RAGE reusable game software components and their integration into serious game engines

Wim van der Vegt, Enkhbold Nyamsuren, Wim Westera

Open University of the Netherlands
(wim.vandervegt, enkhbold.nyamsuren, wim.westera)@ou.nl

Abstract. This paper presents and validates a methodology for integrating reusable software components in diverse game engines. While conforming to the RAGE component-based architecture described elsewhere, the paper explains how the interactions and data exchange processes between a reusable software component and a game engine should be implemented for procuring seamless integration. To this end, a RAGE-compliant C# software component providing a difficulty adaptation routine was integrated with an exemplary strategic tile-based game "TileZero". Implementations in MonoGame, Unity and Xamarin, respectively, have demonstrated successful portability of the adaptation component. Also, portability across various delivery platforms (Windows desktop, iOS, Android, Windows Phone) was established. Thereby this study has established the validity of the RAGE architecture and its underlying interaction processes for the cross-platform and cross-game engine reuse of software components. The RAGE architecture thereby accommodates the large scale development and application of reusable software components for serious gaming.

Keywords: Serious game·reuse·software component·integration·game engine·interoperability·RAGE

1 Introduction

Although games for learning have received attention from researchers and educators for several decades, the uptake of these "serious games" in schools and corporate training has been quite limited. Unlike the leisure game industry, which is an established industry dominated by major non-European hardware vendors (e.g. Sony, Microsoft and Nintendo) as well as major publishers and a fine-grained network of development studios, distributors and retailers, the serious game industry is scattered over a large number of small independent studios. This fragmentation goes with limited interconnectedness, limited knowledge exchange, limited specialisations, limited division of labour and an overall lack of critical mass [1,2]. Moreover, driven by the successes of leisure games, quality standards of serious games as well as their production costs tend to increase substantially, which raises barriers to serious game adoption [3].

In 2014, the European Commission has designated serious games as a priority area in its Horizon 2020 Programme for Research and Innovation. It envisions a flourishing serious games industry that helps to address a variety of societal challenges in

education, health, social cohesion and citizenship, and at the same time stimulates the creation of jobs in the creative industry sector. Funded by the Horizon 2020 Programme, the RAGE project is a technology-driven research and innovation project that will make available serious game-oriented software modules (software assets) that game studios can easily integrate in their game development projects. Serious games studios would then benefit from reusing state-of-the-art technologies, while their development would become easier and faster, and upfront investments during development would be reduced.

In the RAGE project up to 40 advanced software assets are anticipated. These assets cover a wide range of functionalities particularly tuned to the pedagogy of serious gaming, e.g. player data analytics, emotion recognition, stealth assessment, personalisation, game balancing, procedural animations, language analysis and generation, interactive storytelling, social gamification and many other functions. One of the major challenges of RAGE is to ensure portability of the software assets across the wide diversity of game engines, game platforms and programming languages that game studios have in use. In the game industry game engines are the focal point of reuse [4]. They provide core libraries providing functionalities common to most games (e.g., rendering, scripting, networking). To support reusability within specific genres of games, game engines are supplemented with stores of plug-in "assets" [4]. These stores mostly concentrate on reuse of 2D/3D models and animation scripts. In rare occasions, software libraries with auxiliary functionalities are also available. For example, the store for the Unity game engine offers assets for game data analytics (https://www.assetstore.unity3d.com/). However, such libraries are bound to the architecture of the target engine. Furthermore, there is a lack of assets with explicitly pedagogical purposes.

RAGE has addressed these issues by devising a component-based architecture [5,6] that preserves the portability of assets and that supports data interoperability between the assets [7]. In [7] the principles and constituents of the RAGE asset architecture have been described in detail and proofs of concept were presented that demonstrate its compliance with the following basic requirements: 1) minimal dependencies on external software frameworks and 2) interoperability between assets, and 3) portability of assets across different programming languages. This paper focuses on an additional requirement: the portability across different platforms, hardware and game engines. For the validation an existing RAGE Asset is used, the Heterogeneous Adaptation Asset (HAT).

We will first summarise the main features of the RAGE architecture and the set of communication modes it supports. Next, we will introduce the HAT asset and an exemplary game that were used for investigating the asset integration. Thereafter we will discuss the integration of the asset and the game and describe the principal asset classes and the main interaction processes that are required for system integration. Finally, we will discuss the portability of the HAT-asset to other game engines and verify the portability to diverse delivery platforms.

2 The RAGE architecture

The RAGE asset architecture defines a component model (Figure 1) for creating a reusable plug-and-play asset. The component model conforms to common norms of Component-Based Development [5,6,7]: 1) a component is an independent and replaceable part of a system that fulfils a distinct function; 2) a component provides information hiding and used as black box; 3) a component communicates strictly through a predefined set of interfaces that guard its implementation details.

The RAGE architecture [7] distinguishes between server-side assets and client-side assets. Remote communications of server-side assets with either the game engine (client) or a game server are readily based on a service-oriented architecture (SOA) using the HTTP-protocol (e.g. REST), which offers platform-independence and interoperability among heterogeneous technologies. In contrast, client-side RAGE assets are to be integrated with the game engine and are likely to suffer from incompatibilities. Therefore, the RAGE (client) asset architecture relies on a limited set of well-established software patterns and coding practices aimed at decoupling abstraction from its implementation. This decoupling facilitates reusability of an asset across different game engines with minimal integration effort. Figure 1 displays the UML class diagram of the RAGE asset architecture [7].

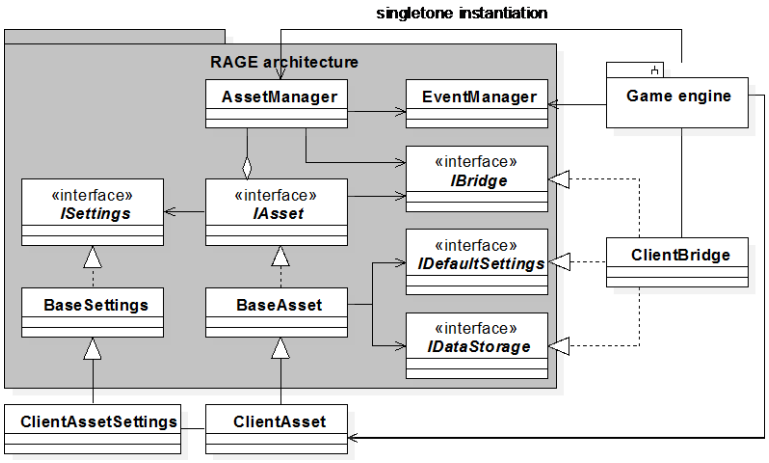

Fig. 1. Class diagram reflecting the internal structure of a client-side software asset.

First, the asset does not provide any functionality related to the game user interface as to avoid platform-dependent code. The asset just provides processing functionality by returning processed data to the game engine (e.g. calculating user performance metrics based on logged behaviours). Second, since various assets may be linked together to express aggregates, a coordinating agent is needed: the Asset Manager, which is implemented as a Singleton, is needed for registration of the assets. It exposes methods to query these registrations. Also, the Asset Manager centralises shared code that is commonly used by multiple assets, such as the name and the type of the game en-

gine, or user login/logout info for assets that would need a user model. For such data, the Asset Manager is the single interaction point with the outside game engine, and thus avoids duplicating code. Third, for allowing an asset to call a game engine method, the Bridge software pattern [8] is used, which is platform-dependent code implementing an interface. Alternatively, the communications could use the Publish/Subscribe pattern [9,10] through the Event Manager, which is initialised by the Asset Manager during its Singleton instantiation. Fourth, the asset offers basic capabilities of storing configuration data (settings), be it delegated through the Bridge to the game engine. Storage also includes localisation data (string translation tables), version information and dependency information (dependency on other assets' versions). Fifth, assets largely rely on the programming language's standard features and libraries to maximise the compatibility across game engines. Therefore, assets could thus delegate the implementation of required features to the actual game engine, for example the actual storage of runtime data.

3 Communications between assets and the game engine

For allowing an asset or its sub-components to communicate with the outside world (e.g. with other assets, the game engine or a remote service), well-defined interfaces are needed. The RAGE architecture support 4 different communication modes, which are connected with asset registration and the use of RAGE architecture methods, the use of game methods, using web services and using Publish/Subscribe events, respectively. These modes will be summarised below are explained below at a generic level. In section 6 we will provide the implementation details of asset registration, the reuse of RAGE architecture methods and the reuse of game engine methods.

3.1 Communications with the Asset Manager and other assets

The Asset Manager has the central role in registering assets. Such registration is needed, because for communication the game engine should be able to locate the assets, as much as each asset should be able to locate other assets. Principal steps of the registering process are:

- Asset creation
  Upon execution the game engine creates the asset by calling its constructor.
- Locating or creating the Asset Manager
  After its creation the asset tries to locate the Asset Manager. If no Asset Manager instance can be found, it creates the instance as a Singleton.
- Asset self-registration
  The asset registers itself at the Asset Manager by the name of its class. In return, it receives a unique identifier, so that multiple instances of the same class can be kept apart.
- Asset ID exchange
  The unique identifier is then returned to the game engine for later use.

The Asset Manager provides an interface for querying this registration of assets. An asset can also query the Asset Manager for other assets by their class names when inter-asset communication becomes necessary.

3.2 Communications through a game method call

For allowing an asset to call a game engine method a Bridge [8] is used. The Bridge includes platform-dependent code that implements one or more interfaces. The following actions are required:

- Bridge creation
  The game engine creates a Bridge and registers it with either a specific asset or with the Asset Manager. The asset can access its own Bridge or the Asset Manager´s Bridge to further communicate with the game engine.
- Calling the game engine
  Upon calling a game engine method, the asset would look for a suitable interface from the Bridge, which then forwards the method call to the game engine.
- Receiving the response of game engine method
  The game engine returns the method's response to the Bridge, which forwards it to the asset.

Overall, the Bridge pattern allows assets to call game engine methods while hiding the game engine's implementation details from the asset. Additionally, polymorphism is supported by allowing a Bridge to implement multiple interfaces, or allowing an asset to access multiple Bridges that implement different interfaces. The asset may identify and select a suitable Bridge and use its methods or properties to get the pursued game data.

3.3 Communications through a web-service call

The Bridge can also be used for the communications of client-side assets with remote services through web services. Obviously, this also applies for client-side assets calling server-side assets. The communication includes the following elements:

- Bridge creation
  If the Bridge was not instantiated yet, the game engine should create it and make it available to the asset.
- Using an adapter
  The Bridge uses an Adapter [11] provided by the game engine, which thus removes the dependency of the asset on specific communication protocols used by remote services, thereby allowing a greater versatility of the asset.
- Sending a request
  In turn the asset could send a request (e.g. load or save data) to the Adapter, which is then to be translated to a suitable format (e.g. REST) and sent to the web service.
- Receiving a response

Eventually, the web service would return its response, which is then received and processed by the asset.

Obviously, the communication with remote services assumes an online connection. 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.

3.4 Communications through a Publish/Subscribe event

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, which could be received by multiple assets.

- Creation of an Event Manager
  An Event Manager is needed, which is a centralised class that handles topics and events. It is initialised by the Asset Manager during its Singleton instantiation.
- Registration of an event
  The game engine registers a publication event at the Event Manager, for instance the broadcast of player performance data, or any other required state data from the game.
- Subscription to the event
  An asset that wants to use such data for further processing would subscribe to the registered event.
- Receiving updates
  Any publication or update of the event by the game engine will then be broadcast by the Event Manager. The assets that have subscribed to the particular event will receive the data and act upon it.

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. The communication can go both ways: asset and the game engine can be either publishers or subscribers. The Publish/Subscribe pattern of communication is more suitable for (asynchronous) broadcasting to multiple receivers than the Bridge-based communication, which realises bilateral communications only.

4 The Heterogeneous Adaptive Gaming asset (HAT)

The Heterogeneous Adaptive Gaming asset (HAT) can be used for real-time adaptation of game features to player skills. The current version of the HAT asset supports adapting game difficulty to player's expertise using the CAP algorithm [12]. The CAP algorithm is based on the Elo rating system [13] that was originally developed to dynamically calculate and match expertise levels of two chess players. Similar to the Elo algorithm, CAP does not require pre-testing to estimate difficulty of items. Instead,

CAP is capable of on the fly estimation of item difficulty and player's expertise parameters. The CAP algorithm is successfully being used in a wide array of games ranging from simple arithmetic games [14] to complex problem solving games such as Mastermind [15].

The HAT asset assumes that a player plays through a sequence of one or more game scenarios. The game delegates the choice of the scenarios to be played to the HAT asset, which after each scenario adapts game difficulty to the player's expertise level. Quantitative ratings need to be assigned to both a player's expertise or skills level, and to the game scenarios' difficulties. After each played scenario, the HAT asset updates the player's expertise rating by taking into account a Boolean value indicating whether the player failed or succeeded in a scenario and the time needed by the player to finish the scenario. If the player performed better than expected then the expertise rating is increased, otherwise it is decreased. Based on the updated player's expertise rating, the HAT asset returns the most suitable difficulty level for the next scenario to the game. For this decision, the HAT asset uses a prefixed probability value indicating the probability that the player finishes the scenario successfully. Based on previous research this probability threshold was set to 0.75, as to balance the challenge provided by the game and player's motivation to continue to play [12], [16]. The player is initially assigned a low expertise rating and, therefore, will be provided with easier scenarios. However, as the player improves by gaining expertise, the expertise rating increases, and more difficult scenarios will be presented. Through this iterative process, the HAT asset ensures that the player is always given a reasonable amount of challenge even if the player gradually improves.

5 The TileZero game

The TileZero game (Figure 2) is a derivative of the popular turn-based board game Qwirkle (released by MindWare, http://www.mindware.com). In recent years, Qwirkle has captured interests of educational researchers for its potential use in developing children's spatial, mathematical, and fluid reasoning skills [17]. The game contributes to capacities to think logically and solve problems from different perspectives. It requires from a player a strategic reasoning ability to form, compare and choose from alternative combinations of moves. Finer grained skills include spatial manipulation of tiles in mind, mental arithmetic of in-game scores, and tactical consideration of other players' possible moves. The same considerations apply to the TileZero game. As the game has simple mechanics and rules that are easy to implement and control, it is a good candidate for testing the asset integration.

The mechanics of TileZero revolves around combining tiles into a sequence. Each tile has a picture of a coloured shape. There are six distinct colours and six distinct shapes resulting in 36 unique tiles. With three copies of each unique tile the total number of playable tiles is 108. Tiles that have not been used yet, are kept in a bag, and players cannot see them.

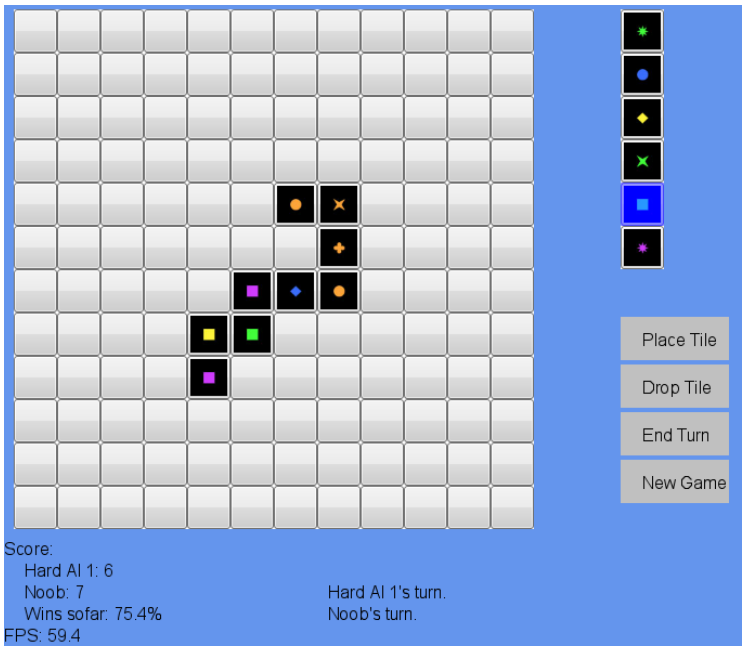

Fig. 2. A screenshot of the TileZero game against Hard AI Player.

TileZero can be played with two to four players. A match starts with three random tiles put in a sequence on a board. Next, each player receives a set of six random tiles. Once tiles are distributed, players start taking turns. During their turn, the players can place one or more tiles on the board and replenish their set from the bag. The player has to follow several rules for tile placement. First, a tile should be placed next to another tile already on the board. Second, any sequence of tiles on the board should have either the same colour and different shapes or vice versa. Third, a player can only place tiles of either the same colour or same shape during a turn. A player receives a score for each tile placed on a board. The score is based on the length of the sequence that the tile forms on the board. The game ends if the bag of tiles is empty and the player put his last tile on the board. The player with the highest score is the winner.

In our implementation of TileZero, a human player plays against one of six available AI opponents. An AI opponent is considered as a scenario. AI opponents have different strategies and thus provide different degrees of challenge to the man player. The six AI opponents in an increasing order of difficulty are Very Easy AI, Easy AI, two versions of Medium AI, Hard AI and Very Hard AI. The TileZero was extended with the HAT asset to match difficulty of an AI opponent to the player's demonstrated expertise level. A beginner player is assigned a low initial rating and therefore, the first few matches will involve Very Easy or Easy AIs. However, as player gains expertise, the HAT asset starts gradually introducing more challenging AIs.

6 Integrating assets with game engines

The TileZero game was implemented on MonoGame v3.0, which is a portable open-source Mono-based and OpenGL-based game engine (monogame.net) [18]. Both TileZero and the HAT asset were written in C# using Visual Studio 2013. The integration of the HAT asset and the TileZero game was based on usage of the Asset Manager and the Bridge pattern for calling game engine methods. The implementation of Web Services and Publish/Subscribe patterns were not needed. In the next sections we will first explain game how to setup game code in MonoGame to be compliant with the RAGE architecture. Secondly, the principal classes required for this integration will be explained. Third, the main interaction processes that are required for system integration and the reuse of libraries are described. Finally, we will discuss the portability of the HAT-asset to other game engines and verify the portability to diverse delivery platforms.

6.1 MonoGame implementation of TileZero

MonoGame uses a simple architecture of 5 methods being called.

- Initialize
- LoadContent
- Update
- Draw
- UnloadContent

When the game starts, the Initialize method is called and the main classes are created and configured. Then the LoadContent method is called which covers the loading of the tile bitmaps. Next MonoGame enters a loop of repetitively calling the Update and Draw methods around 60 times/sec. In the Update method the keyboard and mouse states are examined and processed and forwarded to the game logic. In the Draw method the game model is rendered onto the screen. Finally, when the loop has ended (the end of the game), an UnloadContent method is called to free up previously loaded content.

Instead of directly implementing the HAT adaptation algorithm in the MonoGame code, reuse of the HAT asset requires to declare a separate class (*HATAsset*) wrapping all HAT functionality and thus exposing a minimum number of methods needed. Importantly, the HAT asset itself can already be tested without being embedded in the game. Because the HAT asset does not directly link with the game's user interface, the TileZero game code was separated in two distinct classes, covering the game logic (*TileZeroGame* class) and the display model (*VirtualTileZeroBoard* class), respectively. The *TileZeroGame* class uses the HAT asset to select the appropriate AI for the computer player when a new match is started. It is called by the MonoGame Update method, to process keyboard and mouse input into updates of the *VirtualTileZeroBoard* class. The *VirtualTileZeroBoard* class is used by the Draw method to visualise the user interface of the game.

6.2 HAT asset integration

Figure 3 shows a (simplified) UML class diagram depicting the main classes required for the integration of the HAT asset and the TileZero game.

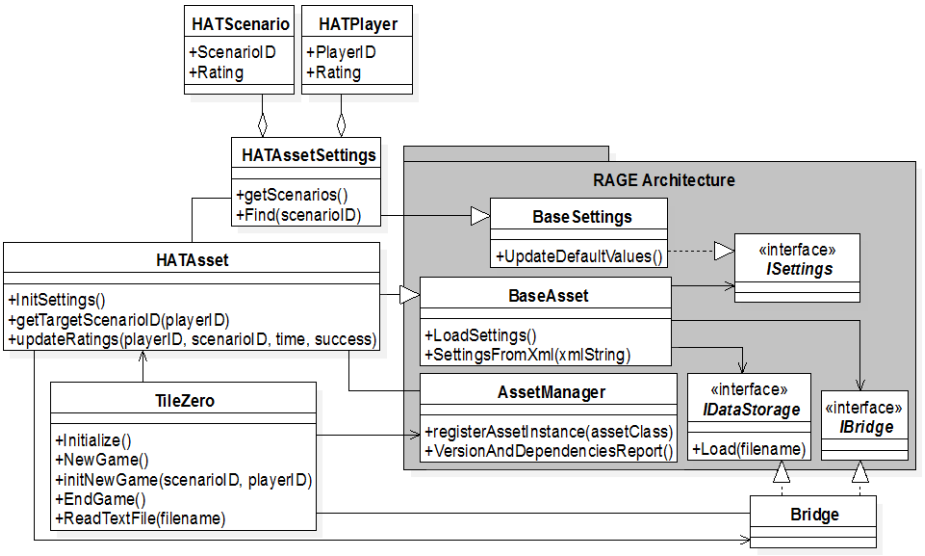

Fig. 3. Class diagram describing the integration of the HAT asset with the TileZero game.

In Figure 3, the *TileZero* class represents the game. The *HATAsset* class represents the core functionality of the HAT asset, which is the adaptation algorithm. To gain access to the standardised functionality of the RAGE architecture, the *HATAsset* class extends the *BaseAsset* class from the architecture. This enables the *HATAsset* class to communicate with the game engine (the *TileZero* class) using the *Bridge* class that implements the Bridge pattern. The Bridge pattern enables the asset to call methods from the game engine without knowing the game´s implementation details. Apart from the *IBridge* interface, the *Bridge* class can realise additional interfaces that allow an asset to delegate common functionalities to a standard library provided by a game engine. For example, the *IDataStorage* interface allows an asset to request the game engine to load or save files.

6.3 The reuse of libraries by using the RAGE architecture

Figure 4 shows the UML sequence diagram reflecting interactions between the HAT asset and the game engine.

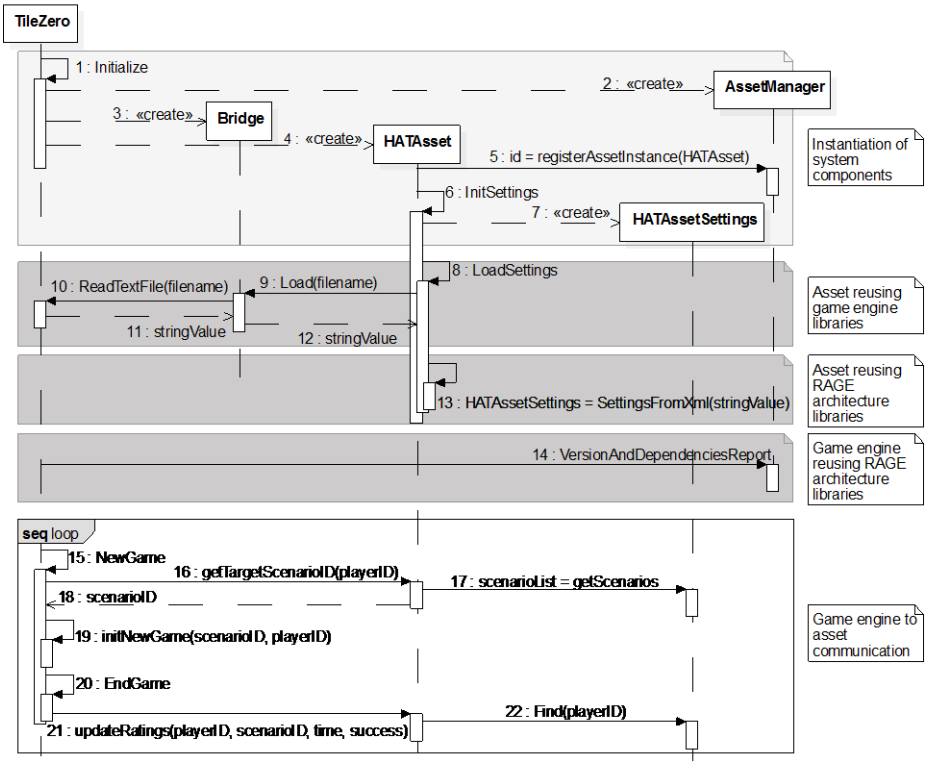

Fig. 4. UML sequence diagram depicting communication processes between the HAT asset and the game engine.

This figure shows five different communication processes, which are labelled at the right hand side. These processes will be briefly explained below, with occasional reference to Figures 3 and 4.

Instantiation of system components. During its initialisation (step 1 in Figure 4), the *TileZero* class instantiates all other components of the system. First, a Singleton of the Asset Manager is created (step 2). Next, an instance of the *Bridge* class is created (step 3) and referred to a newly created instance of the *HATAsset* class (step 4). During initialisation, the *HATAsset* class performs two main operations. First, it registers itself with the Asset Manager and receives a unique id (step 5). Next, it instantiates *HATAssetSetting* class (steps 6 and 7) to load and manage player and scenario settings.

An asset reusing game engine libraries. The HAT asset uses the *IDataStorage* interface to load the asset's settings stored on a local XML file. This process is shown by steps 8 - 12 in the sequence diagram in Figure 4. The *HATAsset* requests the *Bridge*

object to load the file by its name. Contacting the *Bridge* object is a matter of calling the LoadSettings method inherited from the *BaseAsset* class. This method handles details of the call such as ensuring that the *Bridge* object has realised the *IDataStorage* interface. In turn, the *Bridge* object uses libraries from the MonoGame engine to read textual files and it returns to the *HATAsset* the content as a string value. Such delegation of generic functions to game engines has main advantages of avoiding redundancy in code functionality and unnecessarily bloated implementation of an asset software component.

An asset reusing RAGE architecture libraries. One standardised functionality in the *BaseAsset* class is to deserialise XML specified data into instances of a RAGE compliant class for managing settings. In the HAT asset, settings include lists of available scenarios and players together with relevant adaptation parameters such as ratings. These settings are managed by the *HATAssetSettings* class shown before in Figure 3. Within this class, settings for individual scenarios and players are managed as instances of the *HATScenario* and the *HATPlayer* classes respectively. For example, each scenario available in a game is identified in the HAT asset by its ID and assigned a difficulty rating. Because *HATAssetSettings* extends the *BaseSettings* class from the architecture, the HAT asset is able to use the SettingsFromXML method predefined in the *BaseAsset* class (step 13 in Figure 4). This method automatically deserialises the asset's settings from an XML format into an instance of the *HATAssetSettings*.

A game engine reusing RAGE architecture libraries. Functionalities predefined in the RAGE architecture may also be reused by different game engines. One of the core components that offer reusable methods is the Asset Manager that assists the game engine in coordinating multiple assets. The Asset Manager can keep track of all assets by ID or class name and provide basic services relevant to all assets. In this particular example, the Asset Manager is used by the game engine to verify the HAT asset's version and check if it is dependent on any additional library (step 14 in Figure 4).

Game engine to asset communication. Every time the player starts a new match, the game has to decide on the AI opponent to use in the match. The game delegates this decision to the HAT asset as it is shown through step 15 to 22 in Figure 4. The HAT asset treats each AI opponent as a scenario and tries to find one with the difficulty rating that matches the player's expertise rating. As indicated by step 16 in Figure 4, the game requests the HAT asset to return an ID of the AI opponent it should select. This request is accompanied with an ID of the player. As was discussed earlier, the HAT asset maintains players' and scenarios' ratings and IDs in the *HATAssetSettings* class. The HAT asset uses the player's ID to fetch the player´s expertise rating from the *HATAssetSettings* class. Next, it also retrieves the list of all available AI opponents (step 17). Given this information, the asset can find an ID of the AI opponent best suitable for the indicated player. This ID is returned to the game, and a new match starts (step 19). Upon completion of the match, the game requests the HAT asset to update player's rating (step 21). This request includes player and AI IDs,

duration of time the match lasted, and Boolean indication whether player succeeded over the AI opponent. The HAT asset uses these four parameters to recalculate player's expertise rating after each match.

Results of test gameplays. The TileZero and HAT asset were tested by a human player who played multiple consecutive matches against an AI opponent. The HAT asset was used to adapt the game difficulty. Initially, the player was assigned a low initial expertise level and matched against easier AIs. Figure 4 shows how the player's ratings changed during first 29 matches. The figure also depicts the type of AI opponent used in each match. Two main trends can be observed. First, the player's rating shows steady increase indicating a positive overall performance growth of the player. Second, the frequencies of AI types change during 29 matches. The first half of matches shows overall prevalence of Very Easy and Easy AIs, while the second half shows prevalence of Medium and Hard AIs. These two trends together confirm that the HAT asset worked as expected and matched game difficulty to player's expertise.

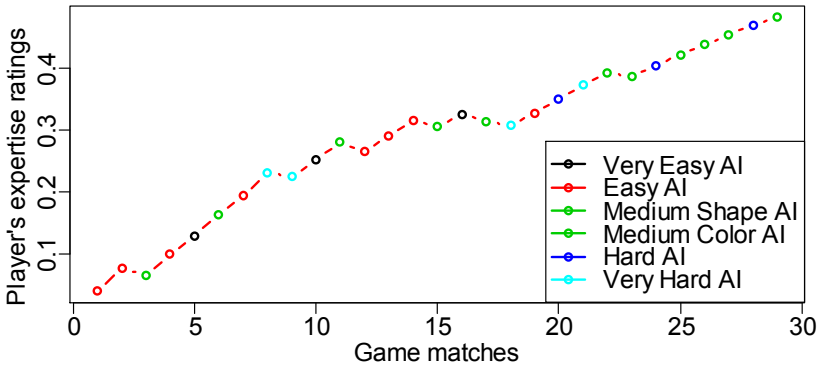

Fig. 5. Player's increasing expertise ratings during 29 matches.

6.4 Portability across game engines and platforms

The principal reason for devising the RAGE asset architecture has been to make available software components that can be reused across different game engines and different platforms. For verifying this, the TileZero project was ported from the MonoGame engine (monogame.net) to both the Unity engine (unity3d.com) and the Xamarin mobile app platform (xamarin.com), which both support the C# implementation. The HAT asset was then added and integrated with each of these new game versions. No extensive user interfaces were implemented in the Unity and Xamarin game versions, as for testing the games' functioning simple buttons for mimicking player's decisions were sufficient. Exactly, because of the decoupling of RAGE assets and the game's user interface, testing of the system integration is completely independent of the user interface. Likewise the portability of RAGE assets across diverse delivery platforms is easily delegated to the game engines´ rendering utilities, which in many cases include cross-platform delivery. Both MonoGame, Unity and Xamarin support a

large number of leading platforms, covering different operating systems and hardware configurations. Successful system integration was established for all three game engines, and proper delivery was verified for Windows desktop, iOS, Android and Windows Phone, be it not in all possible combinations. Some issues were encountered, but these could be easily solved.

First, during the coding of a mock-up game in Unity for Android, XPath could not be used for performing some basic calculations. This was caused by the Mono version that Unity uses. The issue could be solved by replacing XPath by code using the .NET *XmlSerializer* class. It should be noted that this issue is not related to the RAGE asset architecture, but to differences between the Mono and .Net frameworks used.

A second issue was located in the Bridge and occurred when trying to create and access a platform-independent directory in Unity for storing the player's performance data. It turned out the Unity does not allow for this. The Application.dataPath method only provides a read-only directory on iOS. Likewise, the Environment class cannot be used as its main target is desktop. The issue could be solved by using Application.persistentDataPath, which is read-write on all tested platforms. Thereby the *Bridge* class became portable across Unity's target platforms.

Third, in our tests we used a Xamarin Forms project, which allows for referencing to assemblies for using their projects, but it also supports direct referencing to compiled assemblies. Assemblies can be compiled either against a common .Net framework or as a portable assembly. Although Android and iOS allow for both portable (mobile) and non-portable solutions, Windows Phone only allows portable assemblies. The implies that if a Windows Phone project is present, the HAT Asset and the Asset Manager assembly need to be compiled as portable assemblies and used on all respective platform projects.

Fourth, as Unity is using an older .NET version (v3.5) it cannot handle portable libraries. Indeed, .NET version 4.5, as used in Xamarin, is required for portable libraries. Obviously the issue is not an issue of the RAGE architecture.

Fifth, as the format of Visual Studio project files is different for common .Net projects and portable projects, respectively, separate project files are needed for each type of assembly. With some small adjustments the RAGE asset sources can still be shared for both types of assemblies. Two minor coding issues surfaced and were removed. The system libraries used by portable assemblies lack support for some property attributes used in RAGE assets (Category and Description). This was solved by removing these two attributes as they are only used by an experimental configuration editor based on a PropertyGrid and not of vital importance for the game. In the portable projects the affected lines where omitted using C# compiler directives. Also, the two projects have different methods for retrieving properties by reflection. This was addressed by adding some conditional code using C# compiler directives and refactoring the code in such way (using the constructor) that it does not need reflection.

Sixth, the Bridge for multi-target Xamarin Forms projects is composed of a common part and a device specific part. For Android and iOS the Bridge implementation is straightforward. For Windows Phone, however, the preferred file I/O API is asynchronous. This requires that the code in the Windows Phone Bridge waits for the result of asynchronous calls, which could lead to a deadlock. This issue was solved by

including async helper methods that wait for their result in the synchronous interface in a correct way.

Seventh, if an asset's Bridge interfaces such as *IDataStorage* are to be used for all platforms and engines, including Unity, they must be coded synchronously, because the async keyword was included only after the .Net 3.5 framework, and is thus not available in Unity.

Finally, some minor portability issues have been reported before, e.g. confusion of separator characters (e.g. "/" versus "\"), conversion of debug symbol files for Unity, and the compilation of embedded resources in Unity [7].

7 Conclusion

In this study, we have provided further evidence for the validity of the RAGE game asset architecture. We have demonstrated that client-side game technology components that are compliant with the RAGE architecture can be easily integrated with existing game engines and allow for reuse across different engines and platforms. The power of the RAGE architecture is not limited to the potential reuse of assets, but is also based on the efficient reuse of existing libraries, either from the RAGE architecture or from the game engine in use. To maximise the reusability of assets among different games, the assets do not directly link with the game´s user interface and exchange only the basic forms of information with the game engine. In the HAT asset, for example, the code of the asset responsible for difficulty adaptation requires only the exchange of string IDs and a few numerical values such as the duration of a task. The qualifies the integration of RAGE assets as "lightweight", which may promote its adoption.

It should be noted that we have tested the integration of C# coded assets only. In a previous study, we have tested and validated the RAGE architecture by implementing a dummy asset prototype also in C++, Java and TypeScript (JavaScript). Establishing the ecological validity for those languages by integrating real assets in real games for various game engines and platforms needs further investigation. Moreover, in the current study for C# some issues surfaced, be it minors issues. Yet, it demonstrates that cautious and prolonged investigation is needed of the practical factors and conditions that might corrupt seamless asset integration, both for C# and other languages. So far, this study has established the validity of the RAGE architecture and its underlying interaction processes for the cross-platform and cross-game engine reuse of software components. The RAGE architecture thereby accommodates the large scale development and application of reusable software components for serious gaming.

Acknowledgement. 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.

References

1. Stewart, J., Bleumers, L., Van Looy, J., Mariën, I., All, A., Schurmans, D., Willaert, K., De Grove, F., Jacobs, A., Misuraca, G.: The Potential of Digital Games for Empowerment and Social Inclusion of Groups at Risk of Social and Economic Exclusion: Evidence and Opportunity for Policy. Joint Research Centre, European Commission, Brussels (2013).
2. García Sánchez, R., Baalsrud Hauge, J., Fiucci, G., Rudnianski, M., Oliveira, M., Kyvsgaard Hansen, P., Riedel, J., Brown, D., Padrón-Nápoles, C.L., Arambarri Basanez, J.: Business Modelling and Implementation Report 2, GALA Network of Excellence, www.galanoe.eu (2013).
3. Warren, S. J., Jones, G.: Overcoming Educational Game Development Costs with Lateral Innovation: Chalk House, The Door, and Broken Window. The Journal of Applied Instructional Design, 4 (1), 51-63, 2014.
4. Bergeron, B. (2006). Developing serious games. Charles River Media, Hingham MA.
5. Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Sea-cord, R., Wallnau, K.: Technical concepts of component-based software engineering, Volume II. Carnegie Mellon University, Software Engineering Institute, Pittsburgh (2000).
6. Mahmood, S., Lai, R., Kim, Y.S.: Survey of component-based software development. IET software, 1(2), 57-66, 2007.
7. Van der Vegt, G.W., Westera, W., Nyamsuren, N., Georgiev, A., Martinez Ortiz, I.: RAGE architecture for reusable serious gaming technology components. To appear in the International Journal of Computer Games Technologies (2016).
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software, pp. 171-183. Pearson Education, London (1994).
9. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. In: Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP '87), pp. 123–138, 1987.
10. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys (CSUR), 35(2), 114-131, 2003.
11. Benatallah, B., Casati, F., Grigori, D., Nezhad, H.R.M., Toumani, F.: Developing adapters for web services integration. In: Pastor, O., Falcão e Cunha, J. (Eds.) Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings (pp. 415-429). Springer, Berlin Heidelberg (2005).
12. Klinkenberg, S., Straatemeier, M., Van der Maas, H.L.J.: Computer adaptive practice of maths ability using a new item response model for on the fly ability and difficulty estimation. Computers & Education, 57 (2), 1813-1824, 2011.
13. Elo, A.E.: The rating of chess players, past and present (Vol. 3). Batsford, London (1978).
14. Van der Maas, H.J.J., van der Ven, S., van der Molen, V.: Oefenen op niveau: het cijferspel in de Rekentuin. Volgens Bartjens 3, 12-15, 2014.
15. Gierasimczuk, N., Van der Maas, H.L., Raijmakers, M.E.: An analytic tableaux model for Deductive Mastermind empirically tested with a massively used online learning system. Journal of Logic, Language and Information, 22 (3), 297-314, 2013.
16. Eggen, T. J., Verschoor, A.J.: Optimal testing with easy or difficult items in computerized adaptive testing. Applied Psychological Measurement, 30(5), 379-393, 2006.
17. Mackey, A.P., Hill, S.S., Stone, S.I., Bunge, S.A.: Differential effects of reasoning and speed training in children. Developmental Science, 14(3), 582-590, 2011.
18. Pavleas, J., Chang, J. K. W., Sung, K., Zhu, R.: Learn 2D Game Development with C#, (pp. 11-40). Apress, New York (2013).