An Approach for Simplified Subsystem Replacement and Reconfiguration in Multimodal VR, AR and Other Simulation Frameworks

Thomas D. Lepich, Reinhard Möller, Christian John, Thomas Pursche
Faculty of Electrical, Information and Media Engineering,
Bergische Universität Wuppertal, Germany.
Lepich@uni-wuppertal.de

***Abstract*— Nowadays, modern software for the development of augmented and virtual reality applications is designed with the aim to simplify its usability in order to provide services to a wider user base. In this context, our paper presents a novel approach to make the replacement and reconfiguration of a simulation framework's subsystems possible, without being bound by the restrictions, current plug-in strategies incur, or the need of manipulating its source code. Code manipulation requires a deep understanding of software engineering and the framework's software design, including all dependencies among the subsystems. For this purpose, common simulation systems were examined and their restrictions identified. Solutions of different problems in this context were elaborated and are discussed in this paper.**

***Index Terms*— Augmented reality; Virtual reality; Simulation framework; Metaprotocol.**

I. INTRODUCTION

Since the first appearance of so-called *game engines* in the mid nineties [1], not only the capability of the technology is being advanced. Today, modern software for the development of augmented and virtual reality applications is designed with the aim to simplify its usability in order to provide its services to a wider user base. But the extension and reconfiguration of such framework's subsystems is still restricted to the use of mostly naive plug-in architectures with all their limitations, or to the manipulation of the software's source code. The latter is a subject only for highly trained software engineers with profound knowledge of the software's architecture and implementation, especially of the dependencies among all its subsystems.

Figure 1 shows fundamental parts of a simulation application and the distribution of responsibilities in the development process of this kind of software. Simulation software usually consists of a reusable part called *engine*, or in case of games also *game engine* [1, 2]. The engine is normally developed by separate teams of highly specialized software engineers and then licensed to simulation developers like game development studios or individuals. Moreover the application consists of a set of data and some application specific code, with the data being the major part. Further, current engines allow manipulation of this data directly from the engine's user interface. The application specific code itself is reduced to a minimum, often being no more than some sort of short scripts. This approach fosters the simplified use of such frameworks. It is called data-driven architecture [1].

One significant drawback that still remains is the missing ability to reconfigure the engine itself by adding, removing and exchanging its subsystems in a way the application developers know and can manage. With this in mind, we present a possibility to reconfigure a framework for augmented and virtual reality applications using a vocabulary known and mainly used by the developers of simulations, rather than by the software engineers of the framework.

In the context of our research, several simulation frameworks like Unity3D, Unreal Engine or CryENGINE [3-5] were studied to identify commonly used types and their restrictions related to reconfiguration and subsystem replacement. Related standards like OpenGL [6, 7] and other frameworks like the Open Dynamics Engine [8], OpenSceneGraph [9] and the Robot Operating System (ROS) [16] were taken into account. ROS was taken into account because of its completely decoupled subsystem integration where each subsystem is a separate process. It turned out that this constellation can be treated as a collection of different simulations, each bound by the same restrictions discussed in the following sections.

Besides this, related work in the context of modular programming was considered, like OSGi [18], Plux [19, 20] or the Eclipse Platform [17]. Those technologies make heavy use of language specific features like reflection, which isn't provided in languages like C++, which are commonly used for implementation of simulation frameworks. PAL and OPAL [13] use a simple adapter-based concept. An example is given in the next section.

II. ARCHITECTURE OF A TYPICAL SIMULATION ENVIRONMENT AND THE ACCESS TO ITS SUBSYSTEMS

A simulation environment or game engine consists of a quite large number of subsystems. Not all of these subsystems can be adjusted or completely exchanged by the user of the environment. Figure 2 shows typical layers of a simulation environment, each containing a set of subsystems.

A. Runtime Layer

The runtime layer contains software components designated for the playback of the simulation. Most modern simulation environments are data-driven [1]. This means that a simulation is not being defined by the manipulation of code, instead it is defined by a set of data.

Data is in this context every type of media, like images, audio files or animations and others, but also short scripts that are used to describe the simulation logic. This data is then used by interpreter software like different players or the simulation environment's integrated development environment (IDE). Consider Unity3D [3] as a good example for this approach.

Generally, this layer does not need profound access and reconfiguration by the user, except for some minor adjustments, like amongst others the selection of rendering quality. The interpreter software is normally exchanged as a whole part. Often interpreters are capable of using only actually needed subsystems in order to maintain a smaller size of the resulting simulation application. Figure 3 shows this in a simplified diagram.

B. Composition Layer

Direct access is given to the composition layer of the simulation framework. It contains all the necessary types needed to define the entire structure and the behavior of a simulation. These types are generalized versions of the fundamental types found in lower subsystems, or sometimes combined types providing frequently used functionality [3, 4]. They are used to construct simulation objects, to define their functionality and to set up connections and references among the simulation entities. This is normally done in editors with graphical user interfaces instead of being coded by using complex programming languages.

Popular simulation frameworks like Unreal Engine and Unity3D utilize hierarchical data structures for the definition of simulation scenarios, like `AActor` or `GameObject`. Functionality is then added by the use of composition. In those cases functional components are added to the hierarchical structure like `AActorComponent` or Component. Those are the abstract interfaces to the underlying subsystems of the simulation framework.

Especially in the case of `AActor` and `AActorComponent` this concept can be bypassed by the implementation of subclasses. In the first place this seems to be a simplification in terms of implementing a simulation application but it makes an automated evaluation of subsystems along with their communication channels and their purpose nearly impossible.

It is also common practice to define completely specialized data structures for the use in special purpose simulation frameworks, like the ones we presented earlier [10, 11]. See also [12] for further examples. Such concepts are working well in special cases but are not suitable for a general simulation environment.

The vocabulary used in this layer, the utilization of hierarchical data structures for the definition of the simulation layout and composition for the definition of the functionality of simulation objects, works very well in a general purpose simulation framework, and thus is being used in our approach. Besides this it is not only well known from popular environments like Unity3D and Unreal Engine but also from other concepts like file systems with folders and files.

As a consequence this vocabulary needs to be used for the underlying subsystems layer, in order to allow its reconfiguration and manipulation in the same way as with types in the composition layer.

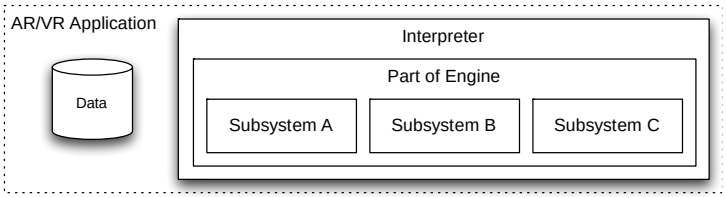

Figure 3: Interpreter using only needed subsystems of the engine and the data defining the simulation

C. Subsystems Layer

In current frameworks, this layer is in most parts not editable by the user. In many cases it is possible to adjust some values but the subsystems are not a subject to be replaced or included as completely new to the entire system. This layer contains software modules like physics engines, renderers or frameworks for the calculation of artificial intelligence. All those subsystems provide types, which are utilized by the composition layer, e.g. rigid bodies, colliders or renderables. Many other types are not propagated to the upper layers; instead they are used for internal calculations only. Figure 4 shows an example of such a constellation.

D. Foundation Layer

The responsibility of this layer is to implement the most fundamental management of the upper layers, namely the loading and unloading of the modules in the subsystems layer. This layer does not need any reconfiguration by the user.

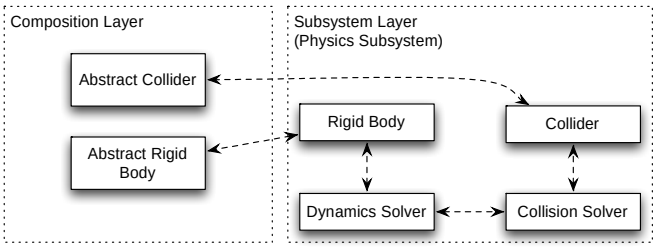

Figure 4: Propagation of different subsystem entities to the composition layer

E. Subsystem Dependencies

Dependencies among subsystems are quite complex. A standardized way for an automated evaluation and description of the communication does not exist. Commonly, subsystems of software are connected directly in the source code or they are implemented in the form of plug-in modules, mostly dynamic libraries, sometimes just scripts using already existing functionality. The implementation of plug-in modules is generally done by coding against interfaces or by implementing subclasses of existing types. Access to

subsystems can be achieved only through interfaces on source code basis. The main disadvantage here is that this approach does not allow automatic evaluation of the communication and the conversion of the results into any human readable form. Also, undesirable direct dependencies to particular subsystems may occur if no adapters are used instead of direct access to types of subsystems.

F. Audio Subsystem Replacement Example

In this example an audio subsystem based on simple OpenAL [14] is going to be replaced with an audio-raycaster like RAYAV [15]. See figure 5. Both subsystems generate the same kind of output but the latter one has additional requirements regarding its input. The raycaster uses world geometry to calculate the resulting audio signal. In a typical simulation world geometry is already present, so that this dependency may be solved without further problems. What is missing is a material description for that geometry, defining the characteristics of the reflections of the audio signal on its surface. In the case of this subsystem replacement a simulation environment should be able to report the missing dependency for each object in the simulation. The proposed approach makes this possible.

G. Physics Subsystem Replacement Example

During simulation development it may become necessary to exchange the physics subsystem. The reasons for this can be a change of requirements on quality, efficiency or functionality. For this reason software frameworks like PAL and OPAL were developed [13]. This kind of software works as an adapter providing a generalized set of functionality. The drawbacks are that these systems are restricted to physics engines only, and that they are reducing the functional range of the implemented subsystems. Further they are still not capable of any automated evaluation of connections or dependencies.

III. RESULTING FRAMEWORK

Our proposed framework defines four fundamental types to be used to implement a simulation environment, System, Scenario, Node and Component.

One of the main types of the proposed framework is System. It is used for the lowest level of resource management, mainly for the allocation and storage of further subsystems. Besides this, its main area of responsibility is the management of Scenario instances, as stated in fig. 6. This type belongs to the lowest layer, the foundation layer.

The type Scenario is also contained in the foundation layer. Its area of responsibility is to maintain a single simulation per instance, each composed of Node hierarchies. See figure 6. This is based on similar concepts like the one used by Unity3D.

Instances of the type Node are responsible for the logical composition of a simulation. Every entity contained in a simulation must be defined using a hierarchy of this type. This concept is well known and widely used so it fits our requirements. It can be found e.g. in a file system in the form of folders. Besides this, in the context of computer graphics, it is used in most scene graphs for the purpose of hierarchical spatial transformations, mostly as specialized and rigid implementations. This concept is also used in other simulation environments, like the `AActor` type in Unreal Engine, `GameObject` in Unity3D or even in other form often as inheritance hierarchies like the types `GameObject`, `MovableObject`, or the type `RenderableObject` and others [1]. Node is integrated into the composition layer.

The type Component is used for the encapsulation of data and services within a simulation. It is used being included into any instance of Node. See figure 7. Similar to Node, Component is defined as a type belonging to the composition layer of the framework.

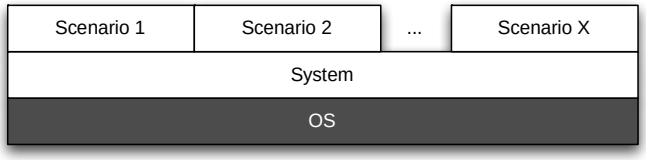

Figure 6: The operating system, system and scenario layers

Subsystem Implementation

The main types in the concept presented so far omit the subsystems layer. But this approach is capable of being used for the implementation of subsystems of a simulation framework exactly in the same way as the simulation itself is being defined within the composition layer of the engine. It is possible to implement all the simulation framework's subsystems using the main data types presented in the previous section. The only difference is that the subsystems now have to be treated as ordinary simulation objects, namely nodes and components. Thus the subsystem layer of the simulation framework can be transferred into, or combined with the composition layer. Parts of subsystems like the examples in figure 4, e.g. the dynamics and collision solver, can be implemented using the Component type.

IV. COMMUNICATION

An important criterion is the communication between components. Especially the possibility must be implemented to automatically interpret and examine component interfaces and dependencies among each other. For this purpose it is necessary to describe the communication and some of its details. This is done with the use of interfaces and protocols.

A. Interfaces and Interface Types

Conceptually, we define an interface as a connection between a single output and a single input of two components, as shown in figure 8. At the current stage of our work we defined a set of different interface types, each working differently in terms of its implementation. Specifically this means, how the interface delivers the data from the source to its destination.

B. Protocol

Protocols describe the communication between two or more components, as shown by figure 8, both in a formal and informal way. The informal part is necessary for human

readability and is currently recommended but optional. The formal description is necessary, and it is done by the definition of every single connection between the inputs and outputs of components. Such a definition needs to contain information about the direction of the connection, the type of the data transfer, the type of data to be transported and its purpose.

Every connection needs to describe its direction by identifying its data source and the target. For this purpose, connections identify a server and a client.

The connection type describes how the data is communicated between the server and the client. The transfer is possible in different ways like e.g. the provision of function pointers, the use of shared memory or other message-based methods.

The Protocol must also describe the type of data being transferred. In this context it is still possible to define completely proprietary connections e.g. by the simple use of shared memory. In this case it is only necessary to exchange a single reference to the memory. The communicating parties can then use this memory in any way. But, a purpose must be specified additionally in order to make it possible to identify the connection among others.

The signature of a connection alone is not sufficient for its definition. A purpose is also required. Otherwise the communicated data cannot carry any useful information. The problem is, if we only describe how the data is structured and transferred, we still don't know what exactly will be the calculated result, or the result's meaning. As an example, consider a simple communication protocol between two participating components. The first sends two numbers to the second one, which then sends another number back as a result. This protocol could describe an addition, subtraction or another kind of calculation.

C. Metaprotocol

In order to provide the possibility for an automated evaluation of all protocols and connections among components, protocols need to be formally described using a predefined set of rules. To ensure this, we use a metaprotocol, which provides a set of interface types a protocol may use. Fig. 9 shows a diagram describing the general structure of the metaprotocol.

V. Conclusion

We presented a new approach to simplify the reconfiguration of simulation frameworks and the replacement of their subsystems without forcing users to know the internal workings and dependencies of all contained subsystems. To achieve this, common simulation frameworks were studied and their restrictions related to the problem identified. Based on those restrictions a solution has been elaborated and discussed. The solution is based on two steps. First step is the combination of the composition layer and the subsystems layer by the definition of necessary types and system architecture. The other step is the definition of the communication channels, which can be automatically evaluated by the simulation system. The resulting framework is currently under development. The implementation is done using C++. As a consequence we have to implement customized reflection mechanics, which can be implemented on the basis of the proposed structure of our framework.

References

[1] M. Gregory, J., "Game Engine Architecture," *1st edn. A K Peters/CRC Press*, 2009.

[2] Zerbst, S., Düvel, O., "3D game Engine Programming," *1st edn. Thomson Course Technology*, 2004.

[3] Lavieri, E., "Getting Started with Unity 5," *1st edn. Packt Publishing*, 2015.

[4] Tavakkoli, A., "Game Development and Simulation with Unreal Technology," *1st edn. A K Peters/CRC Press*, 2015.

[5] Gundlach, S., Martin, M.K., "Mastering CryENGINE," *1st edn. Packt Publishing*, 2014.

[6] Sellers, G., Wright, R.S., Haemel, N.: OpenGL SuperBible, "Comprehensive Tutorial and Reference," *7th edn. Addison-Wesley Professional*, 2015.

[7] Ginsburg, D., Purnomo, B., Shreiner, D., Munshi, A., "OpenGL ES 3.0 Programming Guide," *2nd edn. Addison-Wesley Professional*, 2014.

[8] Smith, R., "Constraints in der Festkörperdynamik*," In: Kirmse, A. (ed.) Spieleprogrammierung Gems*, vol. 4. Carl Hanser Verlag, pp. 253–264, 2004.

[9] Wang, R., Quian, X., OpenSceneGraph 3.0: Beginner's Guide, *Packt Publishing*, 2010.

[10] Lepich, T.D., "Simulation einer Industrieanlage für die Ausbildung im Bereich der SPS-Programmierung," *In: Möller, R. (ed.): Tagungsband, Workshop Sichtsysteme - Visualisierung in der Simulationstechnik.* Shaker Verlag, pp. 123–134, 2007.

[11] Lepich, T.D., "Framework zur Untersuchung und Simulation autonomer Robotiksysteme," *In: Möller, R. (ed.): Tagungsband, 11. Workshop Sichtsysteme - Visualisierung in der Simulationstechnik*. Shaker Verlag, pp. 43–53, 2009.

[12] Hawkins, K., Astle, D., OpenGL Game Programming. *1st edn. Prima Publishing*, 2001.

[13] Boeing, A., Bräunl, T., "Evaluation of Real-time Physics Simulation Systems," *In: Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, ACM, 2007.

[14] Hiebert, G.: OpenAL 1.1 Specification and Reference, 2005.

[15] Karbowniczek, P.: Praca dyplomowa, wersja demonstracyjna silnika audio do gier rayav, 2014.

[16] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y., "ROS: An Open-source Robot Operating System," *In: ICRA Workshop on Open-source Software*, vol. 3, no. 3.2, pp. 5. 2005.

[17] Beck, K., Gamma, E.: Contributing to Eclipse. Addison-Wesley, 2003.

[18] Alliance, O., " OSGi Service Platform, Release 3," *IOS Press, Inc*, 2003.

[19] Jahn, M., Löberbauer, M., Wolfinger, R., and Mössenböck, H., " Rule-based Composition Behaviors in Dynamic Plug-in Systems," *In: Software Engineering Conference (APSEC)*, 17th Asia Pacific, pp. 80–89, 2010.

[20] Wolfinger, R., Dhungana, D., Prähofer, H., and Mössenböck, H., " A Component Plug-in Architecture for the .NET Platform," *In: Modular Programming Languages. Springer Berlin Heidelberg*, pp. 287–305, 2006.

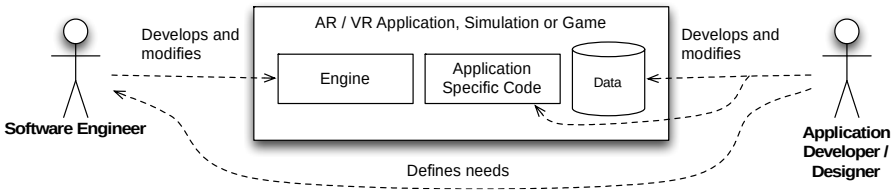

Figure 1: General interaction between developers of the simulation framework and the designers and developers of a simulation application

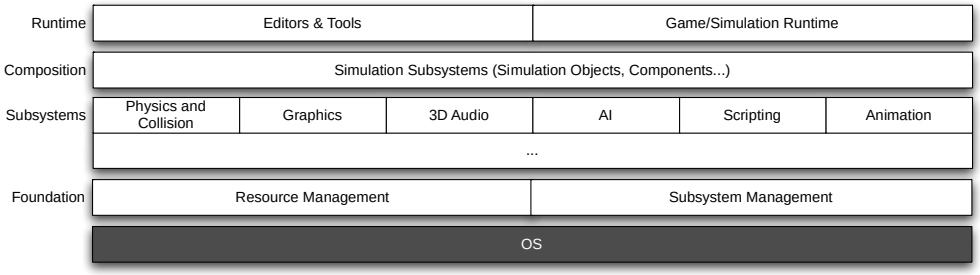

Figure 2: Simplified diagram of typical layers and subsystems of a simulation engine

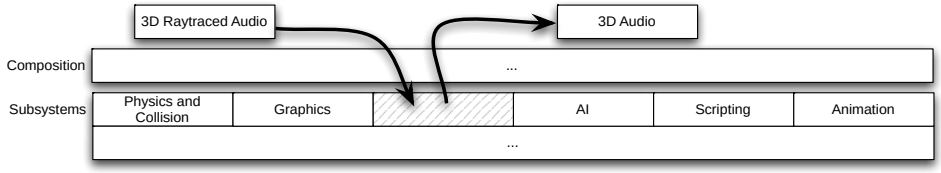

Figure 5: Subsystem exchange

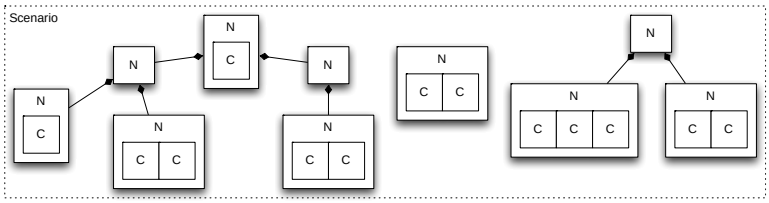

Figure 7: Scenario containing node (N) hierarchies and components (C)

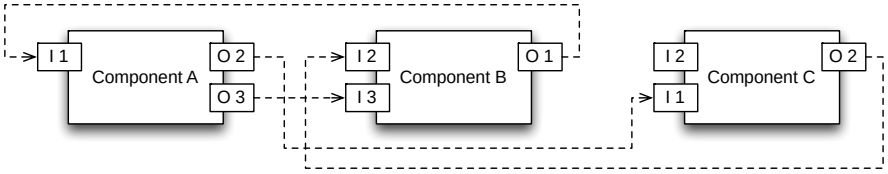

Figure 8: A protocol describing the connections between multiple components

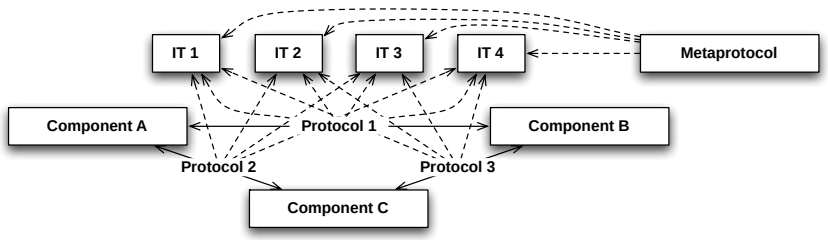

Figure 9: Metaprotocol for the definition of interface types (IT) used by protocols