The USE-VR Platform – A Framework for Interoperability among Different VR Solutions

Benjamin Mesing, Matthias Vahl, Uwe von Lukas

Zentrum für Graphische Datenverarbeitung e.V. (ZGDV),
Joachim-Jungius-Straße 11, 18059 Rostock, Germany

Abstract:

Today, there exists a multitude of different VR solutions, each providing its own advantages and application areas. Needless to say that companies use the VR solutions best suited for their specific purposes. But commercially available VR applications often do not cover all the individual use cases which might occur in different companies – a fact which makes extra customisation necessary. This especially holds true for the shipbuilding industry with its specific needs not addressed by today's VR solutions. Furthermore, VR applications are mostly incompatible among each other which makes the collaboration of different tools impossible, at the moment. With the USE-VR Platform, ZGDV provides a flexible concept for the development of custom modifications which are usable together with different VR applications and which, besides other features, support the collaboration among different VR solutions. This paper will present the USE-VR Platform and illustrate its usage with a use case from the shipbuilding industry.

Keywords: Virtual Reality, Interoperability, Behaviour Integration

1 Introduction and Motivation

VR technologies are steadily gaining foothold in the shipbuilding sector. The advanced performance of modern hardware permits the application of VR in this sector where large models have to be processed. Furthermore, the scepticism of key persons towards VR is slowly replaced by the perception that VR technology can bring major benefits during the whole life cycle of the ship. Various fields for the usage of VR in the maritime industry are discussed in (Nedeß et al, 2005) and (von Lukas, 2006).

Apart from the hardware, it is the VR software which determines the range of scenarios that can be covered by VR solutions. Currently available VR software significantly differ in their available features, ranging from feature-rich and cost-intensive software packages to simple and relatively cheap ones. Since most shipyards operate under high cost pressure, some shy away from bearing the high investments for a feature-rich solution and rather opt for a low cost one.

Moreover, the requirements for VR solutions in shipbuilding significantly differ from those in other sectors like the automotive or aerospace industry. European shipbuilding is dominated by the production of one (or a few) ships of a type instead of producing large series like in automotive industry. Therefore, an extensive overhead for setting up a VR session seems not acceptable for the majority of the actors. For the most part, there is no established process to provide a virtual model at any stage of the design process in the naval architecture. Therefore, much more effort is required to provide an integration of VR sessions into shipbuilding processes. Also, in-

stead of putting the emphasis on high-end rendering, functionality coping with huge and possibly incomplete models and shipyard specific manufacturing tools like cranes and tackles is of major importance. This implies the need for customisation and the development of additional shipbuilding-specific modules – even for feature-rich VR solutions.

This leads to a situation where some shipyards have installed feature-rich VR solutions while others aim at low cost solutions. Within such a heterogeneous system landscape distributed design reviews between partners are hardly possible. Furthermore, each individual solution requires certain adaptations to match the specific needs of the shipbuilding sector. Due to the lack of one leading VR system and the relatively small market for shipbuilding-specific functionality it is a economic necessity to provide the flexibility to reuse such adaptations across different VR systems.

To mitigate those problems, we have developed a platform which supports writing software modules that are usable together with different VR software. The platform is based on available VR software and provides a standard interface for the interaction with this software. Thus, the features of the respective VR software can be used while additional functionality is developed for the platform.

2 State of the Art

Today, there are many toolkits available to develop VR applications, e.g. (Tramberend, 1999; Bierbaum et al, 2001). They provide an abstraction from the underlying display hardware and the interaction devices. Some of the toolkits provide built-in support for distributing and sharing scene information supporting the development of collaborative VR applications. The toolkits themselves provide only the basis for a VR application and require custom applications to be developed on top of it.

The VRML97 standard and its successor X3D (Web 3D Consortium, 2008a) was defined to allow for the description of virtual worlds. In addition to the description of the static part of a scene it provides a standardised way to integrate dynamic aspects into the world by providing a scripting interface as well as an interface for external access. The X3D standard has proven to ensure interoperability among different X3D conform viewers. Behr et al describe how an X3D viewer can be utilised for immersive environments (Behr et al, 2004). However, many vendors of X3D viewers have chosen to provide custom extensions, so that they can offer features not addressed by the standard, and many commercially available VR solutions are not based on the X3D standard at all.

For the purpose of using VR as a tool in industrial production processes, there are various commercially full featured VR applications available provided with extended functionality. Those applications feature a rich set of functions, i.e. collision detection, human models, simple physics and support for collaborative work. They usually provide their own proprietary way to extend the functionality through some vendor-specific SDK.

3 The USE-VR Platform

The development of the USE-VR Platform is driven by various projects of ZGDV with the maritime industry[1]. It is designed to provide an integrated framework which facilitates the simple realisation of custom VR scenarios with minimal effort. The framework provides the architecture to easily set up scenarios for existing VR software. This framework is still under development and will cover three major areas:

1. Streamlining the pre- and post-set-up steps, like data conversion, launching of the actual VR session and processing the results,

1 The work presented here was funded by the German Federal Ministry of Economics and Technology (BMWi) in context of the research project USE-VR.

2. Defining the logic of a given VR scenario, i.e. defining scenario specific behaviour and its integration into the scene, and
3. Developing custom modules for realising additional functionality which operate independently of the VR software being used.

The first point addresses the often complicated process of preparing and evaluating a VR session. It flexibly allows to automate the – sometimes complex – data conversion chain and to connect it with the execution of the VR scenario as well as the modules for the evaluation of the results of the session. The second point aims at the easy realisation of custom VR scenarios by composing building blocks to a hand tailored VR application. The last point refers to the development of interoperable VR modules which can be used together with different VR software. This makes it possible to use the modules together with different VR systems and, therefore, also to connect different systems with each other. We have realised this point by the VR execution platform which offers an abstraction from the concrete VR software and enables custom modules to communicate with the VR software in a standardised way.

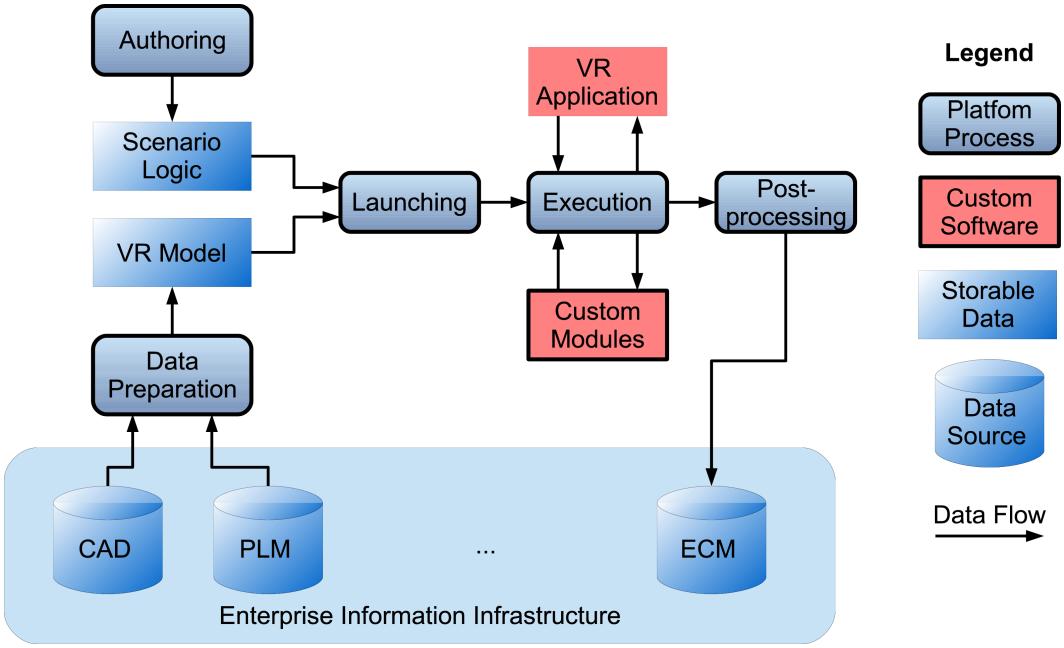

Figure 1. Steps for Executing a VR Scenario

Figure 1 illustrates the typical steps necessary for executing a VR scenario. The parts that will be supported by the USE-VR Platform are highlighted with a bold line. Until now, our main efforts were directed to the VR Execution Platform. In this paper, we describe the VR Execution Platform and show how a custom VR application is built on top of it. Furthermore, we detail on how we achieve interoperability among different VR software systems. Finally, we demonstrate how a VR scenario form the shipbuilding industry can be realised using the platform.

3.1 The USE-VR Execution Platform

The *USE-VR Execution Platform* is not meant to be just another VR software responsible for displaying and connecting interaction devices. Instead, it enables custom modules to communicate with the VR software in a standardised way and, thus, allows these modules to be used in connection with any available VR software.

The principle how the *Execution Platform* achieves platform independence is straightforward. It is based on modules which are responsible for the propagation of information from the

VR software and which allow the modification of information in the VR software. On the one side, *Publishers* are responsible for publishing information from the VR software, e.g. the information that the viewpoint has changed, a collision occurred, or an object was selected. On the other side, *SceneActions* allow to modify information of the VR software or the displayed scene, e.g. moving the camera or an object. *Publishers* and *SceneActions* can be accessed in a standardised way – and, thus, platform independence is achieved.

Custom modules for the *Execution Platform* can be realised as so called *Functions*. *Functions* can access *Publishers* and *SceneActions* to interact with the VR software. Furthermore, they may implement any additional logic, e.g. the connection to an external simulation system like HLA (Dahmann et al, 1997).

For interconnecting different modules (*Functions*, *Publishers* and *SceneActions*), we have chosen the same approach taken by the well known X3D standard (Web 3D Consortium, 2008b). It is based on routes and messages (referred to as events by the X3D standard). Each *Publisher* provides out-slots which are the source of messages. A *Publisher* generates a message when a change of the published property occurs in the VR software. *SceneActions* provide in-slots, which can receive messages. When a *SceneAction* receives a message, it updates a property of the VR software accordingly. *Functions* may have both in- and out-slots. In- and out-slots of the different modules can be connected through routes. An in-slot can be connected to multiple out-slots, and vice versa.

USE-VR Launcher

To realise a concrete VR scenario, a number of *Publishers*, *SceneActions* and *Functions* must be used. Each such module can be instantiated several times. For the simple scenario of synchronising the viewpoint of multiple VR software systems, we need:

- an instance of a `ViewpointPublisher`, publishing changes of the viewpoint from the VR software,
- an instance of a `ViewpointNetworkSynchroniser` *Function* communicating changes of the viewpoint over the network, and
- an instance of a `ViewpointSceneAction` which updates the viewpoint of the VR software when the network synchroniser received new viewpoint information.

The *Publisher* provides an out-slot named *pose* of the type "Pose". A pose contains position as well as orientation information. The network synchroniser provides an in- and out-slot, both named *pose,* and the *SceneAction* provides an in-slot *pose*. Additionally to creating the instances, the slots of the instances must be connected. Figure 2 illustrates the set-up of the modules for the viewpoint synchronisation scenario.

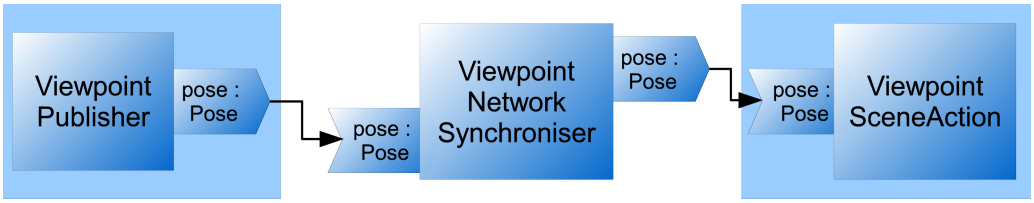

Figure 2. Modules and Connections of Viewpoint Synchronisation Scenario

A launcher file is used to define the instances of the modules required for a scenario and sets up the routes between in- and out-slots. Each instantiated module is assigned a unique instance name. Additionally, for *Publishers*/*SceneActions* the name of the target object from the scene which is published/modified may be given. This allows, e.g. to write a generic `MoveSceneAction` to modify the position of a target object and then parametrise it with the name of the object to be moved. For our simple viewpoint modification scenario this was not necessary, be-

cause the `ViewpointPublisher` and `ViewpointSceneAction` always operate on the active camera. The launcher file for the viewpoint synchronisation scenario is given in Figure 3.

```
<ulf>
  <instances>
    <publisher instanceName="viewpointPublisher"
        extension="ViewpointPublisher"/>
    <sceneAction instanceName="viewpointSceneAction"
        extension="ViewpointSceneAction"/>
    <function instanceName="viewpointSynchroniser"
        extension="ViewpointNetworkSynchroniser"/>
  </instances>
  <routes >
    <route fromNode="viewpointPublisher" fromSlot="pose"
        toNode="viewpointSynchroniser" toSlot="pose"/>
    <route fromNode="viewpointSynchroniser" fromSlot="pose"
        toNode="viewpointSceneAction" toSlot="pose"/>
  </routes>
</ulf>
```

Figure 3. Launcher File for Viewpoint Synchronisation

For all module types it is possible to provide additional attributes upon instantiation. The attributes consist of key value pairs. Thereby, modules can be parametrised with custom parameters.

Adapting to the Target VR Software

As discussed before, the *SceneActions* and *Publishers* provide the "glue" between the VR software and the Execution Platform. While the `ViewpointNetworkSynchroniser`, and, thus, the main part of the logic in our viewpoint synchronisation scenario, is completely independent of the VR software, a specific implementation of the *Publishers* and *SceneActions* must be available for each target VR software. In the launcher file abstract *Publishers* and *SceneActions* are referenced, which need to be replaced with the specific implementations for the target VR software. To support this, it is possible to define a mapping of the abstract modules with regard to their specific implementations within the Execution Platform. The platform evaluates this information and, depending on the respective VR software, instantiates the correct module. This approach follows the concept of the Model Driven Architecture (Miller and Mukerji, 2003) where we find a platform independent model (PIM). From this, a platform specific model (PSM) can be derived for any target platform. For the support of a specific scenario on a new VR software this means that it is merely necessary to provide specific implementations of the used *Publishers* and *SceneActions* and to provide the mapping information.

We have chosen to provide such modules for one commercially available feature-rich VR software, IC:IDO (IC:IDO, 2008), and for the publicly available X3D based browser of the instant**reality** distribution, a VR and Mixed Reality framework developed by ZGDV and Fraunhofer IGD in close co-operation with the industry (Behr et al, 2004).

The IC:IDO software package provides a C++ SDK allowing to access the camera information. Since the USE-VR Execution Platform is written in Java we implemented an interprocess communication (IPC) allowing to transfer messages between the platform and the IC:IDO-side code. On the platform side we implemented a *Publisher*, which receives information about changes of the camera from the IC:IDO side and publishes those on its out-slots. Additionally we implemented a *SceneAction*, that forwards camera information to the IC:IDO side whenever its in-slot receives new information. On the part of IC:IDO, their SDK was used to adapt to camera changes and to modify the camera.

Concerning the instant**reality** X3D browser, we have used the standard VRML97-external authoring interface (EAI) to communicate with the browser (whose scripting interface still is based on the VRML97 standard, though it supports most of the X3D standard). The use of the standard interface ensures the compatibility of our developed modules with every VRML97 conform browser. However, some additional considerations need to be done to create *Publishers* and *SceneActions* for VRML browsers. Many properties of the VRML scene can only be accessed through additional nodes, e.g. the current camera information is not directly accessible and to modify the position of an object, it must be located within a transform node. Therefore, the mechanism for implementing *Publisher* and *SceneAction* for VRML-browsers supports the integration of new nodes into the VRML scene. For example, upon instantiation a `ViewpointPublisher` would add a `ProximitySensor` to the VRML scene giving access to the current camera position and orientation.

The integration of extra nodes is performed in a pre-process on the VRML model before the VR scenario is executed. This is done by a mechanism we have called "weaving". The weaver approach for the automatic enrichment of VRML scenes was first introduced in (Mesing and Hellmich, 2006). Each *Publisher*/*SceneAction* may define a custom piece of VRML code which is woven into the scene once for each instance of the module. The weaving approach renders possible to add a node within or around another one, i.e. allowing to wrap a node in an additional transform node. As a parameter, the VRML code may use the name defined for the instance. This mechanism helps to avoid name clashes when an object is instantiated several times by parametrising names of VRML nodes with the instance name.

As already described in our example of viewpoint synchronisation, we define a *Publisher* which adds a VRML ProximitySensor node to the scene. This enables us to track the camera movement. The *Publisher* connects to the `ProximitySensor` and propagates changes to its out-slot. The *SceneAction*, responsible for the viewpoint modification, adds a VRML Viewpoint node to the scene and modifies its position when it receives a message on its in-slot.

Implementation

The *Execution Platform* is realised in Java under use of the Eclipse RCP (McAffer and Lemieux, 2005), a framework which supports writing modular applications based on plug-ins. The RCP facilitates the definition of custom plug-in types, i.e. so called *extension points*. The concept of *Publishers*, *SceneActions,* and *Functions* is realised as such an extension point. An *extension* realises one extension point and can be thought as an instance of the extension point. For example a `ViewpointPublisher` would be an extension of the *Publisher* extension point. Figure 4 illustrates the hierarchy of the different concepts.

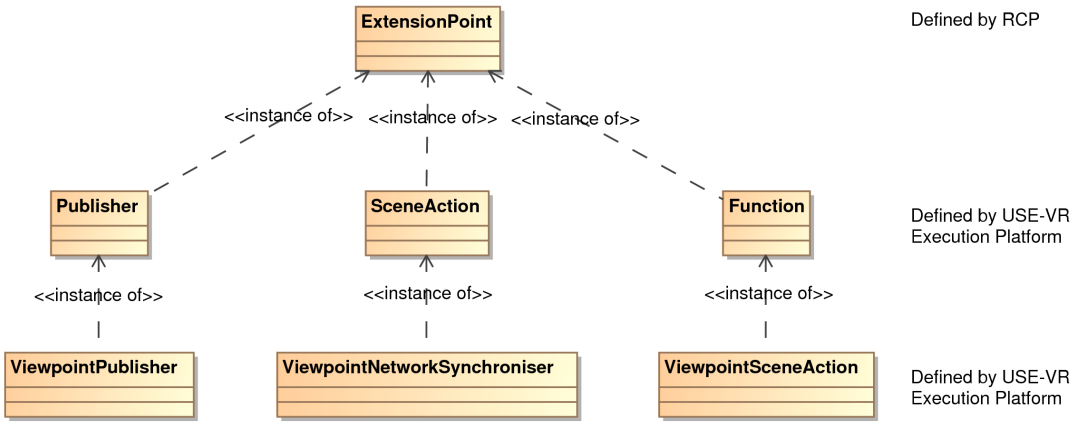

Figure 4. Layer Model for USE-VR Execution Platform

This architecture makes the platform easily extendible. New *Publishers* and *SceneActions* can be defined to access further features of the VR software and can be used in custom scenarios. However, all the *Publishers* and *SceneActions* used to realise a specific scenario must be implemented for every VR software the scenario shall be operated on. The more high-level features of a VR software are made accessible through *Publishers* and *SceneActions*, the more difficult it will become to implement those modules for other VR software. The tradeoff happens between interoperability and the use of specific functionality.

4 Realising an Industry Scenario

In this section, we discuss how a scenario, provided by the shipbuilding industry, can be realised under use of the USE-VR Platform. In the scenario, VR shall be used to determine the optimum time for the installation of a large intermediate shaft in a ship. An intermediate shaft is approximately 10m long and has a weight of several tons. It connects the drive shaft with the main engine. Since space becomes increasingly narrow during the course of the shipbuilding process, a VR model shall be used to determine until when the shaft can be installed (almost) without collision. For a simple realisation of this scenario, it must be possible to move the shaft through the scene, to detect and log collisions, and to record and replay the movement of the shaft. The shaft shall only be movable while it is selected by the user. Figure 5 shows a screenshot of the full industry scenario including a crane runway along which the shaft is moved.

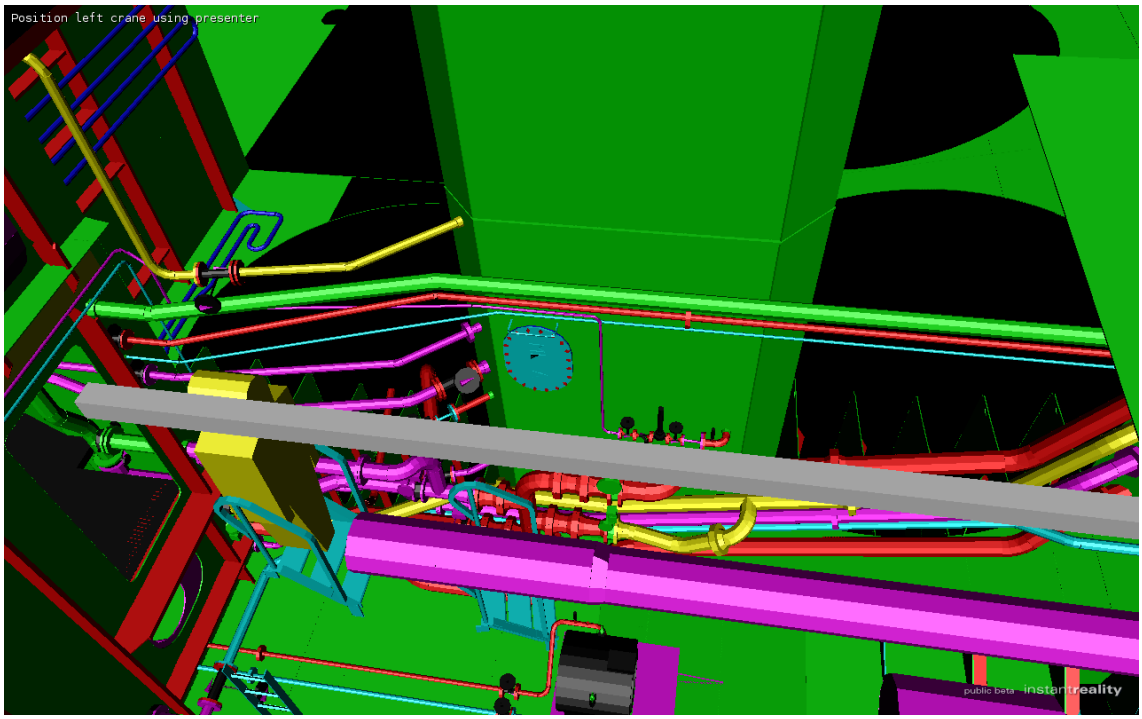

Figure 5. Moving an Intermediate Shaft within the Virtual Environment

Figure 6 illustrates the modules required to realise the scenario. We need one *Publisher* that publishes the name of the selected object whenever an object was selected in the VR software, another one that publishes input information for the position and orientation (e.g. generated by a space mouse or a tracking device), a third one which informs the user about pressing a button to start the replay and, finally, a fourth *Publisher* which informs about collisions that were detected. We also need a *SceneAction* which modifies the position of the intermediate shaft. It should be noted that we rely on some high-level functionality of the VR software here, e.g. the capability to control input devices, to perform an object selection, and to detect object collisions. By

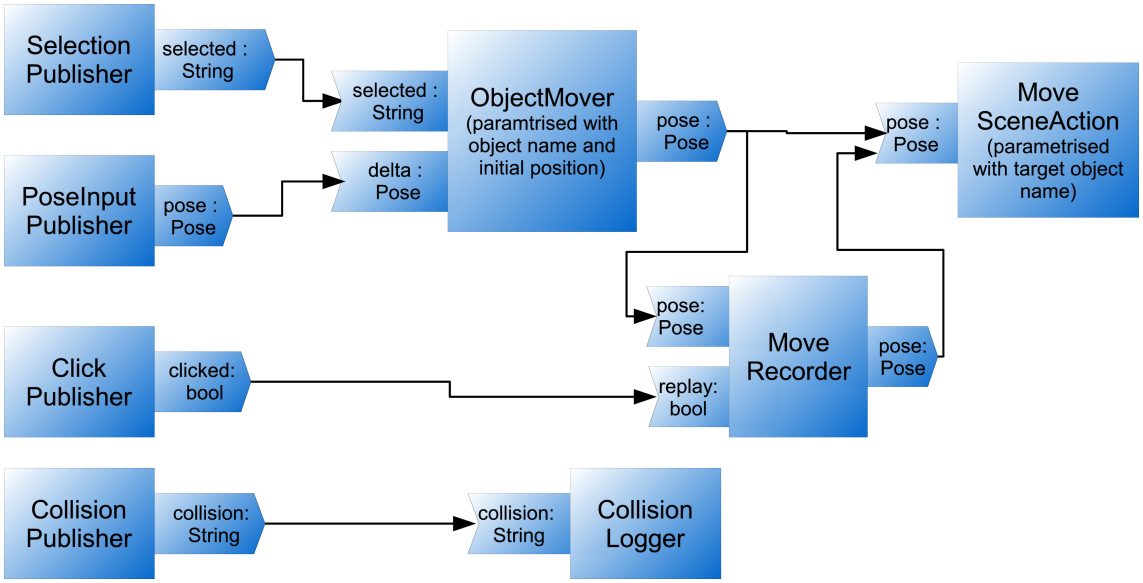

Figure 6. Module Infrastructure for Realising the Industry Scenario

this, we utilise the power of the underlying VR software instead of being forced to re-implement this functionality by ourselves. Only in case the VR software does not provide the required functionality, we have to implement it within the *Publishers*.

The main part of the custom logic is realised by the platform-independent *Functions* depicted in the middle of Figure 6. A mover receives the information about the selected object and the input from the input publisher. If the shaft is selected and a position/orientation delta is received, the mover calculates and publishes the new pose information to its out-slot. Apart from the *SceneAction* which updates the position of the shaft, a record and replay module is connected to the mover. It records the position of the object and, when receiving a replay event, replays the positions recorded before. A collision logger stores all the reported collisions in a file.

For most of the modules it is obvious how an implementation could look like. However, for some of them some further considerations need to be applied. The task of the `ObjectMover` is to calculate the new position of the object whenever a movement request (delta) was received. To achieve this, the mover must keep track of the last position of the object. Since our *Publisher*/*SceneAction* approach does not offer any way to directly access scene information, it requires the mover to store the position/orientation (pose information) itself. This also means that the pose information stored by the mover must always be synchronised with that of the object. To set up the initial pose information of the mover, the shafts position is provided as an attribute when instantiating the `ObjectMover`.

As long as only the mover modifies the objects pose information the synchronisation of the information is no problem, but as soon as another module may modify the pose information we need an extra in-slot within the mover for setting the pose information of the object as well as an additional *Publisher* which propagates changes in the objects pose information. In our scenario, the `MoveRecorder` is also able to modify the object's position, but, in order to simplify matters, we have chosen to allow the pose information to get out of sync.

An alternative approach would have been to provide an abstraction layer which allows for the direct access to scene information. However, we believe that this would have made the task of adding support for a new VR software unnecessarily difficult and we are convinced that the very thin abstraction layer we have defined is more likely to be adopted.

Upon instantiation, the `ObjectMover` module is also parametrised with the name of the object to be moved. It checks if the name of the selected object matches the one to be moved. It

updates the position only when the correct object is selected. The `MoveSceneAction` is also parametrised with the name of the object to be moved and, thus, can be reused in other scenarios.

5 Limitations

One drawback of our approach is that it does not work well in case extensive access to state information of the scene is needed. This is because it defines no way for direct access to the scene graph. Instead, all state information required must be duplicated within the execution platform. This would make it very difficult to implement, for example, a collision detection within the platform. Instead, our approach relies on exactly these high-level operations to be implemented by the VR software. The only way to have full access to the scene graph would be to additionally load the scene information from the source file into the platform. However, this would require additional memory capacity and would raise synchronisation issues if objects can be modified.

Currently, our approach does not provide support for creating geometry on the fly. This may be useful to unitise a scene so as to enable a progressive loading of scene objects such as parts or VR tools. A tedious way to achieve this would be to provide SceneActions for creating special geometric objects, e.g. a *SphereCreatorSceneAction*. A more elegant option would be to define a creator *SceneAction* which receives the geometry to be created on a String in-slot. However, this approach would require to define a common geometry description format. A good candidate for such a format would be X3D, since almost every VR software supports this format for import and thus realising such a *SceneAction* would be relatively simple.

6 Conclusion and Future Work

In this paper, we have outlined the USE-VR Platform, a framework to support the realisation and execution of custom VR scenarios. The platform addresses the authoring and execution of custom VR scenarios including the set-up of the data conversion chain.

The approach of the platform is to utilise existing technology and to allow the integration of the different tools available. We have detailed on the VR execution platform which provides an interface for the development of additional modules. Modules developed for the platform work independently of the VR software that is used. We have discussed how the VR execution platform can be facilitated to realise an industry scenario from the shipbuilding industry and have also indicated the limitations of our approach.

In the near future, we will apply our approach to similar scenarios in order to get additional feedback and we will continue to complete the platform.

A set up tool for the automatic data preparation steps, as mentioned in paragraph 3, is not yet implemented, but it is planned for future work. Furthermore, an easy-to-use authoring component is planned via the integration of an authoring environment as described in (Göbel et al, 2007) and (Balet, 2007).

7 References

7.1 Books

McAffer, J. and Lemieux, J. (2005): *eclipse Rich Client Platform.* Addison-Wesley, Upper Saddle River.

Miller, J. and Mukerji, J. (2003) *MDA Guide Version 1.0.1.* Object Management Group.

7.2 Conference Proceedings

Balet, O. (2007): INSCAPE: An Authoring Platform for Interactive Storytelling. In P*roceeding of the 4th International Conference on Virtual Storytelling*, Springer, LNCS 4871, 176-177.

Behr, J.; Dähne, P. and Roth, M. (2004): Utilizing X3D for Immersive Environments. In *Proceedings of the ninth international conference on 3D Web technology*, ACM Press, 71-78.

Bierbaum, A.; Just, C.; Hartling, P.; Meinert, K.; Baker, A. and Cruz-Neira, C. (2001): VR Juggler: A Virtual Platform. In *Proceedings of the Virtual Reality 2001 Conference,* IEEE Computer Society, 89-96.

Dahmann, J. S.; Fujimoto, R. M. and Weatherly, R. M. (1997): The Department of Defense High Level Architecture. In *Proceedings of the 29th conference on Winter simulation*, ACM Press, 1997, 142-149.

Göbel, S.; Konrad, R.; Salvatore, L.; Sauer, S. and Osswald, K. (2007): U-CREATE: Authoring Tool for the Creation of Interactive Storytelling Based Edutainment Applications. In *Eva 2007 Florence Proceedings*, 53-58.

von Lukas, U. (2006): Virtual and Augmented Reality in the Maritime Industry. In *Proceedings of New Trends in Collaborative Product Design*, Publishing House of Poznan University of Technology, 193-201.

Mesing, B. and Hellmich, C. (2006): Using Aspect Oriented Methods to Add Behaviour to X3D Documents. In *Proceedings of the Eleventh International Conference on 3D Web Technology*, ACM, 97-107.

Nedeß, C.; Friedewald, A. and Kerse, N (2005): Increasing Customer's Benefit using Virtual Reality (VR) - Technologies in the Design of Ship Outfitting. In *Proceedings of the 4th International EuroConference on Computer Applications and Information Technology in the Maritime Industries*, 113-122.

Tramberend, H. (1999): AVOCADO – A distributed Virtual Environment Framework. In *Proceedings of IEEE Virtual Reality,* IEEE. 14-21.

7.3 Internet Sources

IC:IDO (2008) Product information: http://www.icido.com. Last visited: 2008-07-28.

Web 3D Consortium (2008a): X3D specification, http://www.web3d.org/x3d/specifications/. Last visited: 2008-07-28.

Web 3D Consortium (2008b): X3D specification, Section 4.4.8 Event model, http://www.web3d.org/x3d/specifications/ISO-IEC-FDIS-19775-1.2-X3D-AbstractSpecification/index.html. Last visited: 2008-07-28.