

Chapter 2

State of the Art

VR simulation systems are complex software systems that have to manage a large variety of tasks in order to fulfill the desired simulation and rendering goals. In this chapter, state of the art related to acceleration algorithms for graphics applications as well as concepts and principles of scene descriptions and management approaches are presented, which are currently used to realize such systems.

2.1 Acceleration Algorithms for Graphics Applications

One of the major concerns of real-time rendering has always been to optimize the rendering performance. By integrating more and more state of the art rendering techniques into virtual scenes with ever increasing complexity, real-time rendering is still a challenging task to accomplish, even with today's potent graphics hardware. In this section, common strategies for the efficient organization of rendering-related data are introduced.

2.1.1 The Rendering Pipeline

The rendering pipeline is considered to be the core component of all real-time graphics applications. It describes the way of a vectorial, mathematical description of a scene to a rasterized, two-dimensional image, which can be interpreted by an output device like a computer monitor. The rendering pipeline will briefly be introduced in this section, further readings and in-depth descriptions can be found in [7], [372] and [300]. Figure 2.1 illustrates the basic structure of the rendering pipeline. It can roughly be categorized into three conceptual sections, which are called *stages*. Each one of these stages may be a pipeline in itself and can further be divided into sub-stages. The application stage is driven by the graphics application executed on the CPU; thus, the developer has full control over it. At the end of this stage, the data is fed to the geometry and rasterizer stage, which are performed on the GPU with no

direct control of the developer. The geometry stage is responsible for the processing of most of the polygon and vertex-wise operations. The rasterizer stage converts the corresponding vector information into a raster image composed of pixels in order to display them on an output device.

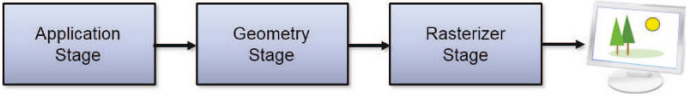


Fig. 2.1 The basic stages of the rendering pipeline.

Over the last two decades, graphics hardware evolved from configurable implementations of the fixed-function pipeline [301] to fully programmable stages using a unified shading architecture, which allow developers to implement their own algorithms and take full control over most stages. In this thesis, the latest technology available at the time, OpenGL 4.4 including shader model 5.0, is used. A detailed illustration of the corresponding rendering pipeline is depicted in Figure 2.2. To invoke the graphics API to draw a group of primitives, drawing functions (*draw calls*) are used. After a draw call, the rendering pipeline is executed using the currently activated shader programs.

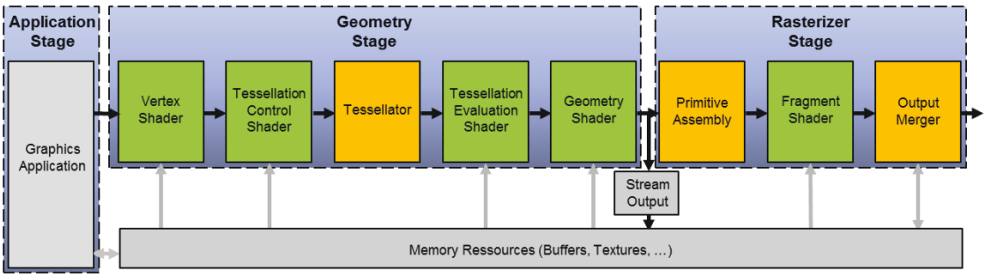


Fig. 2.2 Illustration of the OpenGL 4.4 rendering pipeline. Fully programmable stages are marked in green (e.g. the vertex shader), fixed-function stages are marked in orange (e.g. the tessellator).

Vertex Shader

As illustrated in Figure 2.2, the vertex shader is the first stage of the GPU pipeline. It receives data streams submitted by the application stage for rendering through a draw call. The vertex shader is executed exclusively for single incoming vertices and has no information about adjacent vertices, even if they are part of the currently processed primitive. The vertex shader provides the ability to modify, create or ignore data provided for each incoming vertex, but it can neither create nor delete the vertices itself. The minimal output of a vertex shader must be the location of a vertex.

Tessellation Shaders

The tessellation stage is an optional part of the GPU pipeline. It is diverted into three sub-stages: the *tessellation control* stage, the *tessellator* stage and the *tessellation evaluator* stage; in contrast to the vertex shader, the tessellator stage is able to generate new vertices by interpolating between existing ones. The control stage and the evaluation stage can be programmed directly, while the tessellator stage is a fixed-function stage. The control shader is used to define the way how new vertices are generated by the tessellator stage. The shaders are executed on a per-vertex basis and have further information about the other vertices that are part of the current primitive. The control shader takes two input parameters to control the tessellator stage, named *inner* and *outer tessellation level*. The inner tessellation level governs how often the inside of a primitive is divided while the outer tessellation level governs how many times each edge of the primitive is subdivided. After the tessellator stage has been executed and new vertices were generated considering the settings of the control shader, the output is passed to the tessellation evaluation shader, which computes the new positions and attributes of each generated vertex. The tessellator stage interpolates their positions by using barycentric coordinates only, while the control shader is able to alter these positions in a more flexible way. For example, interpolations based on Bezier patch subdivision can be applied to create smoothly curved patches [225].

Geometry Shader

The geometry stage is also an optional stage, which is executed after the tessellation stage. The geometry shader is executed once per primitive and it is capable of modifying, creating or removing primitives or even discarding them all. A fixed input and output primitive type need to be defined before execution, which do not have to match each other. For example, points can be used as input and triangle strips can be generated as output. The output of the geometry stage can also be streamed into a buffer instead of passing the generated data to the next stage. Stream output was introduced in Shader Model 4.0 and enables iterative data processing since the generated data can be sent back through the pipeline and accessed by the graphics application for further processing. Further pipeline processing can be turned off after stream output, transforming the pipeline to a purely non-graphical stream processor for massively parallel general purpose calculations [233][168][290].

Pixel Shader

The output primitives of the previous stages are clipped and set up for rasterization by fixed-function stages before the pixel shader is executed. The vertex data is interpolated across each triangle's area. In OpenGL, the pixel shader is known as *fragment shader* since color values are not computed for pixels directly; instead, color and depth values are computed for fragments of pixels, depending on how the triangle covers the pixel's cell. The fragment shader's output is limited to the currently processed fragment; however, *multiple render targets (MRT)*

can be attached to the frame buffer to output computation results to assigned buffers. MRT is an important feature for many effects presented later in this thesis, since these render targets can also be used as input textures for subsequent render passes.

One of the key aspects regarding the overall performance of the rendering pipeline is the transition between the CPU-driven application stage and the GPU-driven geometry stage. The data organization and the methods used to provide the data to the geometry stage directly influence the rendering performance; hence, an optimal application stage data management is a vital task for the realization of real-time capable graphics applications.

2.1.2 Rendering System APIs

The application stage provides a stream of primitive drawing commands, state settings and matrix manipulations to the graphics API in order to render the geometry. When developing computer graphics applications, different concepts and APIs are available. They can be categorized into *immediate-mode* and *retained-mode* rendering systems, which are based on different scene processing principles.

Immediate-Mode APIs

In the area of real-time rendering, the two relevant immediate-mode APIs for low-level, machine-oriented graphics programming are OpenGL [300] and Direct3D [372]. Immediate-mode APIs do not hold internal information about geometry and immediately process each supplied primitive with the currently set properties. Immediate mode is the direct access of the rendering system in which only explicitly defined objects are rendered during each render cycle. This kind of API is procedural, it does not store a scene model between frames; instead, the application keeps track of the scene as illustrated in Figure 2.3. To define the way the supplied geometry is being processed, a large number of functions is provided. The rendering performance heavily depends on the application stage (and the user or developer), which have to provide data and commands in a render-friendly way.

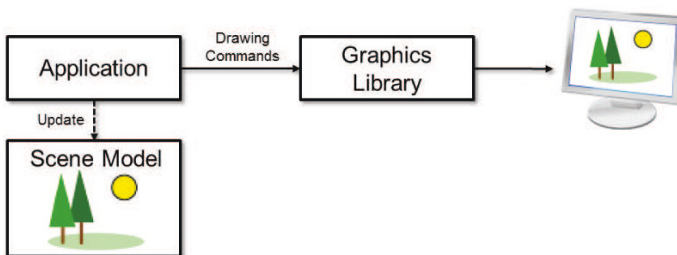


Fig. 2.3 A diagram of an immediate-mode graphics API.

Retained-Mode APIs

Retained-mode APIs are declarative. A graphics library stores and manages a digital model of the scene and the application can construct a scene from provided objects. When a frame should be drawn, the graphics library transforms the scene into a set of drawing commands, which are sent to the underlying, low-level graphics API as illustrated in Figure 2.4. In order to change what is rendered, the application can issue a command to update the scene, e.g. to add or remove an object.

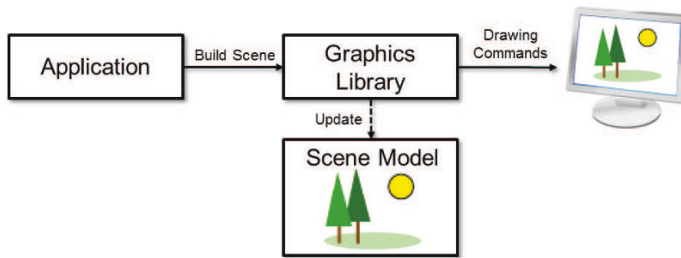


Fig. 2.4 A diagram of a retained-mode graphics API.

The ever increasing complexity of 3D graphics applications demands for methods and approaches of software engineering for modern graphics APIs. An additional software layer is commonly applied, which encapsulates sequences of low-level commands into high-level objects to provide frequently used functionalities for convenient usage. The level of specialization and abstraction of retained-mode APIs varies depending on the addressed application and domain. As illustrated in Figure 2.5, these APIs can further be categorized into *rendering libraries* and *rendering frameworks*.

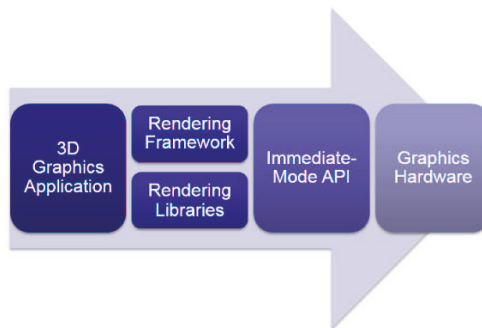


Fig. 2.5 Illustration of the layered structure of a graphics application.

Rendering libraries encapsulate highly specialized rendering algorithms into simplified interfaces. They usually cover a small domain like tree rendering, volume rendering, or flow visualizations and provide functionalities for convenient usage by the developers. Rendering

frameworks provide the infrastructure to realize computer graphics applications and they can integrate one or more rendering libraries. Rendering frameworks can further be categorized into *specialized* and *general* rendering frameworks. Specialized ones make a range of assumptions about the application domain and provide optimized, domain-specific features. General rendering frameworks have no pre-defined domain and provide a broader range of features, but they may not achieve optimal performance for every field of application. A selection of current frameworks will be introduced in Section 2.3.

Last but not least, retained-mode APIs have clear advantages over intermediate-mode APIs when considering the general applicability to and the rendering performance of complex scenes. Due to the fact that they take control over the scene management and organization, a broad range of optimization techniques can be applied in order to prepare the rendering data to be provided to the low-level API in a render-friendly way. Traditionally, two techniques are used in almost any graphics application: *culling* and *batching*.

2.1.3 Culling Techniques

Graphics hardware directly implement culling techniques that remove unseen geometry during rendering pipeline execution using clipping and the z-Buffer. Primitives, which are facing away from the camera are also removed by using *backface culling* [7]. To further reduce GPU workload, CPU-based culling techniques try to minimize polygonal level culling and aim to prevent unseen geometry from being sent down the rendering pipeline. These techniques are traditional tasks performed by the application stage. Due to ever increasing complexity of virtual scenes, culling techniques are essential for every real-time graphics application. Almost any culling technique demand for a hierarchical data structure. A hierarchical data structure is a spatial data structure where the parent entity encloses all child entities. This structure allows to efficiently determine overlapping geometry and is also beneficial to speed up tasks like collision detection or intersection tests, which are very fast to query, typically reaching a performance improvement from $O(n)$ to $O(\log n)$ [7]. The most common spatial data structures in this context are *bounding volume hierarchies (BVHs)* [22], *BSP trees* [25][5] as well as *quad-* and *octrees* [340].

One of the mostly applied visibility culling technique is *view frustum culling (VFC)*. The idea of VFC is to determine and remove objects that cannot be seen by the current camera, e.g. when they lie outside the camera's view frustum. The original idea for VFC was introduced by Clark in the early computer graphics era [53], who used a hierarchical approach to determine invisible surfaces in order to discard them in the application stage prior to rendering. While the basic principle of VFC is clearly defined, a broad range of different implementations have been developed in the last decades, which focus on the efficient implementation for different application areas, e.g. by exploiting the spatial coherence of the scene and enclosing multiple objects in a hierarchically clustered BVH [10][54].

Apart from VFC, another popular culling technique is *occlusion culling*, which tries to determine objects that cannot be seen from the camera due to groups of other objects that lie between them and the camera [318][187]. Occlusion culling depends on how the ob-

jects affect (occlude) each other. Even if current graphics hardware directly support hardware accelerated occlusion queries [29], these approaches are more complex to compute and to realize in an efficient way. A broad range of specialized approaches like *occlusion horizons* [85], *shaft occlusion culling* [291] or *portal culling* [6][331] have been developed, which are optimized for specific application areas. Multiple culling techniques are usually combined to achieve better results. Sekulic suggests to apply VFC with bounding spheres first, followed by a test against bounding boxes if the sphere intersects the view frustum [297]. Finally, the remaining geometry is tested using occlusion culling techniques.

2.1.4 State Sorting and Batching

At first sight, the rendering performance of a graphics application obviously depends on the calculation power of the GPU. Indeed, the GPU faces a large number of processing steps as presented in Section 2.1.1. The performance can actually be limited by the GPU's processing power as well as its fillrate or memory bandwidth; however, a lot of work has been put on GPU-side optimizations of rendering tasks. This includes texture compression, mipmap generation as well as shader code optimizations, which can easily be applied in order to reduce GPU-related performance limitations [47][300]; finally, culling techniques are a popular approach to relieve the GPU by sending as few data as possible down the rendering pipeline.

Apart from the GPU itself, the CPU is also involved for rendering purposes. It has to set up the current *render state*, which defines rendering-related properties like the "appearance" of the geometry to be drawn. Among other things, this includes the shader setup and parameterization as well as texture and lighting setup. After setting the render state, draw calls are sent to the graphics driver that prepares the commands to be sent off to the GPU. The number of these draw calls has a significant impact on the rendering performance. For example, 1000 triangles will be rendered much faster when passed to the GPU as a single draw call instead of 1000 single ones. The cost for the GPU will be very similar in both scenarios; however, the CPU workload will be significantly higher in the second case because of graphics driver overhead with each API call [300]. Referring to the NVIDIA GPU Programming guide [227], reducing the API draw calls is one of the best ways to improve the rendering performance. The importance of CPU-side optimizations and batching has been shown by Wloka [363]. Due to the fact that the calculation power of modern GPUs increases significantly faster than on CPUs [226], modern graphics applications are increasingly becoming CPU-limited, making efficient batching more important than ever before.

Since immediate-mode rendering APIs like OpenGL are huge state machines, similar geometry that use the same render state need to be grouped for efficient rendering. The goal of batching is to create these groups and store them in adjacent memory blocks on the GPU to be able to draw the whole block in one single draw call. The task of the higher-level retained-mode API is to organize the geometry that remains after culling in *vertex buffer objects (VBO)*. They represent a memory block, which can hold the vertices and associated attributes like normals, colors or texture coordinates of multiple objects [332]. VBOs are referenced by *index buffer objects (IBO)*, which hold sets of indices that form the single poly-

gons and build up the shape of the whole object. Batching depends on four main categories, which properties have to be the same for all geometry that should be rendered simultaneously:

- **Geometry properties:** Batched geometry has to consist of the same kind of primitives (e.g. lines, triangles, quads, ...) as well as the same kind of vertex attributes (e.g. vertices+normals or vertices+normals+texture coordinates, ...)
- **Shader / shader properties:** Batched geometry has to be processable with the same shader and similar input properties (uniforms). This mostly implies material properties, which describe the appearance of the geometry.
- **Textures / buffers:** Batched geometry can only refer to the textures and buffers that are currently bound.
- **Transformations:** Batched geometry has to be defined in the same coordinate system or refer to corresponding transformation matrices accessible by the shader.

Due to the named requirements and the fact that only a limited number of textures and buffers can be bound simultaneously, efficient batching is a difficult task to achieve. In order to realize batching in practical applications, different optimization techniques are applied by current graphics applications.

Polygonal Triangulation

Triangulation is a popular approach to address performance issues resulting from multiple draw calls due to the occurrence of different primitive types. In a preprocessing step, the primitives of the whole scene can be converted into a surface description consisting of triangles or triangle strips, only. In an optimal case, the triangulated scene can be processed by a single draw call. A detailed introduction into computational geometry including triangulation is given by Berg et al. [26]. An issue of triangulation is the increased data size of triangulated models because of the dividing of single polygons into multiple elements.

Material Buffer

Excessive shader and shader property state changes related to the rendering of different materials are one of the main reasons for limiting batching possibilities. Before the introduction of the programmable shader pipeline, material properties need to be set through the OpenGL-API prior to geometry rendering. The only way to optimize material state changes was to sort the geometry by materials or to reduce their amount. With the introduction of shaders and multi-purpose buffers on the GPU, material state sorting could drastically be simplified by using a *material buffer* [173]. The idea is to gather material information and upload them to a GPU buffer before rendering. An additional index attribute is added to the geometry, which refers to the corresponding buffer location. During rendering, the material information can directly be retrieved by the shader, completely eliminating the need for material sorting or state changes. Regarding the increasing GPU memory sizes of current graphics cards, the memory

overhead required for the material buffer can be neglected. Trading memory consumption for performance is usually a preferable choice in real-time applications and the material buffer approach is perfectly suitable for high-performance rendering of geometry that consists of a large number of different materials.

Texture Atlas and Bindless Textures

Only a limited number of textures can be bound to the available texture units at a time. The simplest way to reduce the need for texture switching is to use a *texture atlas* [194]. The basic idea is to arrange multiple textures of corresponding objects in a single texture to be able to group together more geometry. Texture atlases are usually created by artists or modelers during scene creation. As described by Hao [117], texture atlases can also be generated automatically, which is a suitable approach to minimize texture state changes for rendering frameworks. On modern graphics hardware, bindless textures provided by the OpenGL *GL_ARB_bindless_texture* extension are available to avoid texture state changes [332]. This extension allows to access texture objects in shaders without first binding each texture to one of a limited number of texture units. The application can query texture handles (64-bit unsigned integers) that can be used to directly access the corresponding texture in the shader. Considering the reduction of the draw call amount, texture state changes can completely be eliminated by providing the texture handles to the shaders through lookup buffers.

Transform Pre-Multiplication

The geometry stage of the rendering pipeline is responsible for the model and view transform of the vertices and transforms them from world space into camera space. Hierarchical data structures are composed of elements, which define their position relative to their parent using a relative transform matrix. These matrices need to be accumulated during traversal to properly transform the corresponding geometry to world coordinates. To minimize corresponding state changes, it is beneficial to separate static objects from dynamic ones. While the relative transform matrix of dynamic objects can change from frame to frame in order to move the element to another position or orientation, the matrix of static ones remains unchanged. All vertices of static objects can be pre-multiplied with the corresponding world matrix and directly written to the GPU buffer in world coordinates. In an optimal case, the pre-transformed vertices can be rendered with a single draw call using just the camera's modelview matrix. This approach can also be applied to sub-trees that remain static between each other by pre-transforming its elements in relation to the parent element; subsequently, the sub-tree can be rendered as a whole by multiplying the parent's transform matrix to the modelview matrix. Due to the resulting reduction of transform state changes, static geometry batching is common in many current rendering frameworks.

2.2 Scene Descriptions and Management for Multi-Domain VR Simulation Systems

As introduced in the previous section, rendering frameworks rely on specific data structures in order to apply optimization techniques to perform best; however, VR simulation systems are not solely built for rendering purposes. When realizing such complex systems, the database concept has to be based on a very flexible data structure to address user input, simulation data and rendering tasks simultaneously. Considering domains like out-of-core rendering [342], procedural rendering or GIS¹ applications, the content of these data structures can dynamically change. Parts of the database need to be created, altered, loaded or discarded in a dynamic fashion; hence, a matching scene description needs to be found, which is suitable to serve as a scene database for a broad range of application domains, regardless if rendering- or simulation-related. In this section, current concepts, approaches and developments are introduced and discussed, which are used to address the named challenges.

2.2.1 Scene Graphs

A scene graph is the most popular retained-mode API in applications related to real-time rendering. It is a object-oriented, hierarchical data structure used to store, organize and manipulate 3D scenes. A typical scene in the area of computer graphics consists of the geometry of all objects, as well as various states that define the way the geometry is being rendered. Scene graphs form a useful abstraction for both scene data and scene behavior. Before the introduction of scene graphs, data and behavior were defined procedurally and changes in one of these areas required code changes. A scene graph encapsulates the scene in a manner that allows for its modification independent from the program code.

Scene graphs have been widely used in computer graphics since the release of IRIS Inventor by Silicon Graphics in 1993. In the paper that describes IRIS Inventor, Strauss introduced how a 3D scene can be described as a tree of nodes [316]. The nodes are divided into shapes, properties and groups, which hold information about the geometry, the appearance and the hierarchies. The main reason to use scene graphs is to hide hardware-specific API implementations from the application and to be able to apply optimization techniques for increased rendering performance. A scene graph provides conceptual means to the developer to define and organize the static contents and dynamic events of a 3D scene. Similarly, it provides all the information needed by the graphics system to efficiently render the scene. Regarding the definition of Hitchner and Sowizral, a scene graph is an abstraction that provides an interface between an application developer (a human) and a rendering system (a computer) [139]. As illustrated in Figure 2.6, any complex graphics application creates and organizes some kind of scene graph.

Scene graphs are usually realized as a *directed acyclic graph (DAG)*, where directed means a one-way parent-child relationship and acyclic means that there can't be cycles (i.e. a child

¹ Geographic Information Systems

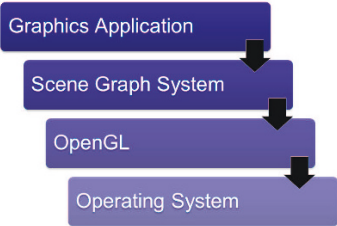


Fig. 2.6 Software layer architecture of modern graphics applications.

can't be one of its own ancestors). Each scene graph has a root node, which represents the scene as a whole including one or more children. The children represent the objects in the scene and can also have zero or more children. Each node can contain transformation data like matrices, which describe their position relative to the corresponding parent node; hence, changes of the parent's transformation matrix also affect the child nodes, which allows to easily translate whole branches of the scene graph. These aspects are illustrated in Figure 2.7, which shows an example of a simple scene graph of a car model.

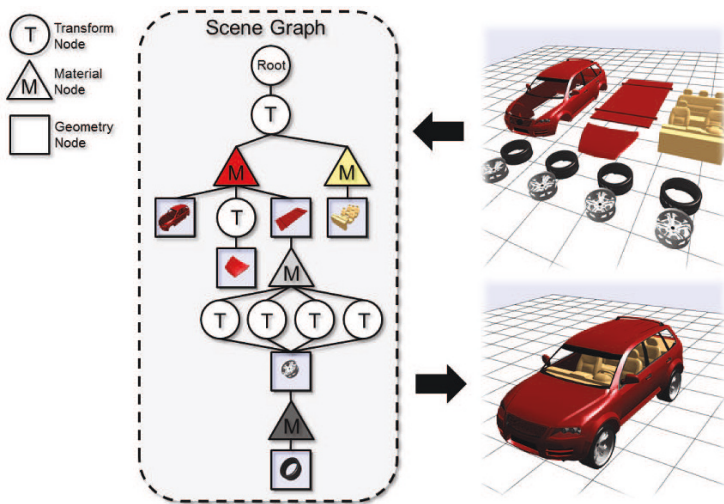


Fig. 2.7 Simple scene graph of a car model.

Another benefit of using scene graphs is the fact that objects can be arranged in logical and spatial order. The hierarchical structure allows for ease of manipulation and the application of culling techniques; moreover, state sorting can easily be realized to minimize render state changes and allow for efficient batching. Over the last decades, scene graphs showed clear benefits improving rendering performance and granting an optimal usage of the available hardware resources by keeping a retained model of the scene.

Because of their loose definition, scene graphs may be implemented in many different ways ranging from simple lists of objects up to very sophisticated spatial databases. While

early scene graphs were mainly applied for performance reasons, recent ones are built in a more universal manner. In an article about the role of scene graphs in the new millennium, Sowizral stated that they became a combined structure for communicating, rendering, execution and modeling [307]. He further states that modern scene graphs *"...allow programmers to shift their attention away from thinking about triangles and vertices to thinking about objects and their arrangement within a scene. They allow programmers to forget about controlling the rendering pipe-line and instead think about content and how best to present it."*[307]. Today, scene graphs are ubiquitous. Most high-level graphics toolkits provide a scene graph API to model 3D scenes and to program 3D applications. An ideal scene graph API should simplify programming of 3D applications, optimize rendering performance, and provide abstraction by encapsulating rendering algorithms. This classical scene graph concept has been adopted by many systems, like OpenSG, OpenSceneGraph, VRS or Java3D; in addition, 3D scene visualization languages like VRML [328] and X3D [41] have been developed. VRML is a format for describing and transmitting 3D objects and worlds composed of geometry and multimedia in a network environment. Today, VRML has mostly been superseded by X3D, which contains improved graphics and networking features and supports more sophisticated ones including multiple data encodings like XML; however, there is no defined way how scene graphs have to be implemented. So far, no "industrial standard" has been established and the very general definition has led to a large number of different scene graph implementations in current software systems, which follow different design principles.

2.2.1.1 Uber-Scene-Graph Approach

A popular scene graph approach was introduced by David H. Eberly [91]. He gives a very general definition of a scene graph and points out that for him a scene graph is just an abstract graph of objects in a tree-like structure with no concrete implementation. Eberly further embraces the scene graph data structure to an *uber-scene-graph*, incorporating as much data as possible. The key idea is to represent any type of scene using a single, all-encompassing scene graph that centralizes all tasks, all responsibilities and all rendering. The Wild Magic engine described by Eberly was well-performing and could efficiently deal with dynamic scenes and common rendering techniques of that time [90]; however, the uber-scene-graph approach was heavily discussed afterwards. In a following article, Forsyth stated that *"Putting the entire world into a big tree is not an obvious thing to do as far as the real world is concerned."*[101]. Bar-Zeew mentioned the vastly different facilities of graphics hardware that was available when many of the earlier scene graph systems were conceived [21]. On account of the limited resources and processing power of GPUs available at that time, a major concern of these scene graphs was to reduce the set of geometry (and the resulting GPU workload) by heavily using CPU-based culling techniques. Considering the limited programmable shading, modern rendering approaches like image-space effects were virtually unavailable, which also hardly fit into these concepts.

Bar-Zeew further stated that *"... The heart of the problem is an overloading of what was once a nice, straightforward performance improvement over immediate mode OpenGL."*[21]. The uber-scene-graph design mixes three unrelated structural ideas into one: The spatial and

hierarchical structure for culling as well as the stateful structure for batching reasons. These are three different trees, which are going to have different arrangements and structures. Most scene graph-related research proceed with the goal of merging these three contradicting trees into one, trying to provide more sophisticated control of this single traversal. Today, the uber-scene-graph approach is considered to be outdated because of the architectural limitations.

2.2.1.2 Multi-View Scene Graphs and Current Trends

Recently, Lander and Gregory pointed out that the choice of data structure to represent the scene is highly dependent on the type of application [182]. One has to find out what the application needs and then select the appropriate data structure. Bar-Zeew describes the concept of a *multi-view scene graph* by removing the fundamental constraint of a single scene graph organization for a visual database: *"It is entirely possible that we can have a single set of objects, call it an object soup, but have two, three, four, or more hierarchies linking these objects into independent and complimentary organizations."*[21]. He pointed out that sticking to a single structure for many years may not be the best choice in order to build a sustainable system. The current trend for the realization of multi-domain VR simulation systems is going to more flexible data structures. Regarding rendering architectures, this trend is moving away from pure scene graphs toward more general *scene databases*, which can be indexed in different ways. Scene databases can not only be used for rendering-related tasks, but also for a broad range of simulations, which may work on data structures not common for rendering systems. In order to realize this general concept for a multi-domain VR simulation system, the concepts of semantic graph databases are an ideal choice.

2.2.2 Semantic Graph Database Principles

All software systems that are based on a model of the environment maintain an internal model of real world objects and processes in order to support a defined application domain. This domain model is fixed and constrained by the data model. For example, the 3D visualization languages VRML and X3D, usually applied in VR applications focus heavily on 3D visualization and interoperability and are limited in terms of the semantic mark-up, which can be attributed to objects. When comparing scene graphs to graph databases, the concept of graph databases is much more general. The main idea is simply to store semantic networks in databases that do not strictly focus on 3D visualization or demand for spatial relations between the contained objects as scene-graphs do. A graph database is the most generic of data structures, capable of representing any kind of data in a highly accessible way. As illustrated in Figure 2.8, graph databases are usually schema-aware and allow a set of nodes (*instances*) with dynamic attributes (*properties*) to be arbitrary linked to other nodes through edges (*associations*).

As early as 1974, J. Abrial introduced concepts for storing and interacting with interconnected data containing elementary facts and rules for practical systems [3]. Abrial describes

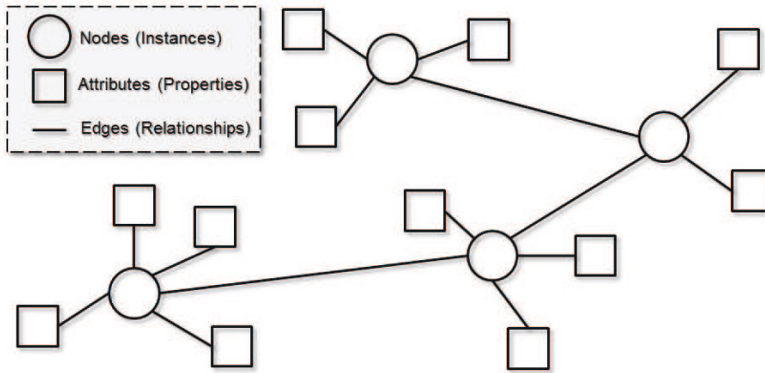


Fig. 2.8 Illustration of the general structure of a schema-aware graph database.

this graph database as a *semantic database*. The semantic database is a schema-aware graph database containing a collection of objects, which definition is "...given by the connections it has with other objects" in order to form "...the model of an evolving physical world." [3]. A few years later, Chen introduced the *entity-relationship model*, which is a database of interconnected *tuples* to unify relational and network databases [50]. At the same time, Rousopoulos and Mylopoulos described rational schemata to map semantic concepts into a database structure by defining a small set of nodes and the relationships between those nodes to describe various knowledge domains [289]. They further define the semantic net as a labeled directed graph of labeled nodes and edges. While labeled nodes should only be used for reference purposes, labeled edges can have a number of associated semantic properties and inferences. The usage of these graph database concepts and, in particular, the simple abstractions of nodes and relationships into connected structures enable the creation of sophisticated models, which are suitable for the application in a broad range of domains.

2.2.2.1 Semantic World Modeling

Semantic graph databases allow for *semantic world models*, which give complex information about the surrounding and enable the interpretation of the meaning of the single instances [2]. Since modern multi-domain VR simulation systems model a wide range of real world phenomena, it is necessary to adopt modeling and development techniques based on semantics. Semantics enable the precise mapping between complex real world phenomena and their corresponding computer models; hence, a semantic world model can be applied for a wide range of simulations and calculations. Semantic world models can describe the environment much more detailed and are not limited to spatial relations or geometry as pre-defines data structures like scene graphs are; however, limited work has been carried out within the software engineering community in relation to the development of semantic-based systems. The interest in graph databases diminished with the emergence of other database models, in particular geographical, spatial and semistructured ones as well as XML [9]. The need to manage infor-

mation with graph-like nature has recently reestablished the relevance of this area. Current business applications are mostly offered as *software as a service (SaaS)*, which have to face the challenge that different users hosted on the same database have different data needs. Following the recent definition of Robinson, Webber and Eifrem, a semantic graph database "... is an online database management systems with Create, Read, Update and Delete (CRUD) methods that expose a graph data model. Graph databases are generally built for use with transactional (OLTP) systems." [266].

This is precisely one of the challenges when combining various simulation and rendering techniques into a holistic multi-domain VR simulation system. While a broad range of graph-based databases have been developed until today, like AllegroGraph², JUNG³ or Neo4J⁴ to name a few, the integration into simulation systems, in particular VR simulation systems is still difficult. Numerous challenges of semantic-based systems still exist due to the very general definition of the semantic data and the difficult management of objects that exist (ontology), specific organizational knowledge of what exists (epistemology) and the required organizational action (pragmatics) [66].

2.2.2.2 Graph-Oriented Object Databases

Object orientation is a very popular approach to address the complexity problem in software systems [223]. *Object databases* directly add database functionality to the object oriented model and language itself, drastically simplifying implementations because of native object persistence. Object databases like db4o⁵, which allow for easy storing and retrieving of any application object are very popular today. Graph databases can also be realized following the object-orientated paradigm to make them practical for software systems. These databases are called *Graph-Oriented Object Databases (GOOD)* [112]. The main difference to object databases is its level of abstraction. While object databases only operate on objects, graph-oriented object databases operate on nodes and edges. One of the first graph-oriented object database was described by M. Gyssens et.al. in the 1990s, "... in which manipulation as well as conceptual representation of data is transparently graph-based." [112].

A recent database grounded on the GOOD principle is HyperGraphDB⁶, which storage mechanism is known as a *directed hypergraph* [149]. A directed hypergraph allows an edge to point to more than two nodes and even other edges; furthermore, nodes and edges can carry arbitrary values as payloads, known as *properties*. Grzybek and Gulliver suggested a semantic

² Allegrograph is a commercial graph database by Franz Inc., <http://www.franz.com/agraph/allegrograph>

³ JUNG (Java Universal Network/Graph Framework) is a Java-based open-source library designed to support the modeling, analysis, and visualization of graph data, <http://jung.sourceforge.net>

⁴ Neo4j is an open-source NOSQL graph database, <http://www.neo4j.org/>

⁵ db4o is an open source object database with native Java and .NET support, <http://www.db4o.com>

⁶ HyperGraphDB is a java-based open-source library for generalized graph databases, <http://www.hypergraphdb.org>

graph-based scene description by using these principles to develop semantically rich software representations of 3D scenes [110]. The presented concept combines graphs and hypergraphs to represent multiple nested relationship categories, which are used to create different types of object relationships within a scene. This description also features object classes and instances, which act in the same manner as their object-oriented counterparts. As illustrated in Figure 2.9, classes and instances can be categorized into user-defined *nodes*, *edges* and *sub-graphs*, allowing a high degree of freedom when developing semantic world models.

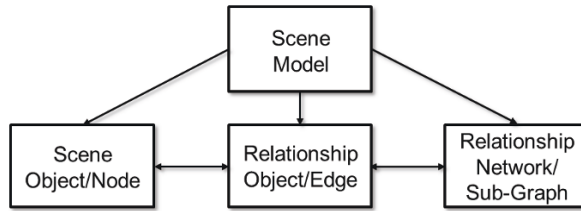


Fig. 2.9 Illustration of the entity relationship model for semantic scene descriptions like suggested by Grzybek and Gulliver [110].

2.2.3 Applying Semantic Scene Descriptions to Graphics Applications

Real-time rendering frameworks are highly advanced and optimized software toolkits that rely on optimized data structures in order to perform best and to efficiently integrate state of the art rendering techniques. Semantic databases usually do not comply with these requirements; hence, it is still a challenge to combine both areas including the flexibility and description power of semantic databases as well as the rendering performance achieved with scene graph structures.

2.2.3.1 Semantically Enhanced Scene Graphs

Regarding multi-domain VR simulation systems, the applied scene graphs have to deal with highly dynamic data due to user input and the progress of the simulation. It is necessary to modify the graphical output according to these changes, however, there is currently no established design pattern for dealing with scene graphs that constantly change [336]. The state of the simulation itself need to be stored in order to detect and react to changes and to update the scene graph accordingly. As illustrated in Figure 2.10, two storage approaches have been established in current applications: *external* and *internal state storage*.

When using external state storage, all states are stored in application-specific data structures. In order to update the scene graph according to state changes, the application maintains references to specific parts of the scene graph, which is only an internal representation of the data to be rendered. In an internal state storage, all states are implemented into the scene

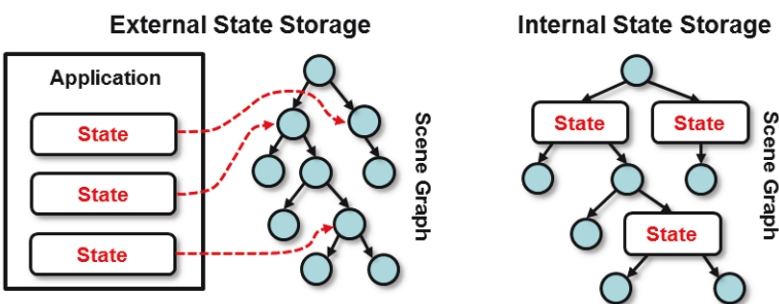


Fig. 2.10 Illustration of external and internal state storage.

graph itself by defining and attaching specific *state nodes*, which update the system state during traversal. This concept of an internal state storage is also known under the term *semantically enhanced scene graphs* [110]. Even if this approach is the most straightforward one and shows clear benefits when applying it to large scenes, the complexity of handling both semantic and rendering-related operations in a combined fashion is still a challenge.

2.2.3.2 Separating Semantic Data from Rendering Data

In order to face the complexity issue with semantically enhanced scene graphs, Mendez et al. suggested to separate semantics from rendering in the scene graph by using semantic tags as attributes [206]. More recently, Tobler picked up this idea and suggested a more complete solution by fully separating semantics from rendering in specific representations [336]. He introduced a *split scene graph architecture* that consists of a separate semantic and rendering scene graph as illustrated in Figure 2.11.

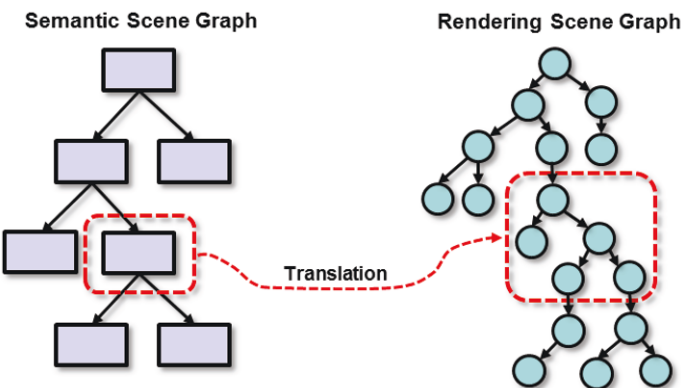


Fig. 2.11 In the split scene graph architecture the semantic scene graph is translated into a rendering scene graph during traversal.

As the name implies, the semantic scene graph only consists of semantic nodes, which describe the scene in a "natural" way. It represents the scene like it was modeled by the user. The graphics application then acts like a compiler by working on this scene graph and translates it into a rendering scene graph during traversal. Optimization techniques as introduced in Section 2.1 can be applied to generate an optimal sequence of rendering operations before they are passed to the graphics API. Each semantic node can also be translated into multiple connected rendering nodes to fulfill the desired rendering task; for example, a user can add a simple car node to the semantic scene graph, which then is translated into multiple geometry, material, texture and shader nodes for the rendering scene graph. Tobler further states that this approach best matches static scenes. In order to realize dynamic scenes, he applies well-known compiler techniques common in languages like C#, Java or even shader languages, where the source code is translated into an intermediate language (IL) for hardware compatibility reasons. Functions and methods are compiled just-in-time whenever they are needed [73]. Similarly, the semantic scene graph is translated into a forest of small rendering scene graph pieces on-the-fly when it is traversed.

To integrate dynamic handling, *rule objects* can be attached to the semantic scene graph, which integrating specific transformation rules into the translation steps. These rule objects can create a rendering scene graph piece that represents the translated semantic node. By integrating a method into the rule objects that is called whenever it is traversed, it can dynamically react to changes in the global state and modify its rendering scene graph piece accordingly. Considering software design patterns, this approach can be seen as implementation of the well-known *Model-View-Controller (MVC)* design pattern [103], as illustrated in Figure 2.12.

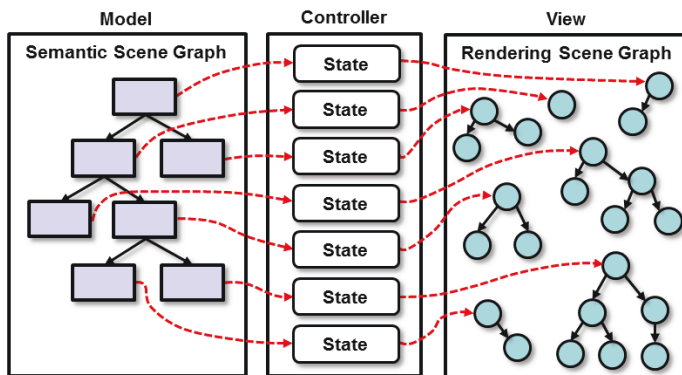


Fig. 2.12 Illustration of the split scene graph architecture implemented by following the Model-View-Controller (MVC) design pattern. During semantic scene graph traversal, the rendering scene graph is built on-the-fly.

By separating the model view from the system view and by using a controller that coordinates the translation, the MVC design pattern allows for the usage of a single model under different environments. It is a suitable concept to separate the user interface from the computation parts. The semantic scene graph represents the model as created by the user, the

rendering scene graph represents the view necessary for the system environment and the rule objects represent the controller, which dynamically translate the model into the system view. This approach is suitable for many kinds of rendering applications that work on semantic datasets and it is not limited to specific domains or environments.

2.3 Survey of Current Graphics and Simulation Frameworks

In this section, current rendering frameworks and systems are introduced that are related to the topics addressed in this thesis. This includes the areas of general rendering, specialized rendering, mobile robotic applications and virtual training. Even if this is just a fraction of current VR and simulation domains, the overview should give an impression of the broad range of available frameworks and applications that rely on VR simulation technology today. A detailed comparison of current mobile robot simulators is given by Staranowicz and Mariottini [313] and by Michal and Etzkorn [209]. In respect to the amount of existing frameworks, not all of them can be named here; hence, a selection of popular frameworks and libraries is given without any claim to completeness:

Rendering Frameworks

- **OpenInventor:** One of the first, wide-spread 3D scene graph systems, developed as a successor of Iris Inventor [316][57]. It is optimized for high-performance rendering using the fixed-function pipeline of early graphics hardware. (<http://oss.sgi.com/projects/inventor>)
- **OpenSG:** Open-source scene graph system for real-time computer graphic applications, which puts its focus on rendering performance, multi-threading and clustering [345]. It also incorporates features of modern programmable graphics hardware. (<http://www.opensg.org>)
- **OpenSceneGraph:** Well established open-source 3D graphics API, which is used in many applications and research projects [45]. Inspired by Iris Inventor, OpenSceneGraph provides more efficient OpenGL state encapsulation for better rendering performance; however, the generation of dynamic scene graph structures is more restricted compared to Inventor and OpenSG [336]. (<http://www.openscenegraph.org>)
- **OGRE3D**⁷: One of the most popular, fully object-oriented open-source rendering engines. It features modern rendering techniques and a clean separation between the scene hierarchy and state sorting. (<http://www.ogre3d.org>)
- **Irrlicht Engine:** Open-source 3D engine with a large development community. It is known for its small size, good performance, state of the art rendering techniques and good compatibility to new and old graphics hardware. (<http://irrlicht.sourceforge.net>)

⁷ Object-Oriented Graphics Rendering Engine

- **Unreal Engine:** The Unreal Engine is a game engine developed by Epic Games and is currently used in a wide range of commercial video games. Older versions are open-source and are also applied in some simulation systems. (<http://www.unrealengine.com>)
- **Unity3D:** Fully featured game engine that also includes physics, sound and tools for game development. Apart from the Unreal Engine, Unity3D is one of the most popular game engines [70]. (<http://www.unrealengine.com>)
- **Visualization Library (VL):** Open source C++ middleware for graphics applications that stays close to the underlying OpenGL 4 API. It provides an intuitive one to one mapping of functionalities in user friendly C++ object oriented framework. VL claims to avoid the architectural limitations of the uber-scene-graph paradigm by promoting the notion of data structure separation and specialization. (<http://www.visualizationlibrary.org>)

Rendering Libraries

- **SpeedTree:** Commercial middleware developed by IDV Inc., specialized for high-quality, high-performance tree and forest rendering. (<http://www.speedtree.com>)
- **SilverLining:** Commercial real-time cloud and sky rendering middleware developed by Sundog Software, which is used in many current video games. (<http://sundog-soft.com/sds>)
- **Graphite:** Commercial middleware developed by Quad Software. It is developed for high-performance terrain surface rendering including water and vegetation layers. (<http://www.quadsoftware.com/>)
- **TrueSKY:** Commercial middleware developed by Simul Software Ltd. It is intended for realistically animated and illuminated sky and cloud rendering. (<http://www.simul.co.uk/truesky>)
- **ViSTA FlowLib:** Visualization framework that provides rendering techniques to visualize unsteady flows in virtual 3D environments. It is part of the ViSTA⁸ virtual reality toolkit [177][11], which aims to provide VR technology into into technical and scientific applications. (<http://sourceforge.net/projects/vistavrtoolkit>)

General Mobile Robot Simulators

- **Player/Stage/Gazebo:** Open-source 3D multi-robot simulator for outdoor environments [170]. It features rigid-body dynamics and sensor simulations through ODE⁹. It uses OGRE3D as rendering framework. (<http://playerstage.sourceforge.net>)

⁸ Virtual Reality for Scientific Technical Applications

⁹ Open Dynamics Engine, library for simulating rigid body dynamics. (<http://www.ode.org>)

- **Webots:** Commercial development environment to model, program and simulate mobile robots in 3D environments [210]. It also uses ODE and features kinematics, rigid-body dynamics and sensor models. (<http://www.cyberbotics.com/overview>)
- **Microsoft Robotics Developer Studio:** Commercial development tool for robot applications, focusing on infrastructure aspects, visual programming and the Kinect¹⁰. (<http://www.microsoft.com/robotics>)
- **Easy-ROB:** Commercial planning and simulation software for robotic platforms in manufacturing plants that operate within work cells. It allows to program and visualize processes like handling, assembly, coating and sealing in a 3D environment. (<http://www.easy-rob.com>)
- **Robotics Toolbox:** Freeware that provides functions to generate trajectories and analyze results from simulated and real robots [60][59]. It is based on MATLAB/Simulink, a commercial block diagram environment for multi-domain simulation and model-based design. (<http://www.mathworks.de/products/simulink>)
- **ROS**¹¹: Open-source, meta-operating system for robots, which provides libraries and tools to develop robot applications. (<http://www.willowgarage.com/pages/software/ros-platform>)

Specialized Mobile Robotics Simulators

- **Simbad:** Java-based open-source 3D robot simulator. It is intended to test general AI algorithms in the context of autonomous robotics and agents for scientific and educational purposes. (<http://simbad.sourceforge.net>)
- **USARSim**¹²: Open-source 3D robot simulator intended as a research tool to simulate urban search and rescue robots and environments [351]. It features sensor simulations and acts as the development basis for the RoboCup¹³ rescue virtual robot competition. An open-source version of the Unreal Engine is used for rendering purposes. (<http://sourceforge.net/projects/usarsim>)
- **VORTEX:** Commercial framework for high-fidelity virtual training on various vehicles, work machines and heavy equipment in dynamic VR environments. (<http://www.vxsim.com>)
- **ROAMS**¹⁴: Research framework developed to simulate exploration rovers on planetary surfaces. It provides the functional simulation of the components and subsystems as well as their interaction with the environment, including kinematics, dynamics and navigation as well as the simulation of sensors and drives [154][145].

¹⁰ The Kinect is a motion sensing input device developed by Microsoft, which allows for gesture-based controlling.

¹¹ Robot Operating System

¹² Unified System for Automation and Robot Simulation

¹³ <http://www.robocuprescue.org/>

¹⁴ Rover Analysis, Modeling, and Simulation

All the named examples differ in the supported features, their graphical representation and their desired application domain. Rendering libraries provide very specific functionalities, they are not built as stand-alone systems and need to be integrated into a rendering framework. Similarly, rendering frameworks need to be integrated into a graphics application in order to realize a VR system. It becomes clear that a multi-domain VR simulation system has to combine a large number of single components in order to fulfill the desired goals of even a single application area like mobile robotics. This number is usually limited because of complexity, compatibility and performance reasons; in particular, applications that integrate state of the art real-time computer graphics demand specific data structures to perform well.

2.4 Discussion

In this chapter, an overview over current acceleration algorithms and scene representations for graphics applications as well as a brief overview of current rendering and simulation frameworks and libraries was given. Even if the listed examples are just a fraction of currently available software, the possibilities and broad applicability of VR-related systems in engineering and in research becomes clear. VR-based simulators like USARSim or Player/Stage/Gazebo make the application area of mobile robotics accessible to a broad range of researchers and students. These systems not require costly real world hardware to test novel approaches in early project stages. Considering the aspect of providing researchers from multiple disciplines with comprehensive data visualization techniques to analyze and explore corresponding data in virtual environments, the goals of the ViSTA/FlowLib framework can be regarded as good example for the work addressed in this thesis. Major differences of all presented frameworks can be seen regarding the desired tasks, the provided features, the manner of presentation as well as the underlying data structure. In particular, the latter point has constrained the realization of a unifying approach that combines of state of the art rendering and complex simulations in a single application. It was shown that real-time graphics applications rely on rendering-centric data structures like scene graphs to efficiently apply optimization techniques like culling and batching. The higher the demands for rendering quality and performance, the more rendering-centric the data structure needs to be, limiting its applicability and flexibility to other domains like complex simulations and vice-versa. This is one of the points where the gap between rendering and simulation can be seen. A current framework that follows the recent trend away from rendering-centric uber-scene-graphs to more flexible multi-view scene graphs is Visualization Library. It separates aspects like transformations, spatial relations or material properties into multiple domain-specific data structures. By further embracing the current trend of multi-view scene graphs, graph-oriented object databases grounded on the principles of semantic graph databases can be seen as a major step toward the realization of modern multi-domain VR simulation systems, which are capable of uniting complex numerical simulations at scientific level with attractive real-time computer graphics that consider aesthetic aspects. Semantic world models can not only be modeled in a "natural" and intuitive fashion, they can also be generated automatically when importing data from available semantic data sources like GIS servers.

Considering current rendering frameworks like Unity3D or OGRE3D that work on a classical scene-graph structure, specific requirements have to be considered to enable the rendering framework to apply optimization techniques. For example, draw call batching is only supported for geometry that is sharing the same material and textures; in addition, object nodes and static flags for corresponding elements need to be defined manually. This information makes the difference between optimized and non-optimized datasets. Since mostly static geometry can be optimized using the named techniques and on account of the fact that each movable object must not be defined as static, further knowledge about the scene must already exist during scene modeling. The definition of too many dynamic nodes has a negative impact on the performance while the definition of too few ones limits the dynamics and interaction capabilities of the scene. In order to maintain a well-performing renderer even for non-rendering-centric semantic datasets, the separation approach presented by Tobler, which uses a semantic and a rendering scene graph, is an ideal choice [336]; however, the rule objects responsible for the transformation of the semantic scene graph into a rendering scene graph need to be defined carefully. While Tobler suggests to define these rules for each node type and execute them directly during traversal, a more extensive approach with a global view of all elements is vital for complex sceneries that consist of multiple thousands of single nodes and dozens of different rendering techniques. To realize this extended approach, an actively managed object-oriented graph database is essential. It has to automatically inform the system about element creation, deletions and data manipulations to perform according reactions on-the-fly. Finally, the system structure itself has to be designed as flexible as the database. On the one hand, the general system concept has to ensure that structural or functional changes in one module do not affect other components. On the other hand, all components have to be able to interact with each other to benefit from arising synergy effects between rendering and simulation modules.

**Bridging the Gap between Rendering and Simulation
Frameworks**

Concepts, Approaches and Applications for Modern
Multi-Domain VR Simulation Systems

Hempe, N.

2016, XI, 244 p. 215 illus., 45 illus. in color., Softcover

ISBN: 978-3-658-14400-5