GraphPool: A High Performance Data Management for 3D Simulations

Patrick Lange, Rene Weller, Gabriel Zachmann
University of Bremen, Germany
{lange,weller,zach}@cs.uni-bremen.de

ABSTRACT

We present a new graph-based approach called GraphPool for the generation, management and distribution of simulation states for 3D simulation applications. Currently, relational databases are often used for this task in simulation applications. In contrast, our approach combines novel wait-free nested hash map techniques with traditional graphs which results in a schema-less, in-memory, highly efficient data management. Our GraphPool stores static and dynamic parts of a simulation model, distributes changes caused by the simulation and logs the simulation run. Even more, the GraphPool supports sophisticated query types of traditional relational databases. As a consequence, our GraphPool overcomes the associated drawbacks of relational database technology for sophisticated 3D simulation applications. Our GraphPool has several advantages compared to other state-of-the-art decentralized methods, such as persistence for simulation state over time, object identification, standardized interfaces for software components as well as a consistent world model for the overall simulation system. We tested our approach in a synthetic benchmark scenario but also in real-world use cases. The results show that it outperforms state-of-the-art relational databases by several orders of magnitude.

CCS Concepts

•Theory of computation → Data structures and algorithms for data management;

Keywords

3D Simulation System; Graph Database; Nested Hash Maps; Simulation Database

1. INTRODUCTION

Today, there are numerous 3D simulation applications available including virtual testbeds for space robotics or industrial automation and many more. The goal of such 3D simulations is usually to simulate a given model and to provide the users visual feedback, most often in real-time. Usually, many independent inhomogeneous software components need to communicate and exchange data in order to simulate the model as well as to provide data for the visual feedback [20, 23]. This data exchange is usually done concurrently in highly parallel manner in order to preserve a fast simulation and immersive visual feedback to the user. Therefore, current simulations rely on some kind of data which is concurrently shared between all software components. For instance the 3D geometries of the objects, like the spacecraft and its' individual components in spaceflight simulations, but also their dimensions, their mass, their position and orientation in the world space and other physical properties that are required for the simulation. During the simulation runs, several components need access to that data, e.g. input/output devices, the renderer, a physically-based simulation component, etc. The data is not always pre-defined and fixed, but it may change during the simulation run. For instance, the physically-based simulation module changes the positions and orientations according Newtons laws of motion. Moreover, it is possible that several simulation components, like input devices controlled by the user and the physically-based simulation, want to access and manipulate the same data at the same time.

To summarize, 3D simulations (and simulations in general) require a data management that is easy to handle and guarantees fast access to data for both, reading and writing while maintaining a consistent simulation state even in heavily concurrent access scenarios [17].

Currently, relational databases are often used for this task. They are well-researched, easy-to-use and deliver out-of-the-box functionality for a consistent data management. Unfortunately, they also have some drawbacks when considering 3D simulation applications.

For instance, they do not scale well to massively parallel access due to their inherent serialization of access queries. Moreover, the relational data model requires a strict definition of a schema (consisting of tables with the defined data fields in row-column format) prior to storing any data. This constraints typical simulation engineering tasks such as capturing new simulation data which was previously not considered or introducing simulation behavior changes due to new data formats and content. Finally, simulation application developers usually use object-oriented programming languages to build 3D simulation applications as handling object-oriented data is nowadays most efficient.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGSIM-PADS '16, May 15 - 18, 2016, Banff, AB, Canada

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-3742-7/16/05. . . $15.00

DOI: http://dx.doi.org/10.1145/2901378.2901379

In contrast to this, the data needs to be collected from many tables (often hundreds or thousands in today's simulation applications) and combined before it can be provided to the application. Similarly, when writing data, the write access needs to be coordinated, separated and performed on many tables [6]. This results in a fundamental mismatch exists between the way a simulation application would like to see its data and the way it's actually stored in a relational database.

We present a novel approach that overcomes these disadvantages of relational database technology for the use in 3D simulations.

Our approach uses wait-free hash map techniques in graph-based schema-less, in-memory resident manner in order to store object-oriented content. As a result, the time-consuming serialization as well as table-based coordination and separation of relational databases are eliminated. Even more, the wait-free hash map techniques allow high performance access even for massive numbers of concurrent read and write operations. Consequently, our data management incorporates a highly responsive low-latency data access for any number of simulation components accessing it. Finally, our approach implements the same functionality as state-of-the-art relational databases such as aggregate queries as well as caching strategies.

The system can be used to flexibly build simulation applications in various fields of applications, like high fidelity end-to-end spaceflight simulations [19] or self-optimizable virtual testbeds [21].

In summary, our contribution of this paper is a centralized data management approach for high performance 3D simulations that incorporates

- high scalability due to wait-free access for all simulation components to the simulation state
- high performance because it is completely in-memory resident
- high adaptability due to graph-based schema-less data storage of object-oriented content

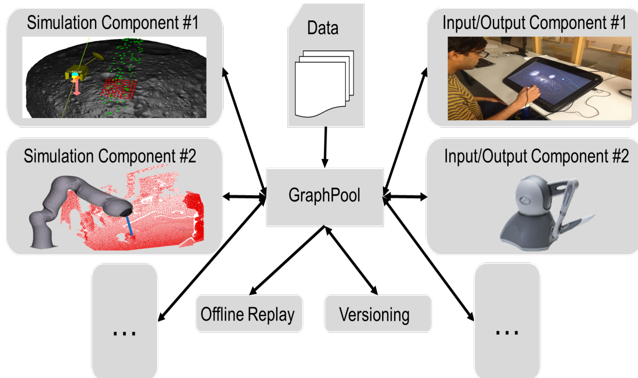

Figure 1: Architecture of a 3D simulation system, using our GraphPool approach: All software components concurrently access the centralized GraphPool which stores the complete simulation data.

Another advantage of our GraphPool is the support of all common kinds of query operations like storing and managing static and dynamic data while enabling sophisticated query types of traditional relational databases. Furthermore, our GraphPool incorporates a versioning mechanism which generates a queryable archive of the complete simulation. As a result, simulation components can be used in an online viewing mode to replay a simulation run step by step, allowing analysis and debriefing. Figure 1 gives an overview of the overall architecture.

2. RELATED WORK

Research in combining database, simulation and rendering methodology has attracted increasing interest in the last decade because databases have been integrated into 3D simulation systems in many different ways. Though many attempts have been made to incorporate database technology into 3D simulation systems, to our knowledge, no one has used in-memory schema-less technology with wait-free access behavior before.

State-of-the-art research in the integration of database technology into 3D simulations use standard full-fledged SQL databases because they are easy-to-use and deliver out-of-the-box functionality for a consistent data management. [17, 13] introduced schema and data synchronization for distributed 3D simulations with a versioning interface. In more basic applications, databases have been used to store additional meta-information (e.g. about scene objects [25, 3]). More sophisticated approaches use the database to store the scene data itself [25], where some do support collaboration [9, 24, 5, 15, 16] while others do not [1, 7]. A flexible support for different data schemata is not widespread among these systems [5, 7, 16]. The simplest realizations allow schema alteration by adding attributes to generic base objects [11]. The more advanced systems support different static [12] or dynamic [7] schemata. However, these data management approaches can only alter their relational table schema based on a new schema delivered by another simulation architecture component (e.g. a simulation server). Consequently, this schema alteration is done manually by hand and is only distributed automatically. In all applications, the table schema alteration is complex and computationally expensive.

To summarize, the above mentioned related studies were focussed on combining traditional relational databases with simulation technology. However, traditional database technology has three main technical limitations:

- the adaptability to object-oriented data due to rigid table-based schema
- the scalability to massive amounts of components accessing the database in real-time manner
- the performance with respect to massively parallel read and write operations due to serialization of access queries

The database research community established in-memory resident databases and the NoSQL ("Not-only" SQL) methodology to compensate for these technical limitations shared by the majority of relational database implementations. NoSQL started out as industry developments in companies such as Amazon, Google, Twitter or Facebook which discovered these serious limitations of relational database technology [10].

In order to overcome these limitations, database architects had sacrificed many of the most central aspects of relational databases, such as joins and fully consistent data, while introducing many complex and fragile pieces into the operations puzzle. They simplified the database schema and introduced various query caching layers. Finally, schema devolved from many interrelated fully expressed tables to something much more like a simple key/value look-up in an attempt to address these new requirements. [6].

Relational and NoSQL data models are very different. The relational model takes data and separates it into many interrelated tables consisting of rows and columns. These tables reference each other through foreign keys that are stored in columns as well. Every piece of data is then stored only once in one table. Consequently, the relational model minimizes the amount of storage space required, which was a key requirement when relational database were created due to expensive hardware [6]. However, space efficiency comes at expense of increased complexity when inserting and looking up data. Developers generally use object-oriented programming languages to build 3D simulation applications as handling object-oriented data is nowadays most efficient. In contrast to this, the data needs to be collected from many tables (often hundreds or thousands in today's simulation applications) and combined before it can be provided to the application. Similarly, when writing data, the write needs to be coordinated, separated and performed on many tables [6]. Consequently, a fundamental mismatch exists between the way a simulation application would like to see its data and the way it's actually stored in a relational database.

Another major difference is that relational technologies have rigid schemas while NoSQL models are schema-less [6]. The relational data model requires a strict definition of a schema (consisting of all tables with the defined data fields in row-column format) prior to storing any data. This requirement makes typical simulation engineering tasks such as capturing new simulation data which was previously not considered or introducing simulation behavior changes due to new data formats and content extremely disruptive and frequently avoided.

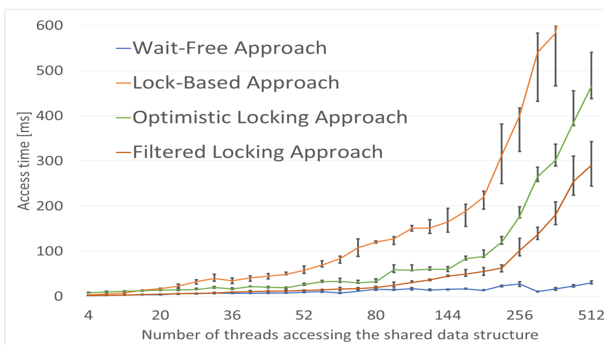

Figure 2: Performance comparison of wait-free and lock-based concurrency control management implementations, which are traditionally used for simulation applications. Adopted from [22].

This is the exact opposite of the desired behavior in the area of simulation and modelling, where developers need to rapidly, and constantly, incorporate new types of data to enrich their simulation models and applications. In comparison, schema-less databases allow the format of the data being inserted or changed at any time, without application disruption [6].

When introducing such a data management we will immediately encounter the well-known problem of concurrent data structures and race conditions which constitutes another challenge of implementing a centralized solution.

In the past, several concurrency control management (CCM) approaches have been proposed to solve this kind of parallel access. In order to avoid problems of traditional lock-based CCMs such as thread starvation or deadlocks, wait-free approaches based on hash maps for realtime interactive systems had been introduced [20, 22, 23]. Wait-free approaches guarantee access to the shared data structure in a finite number of steps for each thread, regardless of other threads accessing the shared data structure by introducing a few atomic operations [18]. This means that these approaches do not need any traditional locking mechanism in order to preserve a consistent data state. Experiments have shown a superior performance of wait-free approaches with respect to traditional locking approaches as Figure 2 illustrates. These wait-free approaches not only support structured data such as arrays or list but also use fast hash key operations in order to find and retrieve the stored data inside the used hash table. Due to their excellent scalability, they are perfectly suited for simulation applications which need to support massive parallel access. Consequently, using wait-free data structures as a data access backbone can highly improve the performance and scalability of a simulation data management.

3. GRAPHPOOL CONCEPT

In this section, we describe the three concepts of our novel GraphPool approach which overcomes the limitations of the presented related work.

First, in order to improve the overall scalability for massive amounts of concurrent simulation components, we introduce a sophisticated wait-free concurrency control management based on hash maps. Second, in order to improve the overall adaptability and performance of the data management, we use object-oriented data formats as data storage backbone. As a result of this, the time consuming separation of data into interrelated tables of relational database technology is eliminated. Third, in order to provide a comparable data management system to common relational databases, we implement relational core and aggregate queries within our approach.

In order to implement these concepts, our approach introduces a central world state (CWS), based on wait-free access using data replication [20, 22]. The CWS is stored in our GraphPool, acting as a centralized data management. This means, the GraphPool is used for storing and managing all parts, dynamic as well as static, of the shared simulation model in a consistent object-oriented data schema. This object-oriented approach correlates to typical 3D construction and environmental data used in 3D simulation applications.

During runtime, every simulation component can replicate any parts of the CWS into its local world state (LWS)

while local changes are tracked and written back into the CWS. These read and write processes execute in wait-free behaviour, without synchronisation [20, 22]. As mentioned above, not only are all static parts of a 3D simulation model (e.g. the 3D environment) are stored in the GraphPool, but also all dynamic objects which are changed by the running simulation. These changes are likewise written to the CWS, hence communicating the new state of the simulation model to the CWS. Consequently, the GraphPool drives the simulation itself as it represents the central communication (dataflow and workflow) hub.

Many more advantages arise from using centralized data management system for a simulation system [17]: Different applications (e.g. for authoring) can be employed using its standardized interfaces, an inherent rights management provides means for fine grained access control, consistent data schema, solution to object identification ("id problem"), and (spatial) queries allow very selective loading and changing of the simulation model.

As described in [20, 22], we developed wait-free synchronization methods based on local and global guarding principles with atomic operations allowing simulation components to efficiently access hash maps. Wait-free concurrency approaches guarantee access to the shared data structure in a finite number of steps for each thread, regardless of other threads accessing the shared data structure. Consequently, wait-free approaches deliver high performance access even for massive numbers of concurrent components as evaluations have shown [20, 22].

In order to achieve this wait-free behavior for all data transactions, we identify all kinds of data that need to be shared between different components, e.g. simulation time or the transformations of the objects in the scene. This data is arranged into logically structured data packets. For instance, the 3D geometry of a car but also its' individual components in a car simulation, such as mass, position, velocity or acceleration are logically arranged into one data packet. We store these data packets in a hash map and assign a set of unique key-identifiers (object-key and member-keys) to each of these data packets. The member-key references the complete data packet while member-keys reference a specific data type within the packet. This combination of identifiers and hash map bucket is denoted as GraphNode. All GraphNodes are registered in our GraphPool and memory is reserved for the data. The GraphPool connects all GraphNodes into a graph-based lookup structure and constructs thereby the CWS. If any component wants to access the data, it simply has to look up the key in the GraphPool.

Hash maps outperform other container types (e.g. lists or arrays) due to their constant lookup, insertion and deletion time of $\mathcal{O}(1)$ which makes them perfectly suitable for a high performance data management. Our nested hash map approach enables row and column based queries for the stored data. From a relational database point of view, a GraphNode is a 1-by-n schema-less table in which all columns n can be accessed separately by n member-keys. Figure 3 illustrates this concept with an simple person data example while Figure 4 illustrates the main concept of the GraphPool consisting of GraphNodes.

As all intermediate states of the simulation are made persistent in the GraphPool, a simulation run can easily be captured by our versioning mechanism. This versioning mechanism generates a time-stamped history of all GraphNodes. These recorded time-stamped GraphNodes represent a queryable archive of the complete simulation. Every simulation component can be used in an off-line viewing mode to replay a simulation run step by step, allowing analysis and debriefing of the complete simulation.

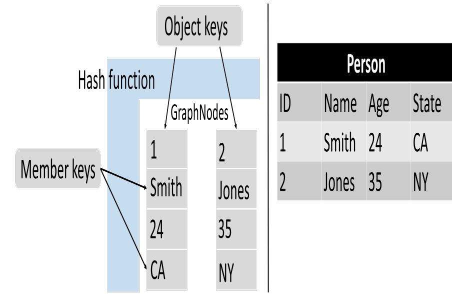

Figure 3: **Comparison of GraphPool approach and rigid table schema: In contrast to a rigid table schema (right), GraphNodes store the data in a nested hash map. The data is then available via object- and member-keys (left).**

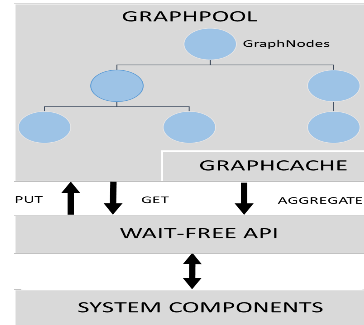

Figure 4: **Access workflow of system components using the GraphPool. The stored data is available is via relational core (put, get) and aggregate functions.**

3.1 Property Graph Model for Nested Hash Maps

In order to allow for relational core and aggregate functions, we arrange the GraphNodes in a property graph structure. We define this graph structure as $\mathcal{G} = \{\mathcal{N}, \mathcal{R}, \mathcal{K}, \mathcal{P}\,\mathcal{L}\}$ with $\mathcal{N}$ nodes, $\mathcal{R}$ relationships, $\mathcal{K}$ keys, $\mathcal{P}$ properties and $\mathcal{L}$ labels.

The property graph contains connected GraphNodes which can hold any number of properties within its hash map. GraphNodes can be tagged with labels representing their different roles in the simulation domain. Labels can serve as a contextualization for GraphNode and relationship properties. Furthermore, labels may also denote constraint or metadata information of GraphNodes.

Every relationship provides a directed, named semantically relevant, connection between two GraphNodes. A relationship always has a direction, a start node, an end node and a type.

The relationship type can be arbitrary, for instance a weight, cost, time interval, distance or inheritance/tree structure. GraphNodes can share any number or type of relationships because they are stored efficiently, without sacrificing performance. In order to enable fast traversion of the GraphNodes, the GraphPool can navigate between GraphNodes regardless of relationship direction.

Furthermore, we follow the consistent rule that no broken links shall be present in the graph. Since a relationship always has a start and end GraphNode, a GraphNode can not be deleted without also deleting its associated relationships. Consequently, an existing relationship will never point to a non-existing endpoint. Moreover, the GraphPool provides a versioning mechanism that archives every previous state of the GraphNodes as a time-stamped version. This mechanism provides transparent access to these historic states. A user interface element or any other component of the simulation system can then set a reference time and the versioning interface takes care of reloading the appropriate versions of the object data. Furthermore, there is no global main loop required; each simulation component can access the GraphNodes, i.e. read or write, at any point in time.

Figures 5 and 6 illustrate our property graph structure.

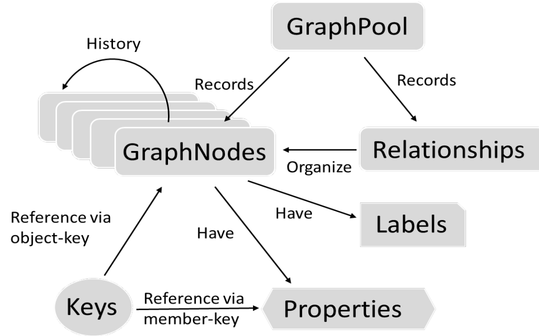

Figure 5: The building blocks of our property graph model: The GraphPool consists of linked GraphNodes which can be accessed via their object- and member-keys.

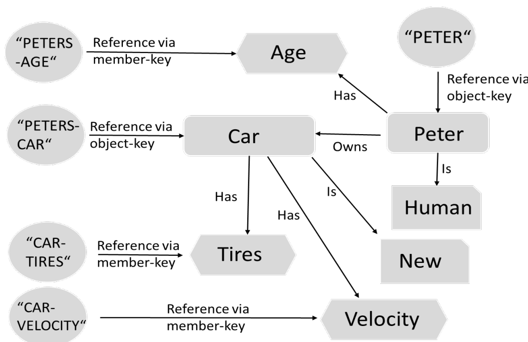

Figure 6: Property graph model example illustrating the use of member- and object-keys, relationships as well as labels.

3.2 Relational Core & Aggregate Queries

In this section, we described how relational core and aggregate queries can be implemented within our property graph structure with caching. The GraphPool has to provide two relational core functionalities:

- Pushing a local world state (LWS) to the central world state (CWS), respectively putting values into the GraphPool
- Retrieving a local world state (LWS) from the central world state (CWS), respectivey getting values from the GraphPool

The CWS is thereby defined by the complete set of GraphNodes which are stored in the GraphPool. As a result, the LWS is a subset of these GraphNodes which a simulation component can access by the GraphPools put and get function.

The put function is used to update a GraphNode via its object-key in the GraphPool. If the object-key is not already stored in the pool, it simply creates a new GraphNode. Otherwise the existing GraphNode will be updated. The value can be retrieved in constant time using our hash function as described below. The get function is used to retrieve an existing GraphNode from the GraphPool.

We presented local [20] and global [22] guarding principles as well as merge strategies [22] for solving this kind of access in wait-free manner. In short, we differentiate between consumer and producer simulation components. Consumer components only read a set of GraphNodes whereas producer components read and write a set of GraphNodes. Therefore, each GraphNode maintains two copies of the data, a producer reference and a consumer reference. These references are used from the corresponding simulation components. This means, that read requests of a GraphNode will return the consumer reference and that write requests of a GraphNode will return the producer reference.

If a consumer wants to read a value, it calls the get function and the GraphPool returns the current consumer copy. This is decided via an access request which every get query has to contain. Moreover, it increments a local atomic marker (see Algorithm 2) of the consumer reference. If the consumer has finished reading, the consumer decrements the local marker again. In addition, it checks whether the local marker is zero and, in case no consumer is reading it anymore, deletes the consumer reference. If the memory can not be directly deleted, the GraphPool will take care of releasing the memory at a later time point as described in [22].

Writing access also begins with a call of the get function in order to retrieve the data which should be manipulated. In this case, the GraphPool returns the producer reference and sets the ownership of the system component. This ownership is an atomic id of the producer reference, which is set and checked in the get and put function. In the get function, the producer reference is marked with the corresponding producer id. When the write operation is conducted, the producer checks whether its id is the current one. If this is not the case, another producer has udpated the producer reference in the meantime (see Algorithm 1). This means that another system component has changed the GraphNode and the changes have to be merged in order to preserve a consistent GraphNode state. The needed merge is then implemented as described in [22]. In short, conflicting producer references are sorted into a producer queue and the GraphPool calls a merge function that processes the merges

of those GraphNodes. In order to do so, every GraphNode contains a merge strategy (e.g. first-come first-serve or averaging the values). Algorithms 1 and 2 illustrate the implementation.

In contrast to the above introduced relational core functionalities which use single data, relational aggregate functions use multiple data. Aggregate functions are essential functions of relational databases. These functions collect in their original implementation the values of multiple columns and rows. They use this collection as input on certain criteria which further filter the result. Typically, selective (equal, not, smaller, greater, between) and numerical (average, min, max, sum) operators are most commonly used for aggregate functions.

Algorithm 3 illustrates the general implementation of an aggregate function in our graph structure. First, the corresponding hash is determined. If the result of the aggregate function was computed before, we take the value from the GraphCache. If not, we recalculate the result of the aggregate function. In order to do so, we collect the corresponding data and apply the associated aggregate function onto this data and store the result in the GraphCache.

Algorithm 1 GraphPool::put($\mathcal{K}$ object-key, $\mathcal{V}$ value)

```
R retired graph node
if K ∈ GraphPool then
    N graph node = GraphPool[K]
    if V_Id = N.Producer.Id then
        N.Producer = V
        R = N.Consumer
        N.Consumer = V_clone
    else
        N.Producer.Queue(V)
        R = N.Consumer
        GraphPool.notify
    end if
else
    GraphPool.insert(pair(K,V))
end if
GraphCache.update(K)
return R
```

Algorithm 2 GraphPool::get($\mathcal{K}$ object-key, $\mathcal{A}$ access)

```
if K ∉ GraphPool then
    return empty
else
    N graph node = GraphPool[K]
    if A is producer then
        N.Producer.Id = A.Id
        return N.Producer.Clone
    else
        N.Consumer.MarkerIncrement
        return N.Consumer
    end if
end if
```

3.3 Wait-Free Caching

Caching is widely used in database technology to store results of expensive aggregate query results. This enables the database to quickly deliver previously computed results. We also provide a caching strategy based on a tree data structure, called GraphCache.

The GraphCache supports two types of workflows. First, if a GraphNode is updated in the GraphPool from a system

Algorithm 3 aggregate($\mathcal{K}$ object-keys, $\mathcal{I}$ member-keys, $\mathcal{A}$ aggregator)

```
H = getHash(K, I, A)
if H_valid then
    return GraphCache.get(H)
end if
C = empty collection
for K_i ∈ K do
    N GraphNode = GraphPool[K_i]
    C += N[I]
end for
R = A(C)
GraphCache.set(H,R)
return R
```

component by calling the put function, the associated stored data in the GraphCache is marked as outdated. Second, if an aggregate query is used, either a cached result is returned or the associated nodes in the GraphCache are marked as valid and the corresponding data is updated.

For the first case, the GraphPool has to support a GraphCache traversal via object-key in order to find those hash values which (partly) consist of the given GraphNode. For the second case, the GraphPool needs to support a traditional cache traversal via hash value in order to find the corresponding cached query result.

Consequently, our GraphCache is accessible via its two roots: the key-root and the hash-root. This enables fast access because unnecessary tree traversal is avoided (see Figure 7).

Due to the main principle of wait-free access of the underlying concurrency control management [20, 22], we propose a wait-free caching approach in order to maintain overall wait-free access control of the GraphPool. When a wait-free data management system is implemented, every access workflow to the stored data has to be wait-free, in order to guarantee the wait-free behavior of the complete system [22]. Therefore, we initialize the complete GraphCache at simulation startup. This initialization at startup has the advantage that cache entry insertion and deletion does not have to be implemented in wait-free manner, but only the update. This update process can be implemented with an atomic boolean, which is used as an indicator that sores whether a cache entries is outdated or not. The GraphCache initialization involves all possible combinations of object-, member-keys and aggregate query types because the queries conducted by the system components are unknown. This results in a tree structure with $o \cdot m \cdot a$ nodes, where o is the number of object-keys, m is the number of member-keys and a is the number of aggregate types.

In detail, the initialization of the GraphCache involves the creation of two root nodes, the key-root $\mathcal{KR}$ and cache-root $\mathcal{CR}$. These roots are created at first. The GraphCache further consists of three node levels: object-keys, member-keys and aggregate types. In order to generate all hash entries for all possible combinations of object-keys, member-keys and aggregate keys, we iteratively combine them: Object-keys are added to $\mathcal{KR}$ as nodes and all member-keys are added to the object-key nodes. Finally, all aggregate types are added to the member-key nodes (see Algorithm 4). After the initialization, the GraphCache can be directly used for caching operations.

Algorithm 4 initGraphCache($\mathcal{O}$ object-keys, $\mathcal{M}$ member-keys , $\mathcal{A}$ aggregate types)

```
KR = key root
CR = cache root
for O_i ∈ O do
  R = node with O_i
  for M_i ∈ M do
    T = node with M_i
    for A_i ∈ A do
      U = node with A_i
      T_{i-childs} += U
    end for
    R_{i-childs} += T
    CR_{childs} += hash(O_i, M_i, A_i)
  end for
  KR_{childs} += R
end for
```

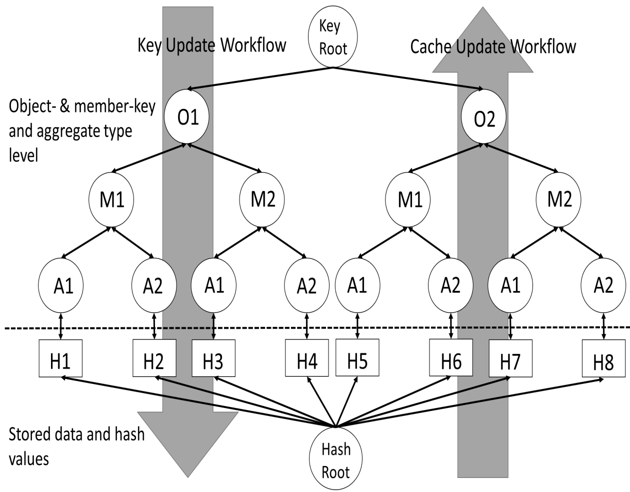

Figure 7: The GraphCache: Updates based on object keys are traversed via the key-root while updates based on hash values are traversed via the hash-root.

Algorithm 5 hash($\mathcal{K}$ object-keys, $\mathcal{M}$ member-key, $\mathcal{A}$ aggregate-type)

```
V = empty hash value
P = prime number
H = hash function
for K_i ∈ K do
  V = V · P + H(K_i);
end for
V = V · P + H(M);
V = V · P + H(A);
return V
```

The GraphCache contains a large number of cache entries for sophisticated simulations. We use a uniform distribution of hash values in order to avoid collisions for cache lookup. In order to deliver such a uniform distribution of hash values, even for massive amounts of cache entries, we use a prime-based hash generation in order to generate unique hash values for all concatenations of object- and member-keys with respect to all defined aggregate functions (see Algorithm 5).

Algorithm 6 prune($\mathcal{O}$ object-keys, $\mathcal{M}$ member-keys, $\mathcal{A}$ aggregate-types)

```
for N ∈ KR do
  if N_key ∈ O then
    remove N and all children from KR
  else
    for C ∈ N_childs do
      if C_member ∈ M then
        remove C and all children from N
      else
        for M ∈ C_childs do
          if M_aggregate ∈ A then
            remove M and all children from C
          end if
        end for
      end if
    end for
  end if
end for
```

However, most of these possible key combinations will never be used during runtime. In order to reduce the memory overhead, we propose a pruning strategy that removes unused nodes from the GraphCache. The main idea is to remove those nodes which have not been used by the simulation application after a predefined timespan.

Consequently, the input for the pruning is a set of object-keys $\mathcal{OK} = \{\mathcal{O}_0, ..., \mathcal{O}_k\}$, a set of member-keys $\mathcal{MK} = \{\mathcal{M}_0, ..., \mathcal{M}_l\}$ and aggregate types $\mathcal{AT} = \{\mathcal{A}_0, ..., \mathcal{A}_n\}$ which have not been used as input for any aggregate query. The pruning is conducted in three phases. First, we remove all child nodes and sub-trees of the key-root $\mathcal{KR}$ which contain a $\mathcal{O} \in \mathcal{OK}$. Second, we remove those nodes which contain a $\mathcal{M} \in \mathcal{MK}$ from the remaining nodes. Finally, we remove those nodes which contain a $\mathcal{A} \in \mathcal{AT}$ (see Algorithm 6).

3.4 Relational Database Import & Export

Currently, most 3D simulation system rely on relational databases. In order to keep the implementation overhead small when moving existing systems to our GraphPool, we present an automatic import and export mechanism. The export of GraphPool data to relational databases can be easily realized in two steps: First, the empty tables for the property graph model objects are generated: Nodes, keys, relationships, labels and history. Second, all GraphNodes of the GraphPool are traversed and for every GraphNode, the label, the relationship and the member variables are stored in the aforementioned tables. Additionally, the history of every GraphNode is traversed in order to store the data in the corresponding table. Switching both steps realizes the import of database tables into our GraphPool.

Algorithm 7 illustrates the GraphPool export implementation and Figure 8 shows the resulting rigid table schema:

Algorithm 7 export($\mathcal{G}$ GraphNodes, $\mathcal{N}$ Nodes table, $\mathcal{R}$ Relationships table, $\mathcal{L}$ Labels table, $\mathcal{K}$ Keys table, $\mathcal{H}$ History table)

for $\mathcal{G}_i \in \mathcal{G}$ **do**
  Store $\mathcal{G}_i.Label$ in $\mathcal{L}$
  for $\mathcal{R}_i \in \mathcal{G}_i.Relationships$ **do**
    Store ($\mathcal{R}_i.Type$,$\mathcal{R}_i.From$,$\mathcal{R}_i.To$) in $\mathcal{R}$
  end for
  for $\mathcal{H}_i \in \mathcal{G}_i.History$ **do**
    Store ($\mathcal{G}_i.Id$, $\mathcal{H}_i.Id$) in $\mathcal{H}$
    Store ($\mathcal{H}_i.Id$,$\mathcal{H}_i.Object$,$\mathcal{H}_i.Member$) in $\mathcal{N}$
  end for
  for $\mathcal{D}_i \in \mathcal{G}_i.Member$ **do**
    Store $\mathcal{D}_i$ in $\mathcal{K}$
  end for
  Store $\mathcal{G}_i.Object$ in $\mathcal{K}$
  Store ($\mathcal{G}_i.Id$,$\mathcal{G}_i.Object$,$\mathcal{G}_i.Member$) in $\mathcal{N}$
end for

KEYS	
KEY	DATA

GRAPHNODE		
ID	MEMBER-KEY	OBJECT-KEYS

LABELS	
ID	Label

RELATIONSHIPS		
TYPE	FROM	TO

HISTORY	
ID	ID

Figure 8: **The resulting rigid table schema for importing and exporting the GraphPool.**

4. CASE STUDY

Our approach enables the implementation of very different categories of 3D simulation applications. Exemplarily, we present the application to a high fidelity dynamics and spacecraft EDL (entry, descent and landing) end-to-end spaceflight mission simulator [19, 2].

More precisely, we adopted our system to a simplified version of ESAs ARCHEO-E2E system [4] that defines a reference architecture for spacecraft engineering feasibility studies. Instruments of the spacecraft, as well as the environment, including the spacecraft's orbit and attitude, are simulated and defined as simulation components within the software architecture. The sensor input (e.g. camera and range finder measurements) for the instruments is synthesized from the simulated environment. In our implementation, all this synthesized data and the current world state (e.g. spacecraft pose, positions of celestial bodies, sensor configurations, scene nodes) are represented as GraphNodes in our central GraphPool. The instruments and the physically-based simulation read and write the entries periodically. Consequently, this scenario has a large amount of concurrent read- and write operations on our GraphPool. Figure 9 shows the visual output of the simulation in which a spacecraft conducts scientific experiments while orbiting an asteroid.

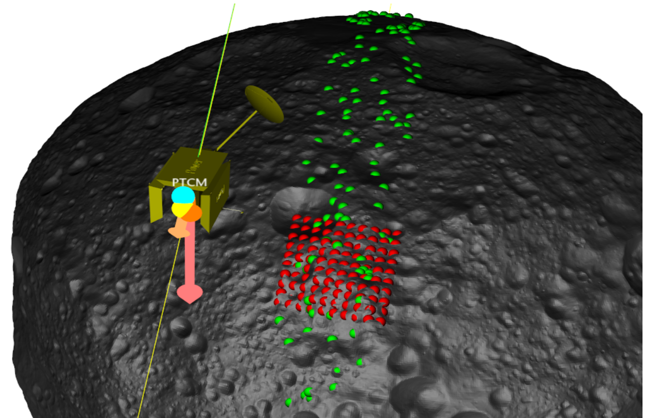

Figure 9: **Use case study: A spacecraft is orbiting an asteroid. Rangefinder (red) and landmark (green) measurements are generated for spacecraft self-localization [14, 8] purposes.**

5. EVALUATION

We implemented our GraphPool in C++. We performed experiments on a machine with an Intel Core i7 quad core processor with enabled Hyperthreading, operated by Windows 7 64 bit and 8GB of memory.

We applied different experiments to measure the performance as well as the quality of our approach. For the quality measurement, we used the use case scenario described above. However, as the scenario is domain-dependent, it can be hardly used to evaluate the performance of our approach. Hence, we additionally implemented a synthetic benchmark for performance measurements.

The GraphPool contained 1000 GraphNodes for the synthetic benchmark. We performed 10,000 read-, write- and aggregate queries for each test. Each test was additionally repeated 100 times and we averaged the resulting timings. The access to the GraphPool was modelled with an equal read/write distribution of concurrent system components. The transactions to the GraphPool and its competitors varied in size from 1 Byte to 1 Megabyte. We compared the performance of our new approach with three competitors. The first competitor was a lock-based implementation of our GraphPool. The other two competitors were a relational SQLite database and a MySQL database. We compared the performance with the traditional on-disk option as well as in-memory resident versions of the databases. Furthermore, we validated if the results from our GraphPool implementations yield the same results as the database competitors.

Our results show, that, in case of a single component, our wait-free GraphPool outperforms all in-memory and relational databases for every query type in several orders of magnitude. However, the traditional lock-based implementation of the GraphPool is slightly outperformed by in-memory resident relational databases. In this case, the lock acquisition introduces a computational overhead with respect to the wait-free implementation. In addition, standard relational databases can not compete with the in-memory databases and GraphPool implementations (see Figure 10).

This performance gain of our GraphPool increases with an increasing number of components accessing the data management systems. In this case, the wait-free access shows its strengths when several components simultaneously access the GraphPool. Like in the single access case, the in-memory relational database slightly outperforms the lock-based GraphPool implementation. The in-memory and relational databases are again outperformed by the wait-free GraphPool by several orders of magnitude (see Figure 11).

Overall, our evaluation underlines the aforementioned technical limitations of 3D simulation systems which rely on relational databases: The relational databases scale not very well with many concurrent components accessing them. Furthermore, our evaluation shows that the hash map backbone of our GraphPool can effectively solve the problems of the rigid table format of relational databases because no transformations of object-oriented data into tables is necessary. Additionally, the wait-free access of the GraphPool improves the overall performance even for massive concurrent read and write operations. In summary, our approach improves the overall system performance of 3D simulations by several orders of magnitude in all access query cases.

6. CONCLUSION

We presented a novel in-memory resident, schema-less, wait-free GraphPool for high performance 3D simulation applications.

Our GraphPool supports all common kinds of data operations like storing and managing static and dynamic parts while maintaining the support of sophisticated query types of traditional relational databases. Additionally, our GraphPool represents a central communication hub that drives the simulation itself. Every simulation run is implicitly logged by the GraphPools versioning techniques. This allows subsequent replay, analysis and archiving of the complete simulation. During runtime, simulation components can replicate any parts of the central world state into their local world state while local changes are tracked and written back into the central world state. Our approach has already proven its feasibility, real-time performance and flexibility in a high fidelity space robotics application.

The results of our synthetic benchmarks show that our approach is able to outperform state-of-the-art relational database driven simulation applications by several orders of magnitude.

We believe that our approach can be applied to a wide variety of 3D simulations (such as industrial automation, process or manufacturing simulations) where it will improve the performance.

In the future, we would like to extend the GraphCache technique. It would be desirable to remove the initial setup phase of the GraphCache while maintaining its wait-free behavior. Probably, this could be done by using unique prime numbers as identifiers for the components using the GraphCache. These identifiers could be used for unique cache access determination, which would result in "private" GraphCaches for every component accessing the GraphPool.

7. ACKNOWLEDGMENTS

This research is based upon the project KaNaRiA, supported by German Aerospace Center (DLR) with funds of the German Federal Ministry of Economics and Technology (BMWi) under grant 50NA1318.

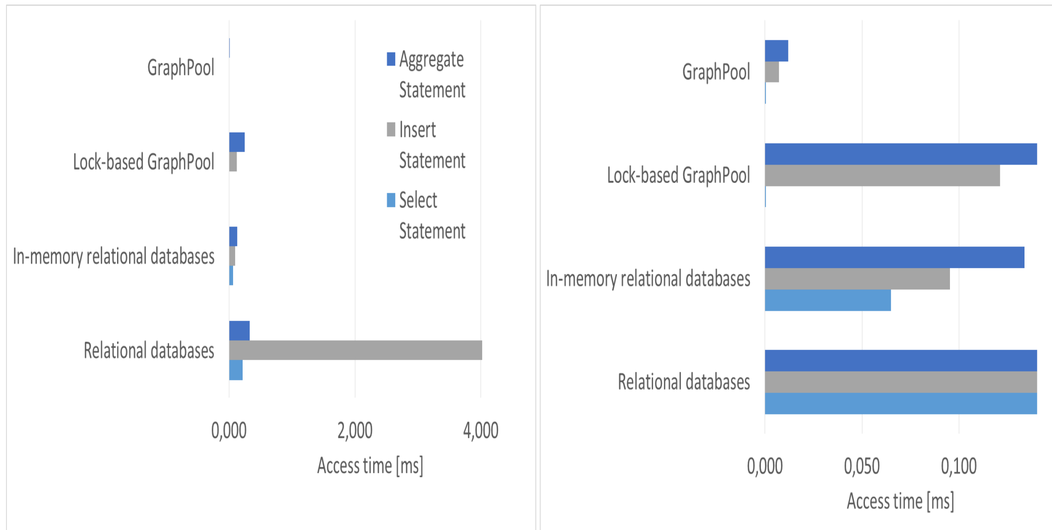

Figure 10: **Performance comparison of core & aggregate queries: Overall (left), our GraphPool outperforms all competitors for all query types for single-component access. In detail (right), in-memory resident relational databases outperform the traditional lock-based implementation of the GraphPool.**

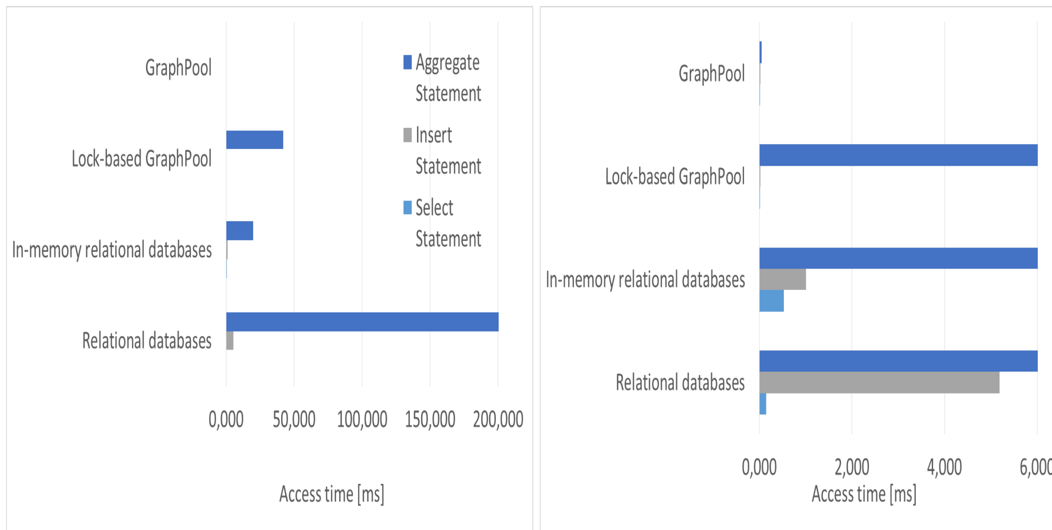

Figure 11: **Performance comparison of core & aggregate queries: Overall (left), our GraphPool outperforms all competitors for all query types for multi-component access. In detail (right), in-memory resident relational databases outperform the traditional lock-based implementation of the GraphPool.**

8. REFERENCES

[1] A. Vakaloudis, B. Theodoulidis. Spatiotemporal Database Connection to VRML. *Proceedings 9th UL Electronic Imaging & Visual Arts Conference*, 1998.

[2] A. Probst, G. Peytavi, D. Nakath, A. Schattel, C. Rachuy, P. Lange, et al. Kanaria: Identifying the Challenges for Cognitive Autonomous Navigation and Guidance for Missions to Small Planetary Bodies. *International Astronautical Congress (IAC)*, 2015.

[3] B. Damer, S. Gold, D. Rasmussen, et al. Data-Driven Virtual Environment Assembly and Operation. *NASA Ames Research Center: VIB Workshop Report*, 2004.

[4] C. de Negueruela, M. Scagliola, D. Giudici, J. Moreno, J. Vicent, A. Camps, H. Park, P. Flamant, R. Franco. ARCHEO-E2E: A Reference Architecture for Earth Observation end-to-end Mission Performance Simulators. *Simulation and EGSE facilities for Space Programmes*, ESA ESTEC, 2012.

[5] C. Watanabe, Y. Masunaga. VWDB2: A Network Virtual Reality System with a Database Function for a Shared Work Environment. pages 190 – 196, 2002.

[6] Couchbase. Why NoSQL? *Whitepaper*, 2014.

[7] D. Schmalstieg, G. Schall, d. Wagner, I. Barakonyi, G. Reitmayr, J. Newman, F. Ledermann. Managing Complex Augmented Reality Models. *IEEE Computer Graphics and Applications*, 27:48–57, 2007.

[8] D. Nakath, C. Rachuy, J. Clemens, K. Schill. Optimal rotation sequences for active perception. In *Proc. SPIE Multisensor, Multisource Information Fusion: Architectures, Algorithms, and Applications 2016*. SPIE Press, 2016.

[9] E. von Schweber. SQL3D - Escape from VRML Island. *SIGGRAPH VRML Consortium*, 1998.

[10] G. Burd. NoSQL. *Whitepaper*, pages 5–12, 2011.

[11] G. van Mare, R. Germs, F. Jansen. Integrating 3D-GIS and Virtual Reality. Design and Implementation of the Karma VI System. *10th Colloquium of the Spatial Information Research Center*, 1998.

[12] J. Haist, V. Coors. The W3DS-Interface of Cityserver3D. *European Spatial Data Research: Next Generation 3D City Models*, pages 63–67, 2005.

[13] J. Rossmann, M. Schluse, R. Waspe, M. Hoppen. Real-Time Capable Data Management Architecture for Database-Driven 3D Simulation Systems. *Database and Expert Systems Applications*, pages 262 – 269, 2011.

[14] J. Clemens, T. Reineking, T. Kluth. An evidential approach to SLAM, path planning, and active exploration. *International Journal of Approximate Reasoning*, 2016.

[15] K. Kaku, H. Minami, T. Tomii, H. Nasu. Proposal of Virtual Space Browser Enables Retrieval and Action with Semantics which is Shared by Multi Users. *21st International Conference on Data Engineering Workshops (ICDEW)*, pages 1259–1259, 2005.

[16] K. Walczak. Dynamic Database Modeling of 3D Multimedia Content. *Interactive 3D Multimedia Content*, pages 55–102, 2012.

[17] M. Hoppen, M. Schluse, J. Rossmann, B. Weitzig. Database-Driven Distributed 3D Simulation. *Proceedings of the Winter Simulation Conference*, 2012.

[18] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, pages 206–209, 1991.

[19] P. Lange, A. Probst, A. Srinivas et al. Virtual reality for simulating autonomous deep-space navigation and mining. *24th International Conference on Artificial Reality and Telexistence (ICAT-EGVE 2014)*, 2014.

[20] P. Lange, R. Weller, G. Zachmann. A Framework for Wait-Free Data Exchange in Massively Threaded VR Systems. 2014.

[21] P. Lange, R. Weller, G. Zachmann. Multi Agent System Optimization in Virtual Vehicle Testbeds. *EAI SIMUtools*, 2015.

[22] P. Lange, R. Weller, G. Zachmann. Scalable Concurrency Control for Massively Collaborative Virtual Environments. *ACM Multimedia Systems, Massively Multiuser Virtual Environments (MMVE)*, 2015.

[23] P. Lange, R. Weller, G. Zachmann. Wait-Free Hash Maps in the Entity-Component-System Pattern for Realtime Interactive Systems. *IEEE VR 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, 2016.

[24] S. Julier, y. Baillot, M. Lanzagorta, D. Brown, L. Rosenblum. BARS: Battlefield Augmented Reality System. *NATO Symposium on Information Processing Techniques for Military Systems*, pages 9 – 11, 2000.

[25] T. Manoharan, H. Taylor, P. Gardiner. A Collaborative Analysis Tool for Visualisation And Interaction With Spatial Data. *Proceedings of the Seveneth International Conference on 3D Web Technology*, pages 75 – 83, 2002.