RAGE architecture for reusable serious gaming technology components

Wim van der Vegt\textsuperscript{1}, Wim Westera\textsuperscript{1}, Enkhbold Nyamsuren\textsuperscript{1}, Atanas Georgiev\textsuperscript{2} and Ivan Martinez Ortiz\textsuperscript{3}

\textsuperscript{1} Open University of the Netherlands, Valkenburgerweg 177, 6419 AT Heerlen, The Netherlands
\textsuperscript{2} Sofia University “St. Kliment Ohridski”, Bul Tzar Osvoboditel 15, Sofia 1504, Bulgaria
\textsuperscript{3} Complutense University of Madrid, Avenida de Seneca 2, Madrid 28040, Spain.

Copyright © 2015 Wim van der Vegt, Wim Westera, Enkhbold Nyamsuren, Atanas Georgiev and Iván Martinez Ortiz. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Abstract

For seizing the potential of serious games, the RAGE project - funded by the Horizon-2020 Programme of the European Commission - will make available an interoperable set of advanced technology components (software assets) that support game studios at serious game development. This paper describes the overall software architecture and design conditions that are needed for the easy integration and reuse of such software assets in existing game platforms. Based on the component-based software
engineering paradigm the RAGE architecture takes into account the portability of assets to different operating systems, different programming languages and different game engines. It avoids dependencies on external software frameworks and minimizes code that may hinder integration with game engine code. Furthermore it relies on a limited set of standard software patterns and well-established coding practices. The RAGE architecture has been successfully validated by implementing and testing basic software assets in four major programming languages (C#, C++, Java and Typescript/JavaScript, respectively). A demonstrator implementation of asset integration with an existing game engine was created and validated. The presented RAGE architecture paves the way for large scale development and application of cross-engine reusable software assets for enhancing the quality and diversity of serious gaming.

1. Introduction

The potential of non-leisure games (serious games) in industry, health, education and the public administration sectors has been widely recognised. An increasing body of evidence is demonstrating the effectiveness of games for teaching and training [1]. While instructional scientists consider motivation as a main driver for effective learning [2-5], games are capable of amplifying the players’ motivation by hooking and absorbing them in such a way that they can hardly stop playing [6]. This motivational power of games is ascribed to their dynamic, responsive and visualised nature, which goes along with novelty, variation and choice, effecting strong user involvement and providing penetrating learning experiences [6]. In addition, serious games allow for safe experimentation in realistic environments, stimulate problem ownership by role adoption, and allow for learning-by-doing approaches, which support the acquisition of tacit and contextualised knowledge [7]. Nevertheless, the complexity of serious game design may hamper the games’ effectiveness [8]. In particular the subtle balance between game mechanics and pedagogical power is not self-evident [9]. Also, various authors [1,10] note that many studies fail to evaluate the educational effectiveness of serious games in a rigorous manner and they call for more randomised controlled trials (involving comparisons between an experimental group and a control group) for increased scientific robustness. Still, the potential of games for learning is widely recognised, stimulating serious game development as a new branch of business.

For various reasons, however, seizing this potential has been problematic. The serious game industry displays many features of an emerging, immature branch of business: being scattered over a large number of small independent players, weak interconnectedness, limited knowledge exchange, absence of harmonising standards, a
lot of studios creating their localised solutions leading to “re-inventing the wheel”,
limited specialisations, limited division of labour and insufficient evidence of the
products’ efficacies [11,12]. Still, conditions for a wider uptake of serious games are
favourable. PCs, game consoles and handheld devices have become low-priced
commodities as are to a lesser extent advanced tools for game creation and the
associated graphics design and media production.

In order to enhance the internal cohesion of the serious game industry sector the
RAGE project (rageproject.eu), which is Europe’s principal research and innovation
project on serious gaming in the Horizon 2020 Programme, makes available a
diversity of software modules (software assets) for developing serious games easier,
faster and more cost-effectively. The pursued software assets cover a wide range of
functionalities particularly tuned to the pedagogy of serious gaming, e.g. in player data
analytics, emotion recognition, stealth assessment, personalisation, game balancing,
procedural animations, language analysis and generation, interactive storytelling,
social gamification and many other functions. Game developers could then simply add
the required assets to their project without the need to do all the programming
themselves. For example, the stealth assessment asset would allow the game
development team to easily incorporate diagnostic functionality that provides metrics
for the progressive mastery of knowledge and skills, based on the tracking and
processing of the players’ behavioural patterns in the game.

Importantly, the creation of the game software assets is not an isolated technical
endeavour, but instead it is positioned as a joint activity of multiple stakeholders
represented in the RAGE consortium. In the consortium computer scientists and IT
developers are working together with serious game developers, educational
researchers, education providers and end users to make sure that the new technology
components are practical and usable, and create added pedagogical value. To this end
the architecture presented in this paper is used for supporting the easy integration of
assets in a set of serious games that will be developed in the project. The games and
the assets included will be empirically tested for their effectiveness in large scale pilots
with end-users. Hence, one of the major technical challenges of RAGE is to ensure
interoperability of the software assets across the variety of game engines, game
platforms and programming languages that game studios have in use. The
incorporation of the software assets should be as easy as possible (e.g. “plug and
play”), without enforcing specific standards or systems as to avoid principal adoption
barriers. In addition, the software assets should allow for being grouped together into
more complex aggregates, e.g. combining an emotion recognition asset with a game-
balancing asset. That is, the assets should comprise a coherent, component-based
system that supports data interoperability between its elements.
This paper takes a technical perspective by describing and validating the RAGE software architecture that aims to accommodate the easy integration and reuse of such interoperable software assets in existing game platforms. The RAGE architecture particularly addresses the structure and functioning of client-side assets. Since exactly client-side assets are supposed to be fully integrated in the game engine, non-compliance issues of assets are likely to occur client-side. This paper will explain the basic requirements and the proposed solution. It presents the proofs of concept that have been created for validating the RAGE architecture in four different programming languages, and it describes how the concrete technical issues encountered in these proofs were overcome.

2. Related work

There are various existing efforts in promoting reusability in both serious games and leisure games. The Unity Asset Store (https://www.assetstore.unity3d.com/) is an example of a successful online market place for game objects. Most of the objects are media objects (e.g. terrains, audio, buildings, weapons), but an increasing number of software modules (e.g. analytics, cloud backend, game AI) are becoming available. Unfortunately, most of the software objects can only be reused in the Unity game engine. Various other online platforms offer reusable game objects, for instance the Guru asset store (https://en.tgcstore.net/), Game Salads (http://gshelper.com/), GameDev Market (https://www.gamedevmarket.net/), Unreal Marketplace (https://www.unrealengine.com/marketplace), and Construct 2 (https://www.scirra.com/store), but their main focus is on user-interface objects and templates.

At a more abstract level Folmer [13] proposed a reference architecture for games. The architecture consists of several layers, such as game interface layer and domain specific layer, and it identifies reusable components within each layer. A similar idea proposed by Furtado et al. [14] describes a software product line-based approach for creating reusable software modules specific to a particular (sub)domain of games. These modules include domain specific reference architectures and languages. The current paper, however, is little concerned with defining a reusability framework for the entire process of game development. Instead, it proposes a "reference architecture" for a particular niche within serious game development that covers enhanced pedagogical functionality. Consequently, while attempting to simplify the reuse of pedagogical components, the RAGE architecture interferes as little as possible with established game development processes.

A recent reusability framework targeting serious games [15,16] relies on a service-oriented architecture (SOA). Within this framework, domain independent features commonly used in serious games are encapsulated into components and implemented
as services. SOA offers several advantages such as decoupling from implementation
details and a high degree of reusability. However, the SOA approach goes with several
limitations, for instance the requirement of being constantly online, diminished
flexibility such as customisation and configuration of services by service consumers,
reduced system performance due to additional overheads associated with SOA and
network calls.

A more versatile reference architecture is needed to ensure seamless interoperability of
the software assets across different game engines, game platforms and programming
languages, whilst the limitations of SOA should be avoided as much as possible.

3. Starting points

3.1. The asset concept

The idea of making available a set of software assets seems to neglect the fact that the
productive reuse of software requires the software to be complemented with
installation guides, metadata, product manuals, tutorials examples, documentation and
many other things. For describing these elements we will comply with the W3C
ADMS Working Group [17] by referring to the term “asset”. Assets are not limited to
software, but are considered abstract entities that reflect some “intellectual content
independent of their physical embodiments” [17] or even more abstract, an asset
would be “a solution to a problem” [18]. Not every asset includes software: indeed, the
Unity Asset Store offers a variety of solutions, either media assets such as 3D-objects,
audio, textures or particle systems, or software assets such as editor extensions, game
AI and physics engines.

The RAGE assets discussed in this paper are software assets. They may either contain
a source code file or a compiled program file, and may contain various other artefacts
(cf. Figure 1).
The software artefact inside the asset is called “Asset Software Component”. For being able to retrieve an asset from a wide variety of assets that are stored in an asset repository (the RAGE asset repository), it contains machine-readable metadata. In addition to keyword classifiers and asset descriptions the metadata include information about versions, asset dependencies and programming language used among other things. The asset may include various additional artefacts that are not to be compiled (tutorials, manuals, licences, configuration tools, authoring tools and other resources). While the constituents of the asset are stored separately in the RAGE repository, the asset can be packaged for distribution.

In the rest of this paper we will interchangeably use the term “Asset Software Component” and the term “Asset” (as a short-hand notation) for indicating the software module that is to be linked or integrated with the game.

3.2. Basic requirements

For being able to fulfil the ambition of the RAGE project to create an interoperable set of advanced game technology assets that can be easily used by serious game studios we have expressed the following basic requirements.

3.2.1. Interoperability between assets

Since game developers may want to group multiple assets into more complex aggregates, data storage of assets to be used in their games and data exchange between assets should be well-defined.
When creating a new asset, asset developers should not replicate functionalities of existing assets but instead should be able to exploit these. This requires a consistent approach to nomenclature of variable names, objects and methods to be used across the whole set of assets, so that correct calls and references can be made.

3.2.3. Extendibility

The architecture should be robust over extending the set of assets with new assets. Developers may want to create new assets and add these to the existing set of assets. For allowing new assets to interact with existing ones, the interoperability conditions should be preserved.

3.2.4. Addressing platform and hardware dependencies

Different hardware (e.g. game consoles) and operating systems may require different technical solutions. Browsers, actually the pairing of browsers and operating systems, display an even wider diversity with respect to browser brands, settings and versions. This suggests that the programming for web browsers should be conservative as to avoid browser version issues as much as possible. Hence, direct access to the game user interface and/or operating system should be avoided.

3.2.5. Portability across programming languages

Using only well-established software patterns in RAGE would increase the portability as there will be valid existing code available from the start. Nevertheless, portability across different programming languages may be affected by the different natures and execution modes of the supported languages, for instance interpreted languages such as JavaScript versus compiled languages as C++, C# and Java. Also multi-threading versus single threading could hamper code portability.

3.2.6. Portability across game engines

The implementation of an asset in a supported programming language should be portable across different game engines and possibly even across hardware platforms and operating systems as much as possible. For this it is important to rely on the programming language’s standard features and libraries to maximise the compatibility across game engines. Assets could thus delegate the implementation of required
features to the actual game engine, for example the actual storage of runtime data: as the game engine knows where and how data can be stored, assets could delegate the actual storage operation to the game engine and need not have their own stores.

### 3.2.7. Minimum dependencies on external software frameworks

The usage of external software frameworks (such as jQuery or MooTools for JavaScript) should be avoided as much as possible because of their potential interference with the game engine code. Also namespace conflicts may readily occur when external frameworks were used.

### 3.2.8. Quality assurance

For preserving the integrity of an ever-growing system of assets basic quality assurance requirements will be needed. With respect to coding RAGE assets require a consistent rather than mandatory coding style and expects documentation on class level or method level in order to accommodate debugging and maintenance. The inclusion of well-established software patterns and compliance with (subsets of) existing standards in order to minimise overheads should have priority. Namespaces (or Modules) should be used to prevent name conflicts between RAGE assets and game engine code.

### 3.2.9. Overall simplicity

Software complexity should be avoided as to preserve practicability and maintainability. Preferably, assets should address only core functionality, which is in accordance with good practices of software development. Therefore, as a general starting point, the architecture and the integration points should be as simple as possible.

### 4. Component-based design and development

Since RAGE assets are positioned as being used as reusable software components (plug-and-play), we adopt a component-based development approach (CBD) [19,20].
In accordance with CBD, the RAGE asset architecture defines a component model for creating a reusable asset. The component model conforms to common norms of CBD, which are summarised as follows:

- A component is an independent and replaceable part of a system that fulfils a distinct function [20,21]. Following this norm, it is assumed that an asset provides a game developer with access to a concrete functionality that produces added value. For RAGE this added value will ideally (but not exclusively) be within the context of learning and teaching.

- A component provides information hiding and serves as a black box to other components and technologies using it [22,23]. In accordance with this norm a game developer does not need to know how a particular functionality was implemented in the asset.

- A component communicates strictly through a predefined set of interfaces that guard its implementation details [20,21] [24]. Within the context of RAGE, it is assumed that a minimal intervention from a game developer is needed to integrate assets with a game engine.

Various existing CBD frameworks are available such as Enterprise JavaBeans [25] and CORBA [26]. However, we have chosen not to use these for several reasons. Although these frameworks simplify component integration and management, cross-platform support is still an issue. For example, Enterprise JavaBeans is limited to the Java platform. While CORBA is a language independent framework, mapping between different languages using an interface-definition language remains problematic. More critically, these frameworks are overly general and complex, which makes them unsuitable for addressing needs specific to the domain of serious game development. For example, game engines are de facto standard platforms for component control and reusability in game development, which creates a direct conflict with EJB or CORBA. For these and other reasons, existing CBD frameworks find very limited adoption in game development. RAGE will offer a less centralised approach by including a lightweight control component in the form of an Asset Manager that is designed to complement game engines rather than to take over their functionalities. The Asset Manager is discussed in more details later in this article.
5. RAGE architectural design

5.1. Client-side assets versus server-side assets

Assets can be positioned both client side and server side. Figure 2 sketches the general lay-out of an asset-supported game system.

![Distributed architecture of client-side assets and server-side assets supporting a game](image)

On the client side the player has access to a runtime game (game engine). The game may incorporate client side assets that provide enhanced functionality. On the server side a game server may be supported by server-side assets. Server-side assets may either be integrated with the game server or reside on remote servers.

Client-side assets should be preferred when frequent and direct interactions with the game engine on the local computer are required, or when results should be immediately available. As a general guideline any processing that can be done client side, e.g. the processing of temporary, local data, should in fact be done client side, as to reduce external communications as much as possible. However, when extensive processing is required, e.g. for speech recognition, a server-side asset would be appropriate in order to avoid poor game performance, e.g. reduced frame rate or reduced game responsiveness. Obviously, assets that collect and process population data (learning analytics) and assets that require real time synchronisation across multiple remote players need to be server side as well.
Communications between the game engine and the various servers are readily based on the http protocol, e.g. REST, SOAP. On the client-side all assets are fully integrated with the game engine. This allows for direct interface methods, thus avoiding the impediments of SOA. The principal architectural challenges are on the client side. The easy and seamless integration of multiple collaborating client-side assets as well as the portability of assets across multiple game engines, platforms and programming languages requires a sound overall component architecture. This client-side architecture will be elaborated in the next sections.

5.2. Programming languages

Given the wide variety of computer programming languages a full portability of assets across programming languages will not be possible. It is inevitable to condense the spectrum of covered programming languages to those languages that are mostly used for creating games. This mainly holds for client-side code which is highly dictated by the game engine used, while server-side solutions are able to manage more diversity. For a start RAGE assets will be developed supporting two core code bases: C# (for desktop and mobile games) and HTML5/JavaScript (for browser games), which are predominant products of compiled languages and interpreted languages, respectively. According to a survey among European game studios [27] the most popular programming language is C# (71%), followed by C++ (67%), JavaScript (48%), objective C (33%) and Java (33%), which is quite similar to the latest Redmonk programming languages rankings [28]. The growth of C# has accelerated significantly since 2014, when Microsoft released its .NET core framework as open source, supporting any operating system and pushing the multiplatform nature of C#. Major game engines or multi-platform tools (e.g. Unity/Xamarin) use C# as their core language. C++ is still being used a lot, but it is more complex than C# or Java and often relies on platform-dependent constructs. With respect to browser-based games HTML5/JavaScript is considered a de facto standard. However, for overcoming the lack of strictness of JavaScript the RAGE project has adopted TypeScript (http://www.typescriptlang.org/) instead as the primary development language, which directly translates into JavaScript. TypeScript is used as a superclass of JavaScript that adds static typing, which can be used by Integrated Development Environments and compilers to check for coding errors.

5.3. The Asset Software Component’s internal structure

Figure 3 displays the UML class diagram of the Asset Software Component.

5.3.1. IAsset

The IAsset class, which is defined as an interface, provides the abstract definition of the Asset Software Component including the fields, properties and methods required for its operations and communications.

5.3.2. BaseAsset
BaseAsset implements the set of basic functionalities following the definitions provided by IAsset. Moreover, BaseAsset exposes standardised interfaces that delegate the storage of component's default settings and run-time data used by the component to the game engine.

5.3.3. ISettings

In accordance with the abstract definition in the IAsset interface this interface ensures that every Asset Software Component has the basic infrastructure for managing a unique component ID, type, settings, version information, etc.

5.3.4. BaseSettings

This class realises the ISettings interface. It serves as a base class for the component’s configuration settings.

5.3.5. IBridge

IBridge provides a standardised interface that allows the component to communicate with external technologies such as the game engine or a remote service.

5.3.6. ClientBridge

This bridge realises the IBridge interface. It mediates the direct communication from an Asset Software Component to the game engine (for calls from the game engine to the component mediation is not required). In addition, ClientBridge may implement additional interfaces like IDataStorage via polymorphism. Thus, the same bridge object may provide the component with more specialised functionalities, such as allowing it to retrieve default settings from the game engine.

5.3.7. IDefaultSettings
The IDefaultSettings interface delegates the management of component's default settings to the game engine. It can be used when configuration parameters are stored in an external file rather than being hard-coded in a BaseSettings subclass.

5.3.8. IDataStorage

The IDataStorage interface delegates the management of run-time data used by the component to the game engine. It can be implemented to access a local file or to query a database.

5.3.9. AssetManager

The AssetManager takes over the management of multiple RAGE assets during both compile-time and run-time. The game engine instantiates all necessary assets and creates a singleton of the AssetManager that simplifies the tracking of all instantiated assets. The AssetManager covers the registration of the assets used by the game, it locates the registered assets on request and it offers some common methods and events that may be used by the assets, e.g. requesting the name and type of the game engine. The Asset Manager is discussed in more details in next section.

5.3.10. EventManager

Alternatively, indirect (multicast) communication between various elements is also possible via the EventManager that is initialized by the AssetManager.

The RAGE asset architecture can thus be summarised as follows:

- The IAsset class defines the requirements for the RAGE asset as a reusable software component.
- For game developers the implementation details of an Asset Software Component are hidden by a set of interfaces.
• Likewise, the implementation details of an external technology are shielded for the Asset Software Component by one or more interfaces, for example the storage of asset data in the game engine through IDataStorage, which is implemented on a bridge.

• The architecture supports various means of communication, e.g. between assets, via the bridge interfaces or via the EventManager.

5.4. The AssetManager

As indicated above, the AssetManager is the coordinating agent for the management of multiple RAGE assets. Obviously, the AssetManager complies with the singleton pattern as only one coordinating agent is needed. The AssetManager covers the registration of all Asset Software Components that are used by the game engine. For achieving this it should expose methods to query this registration so that each Asset Software Component is able to locate other components and link to these. For avoiding duplicate code in each Asset Software Component the AssetManager could also be the provider of basic services that are relevant for all assets, such as the game engine’s heartbeat, which indicates the game’s proper operation, or user login/logout info for assets that need a user model. It centralises the code and requires only a single interaction point with the game engine. These data could be broadcast and transferred to Asset Software Components that have subscribed themselves for this event. Likewise, the AssetManager could also coordinate the link between the Asset Software Component and its data storages (which in fact requires reference to the game engine, because an Asset Software Component would not have storage capacity by itself).

5.5. Asset communication

Asset Software Components need to communicate with the outside world, for instance for receiving input from a user or game engine, for sending the results of their calculations to the game engine, for making web calls to query server-based services or for linking up with other RAGE assets. For allowing an Asset Software Component (or its sub-components) to communicate with the outside world, well-defined interfaces are needed. A set of standard software patterns and coding practices are used for accommodating these communications. These include the “Publish/Subscribe” pattern, the “Bridge” pattern, Asset method calls and web services (cf. Table 1).

Table 2: Communication modes of client-side assets
### Client-side requests

<table>
<thead>
<tr>
<th>Communication modes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Asset to asset</strong></td>
</tr>
<tr>
<td>These communications requires a once-only registration of the Asset Software Component at the AssetManager:</td>
</tr>
<tr>
<td>— Publish/Subscribe</td>
</tr>
<tr>
<td>— Asset method call</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Asset to game engine</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>With bridge:</td>
</tr>
<tr>
<td>— Web services (e.g. outside world)</td>
</tr>
<tr>
<td>— Game engine functionality</td>
</tr>
<tr>
<td>— Hardware, Operating System</td>
</tr>
<tr>
<td>Without bridge:</td>
</tr>
<tr>
<td>— Publish/subscribe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Game engine to asset</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>— Asset Method calls</td>
</tr>
<tr>
<td>— Publish/subscribe</td>
</tr>
</tbody>
</table>

The various communication modes will be briefly explained below.

#### 5.5.1. Asset Software Component’s methods

Each Asset Software Component may dynamically communicate with any other asset in the environment. This capability is enabled through registration by the Asset Manager. An Asset Software Component can query the Asset Manager for other Asset Software Components by referring to either a unique ID and/or a class name. Once the requested Asset Software Component is encountered, a direct communication can be established between two components without the need for further mediation by the Asset Manager.

#### 5.5.2. Bridges

For allowing an Asset Software Component to call a game engine method the bridge software pattern [29] is used, which is platform-dependent code exposing an interface. The game engine creates a bridge and registers it either at a specific asset or at the AssetManager. The BaseAsset then allows an Asset Software Component easy access...
to either one of these bridges to further communicate with the game engine. Overall, the bridge pattern is used to mediate a bi-directional communication between the Asset Software Component and the game engine while hiding game engine’s implementation details. Additionally, polymorphism is used by allowing a bridge to implement multiple interfaces. The Asset Software Component may identify and select a suitable bridge and use its methods or properties to get the pursued game data.

5.5.3. Web services

Communication through web services assumes an online connection to a remote service. Web services allow an Asset Software Component to call a service from the game engine by using a Bridge interface. In principle, an Asset Software Component may not implement the communication interface itself and instead may rely on the adapters provided by the game engine [30]. Such approach would remove the asset’s dependency on specific communication protocols used by remote services, thereby allowing a greater versatility of the asset. Within the RAGE project automatic coupling with services will be supported by using the REST communication protocol [31]. When a service is unavailable, e.g. when the game system is offline, the interface should be able to receive a call without processing it or acting on it. It is reminded here that server-side communications as indicated before in Figure 2 can all be implemented as web services.

5.5.4. Publish/Subscribe

Communications can also be arranged using the Publish/Subscribe pattern, which supports a 1-N type of communication (broadcasting). An example would be the game engine frequently broadcasting player performance data to multiple assets. The RAGE architecture allows for including Publish/Subscribe patterns where both Asset Software Components and the game engine can be either publishers or subscribers. The Publish/Subscribe pattern requires an EventManager, which is a centralised class that handles topics and events. The EventManager is initialised by the AssetManager during its singleton instantiation. Once initialised, either an asset or the game engine can use the EventManager to define new topics, (un)subscribe to existing topics, or broadcast new events. According to the Publish/Subscribe design pattern, subscribers do not have knowledge of publishers and vice versa. This allows an asset to ignore implementation details of a game engine or other assets. Additionally, this mode of
communication is more suitable for asynchronous broadcasting to multiple receivers than the bridge-based communication, which realises bilateral communications only.

5.5.5. Composite communications

The basic patterns explained above enable composite communication modes, which are composed of multiple stages. For instance, Asset Software Components may use the game engine as an intermediate step for their mutual communications, by using bridges, web services, or Publish/Subscribe. Also the AssetManager may act as a communication mediator. Once registered at the AssetManager, an Asset Software Component could use the AssetManager’s set of commonly used methods and events in order to minimise the number of the game engine interaction points. In many cases it is more efficient to implement widely used functionality in the AssetManager than implementing it in every individual asset.

6 Technical validation

For the technical validation of the RAGE architecture a basic software asset has been developed for all four of the selected programming languages. This basic Asset Software Component included all elementary operations and patterns, e.g. registration, save, load, log, among other things. The assets should meet the following test requirements:

1. Once created, the Asset Software Components should induce the creation of a single AssetManager, which enables the components’ self-registration.
2. The AssetManager should be able to locate Asset Software Components and generate the associated versions and dependency reports.
3. Asset Software Components should be able to directly connect through a method call.
4. Asset Software Components should be able to call game engine functionality. The bridge code between the Asset Software Components and the game engine should provide some basic interfaces, such as simple file i/o and access to web services.
5. The Publish/Subscribe pattern should allow Asset Software Components to both broadcast and subscribe to broadcasts, e.g. transferring an object.
6. The system should support multiple interactions, e.g. a dialog system.
7. The system should check for (default) settings and their serialisation to XML (for C#) and should be able to include default settings at compile time.
Tested implementations of the basic asset in all four programming languages can be found on GitHub (https://github.com/rageappliedgame). All implementations proved to meet the specified requirements. Yet, a number of language-dependent issues were encountered (and solved) that deserve further attention. In addition, the Unity game engine was used as an integration platform for the C# asset version. A number of engine dependent issues were identified and solved as well.

6.1 Issues in C#

Characters.

First, since Windows and OS X have different directory separator characters, forward slash (/) and backslash (\), respectively, portability fails. Problems can be avoided, however, by using the Environment class in C# that dynamically returns the correct separator rather than using hardcoded separators. Second, the mono version used in Unity (v5.2) silently turns UTF-8 XML into UTF-16 during parsing, leading to problems during de-serialisation of version info. This issue can be bypassed by omitting the parsing procedure and directly serialising the XML files.

Debugging.

First, Mono’s Debug.WriteLine method does not offer a syntax format such as String.Format. In order to obtain formatted diagnostic output messages during asset development, the String.Format method must be used explicitly. Second, debugging in Unity needs a different format of the Debug Symbol files generated during compilation with Visual Studio. Mono provides a conversion tool for this. Third, the pdb to mdb debug symbol converter of Unity (v5.2) cannot convert debug symbols created by Visual Studio 2015. A workaround is using Visual Studio 2013 or patch and recompile this utility. Finally, Mono and .Net display slight differences in the method names that are used for diagnostic logging. This problem can be solved easily by using the bridge for supplying the actual logging methods.

Compilation.
In Unity the assemblies with Embedded Resources (i.e. RAGE Asset Software Components have their version data and localisation data embedded) cannot be compiled as upon compilation Unity automatically removes the embedded resources. The resources can be compiled with Visual Studio though, whereupon they can still be used in the Unity engine.

### 6.2 Issues in TypeScript/JavaScript

**Characters.**

TypeScript and JavaScript rely on using a forward slash (/) directory separator on all platforms (Windows uses the backslash but allows using a forward slash). As TypeScript/JavaScript code will mainly be used for web-based games and will not try to access local files, this issue will be of little practical significance.

**Interfaces.**

TypeScript implements interfaces but uses these only at compile time for type checking. The resulting JavaScript is not capable of checking the existence of an interface as such. For allowing the asset to select the bridge to be used, the asset should instead check for the interface method that needs to be called.

**Settings.**

During deserialisation of JSON data into asset settings JavaScript will only restore the data but will not recreate the methods present in the class. As a consequence computed properties will fail. A workaround is either to avoid using the methods and/or the computed values, or to copy the restored data into a newly created settings instance.

### 6.3 Issues in Java

**Characters.**
Because of the different directory separator characters in Windows and OS X, forward slash (/) and backslash (\), respectively, portability fails. Problems can be avoided by using the File.separator field, which dynamically returns the correct separator, instead of using hardcoded separators.

Properties.

Java does not support the properties concept of C# and TypeScript, but relies on naming conventions instead (get/set methods).

Default values.

Java has no standard implementation for default value attributes. As a consequence default values have to be applied in the constructor of the settings subclasses.

6.4 Issues in C++

Characters.

Also in C++ portability across Windows and OS X is hampered by the different directory separator characters. Problems can be avoided by creating conditional code for pre-processor directives that provide hard-coded separators tuned to the platform of compilation.

Properties.

C++ does not support the concept of properties. Instead it relies on naming conventions (get/set methods).

Default values.
Like Java, C++ has no standard implementation for default value attributes, so default values have to be applied in the constructor of the settings subclasses.

Singleton.

During testing the new C++ 2013 singleton syntax led to crashes in Visual Studio, so it had to be replaced with a more traditional double checked locking pattern.

Web services

Although web service calls, e.g. with REST or SOAP, are a well-established approach to client-server communications and other remote communications, problems may arise because of slight semantic differences on different platforms. For instance, the popular JSON data format embedded in the web service protocols may suffer from this ambiguity, in particular with respect to the symbols of decimal separator, thousands separator, list separator, quote, data-time formats, null versus undefined, character encoding (e.g. UTF-8), prohibited characters in filenames, the line feed and carriage return. XML-converted data are less sensitive to these issues.

7 In conclusion

In this paper we have reported the design of the RAGE architecture, which is a reference architecture that supports the reuse of serious gaming technology components across different programming languages, game engines and game platforms. An asset would offer a standardized interface that can be directly implemented by a game engine (via the bridge pattern) or it uses the asset’s event manager for a publish/subscribe event. Proofs of concept in four principal code bases (C#, Java, C++ and TypeScript/JavaScript) have validated the RAGE architecture. In addition, the C# implementation of the test asset was successfully integrated in the Unity game engine, which demonstrates the practicability and validity of the RAGE asset architecture. The RAGE project will now start to develop up to 30 dedicated serious gaming assets and use these in customer-driven serious games projects. RAGE will make these assets available along with a large volume of high-quality knowledge resources on serious gaming through a self-sustainable delivery platform and social
space. This platform aims to function as the single entry point for different stakeholders from the serious gaming communities, e.g. game developers, researchers from multiple disciplines, online publishers, educational intermediaries and end-users. RAGE thus aims to contribute to enhancing the internal cohesion of the serious games industry sector and to seizing the potential of serious games for teaching, learning and various other domains.

8 Acknowledgements

This work has been partially funded by the EC H2020 project RAGE (Realising an Applied Gaming Eco-System); http://www.rageproject.eu/; Grant agreement No 644187.

9 References