhs. Technical report, Department of Computer Science, University of North Carolina at Chapel Hill, 2006.
- [6] F. Mannuß and A. Hinkenjann. Towards better quality in virtual environments. In *Proc. of Eurographics Workshop on Virtual Environments*, pages 223–224, Aalborg, Denmark, 2005.

- [7] F. Mannuß, A. Hinkenjann, and J. Maiero. From scene graph centered to entity centered virtual environments. In *Proc. of Workshop on Software Engineering and Architectures for Realtime Interactive Systems, in conjunction with IEEE VR 2008*, Reno, USA, 2008.
- [8] M. Shevtsov, A. Soupikov, and A. Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum*, 26:395–404(10), September 2007.
- [9] M. Teschner, S. Kimmerle, G. Zachmann, B. Heidelberger, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnat-Thalmann, and W. Strasser. Collision detection for deformable objects. In *Eurographics State-of-the-Art Report (EG-STAR)*, pages 119–139. Eurographics Association, Eurographics Association, 2004.
- [10] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports*, 2007.
- [11] R. Weller and G. Zachmann. Kinetic separation lists for continuous collision detection of deformable objects. In *Third Workshop in Virtual Reality Interactions and Physical Simulation (Vrphys)*, Madrid, Spain, 6–7 November 2006.