A Framework for Developing Flexible Virtual-Reality-centered Annotation Systems

Sebastian Pick* Torsten W. Kuhlen*,†

*Virtual Reality Group, RWTH Aachen University
*JARA - High Performance Computing
†Jülich Supercomputing Centre

ABSTRACT

The act of note-taking is an essential part of the data analysis process. It has been realized in form of various annotation systems that have been discussed in many publications. Unfortunately, the focus usually lies on high-level functionality, like interaction metaphors and display strategies. We argue that it is worthwhile to also consider software engineering aspects. Annotation systems often share similar functionality that can potentially be factored into reusable components with the goal to speed up the creation of new annotation systems. At the same time, however, VR-centered annotation systems are not only subject to application-specific requirements, but also to those arising from differences between the various VR platforms, like desktop VR setups or CAVEs. As a result, it is usually necessary to build application-specific VR-centered annotation systems from scratch instead of reusing existing components.

To improve this situation, we present a framework that provides reusable and adaptable building blocks to facilitate the creation of flexible annotation systems for VR applications. We discuss aspects ranging from data representation over persistence to the integration of new data types and interaction metaphors, especially in context of multi-platform applications. To underpin the benefits of such an approach and promote the proposed concepts, we describe how the framework was applied to several of our own projects.

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

1 INTRODUCTION

Note-taking is an imporant step of the data analysis process [16]. To provide the right means for note-taking in virtual reality (VR) in context of different application scenarios, a variety of specialized annotation systems have been developed in the past. Examples include systems for architectural walkthroughs [8], flow analysis [1], and oil platform construction reviews [7]. While every annotation system is geared toward a specific application, they usually share a common set of core functionalities. These include—but are not limited to—annotation creation and presentation, metadata in- and output (e.g., text, audio or sketches), and persistent storage.

However, even though a common set of core functionalities can be identified, the implementations of annotation systems vastly differ even from a high-level viewpoint. These differences usually arise due to one of three reasons for specialization: **S1**—*functional requirements*; **S2**—*application-specific extensions*; and **S3**—*platform characteristics*. Core components are subject to *functional requirements* including specialized data types (e.g., audio-synchronized camera paths [7]) or different requirements toward the persistence layer (e.g., distributed [9, 7] versus local storage [8]). Additionally, *application-specific extensions* might also be required, providing functonalities like automated annotation positioning (e.g., [10]) or structured annotation representations (e.g., [7]). Finally, diversity is further increased by differences in available VR platforms, ranging from desktop VR setups, over head-mounted displays (HMDs), to CAVE automatic virtual environments (CAVEs)—each featuring unique *platform characteristics* that entail certain implications for application design, like adapted metadata in- or output techniques (e.g., text input via keyboards for desktop VR versus speech recognition for CAVEs). No matter the reason, special requirements demand specialized annotation system implementations.

This situation gives rise to various challenges from a software engineering standpoint. First, individual components' interfaces have to be designed such that they can be combined with other components. This is especially important if components are to be exchangable. Second, if adaptability is a key requirement—e.g., to support multiple VR platforms—affected components have to be designed accordingly. These two aspects and the fact that annotation systems tend to quickly grow in complexity make the design of reusable annotation system building blocks difficult. As a results, it might become necessary to build annotation systems from scratch only to fulfill application-specific requirements.

Clearly, the prospect of having to rebuild annotation systems from scratch is economically not feasible. We have faced these issues in various of our own projects that require annotation capabilities, all of which were subject to one or more of the reasons for specialization (S1–S3) mentioned above. In order to circumvent the aforementioned issues, it would be worthwhile to have building blocks that are applicable to a variety of use cases. To achieve such a reusability without restricting extensibility, we argue that two essential requirements have to be fulfilled:

1. **R1**—Provide a set of resuable core components that anticipate the aforementioned reasons for specialization.
2. **R2**—Allow easy access to all annotation data to facilitate the development of entirely new extensions.

R1 is important to minimize re-implementation efforts in face of S1 and S3. R2, on the other hand, ensures that it is possible to add new functionality according to S2.

In this paper, we propose an annotation system framework that fulfills both requirements. We present details on the structure of the framework and give examples of how it was and is being applied to our own projects. To the best of our knowledge, such a discussion has not yet taken place for VR-centered annotation systems. Usually, the focus is on interaction metaphors (e.g., [4, 7, 17]), high-level concepts (e.g., [3]) or system overviews (e.g., [8, 3, 7]). Only rarely, generic workflows and system architectures are discussed (e.g., [1]). However, even if such a discussion takes place, it does not cover the software engineering challenges that arise when implementing concrete annotation systems.

Therefore, the contribution of this paper is two-fold. First, we give details on an approach that can be used as grounds to facilitate

*{pick | kuhlen}@vr.rwth-aachen.de

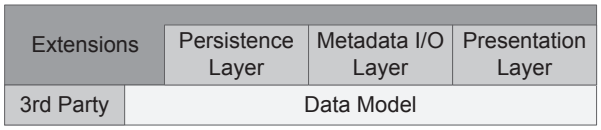

Figure 1: An overview of the framework's architecture. Core components include the data model—which is the base for all other components—, the persistence layer, the metadata I/O layer, and the presentation layer. Extensions are build on top of these core components and may use external dependencies, like 3rd party libraries.

the creation of annotation systems. Second, we empirically demonstrate its usefulness by discussing its application to actual projects.

The rest of the paper is structured as follows. In the next section we present our framework (see sec. 2). After this, we describe how the framework was applied to some of our own projects (see sec. 3), before concluding with a discussion (see sec. 4).

2 Annotation System Framework

This section contains the description of our annotation system framework. It is split into three major parts. First, an overview of the entire framework is given and its design guidelines are explained (see sec. 2.1). After that, more details on every core component are provided (see secs. 2.2 to 2.5). Finally, the development of extensions is briefly discussed (see sec. 2.6).

2.1 Overview

The underlying principle guiding our framework design is based on the fact that annotation data is at the core of the general note-taking process [16]. All operations that are performed during note-taking can be understood as processing annotation data in one way or another, including its creation, modification, and transformation. As a direct consequence, an annotation data model is at the heart of our framework (see fig. 1). All other components are developed directly on top of it. In accordance with the requirements from Section 1, all components are designed such that they can be reused in different application scenarios, but still retain extensibility and customizability. To this end, individual components represent rather lightweight structures that are loosely coupled to facilitate reusability.

To achieve the goal of providing reusable building blocks, our framework is only comprised of concepts that could be factored into generalized components. The resulting core components consist of the aforementioned data model (see sec. 2.2), a persistence layer (see sec. 2.3), a metadata handling system (see sec. 2.4), and a layer for presentation of individual annotations' data (see sec. 2.5). Any additional component is realized by means of these core components and understood as an extension. Overall, the framework offers three distinct points for specialization. Framework users can (1) use integrated points for customization, (2) define specializations of existing components, or (3) develop new extensions based on core components.

The limitation to the above core components does not mean that only these represent essential aspects of an annotation system. However, we argue that they constitute the most commonly used ones and therefore deserve the most attention w.r.t. the aforementioned requirements. As a result, our framework features other components that have proven essential to many applications but are still regarded as non-core components as they can be considered as too specific. One major example for this are the annotation layout facilities (see sec. 2.6).

VR support is an important design factor for our framework. While this is not explicitly reflected in the design of all components, it still has a substantial impact on some. This impact will be discussed in the respective sections. Also, it must be noted that our implementation of the discussed framework is done using a specific

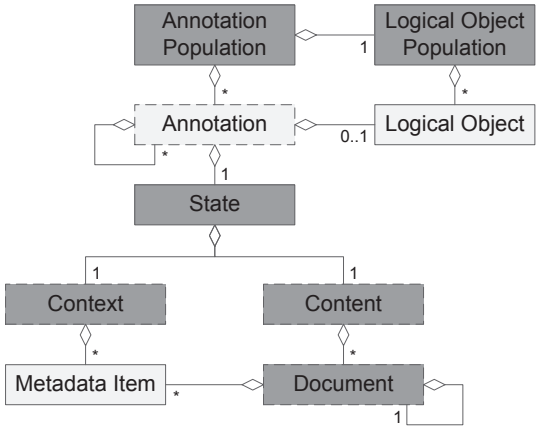

Figure 2: All data model components. Extensibility is achieved by integrated points for customization (dashed edges) and specializations of existing types (light gray).

VR toolkit, i.e., the ViSTA VR toolkit [2]. Nevertheless, we deem the framework's design general enough to be applicable to other toolkits and will thus not discuss toolkit-specific aspects (e.g., rendering facilities or network protocols). Such issues should be addressed using established abstraction techniques.

2.2 Data Model

The main objective of note-taking is to capture insights about a certain object of investigation. As such, it is a process of generating metadata and organizing it in a way that is meaningful to the investigator. Facilitating metadata organization is the main purpose of the data model. However, this organization is not limited to the investigator's intentions, but also has to consider other framework components. Each component has to be able to access the information required for its respective operations. For these reasons, the data model represents the most central component of the framework being the base for all other components. Ensuring that the requirements from Section 1 are fulfilled is hence of special importance. The data model is depicted schematically in Figure 2.

At the very base of the data model—being its most elemental part—are the *Metadata Items* (MIs). Each MI represents a primitive metadata entity, like a text fragment, an image, or an audio recording. New MIs are easily introduced by specializing an MI base class. Internally, MIs are allowed to hold any type of data and can organize it in any way required. Access to that data is granted by means of a reflection mechanism. The reflection mechanism has MIs expose relevant data fields by means of a series of keys. These keys can be used to query a data field's type and to retrieve its data payload. The same mechanism is used to supply data to MIs.

One design goal is to create MIs that store only a minimal set of metadata, instead of complex combinations of it. For example, instead of realizing a type like a 'labeled screenshot' as a single MI, it is encouraged to split it into an MI for text (label) and image (screenshot) each. The reason for this is that combining primitive types scales better, as they can be reused more easily in other contexts. A structuring mechanism that enables the combination of MIs into complex types is discussed below.

To organize MIs of one annotation, we employ an approach based on the Context-Content Model (CCM) introduced by Assenmacher et al. [1]. The CCM groups MIs either into the *context*—which holds contextual information, like the viewer position at annotation creation—or the *content*—which contains the actual user-generated note-taking metadata. In contrast to the original CCM, our approach allows for a more advanced organization of MIs. As contextual metadata is usually generated automatically, it is orga-

nized in the context in the same way as in the original CCM, i.e., in an unordered list without further structure. For content-related MIs, however, we employ a document-style hierarchical structure. The reason for this lies in the fact that note-taking metadata itself usually has a document-like structure, like the aforementioned 'labeled screenshot' example. As a result, MIs are first assigned to a *document* before storing them in the content. A document can hold an arbitrary number of MIs to facilitate grouping of related items, e.g., a series of screenshots. In addition, a document can be assigned a parent document, which is used for the creation of hierarchies. This way, the aforementioned 'labeled screenshot' can be realized by first creating a document containing a text MI and another containing an image MI. Next, the image document is made the parent of the text document. Note, that this organization does not imply a potential graphical layout but only a logical one.

Together, the data stored in context and content makes up an annotation. However, instead of directly associating context and content to an annotation, they are first combined into a so-called *state*, of which an annotation can hold an arbitrary number. These states are used to reflect discrete changes in annotation data. The main application scenario for this is time-variant data. Here, states can be used to change the annotation data according to changes in the annotated object. In our framework this is realized by a specialized annotation type. It requires from each of its states that their contexts contain timing information, which is then used to determine the currently active state. This way, states are an incarnation of the annotation mechanism for time-varying data as presented in [1]. In general, specialized annotation types are only required in order to determine the active state according to a given condition.

In addition to the data stored in annotations, every annotation can also store links to related annotations. This way, complex relationships can be expressed. One application scenario for this was discussed in [7]. By relating annotations to each other, an issue tracking system was realized. It allows users to have issue-related discussions by adding new annotations to an issue and relating them to other annotations of that issue, e.g., to reply to previous remarks.

To extract annotatable objects from the data set under investigation, we employ the concept of *Logical Objects* (LOs) introduced in [1]. An LO represents an arbitrary subset of scene data that an annotation relates to. It can, therefore, be understood as an adapter between scene data and the annotation framework. Often, an LO contains the information that is required to reconstruct relevant aspects of the scene data independently from the original data set. Developers can introduce arbitrary new types of annotatable objects by specializing an LO base class, which only imposes a minimal interface. No constraints are placed on the data that can be contained in an LO. It has to be noted that it is optional for an annotation to relate to an LO, as such a relation cannot necessarily be established in every situation. One such situation, which was also discussed in [1], is the creation of overview screenshots of the entire scene.

At the top of the data model reside the *annotation population* and the *logical object population*. Both constitute collections for annotations and LOs, respectively. While populations are also responsible for managing the lifetimes of entries added to them, their main purpose is to group related annotation data and provide a central point of access. If entries are added to or removed from a population, appropriate messages are emitted using an observer pattern [5] such that dependent components can directly react to changes. This behavior is exploited by various (core) components as explained in the following sections. In addition, basic data consistency is also ensured. This means that if an annotation is added to a population, it is checked whether its LO is contained in the associated LO population. If this is not the case, it will be automatically added, thereby ensuring accessibility of all referenced LOs.

The described data model fulfills both requirements from Section 1. In accordance to R1, it is possible to make any type of object annotatable and store arbitrary types of metadata in annotations. Structuring mechanisms further allow to express advanced relations between metadata primitives and entire annotations. As all annotation data is accessible, the creation of extensions is also ensured, thus fulfilling R2.

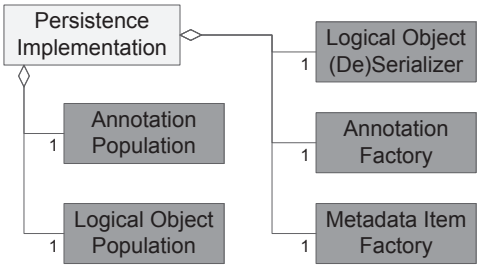

Figure 3: The persistence layer provides common functionality required to store and restore annotation data (dark gray) to concrete persistence implementations (light gray).

2.3 Persistence Layer

Similarly important as the organization of annotation data at runtime is its persistent storage, such that it can be retained for future access. Consequently, the persitence layer is designed such that specific persistence implementations are enabled to handle any kind of annotation data. A specific persistence implementation is granted access to annotation data by means of the annotation and LO populations. The most important operations, i.e., preparing objects for storage and restoring them from persisted data, are facilitated by serialization and factory approaches (see fig. 3).

In general, every storage and retrieval operation happens in two phases. First, LOs are processed, after which annotations are. This order is enforced for all persistence implementations using the template method design pattern [5]. The reason for this is the dependence of annotations on LOs. An LO is part of an annotation's data and can potentially be referenced by multiple annotations at once. Thus, first processing LOs ensures that annotations can rely on related LOs' proper handling before being processed themselves. Thereby, data handling issues, like data corruption (storage) and incomplete annotation data (retrieval) can be avoided.

Storage and retrieval look similar for all annotation data. They involve a variety of (de)serialization operations in conjunction with implementation-specific storage operations. Storage operations are hidden in the concrete persistence implementation and no constraints are placed on their realization. (De)serialization operations, on the other hand, have to be designed in a generalized fashion, since data types can be specialized. Consequently, our framework provides such generalized (de)serialization facilities for all user-definable types, i.e., LOs, annotations and MIs.

As mentioned in Section 2.2, the design of LOs can be freely chosen and is thus transparent to persistence implementations. Consequently, LO (de)serialization also has to happen in a transparent fashion. To this end, for every LO a factory-like (de)serialization function has to be specified by its implementer. These functions are organized in the logical object (de)serializer (see fig. 3). To store an LO, it is passed to the logical object (de)serializer, where the matching (de)serialization function is determined and the LO serialized accordingly. To restore an LO, the serialized data and the LO's type identifier are passed to the logical object (de)serializer, which creates a concrete LO instance from it.

Compared to LOs, persisting annotations is relatively simple. Since annotation specializations are only required for state management, only the annotation type has to be stored to be able to restore it. For this, a factory approach is chosen for which implementers only register annotation types that are to be used.

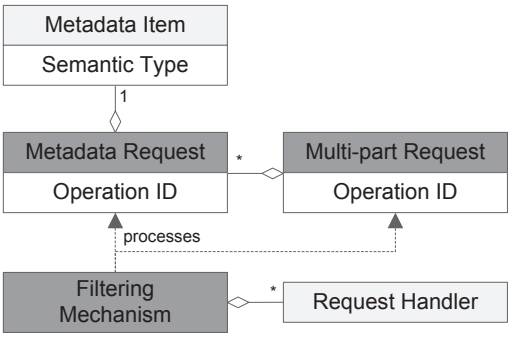

Figure 4: The metadata handling system allows to formulate arbitrary metadata in- and output requests based on the available Metadata Items. Every request is matched semi-automatically to available in- and output techniques by means of a user-in-the-loop approach.

MIs are (de)serialized exploiting their reflection mechanism. Persistence implementations retrieve the contents of each of an MI's data fields and are free to store them in any way desired. For this to work, it must be ensured that contents of every data field can be serialized. To this end, the reflection mechanism always returns contents in a serialized form. Primitive types, like numerical values, are retrieved in their native form. Complex types, like image data, on the other hand, is provided in a pre-serialized form using an approach equal to the one used for LOs. To restore MIs, first a factory approach is used to create a concrete MI instance, after which its fields are restored.

In our framework implementation, serialization data occurs in the form of byte arrays. For persistence implementations that cannot directly handle binary data, we use a base64 encoder to transform it into strings. A concrete example for such an implementation is one based on SOAP web services.

The design of the persistence layer allows for the use of arbitrary storage approaches for two reasons. First, all relevant annotation data can be accessed through the annotation data populations. Second, the generalized serialization facilities reduce storage operations to organizing chunks of already serialized data. Therefore, new implementations can be easily created and reused, whereby requirements R1 and R2 are fulfilled.

2.4 Metadata Handling

As stated before, metadata is at the heart of the note-taking process and represented by MIs in our framework. Consequently, appropriate workflows for the in- and output of metadata—so-called *Metadata Input/Output Operations* (MIOPs)—have to be provided. Unfortunately, designing such workflows is not trivial and rigid, pre-defined workflows are not a viable solution.

Instead, it is desireable to be able to create adaptable, easy-to-extend workflows for three reasons. First, workflows have to be adapted to the used VR platform. Usually interaction techniques vary across platforms for technical reasons and choices often depend on subjective or task-related reasons. Hence, workflows should only offer techniques that are suited for a given MIOP. Second, not all types of metadata can be captured using single interaction techniques. For example, creating a 'labeled screenshot' requires the use of at least two techniques. Third, new MIs might have to be added, requiring the extension of existing workflows.

We argue that a mechanism that facilitates metadata in- and output in face of these issues has to fulfill three requirements. (1) It has to automatically identify suitable techniques—so-called *Metadata Handling Techniques* (MHTs)—for a given platform and MIOP. (2) It has to automatically arrange MHTs into workflows if required. (3) It has to be extensible w.r.t. new MHTs and MIs. In the following, we discuss how such an approach is realized in our framework.

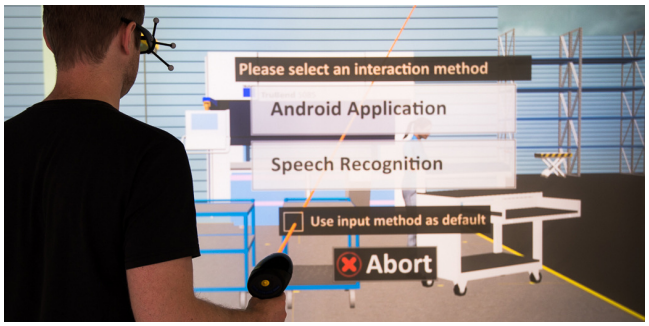

Figure 6: A user is prompted by the metadata handling system's filtering mechanism to choose one of several available text input techniques (see sec. 2.4.3). Here, the system is applied to the flapAssist application [12] (see sec. 3.3).

We first give an overview of the approach (see sec. 2.4.1). Then we explain how MIOPs are triggered (see sec. 2.4.2) and appropriate MHTs selected (see sec. 2.4.3).

2.4.1 Approach Overview

In general, our approach works similar to Android's Intent system [6]. First, a request for a MIOP (Android: Intent) is issued to the system. Next, a filtering mechanism identifies MHTs (Android: Activities) that can handle the request. In case multiple MHTs were identified, the user is prompted to make a choice.

The idea of introducing a filtering mechanism helps to fulfill the requirements introduced at the beginning of this section. Instead of presenting the user with a fixed choice of MHTs, the filtering mechanism can first identify those techniques that are suitable for a given request, the used platform, and the present in- and output devices. This way, workflows are automatically formulated by the system avoiding the need to manually design them. Finally, the user can choose from identified MHTs based on her preferences and the given MIOP, in case multiple alternatives are available.

However, in order to be applicable to the selection of MHTs, the Intent concept has to be extended. Figure 4 gives an overview of the components of our system, which are described below.

2.4.2 Metadata Request Definition

MIOPs are handled in various steps and initially triggered by formulating a *Metadata Request* (see fig. 5). The request consists of an operation identifier, which indicates whether an in- or an output operation is to be performed, and an MI, which indicates the kind of metadata to be handled. In contrast to Android's Intents, a request is inherently tied to a data object representing the metadata, i.e., the MI. More complex requests like 'create labeled screenshot', are formulated as an ordered list of individual requests, called *Multi-part Metadata Requests*, which are processed one after another.

The MI provides two important pieces of information. First, its data entries define the data format in which metadata is organized, e.g., as a pixel image. Second, each MI is assigned a so-called *semantic type* which allows to semantically interpet the data stored within an MI. For example, it indicates whether a pixel image represents a screenshot or a sketch. The semantic type is an integral part of every MI in addition to the data entries already discussed in Section 2.2. Together, both pieces of information are important to determine a suitable MHT, as described in Section 2.4.3.

2.4.3 Filtering Mechanism

Requests are passed on to the filtering mechanism that determines applicable MHTs in form of *Request Handlers*. Each handler represents a single MHT and has to fulfill three tasks. First, it exposes

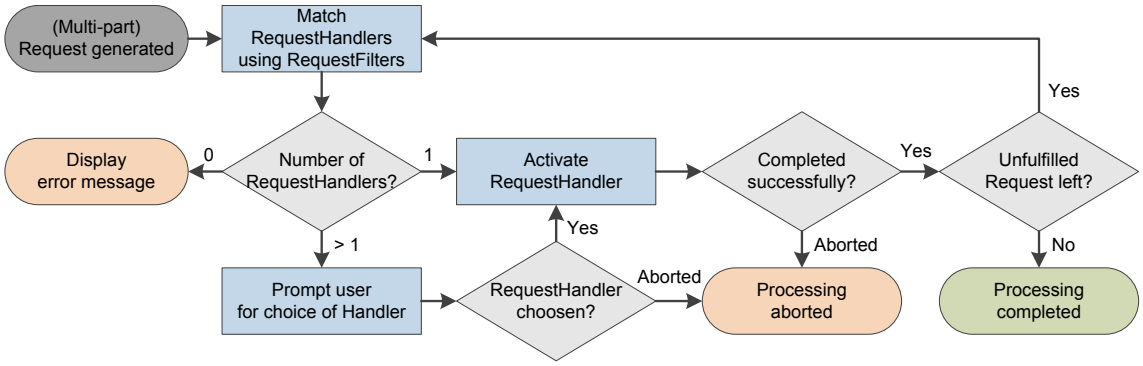

Figure 5: Metadata Input/Output Operation handling is triggered by generating a Metadata Request (dark gray box). Next, the filtering mechanism attempts to fulfill the request by identifying and activating a suitable Request Handler (blue boxes). Based on the outcomes of these steps (gray diamonds) request processing is either aborted or successfully completed (red and green boxes).

a technique's capabilities in form of *Request Filters* and platform information. Both are used to determined whether an MHT can be used to handle a request or not. Second, it is responsible for transferring the metadata between the MHT and the MI, including any required data conversion. Third, it activates the associated MHT if a new request is passed to it and deactivates it again after the request was handled. Based on the number of identified handlers, the filtering mechanism emits an error to the user (no handlers found), passes the request to the handler for processing (exactly one handler found), or prompts the user to choose among available handlers (more than one handler found; see fig. 5).

In general, the filtering mechanism works similarly to Android's Intent system [6]. However, the role of handlers differs from those of Android's Activities. Instead of representing full-fledged MHTs themselves, they are only an interface between the actual technique and the filtering mechanism. Hence, the number of adaptations necessary to use techniques with the annotation framework that are otherwise independent of it can be reduced.

Apart from the handlers, the rules used by the filtering mechanism to match requests with handlers were also modified compared to Android. The filtering happens in two steps. First, the handler's platform information is checked against the used platform. This way, it is determined whether a technique is applicable to a given platform and whether all required in- and output devices are present. If this check fails, the handler is deemed unsuitable; if it succeeds, the second filtering step commences.

In this step, the request is matched to all of the handler's filters. To this end, each filter holds the same information as a request to indicate processible request configurations. This includes an operation identifier, a semantic type, and a data format (see sec. 2.4.2). Matching a request to a filter can have one of three outcomes:

1. **No match** Operation identifiers do not match, or they match, but neither data format nor semantic type match.
2. **Weak match** Operation identifiers and data formats match, but semantic types do not.
3. **Strong match** Operation identifiers, data formats, and semantic types match.

A handler is considered suitable for a request, if at least one of its filters yields either a weak or strong match. The idea behind this is that while only strong matches indicate a fully compatible handler, weak matches can still be used as fallback techniques. For example, a viewer for generic images can still be used for viewing sketches even though it cannot provide editing functionality. For this reason, the list that is shown to a user to prompt her to choose from available MHTs, consists of all techniques for which a weakly or a strongly matching filter was found.

The metadata handling system fulfills requirements R1 and R2. The abstraction by Reuqest Handlers allows to use any type of MHT desired, especially those that are implemented in terms of other toolkits or 3rd party libraries. At the same time, handlers constitute reusable building blocks. Issues that arise in context of multi-platform VR applications are also addressed.

2.5 Presentation Layer

So far only creation, handling, and storage of annotation data have been discussed. Another essential aspect, however, is its presentation. Depending on the data stored in an annotation and its organization, the annotation's *presentation form* can differ. For example, a text might indicate that a label is to be displayed, while a screenshot together with viewing information indicate a viewpoint annotation (see fig. 7). In general, the presentation form of annotation data depends on the purpose for which an annotation was created.

However, explicitly assigning a presentation form to an annotation at creation is undesireable for various reasons. First, naïve solutions easily lead to a rigid design that is difficult to maintain. Second, it must be ensured that the link between annotation and presentation form is restored when annotations are loaded using the persistence layer. To avoid these issues, we employ an approach that decouples the creation of an annotation from the choice of a presentation form. First, an annotation is created and added to an annotation population. Next, a *presentation manager* observing that population retrieves the annotation and generates an appropriate *presentation form* using a factory design pattern (see fig. 8).

This factory approach uses two pieces of information to deter-

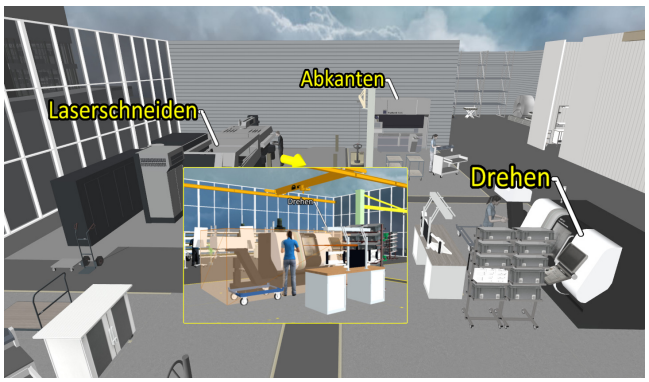

Figure 7: The factory layout planning application flapAssist [12] uses annotations to capture layout-related comments. Labeled (top) and viewpoint annotations (bottom middle) are shown in yellow.

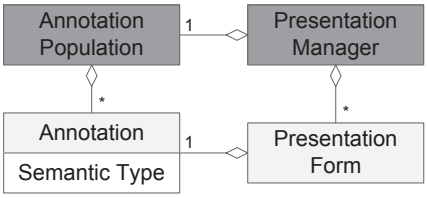

Figure 8: In the presentation layer, a presentation manager generates presentation forms for annotations added to a population. It does so by considering the annotation's class and semantic types.

mine the presentation form. First, the annotation type is used as it was already discussed in Section 2.2. In addition, each annotation also holds a *semantic type* similar to the one used for metadata handling (see sec. 2.4). It is used to indicate the purpose of an annotation. For the two aforementioned types of annotations shown in Figure 7—i.e., labeled and viewpoint annotations—two different semantic types were used, while using the same annotation type.

This abstraction addresses both of the aforementioned issues and preempts the need of rigidly linking annotations to presentation forms. First, links can be changed by configuring the presentation manager accordingly. Second, the presentation form is automatically created when annotations are restored by a persistence implementation. It is even possible to change the presentation form after annotations have been persisted. Technically, each presentation form is based on a common base class that offers facilities to access the associated annotation's data. Each concrete presentation form has to verify whether the supplied annotation data contains the required information. If this is not the case, the presentation form creation is aborted. Aside from these checks, presentation forms can be implemented in any way desired. The choice for what annotation data is presented and how, is entirely within the responsibility of the concrete presentation form.

The above design facilitates the creation of arbitrary types of presentation forms for annotation data. It offers a flexible link between the annotation data itself and what presentation form is to be used for it. Consequently, requirements R1 and R2 are both fulfilled.

2.6 Extensions

In the previous sections the core components of the annotation framework have been discussed. Here, we briefly describe how they can be utilized to implement additional extensions. In general, additional components are considered to be extensions, if they add functionality that makes very specific assumptions thus making it difficult to generalize or extent them.

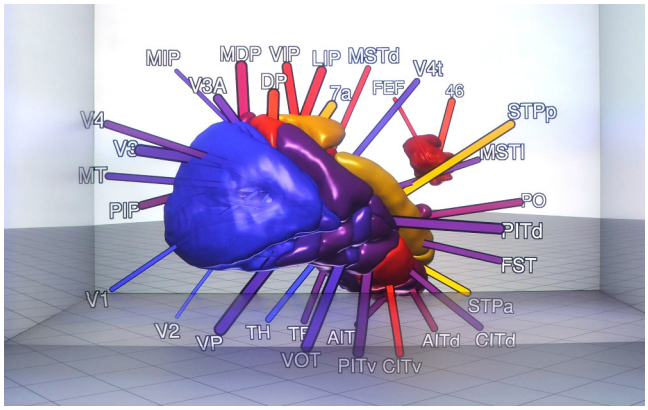

Figure 9: The VisNEST application [11] for exploring brain activity data. It utilizes an automated layout algorithm [13] and encodes activity data of brain areas in annotations' presentation forms, i.e., their leading lines' width and color (see sec. 3.2).

One example for such an extension are automated annotation layout algorithms. The purpose of these algorithms is to automatically position annotations within the virtual environment (VE) such that visual accessibility is ensured and the perception of annotations improved. One such algorithm is shown in Figure 9, which—being targeted at clusters of centralized objects—produces radial layouts [13]. All layout algorithms currently available in our framework—including radial, column and offset layouts—make certain assumptions about annotations and LOs. For example, annotation forms that shall be used with a layout algorithm need to allow to change their transformation (see sec. 2.5). Also, associated LOs have to provide certain types of information, e.g., a so-called *anchor point*, which is used for connecting leading lines or for determining relative annotation positions (see fig. 9).

Technically, the layout extension depends on the data model and the annotation presentation layer components. To transfer newly created annotations to a layout algorithm, a filter mechanism is employed. The filter mechanism observes the presentation manager and the LO population and ensures that only those annotations and LOs are added to a layout algorithm, that are of the correct presentation form and contain the required metadata. As layout algorithms retrieve the data they work on from the populations, it is possible to swap layout algorithms at any time.

In general, the layout extension fulfills requirement R1 as it is based on core components of the framework. On the other hand, R2 is usually not fulfilled as layout algorithms often have to be adapted to meet specific needs.

3 Results

In this section we describe how the discussed framework has been applied to several applications. We point out specific needs of each application in context of the annotation framework with the goal to promote the concepts discussed in the previous section.

3.1 Exploration of Air Traffic Effects

In the *Virtual Air Traffic System Simulation* (VATSS) project [14] an immersive virtual environment (IVE) targeting CAVEs was developed, which allows users to experience the effects of air-traffic-related noise emissions. The purpose of this application is to facilitate the communication between different groups of stakeholders during the approval process of air-traffic-related projects, like the addition of runways to existing airports or the introduction of new types of aircraft. To this end, users are able to not only look at visualizations of noise emissions, but also to experience them by means of a real-time 3D auralization approach (see fig. 10).

The VATSS application uses annotations to enable users to mark points of interest on the ground, e.g., to mark the house of an affected resident. These markers can then be used to analyze simulation scenarios, e.g., by comparing the noise levels from different simulations at the marked locations. In total, two different types of markers can be created: location and threshold markers. Location markers represent location bookmarks to points of interest. Users can freely choose names for these markers by means of a speech-recognition-based text input system. Subsequently, these markers can be used to indicate the destination for a semi-automated traveling technique, such that users can easily move there in order to listen to auralized noise emissions. The threshold marker, on the other hand, indicates if a certain noise level has been exceeded at a point on the ground. When it is placed, the user is prompted to choose a scalar noise level threshold by means of a slider-based interaction technique. Whenever the noise threshold is exceeded, the marker visually signals this to the user. Additionally, when a user clicks on such a threshold marker, those segments of an airplane's trajectory are highlighted, for which the threshold at the marker's location is exceeded (see fig. 10). This way, users are made aware

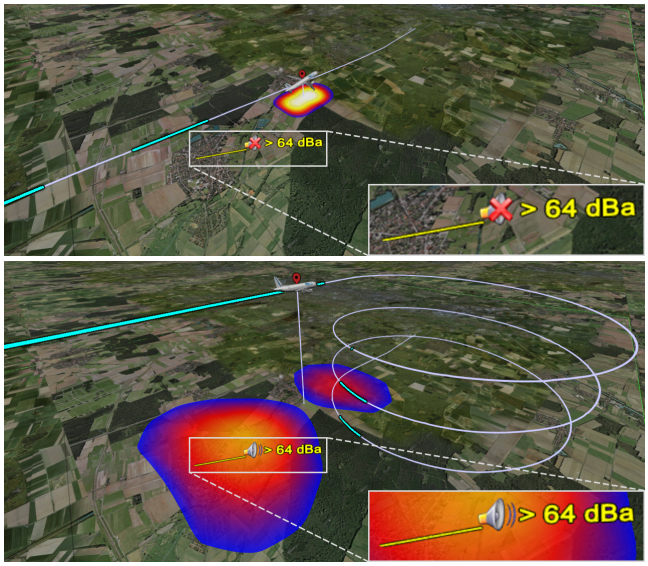

Figure 10: The Virtual Air Traffic System Simulation application [14] used to communicate effects of air-traffic-related noise emissions. It supports data annotation for analysis and comparison of air traffic simulations (see sec. 3.1). Here, a marker indicates whether the noise level at a given location exceeds a certain threshold. Clicking on it highlights those airplane trajectory segments, for which the threshold is exceeded (highlighted in blue).

of changes in noise levels at user-defined locations while looking at the running simulation or when comparing simulations.

For the realization of the above functionality, all of the framework's components were used. To define the two different marker types, a common annotation type and two presentation forms were created (see sec. 2.5). For location markers the labeled annotations were used, while a specialized form of them was derived for threshold markers. Placement of annotations happened by means of a static offset layout algorithm. Whenever an annotation was created, first, an appropriate LO was instantiated for it. For location markers, the LO's data only consisted of a vector indicating the point on the ground. For threshold markers the time-dependent noise data at the given location was also stored for each associated simulation. Annotation of time-dependent data was done using the state concept introduced in Section 2.2. Immediately after LO creation, a metadata request for either text input (location marker) or definition of a scalar floating point value (threshold marker) was emitted. For the aforementioned input techniques, appropriate handlers were registered with the metadata handling system (see sec. 2.4). Annotation data was persisted using an XML-based persistence implementation (see sec. 2.3). The semi-automated traveling technique for location markers was developed as an extension (see sec. 2.6).

3.2 Exploration of Brain Activity Data

The application *VisNEST* [11] is being developed as a tool for the exploration of brain activity data generated by the NEST simulator. To this end, VisNEST provides various visualizations that offer different views onto acivity data.

In this setting, annotations are used to augment visualizations with additional contextual information. For example, annotations are used to provide anatomical names for geometrical representations of brain areas (see fig. 9). Users can explore the brain model by means of navigation and direct manipulation techniques, like dragging around individual areas to reveal views onto hidden ones. In addition to providing anatomical names for brain areas, annotations also encode an area's current activity w.r.t. the time-dependent simulation data. Depending on the activity, the color and width of an annotation's leading line is modulated (see fig. 9).

In contrast to the VATSS application, VisNEST uses the annotation framework only for data presentation and does not allow to create new annotations. Overall, only the data model, the presentation layer and the layout extension were used. To encode the brain activity data, a specialization of the labeled annotations was derived. For this, each LO contains the time-dependent simulation data for its respective brain area. Annotation placement was realized using a radial layout algorithm [13], which also ensures the correct annotation placement during direct manipulation operations, like dragging. Furthermore, annotation data is not persistently stored, but generated on the fly from the brain activity data that was loaded.

3.3 Factory Layout Planning

The *Factory Layout Planning Assistant* (flapAssist) application [12] is one of several applications currently developed in the context of virtual production within the Cluster of Excellence "Integrative Production Technology for High-Wage Countries" project. Its purpose is to facilitate the factory layout planning process by means of visualization and VR approaches. To this end, flapAssist interfaces with a commercial layout planning tool to instantly retrieve planning data and enable immediate virtual walkthroughs within an IVE. Additional decision-relevant information is made accessible in form of visualizations.

Within flapAssist, annotations are used to capture insights gathered and decisions made during virtual walkthroughs regarding the factory layout. To this end, users can create different generic annotations, like labeled or viewpoint annotations (see fig. 7), and store various types of metadata in them, e.g., audio comments or sketches. While the main target platform for flapAssist are CAVE-like IVEs, it also has to support non-immersive desktop systems. The reasons for this is, that insights generated during a virtual walkthrough have to be re-traceable even if an immersive VR system is not available. This is the common case as planners involved usually do not have access to VR hardware apart from special planning sessions. To ease this exchange of information between systems and planners that might even be remotely distributed, the so-called Virtual Production Intelligence Plattform (VPIP) [15] is being developed by project partners. It constitutes a web-based data integration platform that is used for distribution of any planning-relavant data among involved parties. For this reason, it has to be possible to also exchange annotation data through it. One feature currently under development, is a discussion system similar to the one from [7].

flapAssist makes extensive use of all framework components. First, several annotation types, like labeled and viewpoint annotations, can be created using the presentation layer (see sec. 2.5). The user can add arbitrary MIs to annotations by generating the appropriate metadata request via a context menu invoked on annotations. To support the metadata input within CAVE-like IVEs and also on desktop systems, the metadata handling system is heavily used (see sec. 2.4). On desktops, users can use established means, e.g., to capture screenshots or enter texts. For CAVEs, a variety of techniques is combined, e.g., text input is supported by speech recognition as well as by means of an Android-based application (see fig. 6). The latter is also used to record or playback audio comments. For annotation positioning an offset layout algorithm is currently being used. Persistent storage of annotations can be configured to store annotation data either locally or remotely by simply swapping persistence implementations. For local storage, the XML-based approach developed for the VATSS application (see sec. 3.1) has been reused. For remote storage, on the other hand, an implementation based on SOAP web services was created, which is compatible with the aforementioned VPIP data integration backend. The discussion system is currently being developed on top of the data structuring mechanisms from the data model (see sec. 2.2).

4 Discussion and Conclusion

We have described an annotation system framework that helps to speed up the creation of VR-centered annotation systems. It does so, by providing various core components that provide common functionality present in almost every annotation system. At the same time, these components are designed for an increased degree of extensibility and customizability, such that annotation systems can be easily adapted to application-specific needs. To promote our design choices, we have discussed how the framework was used in different contexts and how it facilitated the realization of application-specific needs.

Overall, the framework provided sufficient flexibility to apply it to a variety of application scenarios. This means that requirements put forward by each application could be realized either by harnessing the provided points for specialization, or by implementing new extensions relying on the basic mechanics provided by the core components. Due to the loose coupling between core components, a library of reusable building blocks was developed during the creation of application-specific annotation systems. This includes—among others—specialized persistence implementations, layout algorithms, annotation presentation forms, and metadata handling techniques. Some of these building blocks were already reused by the described applications. Therefore, we argue that our architecture fulfills the requirements as introduced in Section 1.

Nevertheless, while the framework has already been successfully applied, it is still undergoing active development. The design of the framework is based on the identification of recurring patterns across different application scenarios. These recurring patterns are then extracted into generalized concepts, which—so far—have resulted in the presented components. Consequently, core components only reflect those aspects of an annotation system that could be generalized to a point where they are applicable to other application scenarios as well. Even though the framework already covers various aspects, some areas have not yet been touched at all. Furthermore, some components can be extended by additional functionality.

One example for still uncovered functionality is the segmentation of scene data into annotatable objects—the aforementioned Logical Objects (LOs). In several scenarios this segmentation might happen as a pre-processing step as proposed by some approaches. Segmentation can, however, often only happen at runtime as it depends on the user's intentions to annotate. So far, it is the repsonsibility of the application to generate LOs without any framework-support apart from the data model, e.g., as described for the VATSS and flapAssist applications. To further extend our framework, we are currently working on a generalized approach to LO generation and similarly useful components.

An example for a component that can be further improved is the presentation layer. So far, different presentation forms are discerned solely by an annotation's semantic type. While this allows to realize any desired presentation form, it is a rather coarse approach as it allows to influence data presentation only on the level of entire annotations. Thus, new presentation forms have to always consider the entire annotation data, even if only the presentation of a subpart of it has to be changed. Instead, it would be desireable to be able to specify presentation forms for subsets of an annotation's metadata, e.g., only a certain document or an individual metadata primitive. To realize such a fine-grained approach, we are currently working on integrating techniques used in automated user interface generation into the presentation layer.

In summary, using the presented annotation system framework has already significantly helped us to realize annotation systems for specific application scenarios. We could reuse existing building blocks and could focus on application-specific functionality during development, relying on the provided low-level functionality instead of having to re-build it. As a result, we argue that adapting the proposed concepts is worthwhile. At the same time, we are working on extending the framework to further increase it usefulness. Overall, our framework is a good basis for the creation of application-specifc, VR-centered annotation systems.

Acknowledgements

This research was funded by the German Research Foundation DFG as part of the Cluster of Excellence "Integrative Production Technology for High-Wage Countries".

References

[1] I. Assenmacher, B. Hentschel, C. Ni, T. Kuhlen, and C. Bischof. Interactive Data Annotation in Virtual Environments. In *Proc. of the Eurographics Conf. on Virtual Environments*, pages 119–126, 2006.

[2] I. Assenmacher and T. Kuhlen. The ViSTA Virtual Reality Toolkit. In *Proc. of the IEEE VR Workshop Software Engineering and Architectures for Realtime Interactive Systems*, pages 23–28, 2008.

[3] D. Bowman, C. North, and J. Chen. Information-rich virtual environments: theory, tools, and research agenda. *Proc. of the ACM Symposium on Virtual reality Software and Technology*, pages 81–90, 2003.

[4] J. Chen, P. Pyla, and D. Bowman. Testbed evaluation of navigation and text display techniques in an information-rich virtual environment. *Proc. of the IEEE Conf. on Virtual Reality*, pages 181–188, 2004.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] I. Google. Android Intents and Intent Filters, 2014. http://developer.android.com/guide/components/intents-filters.html, last visited on 2014/11/21.

[7] J. Guerreiro, D. Medeiros, D. Mendes, M. Sousa, J. Jorge, A. Raposo, and I. Santos. Beyond Post-It: Structured Multimedia Annotation for Collaborative VEs. In *Proc. of the Eurographics Conference on Virtual Environments*, pages 55–62, 2014.

[8] R. Harmon, W. Patterson, W. Ribarsky, and J. Bolter. The Virtual Annotation System. In *Proc. of the IEEE Virtual Reality Annual International Symposium*, pages 239–245, 1996.

[9] T. Jung, M. D. Gross, and E. Y.-L. Do. Annotating and Sketching on 3D Web Models. In *Proc. of the ACM Conference on Intelligent User Interfaces*, pages 95–102, 2002.

[10] S. Maass and J. Döllner. Efficient View Management for Dynamic Annotation Placement in Virtual Landscapes. In *Springer Smart Graphics*, pages 1–12, 2006.

[11] C. Nowke, M. Schmidt, S. J. V. Albada, J. M. Eppler, R. Bakker, M. Diesmann, B. Hentschel, and T. Kuhlen. VisNEST Interactive Analysis of Neural Activity Data. *IEEE Symposium on Biological Data Visualization (BioVis)*, pages 65–72, 2013.

[12] S. Pick, S. Gebhardt, K. Kreisköther, R. Reinhard, H. Voet, C. Büscher, and T. Kuhlen. Advanced Virtual Reality and Visualization Support for Factory Layout Planning. In *Proc. of the Conference Entwerfen Entwickeln Erleben*, 2014.

[13] S. Pick, B. Hentschel, M. Wolter, I. Tedjo-Palczynski, and T. Kuhlen. Automated Positioning of Annotations in Immersive Virtual Environments. In *Proc. of the Joint Virtual Reality Conference of EuroVR - EGVE - VEC*, pages 1–8, 2010.

[14] S. Pick, F. Wefers, B. Hentschel, and T. Kuhlen. Virtual Air Traffic System Simulation—Aiding the Communication of Air Traffic Effects. In *Poster Proc. of IEEE Virtual Reality*, pages 133–134, 2013.

[15] R. Reinhard, C. Büscher, T. Meisen, D. Schilberg, and S. Jeschke. Virtual Production Intelligence—A Contribution to the Digital Factory. In C.-Y. Sum, S. Rakheja, and H. Liu, editors, *Intelligent Robotics and Applications*, volume 7506 of *Lecture Notes in Computer Science*, pages 706–715. Springer Berlin Heidelberg, 2012.

[16] R. Springmeyer, M. Blattner, and N. Max. A Characterization of the Scientific Data Analysis Process. In *Proc. of the Conference on Visualization*, pages 235–242, 1992.

[17] M. Tsang, G. W. Fitzmaurice, G. Kurtenbach, A. Khan, and B. Buxton. Boom Chameleon: Simultaneous capture of 3D viewpoint, voice and gesture annotations on a spatially-aware display. In *Proc. of the ACM Symp. on User Interface Software and Technology*, pages 111–120, 2002.