Cherry-Picking RIS Functionality – Integration of Game and VR Engine Sub-Systems based on Entities and Events

Dennis Wiebusch* Ulm University
Chris Zimmerer† University of Würzburg
Marc Latoschik‡ University of Würzburg

ABSTRACT

Modern game engines provide a variety of high-end features and sub-systems which have made them increasingly interesting for AR/VR research. Here, it often is necessary to combine features from different sources. This paper presents an approach based on entity-event state decoupling and exchange. The approach targets the combination of sub-systems from different sources which simulate functionally coherent aspects of the virtual objects like physics, graphics, AI, or developer services like state editing. The approach decouples specific internal representations using a semantic description layer for identifiers, data types, and potential relations between them. We illustrate the main concepts using examples from the combination of the Unreal Engine 4, the Unity engine, and own research software and illustrate performance related aspects as a guideline for the choice of an appropriate transport layer.

Index Terms: D.2.13 [Software]: Software Engineering—Reusable Software

1 INTRODUCTION

Various software solutions support the development of Real-Time Interactive Systems (RIS) for Virtual and Augmented Reality (VR/AR). Some solutions support dedicated functions like input processing, graphics and audio rendering, or simulation of physics and Artificial Intelligence (AI). Others provide complete software frameworks which usually include a combination of beforementioned functions. For example, many of today's game engines matured to provide excellent features and state of the art simulation subsystems for many VR/AR research projects. Lately, VR also has become popular as an option for computer games. Hence, many engines directly support VR devices like head-mounted displays (HMDs) and efforts have been made to enable their use in CAVEs [14, 19, 24].

Employing game engines in VR research has been advocated for some time [17] and the undeniable benefits of these engines render them interesting for HCI and VR researchers. However, there is no one-size-fits-all, specifically in innovative research projects. The major target of game engines are computer games or game-related VR setups not comparable to research setups. For example, research projects often include extended input device integration (e.g., for full-body tracking), output rendering channels and devices (e.g., for haptic rendering), or specific simulation software driving the application's main functionality (e.g., for large data set visualization). Maybe sooner or later some of these functions will also become commercially interesting and will be integrated into the engines as a full supported feature. Until this happens, however, there always will be a requirement to combine features from different (software) sources.

*e-mail:dennis.wiebusch@uni-ulm.de
†e-mail:chris.zimmerer@uni-wuerzburg.de
‡e-mail:marc.latoschik@uni-wuerzburg.de

Today's game and VR engines have become quite complex pieces of technology which has been a major motivation for RIS research. The resources needed to develop all necessary features by one-self while still providing state-of-the-art quality are basically out-of-reach for individuals or small teams. This can be asserted for industry and research labs alike. Hence, integration of existing software into one's own target system has become a mandatory and reoccurring pattern in software engineering. This paper proposes a software integration concept for connecting different subsystems of VR and game engines based on entity-event state decoupling and exchange. The approach specifically proposes a semantic abstraction layer. In contrast to purpose-built plugins or tailored interfaces it provides a well defined interface fostering reusability and exchangeability. We illustrate the main concepts using examples from the combination of the Unreal Engine 4, the Unity engine, and own research software and illustrate performance related aspects as a guideline for the choice of an appropriate transport layer.

2 RELATED WORK

Game engines have been used in various application and research scenarios, see, e.g., [17, 22, 14, 19, 24]. At this time, the most commonly used engines are Unity [32], the Unreal Engine [9], and the CryEngine [7] in different versions.

For example, the Unreal Engine has been used in the area of interactive storytelling for some time [5], for applications involving gesture interaction [6], was applied to haptic devices [28], and used in the context of intelligent virtual environments [18]. It also was used in applications for teaching engineering and construction [13], virtual teaching [20], and smart prototyping [23]. The latter project used Unity, too, however, the authors report that they found the integration of C++ libraries to be easier with the Unreal Engine.

Unity was used in the context of serious games [26] and to explore the possibility to automatically generate VEs that are adapted to the user's physical environment [29]. Moreover, it was used to integrate cognitive models in VR applications [27] as well as for interactive storytelling involving a custom-built hardware device [11].

The indicated vast amount of application areas of game engines in VR research complicates the choice of the optimal engine. This is in line with our initial statement that reuse of code and framework is of concern when choosing an engine. In this context, methodologies for selecting engines have been proposed for serious games [25] and virtual environments in general [33].

The *m+m* (Movement + Meaning) framework [2] provides a similar approach to the one presented in this paper. However, *m+m* mainly targets real-time interactive systems focused on movement data. It supports the Unreal Engine as well as Unity as output engines and can connect to different sensors, e.g., Microsoft Kinect and Leap Motion. As opposed to the m+m framework, the proposed approach is not focused on movement data but on the utilization of game engine subsystems in general.

The *#FIVE* framework [4], although it is targeted at the development of collaborative, interactive virtual environments (and not on general interfacing of different game engines), bears resemblance with the presented work, since it integrates with the Unity Engine via a DLL file and allows to attach other modules via its API.

There are other approaches to interface game engines, like intercept-based techniques targeting rendering subsystems, as presented by [36], or like the AnyHaptics framework [28] (dedicated to the area of haptics) and the Physics Abstraction Layer [3] (dedicated to the area of physics). As opposed to these, the presented approach targets the general utilization of subsystems. It requires to implement a plugin to interface with the subsystems of the game engine in question. Since this is possible with most current game engines it is not considered a limitation. This way, it enables access to more aspects of the subsystem and–since it uses the official APIs–is easier to maintain.

Our implementation is based on an entity model and an event system, two approaches that were found useful by others [30, 21]. More specifically, it relies on the Entity-Component-System (ECS) pattern [1] as well as the actor model [12] to achieve a highly modular, extensible architecture. In addition, it features integrated semantics at a core layer, which thus is connected to the subsystems of the used game engine. This way, it provides a step towards the integration of semantics into computer games, as advocated by [31].

One of the presented examples is the PEARS framework [8], which employs the presented approach to take advantage of the physics simulation and rendering systems of the Unity Engine. Another example application utilizes the VR capabilities of the Unreal Engine [9] and is based on Bolt's famous "Put-That-There".

3 Concept

There are different vocabularies for describing elements of a Real-Time Interactive System (RIS). This becomes especially apparent in comparing the terms "physics engine" and "game engine," the first usually being a part of the second. In order to avoid misunderstandings, we specify the main terms that will be used in the remainder of this paper below. We largely adopt the terms introduced by the ECS pattern:

Entity describes an object or concept relevant to the simulation. It can be of physical, virtual, or conceptual nature. Entities are composed of components.

Component refers to a certain aspect of an entity. This can be a single data property (e.g., position) or more complex characteristic (e.g., the representation of a rigid body). Components can share data properties.

Event identifies an incident that is restricted to a single point in time during the simulation. This includes the occurrence of value updates as well as more meaningful incidents (e.g., a collision).

System (also subsystem) denotes a piece of software that is dedicated to a particular purpose (e.g., physics simulation) and usually part of an engine or implemented in a dedicated software library. Despite their distinct purpose, systems often are closely coupled to achieve higher performance. Alongside application specific code, systems receive and handle events.

Engine denominates a software framework that is used by game designers to create virtual environments. Engines are composed of systems.

3.1 Fundamental Observations

A connection to one or more subsystems of one or more game engines can either be achieved by creating plugins for the system(s) of the respective engine or by creating a network connection that is used to transfer information between the engine and the application. The first approach involves less communication overhead, whereas the networked variant allows to combine the computing power of multiple machines and different systems. This aspect will be disregarded and concerned as the *transport layer* (cf. Figure 1), since both approaches are conceptually equivalent: in any case information about the current simulation state is shared between the utilized systems and the application.

Instead, the main concern is the representation of the simulation state as well as of changes to it. Since it is extremely unlikely that two engines and/or software libraries share the same representation, a conversion process between the representations is required.

The representation of the simulation state includes both the utilized data structures and the underlying paradigm. For example, a rotation can be defined by a matrix that is stored in column-major format. The matrix can represent an entity's local rotation (relative to its parent node in the scene graph used by the rendering subsystem) or the rotation in world coordinates (in a flat data structure used by the physics subsystem). In addition, access to these properties depends on the initial developer's choice of identifiers.

This leads to the fact that each project needs to specify a layer between the developed application and the game engine(s) in use. Moreover, when the used engine or one of its subsystems shall be replaced by that of another engine (or a newer version of the currently used one), this connection has to be established again.

We propose a configurable layer that facilitates the arbitrary combination of subsystems and software libraries. It consists of a thin layer that is integrated into the application and the used game engine(s). Obviously, such a layer is prone to changes of the engines' API. However, this is unavoidably true for every application and abstraction layer. The beneficial aspect of such a layer is that only the very specific part for the changed subsystem has to be adapted, leaving existing applications untouched.

Although the approach allows to combine subsystems arbitrarily, that combination remains to be chosen reasonably. For instance, separating a cloth simulation system from a rendering system, possibly even by using a network transport layer, will inevitably lead to performance reduction. Moreover, the approach cannot compensate for inexistent features in engines, but at best ignore an application's request for such features (if the developer does not decide to implement them manually). Consequently, even though subsystems can be exchanged easily, their initial feature set has to be sufficient to make such a replacement reasonable. In the end, the intention of the approach is to facilitate the flexible utilization of game engine subsystems in research projects.

3.2 Architecture

There are two main concepts underlying the proposed approach:

1. The simulation state is synchronized via an ECS pattern-based representation in the abstraction layer, resulting in a star network like topology (cf. Figure 1).

2. Communication between subsystems and the application is performed on an event basis, reducing the amount of concepts involved in the synchronization process to a minimum (cf. Figure 2).

State Representation

The ECS pattern fits the task of sharing a representation between different engines best, since it decomposes entities into components that can be accessed by the respective subsystem. It does not force any hierarchy (e.g., scene graph representation) on the developer and thus provides freedom to choose an appropriate representation.

We additionally apply a semantic extension to the pattern as suggested by [35], allowing to specify identifiers and types outside the application, thus facilitating reuse. The semantic extension, amongst other things, adds human readable type information to every value that is synchronized via the abstraction layer. This way, selecting converters, dispatching value updates, selecting components, etc. is immensely simplified, since values keep the associated meaning even when passed through multiple functions.

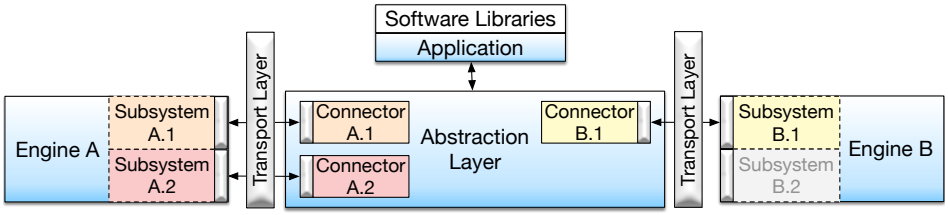

Figure 1: Schematic overview of the proposed approach. Parts that include additional code, i.e. layers for each subsystem in the engine, the transport layer, as well as the central abstraction layer which synchronizes the application state, are shown elevated.

For engines that apply the ECS pattern, like Unity, the amount of code to be written for each subsystem of an engine is very small (cf. [34]), since only the translation of the incoming events into the local state representation has to be implemented. This process is quite straightforward, as the direct identification of components in entities is possible. Engines and libraries that do not apply the pattern require an additional lookup and integration process, that includes finding the respective property in the local state representation (see section Entity-Component Registry below).

Conversion between Representations

One specific feature of the proposed approach is the centralized conversion of representation formats in the abstraction layer (cf. Figure 3). In the initially mentioned example this means that a rotation matrix in the application is automatically converted into a quaternion representation (for game engines that adopt this representation) without being noticed by the developer. For example, the common right handed matrix representation of OpenGL based applications is internally transformed into a left handed quaternion representation for Unity in our implementation.

This conversion process is supported by a centralized definition of types. The conversion operates on data types combined with semantic information, such that it is possible to differentiate between a floating point number that, e.g., identifies a radius and one that identifies an angle. This is important in case the scaling between two systems is different, since the radius would have to be scaled but the angle would not.

A further benefit of the annotated types is the opportunity to connect symbol based AI subsystems to the engine. Otherwise, their integration would require considerable effort. Such AI subsystems could involve a reasoning module that detects an entity's type by its composition of components and an action planning module that subsequently detects possible action sequences that would not be detectable without the knowledge about this inferred type.

Entity-Component Registry

An important aspect is the presence of value-change triggers regarding component modifications as well as lookup functionality for entities and components. Without these features the necessity of synchronizing the complete simulation state on a frame-by-frame basis would arise. The state representation inside the abstraction layer is designed to provide these features, wherefore the propagation of changes and events can easily be achieved. On the contrary, notification of value changes is provided only by few game engines, requiring the implementation of an entity-component registry. This registry contains references to the set of synchronized entities and components that is continuously iterated to check for value changes. Moreover, it is used for mapping identifiers of entities and components between the abstraction layer and the internal representation.

All of the described aspects (ECS-based state representation, converters, and the entity component registry) are specific to each respective system. For this reason, a software package that is designed for a certain system and complies with the software interfaces of the abstraction layer needs to implemented for each system once and can then be shared between—and possibly extended by—developers. We created two prototypes of such packages for Unity and the Unreal Engine 4 and implemented corresponding demo applications, which are presented in section 4.

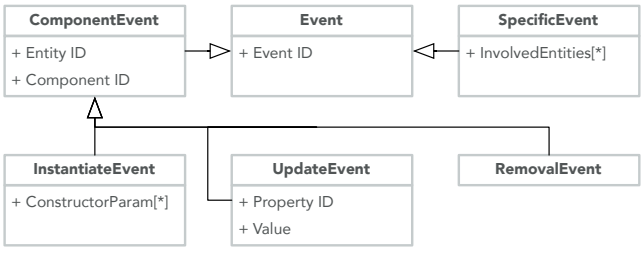

Figure 2: Hierarchy of event types used by the proposed approach.

3.3 Entity Lifecycle

The lifecycle of an entity at large consists of three phases (instantiation, update, and removal), which are described below.

Instantiation

The creation of entities is initiated via an instantiation event, which contains an entity ID and a component ID for later matching processes (cf. Figure 2). Configuration is performed inside the abstraction layer, based on descriptions of entities (i.e. sets of component descriptions and initial values). In this context, a subsystem is only given the information it needs for the entity's creation. For example, a rendering subsystem is informed about the model to be loaded, but it does not receive any information about its physical properties. In this way, the synchronization overhead is minimized and no irrelevant data has to be stored by an engine.

The mentioned component descriptions implement a simple software interface that is provided by the abstraction layer (cf. [34]). These descriptions, the converters, as well as an actor that applies converters and sends events via the transport layer, form the *Connector*s in Figure 1. Such connectors are part of the beforementioned package that is created once for each system and shared between developers.

During the creation of an entity, the components specified by the descriptions are queried from the engine. More specifically, an engine is requested to create its internal representation of the entity,

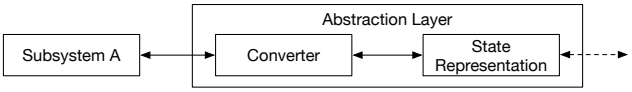

Figure 3: Overview of the conversion process: converters specified in the abstraction layer (cf. Listing 2) transform the local state representation in to the subsystem's format.

which at least contains the requested component(s) but may consist of more elements if needed by the engine (for example, every gameobject in unity has a transform component, even though it may not require a position to be functional). A mapping between the sent IDs and the internal representation is established via the previously mentioned registry. Finally, this mapping is communicated to the abstraction layer. Note that this permits connections to entities and components that already exist in the simulation of the engine. Ultimately, fully set up virtual environments can be augmented with additional functionality.

Component Updates

If a property change is detected (see Entity-Component Registry in section 3.2), this is communicated via the transport layer. Inside the abstraction layer the correct converter (cf. section 3.2) is used to transform data from the subsystem's representation into the central one. The data then is converted into the internal format of each subsystem that registered for updates of the respective component and an update event is dispatched.

Although the central representation can be chosen arbitrarily (since converters can be exchanged), it is advisable to choose a representation that results in as few conversions as possible. More specifically, choosing the representation used by the system that emits the most updates regarding a specific component (e.g., the physics engine with respect to position updates) makes sense, since this eliminates the need for converting the incoming update events.

Note that the major part of a system's internal representation is not involved in the update (nor in the instantiation) process. For example, it makes sense to store meshes and sounds in external files, send identifiers (i.e. file names) to the respective system and let it load that file, as opposed to creating an internal representation that is then converted and sent to the graphics and sound systems.

In most cases, this system's representation (the loaded model or sound) will never be changed, except for removal or replacement operations of the whole asset. If, however, such an asset would be required to be modified at runtime (e.g., a point cloud that is modified by one system and then visualized by another), a converter supporting a common representation and (if networking is involved) a representation-specific serialization process would have to be added to the abstraction and the transport layer.

The most common, intermediate case is to parametrize the systems' data and update these parameters. In the simplest case this would result in a single update (e.g., position) but also multi-parameter updates are conceivable. For instance, updated parameters of a shader can be communicated to the respective system, as opposed to sending an updated version of the shader code (which, due to the textual nature of shader code, would easily be possible, too). Since converters and subsystem connection is engine specific, such custom optimizations can be implemented without affecting other parts of an application.

Removal

Removal of entities and/or components is straightforward, since it only requires the notification of all subsystems (via an event, cf. Figure 2) to remove the respective components. Even though it is more common to remove complete entities, the removal of single components (i.e. removing functionality) is easily possible, too, since communication is performed on a per system-basis (as opposed to engine-basis). On request, systems remove (or disable) a component from their internal representation and optionally signal the successful execution to the abstraction layer, which in turn updates its representation.

Disabling and (re-)enabling components would be possible by adding two more event types. Most of this functionality has to be implemented inside the respective system or engine if it is not supported from the beginning.

Figure 4: *Big Bang* is a virtual reality application in which the user can bring his own universe to life, by creating, modifying and deleting planetary objects via a multimodal speech and gesture interface.

4 Implementation

All introduced software layers have been implemented demonstrating the general feasibility of the presented approach. The abstraction layer (L1) has been realized with the open source framework *Simulator X* [16]. A transport layer building on network sockets (L2) and two respective subsystem layers (L3)–one for the Unreal Engine and one for Unity–have been developed. Furthermore, two proof of concept applications have been implemented with this architecture: *Big Bang*, a multimodal "genesis application" inspired by Bolt's *Put-that-there*, which utilizes the VR capabilities of the Unreal Engine, as well as *PEARS*, a semantically enriched physics simulation using Unity.

In the following sections the use case of the Big Bang application serves as an example for the subsequently presented implementation of the layers L1–L3. Concerning L3 we focus on the development done in Unreal, since the respective layer in Unity has been implemented analogously.

Application

Big Bang is a virtual reality application in which the user creates, modifies, and deletes planetary objects via a multimodal speech and gesture interface (cf. Figure 4). The HTC Vive is used to create a fully immersive experience. Typical interactions include the instantiation of planetary objects ("Create a planet [pointing] there."), updating positions ("Move [pointing] that planet [pointing] there.") as well as their removal ("Destroy [pointing] that planet").

Big Bang has been implemented with Simulator X for practical reasons: Firstly, multimodal interfaces can easily be developed with the engine's integrated multimodal processing system *miPro* [10]. Secondly, this way the necessity to develop another subsystem layer that connects the application to the abstraction layer is omitted.

Furthermore, Big Bang emphasizes the benefits of cherry-picking RIS functionality. Combining miPro's capabilities with Unreal's well-known advantages, e.g., high quality graphics, yields a more immersive and enjoyable application.

L1: Abstraction Layer

The abstraction layer has been implemented as a part of Simulator X. The engine adopts the ECS pattern and therefore subsystems that control associated components can be attached to and removed from it. Moreover, Simulator X incorporates semantic based software techniques on a core level. In this way, a common ground for data types and names is created, facilitating the synchronization of different state representations.

```
1  class IntegrationExample() extends SimXApplication {
2    def applicationConfiguration = Config withSubsystem
3      //UnityConnector(unityName, "127.0.0.1", 9000) and
4      UnrealConnector(unrealName, "127.0.0.1", 9000) and
5      MiPro(mmiName)
6
7    handle[MiProMessages.Create]{msg =>
8      Planet(msg.name, msg.transform).realize()}
9    handle[MiProMessages.Update]{msg =>
10     msg.entity.set(msg.transform)}
11   handle[MiProMessages.Removal]{msg =>
12     msg.entity.remove()}
13 }
14 case class Planet(name : String, transform : ConstMat4f)
15 extends EntityDescription(name,
16   UnrealAsset(
17     transformation = transform,
18     asset          = "assets/unreal/planet.fbx"),
19   //Uncomment for Rendering in Unity
20   //UnityAsset(
21   //transformation = transform,
22   //asset          = "assets/unity/planet.obj"),
23   Multimodal(selectable = true))
```

Listing 1: A simplified `IntegrationExample` based on the *Big-Bang* application showcasing key aspects of the abstraction layer.

In our implementation of the abstraction layer a component is a collection of state variables, which can be added, updated, and removed. Access is performed via human readable symbols (like `transformation`) that contain type information and are provided via the core level semantic layer. For observing state changes callbacks that are invoked when a variable's value is modified can be registered. Concurrent access to the central application state is achieved by message-based communication, which is implemented based on the actor model [12]. The simulation state is represented by the entirety of all entities present in an application.

The so called `UnrealConnector` has been implemented as an actor that can be seamlessly integrated into the abstraction layer by implementing the layer's software interface for connectors. It undertakes four primary tasks: it spawns a dedicated networking actor that communicates via the transport layer with the Unreal Engine, observes instantiations, observes component updates, and performs removal of entities from the simulation.

A dedicated component description, called `UnrealAsset`, which contains the relevant information for the Unreal engine to render the entity, is provided with the `UnrealConnector`. Only entities containing this component are monitored by the `UnrealConnector`, reducing computational complexity.

Listing 1 showcases key aspects of the implemented abstraction layer based on Big Bang. All involved systems and/or connectors are defined in the `applicationConfiguration` (lines 2–5). In this case these are the miPro subsystem of Simulator X and the `UnrealConnector`. The composition of entities is described in so called `EntityDescriptions` (lines 14–23), which are instantiated at runtime (in lines 7–12). They consist of a set of component descriptions, which specify the state variables that ultimately make up the entities. In case of the shown planet entity, `Multimodal` describes properties that are relevant for multimodal processing, e.g, if an entity is selectable by means of interaction (line 25), while `UnrealAsset` determines the model to be loaded and rendered by the Unreal Engine as well as its initial transformation (lines 16–18). Instead of using the Unreal engine, rendering can be performed with Unity by swapping the commented code (lines 3 & 19–22) with the respective Unreal parts.

Lines 7–8 showcase the instantiation of a planet entity in response to miPro successfully recognizing a user command.

```
1 new Converter(Position)(types.Transformation){
2   // convert from local to global type
3   override def convert(i: Mat4): Vec3 =
4     Vec3(i(3).x, i(3).y, -i(3).z)
5
6   // convert from global to local type
7   override def revert(i: Vec3): Mat4 =
8     Mat4(Mat4x3.translate(Vec3(i.x, i.y, -i.z)))
9 }
```

Listing 2: A code excerpt depicting a converter which automatically transforms a right handed transformation matrix into a left handed position vector.

Since `Planet` contains the `UnrealAsset` component, the `UnrealConnector` is informed by means of the underlying actor system. The passed information is converted and conveyed to the connection actor, which ultimately emits an instantiation event to the Unreal Engine. After the planet entity's representation is instantiated in the abstraction layer, the `UnrealConnector` internally observes the state variables `transformation` and `asset` by registering a callback function. The former is, for example, triggered in response to the transformation update indicated in lines 9 and 10, subsequently converting new values and dispatching the updates via an associated event. Similarly, when the planet entity is deleted in lines 11–12, a removal event is emitted and eventually processed by the Unreal Engine.

The conversion of information is a central aspect when exchanging data between multiple engines. For example, Big Bang uses a single transformation matrix to determine the position, scaling and orientation of an entity in a right handed coordinate system, whereas Unreal uses three dedicated vector/quaternion representations and a left handed coordinate system. Due to the integrated semantics and type information our implementation permits the automatic application of converters. Listing 2 exemplifies such a converter, which transforms respective data from one representation to another each time the state is retrieved from or written to the abstraction layer.

L2: Transport Layer

To convey information from one engine to another, we chose a client server architecture using network sockets, requiring a thin connection layer in all involved systems (L1 & L3). Consequently, engines can be run on different machines in a local network, on the one hand increasing the overall computational power but on the other hand increasing the event passing time. Eventually, this approach was chosen since it allows flexible combination of different engines, even if these depend on different operating systems.

Due to the proof of concept nature of the implementation, some simplifying decisions have been taken: The current implementation utilizes the TCP protocol, obviating the need of network package management. Moreover, JSON was chosen as the exchange format because its functionality is capable of serializing and deserializing basic data types and objects into human readable, string based messages, facilitating debugging processes.

This obviously leaves room for optimizations: JSON produces overhead and is slower in terms of parsing than a binary format and the UDP protocol allows for faster transmission speed and smaller header sizes. The consequences of these design decisions regarding the performance of the current implementation are indicated in section 5. Note that a networking architecture might not be feasible in all scenarios. A transportation layer implementation for passing information on one machine, building on ECS-based data structures, e.g., [15], offers a faster and higher data throughput.

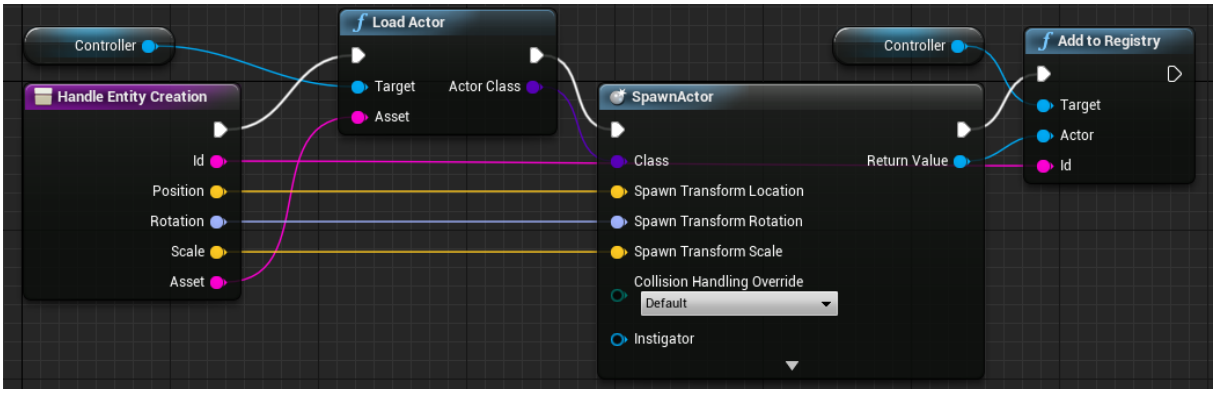

Figure 5: Simplified excerpt of an Unreal blueprint showcasing the reaction to an incoming instantiation event.

L3: Subsystem Layer

In Unreal functionality can be implemented via C++ and so called *Blueprints*, a visual scripting approach. Instead of writing code, the developer can create nodes and connect them in a visual editor. For the Unreal subsystem layer a hybrid approach was chosen: core functionality is implemented in C++, whereas the program's behavior is designed with blueprints.

The subsystem layer in Unreal is implemented in a dedicated `PlayerController`, which is a low-level class representing the human player. It was chosen for implementing the layer on account of Unreal's authority system for accessing individual elements of the simulation and the `PlayerController`'s persistence throughout the application (e.g., in multiple levels of a game). In addition, a dedicated `ConnectionController` extending the `PlayerController` was implemented.

This `ConnectionController` starts a socket on a predefined port, waiting for the respective counterpart to establish a connection and listening to the aforementioned three event types. Figure 5 illustrates Unreal's reaction to an `InstantiateEvent` using Blueprints: The controller loads an actor class, which in Unreal corresponds to an entity, with the transmitted asset and subsequently spawns the actor. Eventually, the actor is added to the entity registration with the corresponding identifier.

On receiving an `UpdateEvent` the controller identifies the affected actor by means of the transmitted entity id (cf. Figure 6). Thereafter, the transmitted changes, e.g., position updates, are directly set due to the aforementioned conversion of the abstraction layer. Lastly, when receiving a `RemovalEvent` the controller removes the actor from Unreal's simulation state.

4.1 Multi-System Examples

A simple example is shown in the top of Figure 7: Unity and Unreal render the same scene (a ball jumping on a table) being connected

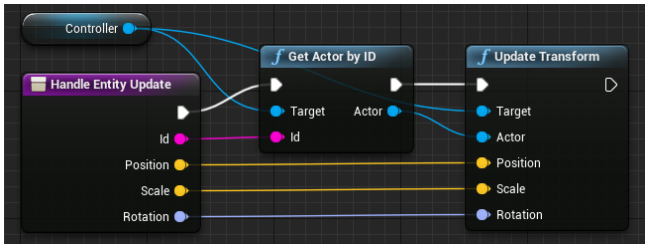

Figure 6: Simplified excerpt of an Unreal blueprint showcasing the reaction to an incoming update event.

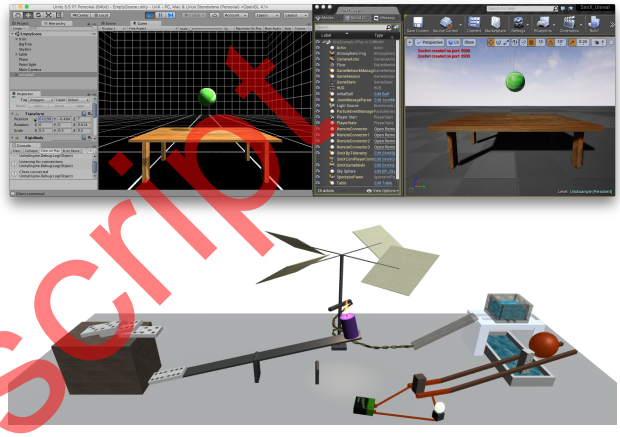

Figure 7: The scenes from the multi-system examples. Top: a simple scene synchronized between Unity and Unreal. Bottom: PEARS combining the Unity renderer and physics as well as different custom-made AI systems that support the physical simulation.

by the proposed approach. In this example, the physical simulation is performed by Unity and results are passed to Unreal.

A more complex example is proviced by the PEARS framework [8], in which the rendering and physics systems of the Unity engine are used. The connection is established via TCP sockets and uses JSON encoded messages to communicate between abstraction layer and the involved systems. In addition to the used subsystems of Unity, multiple AI modules are used to extend available capabilities regarding the physics simulation. This includes thermodynamics, electricity, and chemical properties.

An exemplary PEARS scene is shown in the bottom of Figure 7: Common rigid body physics (falling domino blocks, rotating windmill, etc.) is provided by the Unity physics subsystem, which interacts with the additional AI physics subsystems of the PEARS framework (e.g., airflow by the candle and a electrical circuit to be closed when the red ball connects the battery with the lamp). The details of this interaction have been published in [8].

As mentioned before, the implementation of the Unity connection has been performed analogously to the Unreal connection, consisting of the three layers L1–L3. These include a `UnityConnector`, and `UnityAsset`, as well as the subsystem layer in Unity (which here is a `MonoBehaviour` that is attached to the main camera). Due to the common representation in the abstraction layer, most parts of the implementation of the transport layer could be re-used.

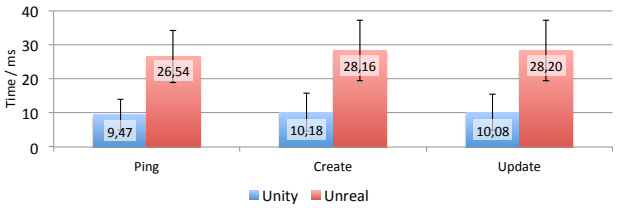

Figure 8: Results from the Unity and Unreal benchmarks.

5 Evaluation

In order to acquire metrics from the interconnection between two systems and to quantify the implementation's qualities, preliminary benchmarks have been conducted.

5.1 Benchmarks

All benchmark runs were performed on a single machine (Intel Core i7 4870HQ, AMD Radeon R9 M370X, 16GB Ram, Mac OS 10.12.3). The maximum framerate was set to 60Hz.

Our benchmarks aimed at measuring the time for executing a single operation in the creation and update phases of the entity lifecycle, excluding communication overhead. In order to do so, the timespan between initiating an operation and receiving the associated acknowledgment message was measured. The latter was sent after the receiver (Unity or the Unreal Engine) had integrated the new information into its state representation. Since access to the game state is only permitted at certain points in the respective engine's game loop (e.g., in the Update method of a Unity script), the raw measurements include the time that passed before this point was reached (i.e. until the previous frame was finished).

To estimate communication overhead, three variants of ping-type measurements were performed: First, the ping command-line tool was used to measure a base line of for pinging the local machine (0.085 ms, SD 0.015). Second, a ping-type message was implemented, to measure the plain processing time for serializing, sending, deserializing, and processing, which is not dependent on the opposite engines game loop. Processing this type of events consists in plain (de-)serialization and message sending, without any further computations. This was performed for Unity only, since our Blueprint-dependent implementation for the Unreal Engine does not support asynchronous message handling. The average processing time was 0.46 ms (SD 0.35). Finally, to distinguish the serialization and communication from actual processing of the transmitted events, a third ping-type measurement was performed, where the ping message was processed in above-described point in the engine's game loop (results are shown in column "Ping" of Figure 8).

In the actual create-update benchmark run, 100 spheres were created and position, rotation, and scale updates were sent to them. Beside these spheres only one other object (a tree) was shown in the scene, to create some (but few) constant load for the renderer. The transport layer in both cases was the before-mentioned JSON-over-TCP network connection. Results of the benchmark for the connection to Unity and the Unreal Engine are shown in Figure 8.

With the Unity connection more than 90% of the events were processed within 16 milliseconds (~1 frame for 60Hz update rate), more than 99% within 32 ms. The system was capable of processing more than 20.000 ping-type events per second and maintaining these processing times. With the Unreal connection only 0.6% of the events were processed within 16 milliseconds, 75% of the events were processed within 32 ms and more than 99% within 64 ms. Despite the longer response time, the Unreal connection was capable of processing more than 40.000 ping-type events per second and maintaining the shown processing times. It has to be noted that all measurements include two-way communication.

5.2 Discussion

First of all, the results from the Unity ping measurements indicate that for Unity an overhead of about 0.68 ms is added to the expected average ping time of 8.79 ms (~8.33 ms until end of frame plus 0.46 ms on average for asynchronous ping processing). The fact that much more than 16 messages could be processed per frame indicates that this additional time is rather an offset than processing related. It is assumable, that this offset is caused by the first-in-first-out processing of messages: messages that are received directly before a frame is finished (and thus can be processed instantly) are delayed until all other pending messages have been processed. Some minor additional overhead is introduced by message handling operations.

Consequently, we assume that the times measured in the benchmarks are composed of processing time for all messages, the time for transport and (de-)serialization, and the average time until the end of the current frame:

$$t_{measure} = n_{msg} \cdot t_{proc_{single}} + t_{transp} + \frac{t_{frame}}{2}$$

Hence, the average processing time (excluding serialization and transport) for Unity connection is approximated by

$$t_{proc_{single}} = \frac{t_{measure} - t_{transp} - 0.5 \cdot t_{frame}}{n_{msg}} \approx \frac{t_{measure} - 8.79}{100}$$

This leads to an approximated average overhead of 13.9 μs and 12.9 μs for creation and update operations, respectively.

The results for the Unity connection as compared to the Unreal Engine connection differ by a factor of about 3. The main reason for this is the more sophisticated implementation of the Unity connection, which has undergone performance optimizations (e.g., concurrent processing of incoming events and shortened JSON keys).

After all, our results are encouraging: measurements for Unity indicated that its performance is sufficient, e.g., to synchronize positions of 200 constantly moving entities at 60Hz. Regarding the connection to the Unreal engine there is room for improvements. Nevertheless, although not being within the one-frame limit, performance was sufficient to use the it in the Big Bang application.

Reasons for the poorer performance of the connection to the Unreal engine likely is the fact that processing messages was performed in the main thread, as the Blueprint visual scripting was (partly) used. The connection to the Unity engine ran a dedicated thread for this task. This also explains why nearly no message was processed by the Unreal connection within on frame, while this was the case for 90% of the messages with the Unity connection.

In addition to these results, the aspects of software reuse, including effort analysis for exchanging a rendering system using the Unity connection have been evaluated in chapter 6.2 of [34].

6 Conclusion

In this paper we presented a software integration concept for connecting subsystems of different VR and game engines: A semantic abstraction layer, based on the ECS pattern, decouples specific internal representations and provides a well defined, extensible interface fostering reusability and exchangeability. Information is communicated between engines via an event based transport layer. A centralized conversion within the abstraction layer automatically adjusts different representation formats. Our reference implementation connecting the Unreal Engine 4, the Unity engine, and Simulator X showcases the concept's general feasibility.

While the ECS pattern turned out to be a suitable choice for modeling the semantic abstraction layer, implementing automatic type conversions, and representing a sharable simulation state, benchmark results indicate room for improvement of the transport layer's implementation.

The proof of concept applications *Big Bang* and *PEARS*, however, show promising results emphasizing the usefulness of cherry-picking RIS functionality in innovative research projects.

Future work will include optimization in terms of performance. This revision will specifically include the replacement of the JSON format in favor of a purpose-built binary format. We furthermore plan a revision of the Unreal connection, which yielded relatively poor benchmark results as compared to its Unity counterpart. And finally, we are going to continue utilizing our implementation of the presented abstraction layer to use state of the art game engine features in future research projects.

REFERENCES

[1] T. Alatalo. An entity-component model for extensible virtual worlds. *Internet Computing, IEEE*, 15(5):30–37, Sept 2011.

[2] U. Bernardet, D. Adhia, N. Jaffe, J. Wang, M. Nixon, O. Alemi, J. Phillips, S. DiPaola, P. Pasquier, and T. Schiphorst. M+M: A Novel Middleware for Distributed, Movement Based Interactive Multimedia Systems. In *Proceedings of the 3rd International Symposium on Movement and Computing*, MOCO '16, pages 21:1–21:9. ACM, 2016.

[3] A. Boeing and T. Bräunl. 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*, GRAPHITE '07, pages 281–288. ACM, 2007.

[4] R. Bouville, V. Gouranton, T. Boggini, F. Nouviale, and B. Arnaldi. #FIVE: High-Level Components for Developing Collaborative and Interactive Virtual Environments. In *Proceedings of Eighth Workshop on Software Engineering and Architectures for Realtime Interactive Systems*, SEARIS, 2015.

[5] M. Cavazza and F. Charles. User Interaction for Interactive Storytelling. *Handbook of Digital Games and Entertainment Technologies*, pages 415–428, 2017.

[6] A. Clark and D. Moodley. A System for a Hand Gesture-Manipulated Virtual Reality Environment. In *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '16, pages 10:1–10:10. ACM, 2016.

[7] Crytek. Cryengine 5, 2015. http://www.cryengine.com.

[8] B. Eckstein, J.-L. Lugrin, D. Wiebusch, and M. Latoschik. PEARS - Physics Extension And Representation through Semantics. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(2):178–189, 2015.

[9] Epic Games. Unreal Engine 4, 2014. https://www.unrealengine.com/.

[10] M. Fischbach, D. Wiebusch, and M. E. Latoschik. Semantics-based software techniques for maintainable multimodal input processing in real-time interactive systems. In *9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*. IEEE Computer Society, 2016.

[11] J. Gugenheimer, D. Wolf, G. Haas, S. Krebs, and E. Rukzio. SwiVRChair: A Motorized Swivel Chair to Nudge Users' Orientation for 360 Degree Storytelling in Virtual Reality. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 1996–2000. ACM, 2016.

[12] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[13] T. Hilfert and M. König. Low-cost virtual reality environment for engineering and construction. *Visualization in Engineering*, 4(1):2, 2016.

[14] A. Juarez, W. Schonenberg, and C. Bartneck. Implementing a Low-Cost CAVE System Using the CryEngine2. *Entertainment Computing*, 1(3):157–164, 2010.

[15] P. Lange, R. Weller, and G. Zachmann. Wait-free hash maps in the entity-component-system pattern for realtime interactive systems. In *IEEE VR 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, 2016.

[16] M. E. Latoschik and H. Tramberend. Simulator X: A Scalable and Concurrent Architecture for Intelligent Realtime Interactive Systems. In *IEEE Virtual Reality Conference*, pages 171–174, 2011.

[17] M. Lewis and J. Jacobson. Game Engines in Scientific Research. *Commun. ACM*, 45(1):27–31, 2002.

[18] J.-L. Lugrin and M. Cavazza. Making Sense of Virtual Environments: Action Representation, Grounding and Common Sense. In *Proceedings of the 12th International Conference on Intelligent User Interfaces*, pages 225–234. ACM, 2007.

[19] J.-L. Lugrin, F. Charles, M. Cavazza, M. Le Renard, J. Freeman, and J. Lessiter. CaveUDK: a VR game engine middleware. In *Proceedings of the 18th ACM symposium on Virtual Reality Software and Technology*, pages 137–144. ACM, 2012.

[20] J.-L. Lugrin, M. E. Latoschik, M. Habel, D. Roth, C. Seufert, and S. Grafe. Breaking Bad Behaviours: A New Tool for Learning Classroom Management using Virtual Reality. *Frontiers in ICT*, 3:26, 2016.

[21] F. Mannuß, A. Hinkenjann, and J. Maiero. From Scene Graph Centered to Entity Centered Virtual Environments. In *IEEE VR Workshop on Software Engineering and Architectures for Realtime Interactive Systems*, pages 37–40, 2008.

[22] A. C. A. Mól, C. A. F. Jorge, and P. M. Couto. Using a Game Engine for VR Simulations in Evacuation Planning. *IEEE Computer Graphics and Applications*, 28(3):6–12, May 2008.

[23] M. Müller, T. Günther, D. Kammer, J. Wojdziak, S. Lorenz, and R. Groh. *Smart Prototyping - Improving the Evaluation of Design Concepts Using Virtual Reality*, pages 47–58. Springer International Publishing, 2016.

[24] M. Nakevska, C. Vos, A. Juarez, J. Hu, G. Langereis, and M. Rauterberg. Using Game Engines in Mixed Reality Installations. In *International Conference on Entertainment Computing*, ICEC, pages 456–459. Springer, 2011.

[25] P. Petridis, I. Dunwell, S. de Freitas, and D. Panzoli. An Engine Selection Methodology for High Fidelity Serious Games. In *2010 Second International Conference on Games and Virtual Worlds for Serious Applications*, pages 27–34. IEEE, 2010.

[26] H. Prendinger, N. Alvarez, A. Sanchez-Ruiz, M. Cavazza, J. Catarino, J. Oliveira, R. Prada, S. Fujimoto, and M. Shigematsu. Intelligent Biohazard Training Based on Real-Time Task Recognition. *ACM Transactions on Interactive Intelligent Systems*, 6(3):21:1–21:32, 2016.

[27] P. R. Smart, T. Scutt, K. Sycara, and N. R. Shadbolt. Integrating ACT-R Cognitive Models with the Unity Game Engine. In *Integrating Cognitive Architectures into Virtual Character Design*, chapter 2, pages 35–64. IGI Global, 2016.

[28] D.-J. Song and J. Park. AnyHaptics: A Haptic Plug-in for Existing Interactive 3D Graphics Applications. In *Proceedings of the 20th ACM Symposium on Virtual Reality Software and Technology*, VRST '14, pages 27–30. ACM, 2014.

[29] M. Sra, S. Garrido-Jurado, C. Schmandt, and P. Maes. Procedurally Generated Virtual Reality from 3D Reconstructed Physical Space. In *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology*, VRST '16, pages 191–200. ACM, 2016.

[30] A. Steed. Some Useful Abstractions for Re-Usable Virtual Environment Platforms. In *IEEE VR Workshop on Software Engineering and Architectures for Realtime Interactive Systems*, pages 33–36, 2008.

[31] T. Tutenel, R. Bidarra, R. M. Smelik, and K. J. D. Kraker. The Role of Semantics in Games and Simulations. *Comput. Entertain.*, 6(4):57:1–57:35, 2008.

[32] Unity Technologies. Unity 5, 2015. http://www.unity3d.com.

[33] M. Westhoven and T. Alexander. *Towards a Structured Selection of Game Engines for Virtual Environments*, pages 142–152. Springer International Publishing, 2015.

[34] D. Wiebusch. *Reusability for Intelligent Realtime Interactive Systems*. PhD thesis, Universität Würzburg, 2015.

[35] D. Wiebusch and M. E. Latoschik. Decoupling the Entity-Component-System Pattern using Semantic Traits for Reusable Realtime Interactive Systems. In *IEEE VR Workshop on Software Engineering and Architectures for Realtime Interactive Systems*, IEEE VR, 2015.

[36] D. J. Zielinski, R. P. McMahan, S. Shokur, E. Morya, and R. Kopper. Enabling closed-source applications for virtual reality via OpenGL intercept-based techniques. In *2014 IEEE 7th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 59–64, March 2014.