View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Open University of the Netherlands Research Portal

Realising and Applied Gaming Ecosystem

Research and Innovation Action

Grant agreement no.: 644187

D1.4 – First version applied gaming asset methodology

RAGE – WP1 – D1.4

Project Number	H2020-ICT-2014-1
Due Date	M18: 31 July 2016
Actual Date	July 2016
Document Author/s	Westera, W., Stefanov, K., Van der Vegt, W., Nyamsuren, E., Moreno Ger, P., Freire, M., Georgiev, A., Grigorov, A., Boytchev, P., Griffiths, D., Hollins, P., Fernández Magnón, B., & Martinez Ortiz, I.
Version	1.0 intermediate version for submission
Dissemination level	PU
Status	Final
Document approved by	WimW

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644187

Document Version Control			
Version	**Date**	**Change Made (and if appropriate reason for change)**	**Initials of Commentator(s) or Author(s)**
0.1	July 2015	Initial version D1.1 draft	WimW
0.2	August 2015	Revision after input WP1, particularly SU	WimW, Krassen
0.3	May 2016	Renamed to D1.4 for preparing intermediate submission to project review	WimW
0.4	June 2016	Integration of contributions from Krassen, Eric, Pablo	WimW
0.5	June 7 2016	Inclusion of chapter 6 (Dai) Comments from Pavel, Krassen	WimW
1.0	July 20 2016	Processing of internal reviewers' comments	WimW

Document Change Commentator or Author		
Author Initials	**Name of Author**	**Institution**
WimW	Wim Westera	1.OUNL

Document Quality Control			
Version QA	**Date**	**Comments (and if appropriate reason for change)**	**Initials of QA Person**
0.5	July 4	Internal review	Christina, Alexander
0.5	July 4	Internal review	Johan

TABLE OF CONTENTS

Page

EXECUTIVE SUMMARY **6**
1 INTRODUCTION **7**
1.1 Identifying the problem space of asset creation, delivery and maintenance 7
1.2 Challenges and principles 9
1.3 Structure of this document 11
2 MAIN CONCEPTS **12**
2.1 Terminology 12
3 BASIC NEEDS ASSESSMENT **14**
3.1 Method 14
3.2 Results 14
4 RESTRICTING THE TECHNOLOGY RANGE **16**
4.1 Method 16
4.2 Results 16
4.3 Conclusion 16
5 ASSET SYSTEM ARCHITECTURE **18**
5.1 Architecture requirements 18
5.2 The RAGE Component-Based System Architecture 18
5.3 RAGE assets 19
5.3.1 The internal structure of a client-side Asset Software Component 20
5.3.2 Technical validation of the client-side asset architecture 21
5.4 Server-side architecture 21
5.4.1 A2: Authentication and Authorization Asset 22
5.4.2 Simple User Model (SUM) and cross-asset communication 23
6 OPEN INTEROPERABILITY STANDARDS **24**
6.1 Goals 24
6.2 Concepts 24
6.3 Mandation framework 24
6.4 Target outcome 1: Strategy for inter-asset interoperability 24
6.5 Target outcome 2: Strategy for interoperability with external systems 24
7 DESIGNING A SHARED METADATA SET **26**
8 ASSET REPOSITORY INFRASTRUCTURE **29**
8.1 Front-end tools to access the software asset repository 31
8.1.1 RAGE Taxonomy tools 31
8.1.2 RAGE metadata editor 31
9 RAGE ASSET PACKAGING **32**
10 ASSET DEVELOPMENT AND MAINTENANCE **33**
10.1 Asset quality assurance 34
10.2 Asset tools and infrastructure 34
10.2.1 Coding tool 34
10.2.2 Metadata tools 35
10.2.3 Configuration and authoring tool 35
10.2.4 Source code management tool 35
10.2.5 Asset packaging, storage and validation tool 35
10.2.6 Asset distribution and support tool 36
10.2.7 Installation and integration tools 36
11 ASSET SOFTWARE LICENSES **37**
12 LIST OF ANNEXES **38**
Annex 1 Needs interviews 38
Annex 2 RAGE Technology Report 38
Annex 3 RAGE Asset System Architecture 38
Annex 4 RAGE reusable game software components and their integration into serious game engines 38
Annex 5 RAGE Vocabulary and metadata categorisation 38
Annex 6 RAGE Metadata Model 38

Annex 7 RAGE Software Asset Model and Metadata Model ... 38
Annex 8 RAGE Asset Repository Infrastructure ... 38
Annex 9 RAGE Asset Package ... 38
Annex 10 Software licencing ... 39
13 REFERENCES ... 40

LIST OF FIGURES

Figure 1. The asset problem space ... 8
Figure 2. Outline of the RAGE asset-based architecture and its diverse data communication routes. ... 19
Figure 3. Exemplary lay-out of a RAGE Asset ... 19
Figure 4. Internal structure of the asset software component (Van der Vegt et al., 2016a) ... 20
Figure 5. Overall server-side architecture. All remote communications necessarily pass through the A2 asset. ... 22
Figure 6. Internal structure of the A2 asset. ... 22
Figure 7. The RAGE metadata schema ... 27
Figure 8. Asset Repository Architecture ... 30
Figure 9. The Asset Package ... 32

LIST OF TABLES

Table 1. Description of RAGE metadata schema elements ... 28

LIST OF ABBREVIATIONS

A2	RAGE's Authentication and Authorization Asset
ADL	Advanced Distributed Learning: US-based creator of e-learning standards organisation, e.g. xAPI
ADMS	Asset Description Metadata Schema specified by the World Wide Web Consortium (W3C)
API	Application Programming Interface
BSD-2	Berkeley Software Distribution open software licence
CA	Consortium Agreement
CRUD	Basic content manipulation features: create, read, update, delete
CSS	Cascaded Style Sheet: a style sheet language for markup language
DoA	Description of Action
EMB	RAGE's Executive Management Board
GitHub	A popular open software community platform (github.com)
IDE	Integrated development environment
IEEE	Institute of Electrical and Electronics Engineers
IEEE LOM	Learning Object Metadata standard from IEEE
IMS CP	IMS Content Packaging: an e-learning content standard
IMS LTI	Learning Tools Interoperability standard from the IMS Consortium
Json	JavaScript Object Notation: a data-interchange format
LOM	Learning Object Metadata standard from IEEE
LRS	Learner Record Store: xAPI-based data storage system
MIT	Open software licence from the Massachusetts Institute of Technology
OAI-PMH	Open Archives Initiative Protocol for Metadata Harvesting
OS	Open Source software
QA	Quality Assurance
RAS	IBM's Reusable Asset Specification

RDF	Resource Description Framework: a standard model for data interchange on the We
REST	Representational State Transfer, used for web services
RMS	RAGE Metadata Schema
SDK	Software Development Kit
SKOS	Simple Knowledge Organization System: W3C recommendation for representing thesauri, classification schemes, taxonomies, subject-heading systems, or any other type of structured controlled vocabulary.
SMB	RAGE's Strategic Management Board
SOA	Services-Oriented Architecture
SPARQL	RDF-based semantic query language for databases
SUM	Simple User Model: approach toward shared storage of user data across the RAGE asset framework
TEL	Technology-Enhanced Learning
UML	Unified Modeling Language
VLE	Virtual Learning Environment
W3	World Wide Web Consortium
WP	Work Package
xAPI	Experience API, e-learning data standard for the exchange of content and events
XML	Extensible Markup Language, a text-based text-based format used to share data on the World Wide Web and across other IT systems.
XSD	XML Schema Definition: defines the elements of an XML data file

EXECUTIVE SUMMARY

This deliverable (D1.4) is an intermediate document, expressly included to inform the first project review about RAGE's methodology of software asset creation and management. The final version of the methodology description (D1.1) will be delivered in Month 29.

The document explains how the RAGE project defines, develops, distributes and maintains a series of applied gaming software assets that it aims to make available. It describes a high-level methodology and infrastructure that are needed to support the work in the project as well as after the project has ended. Based on this WP1 deliverable five scientific papers were prepared, three of which have been accepted-published (cf. Annexes 1,3,4), and two are under review.

RAGE cannot directly adopt and reuse existing approaches and methodologies used by game engine vendors and game development companies, as RAGE focuses on 1) software assets rather than media assets, 2) the integration of assets in multiple game platforms, and 3) linking multiple assets together for enhanced functionality (e.g. combining player data collection and game adaptation).

This document makes the challenges and principles of the asset methodology explicit and uses these as the basis for the methodology design. Detailed elaborations are provided in 5 scientific papers: 3 of the papers have been published, 1 paper is accepted for publication and 1 paper is still under review at the time of writing. The published or accepted papers are included in the annexes.

An in-depth analysis of the technical landscape of game engines, platforms and programming languages is presented and used to restrict asset development to a few primary code bases that still would allow to reach out to a maximum number of game developers and their platforms. RAGE will particularly focus on C# and TypeScript (typed JavaScript).

An asset system architecture was designed to support both server-side assets and client-side assets. Platform and hardware dependencies were avoided as much as possible as to achieve maximum portability between game engines, programming languages and platforms. Server-side assets will be using web services (Rest). The client-side asset architecture was validated for multiple platforms and languages (java, C++, as well as C# and TypeScript/JavaScript).

Interoperability standards and specifications have adopted a case-based approach, focussing on asset interoperability in the RAGE pilots. Main focus is on shared nomenclature across assets. Interoperability with external systems focuses on the application of xAPI. A VLE demonstrator is in being prepared with the Hull College Group.

An asset metadata XML schema was designed to accommodate search in the asset repository, and to include dependencies, software versions, ownership and licensing information. The schema is based on a core subset of RAS and extends it with elements from ADMS, IEEE LOM and metadata related to the applied games domain.

The RAGE asset development methodology is presented, while it assumes neutrality towards different software development environments, programming languages and methodologies. Starting points for asset quality assurance have been specified for further elaboration and implementation. The asset repository architecture is presented as well as the approach toward data-interoperability.

After detailed analysis the Apache version 2.0 license has been proposed as the default license to be granted to RAGE software assets, because it offers protection and openness, and it allows for commercial exploitation. The proposal has been officially adopted and confirmed by the RAGE Strategic Board.

1 INTRODUCTION

Repository, assets, asset development and asset management
One of the key ideas of RAGE is to make available an asset repository system that allows for developing, sharing and reusing applied gaming assets. The assets are advanced game technology components (software), enriched and transformed to support applied game development. The assets cover pedagogically oriented functions that support game-based learning, particularly the capturing and analysis of user data and the support of strategic interventions and social representations in the game (cf. Objective 1 in the Description of Action). The assets go with value adding services and attributes that facilitate their use: instructions, tutorials, examples and best practices, instructional design guidelines and methodologies, connectors to major game development platforms, data capacity, and specific content authoring tools/widgets for game content creation.
The asset repository system will be embedded in a community environment (the RAGE Ecosystem) that will offer centralised access to a wide range of additional applied gaming resources, e.g. training materials, best practices, services, user-posted products, etc. Essentially for creating a sustainable community system RAGE will tap on to the intelligence of the crowd and eventually provide a set of asset creation aids that allow external developers to create and upload their own assets to the repository system.

Purpose of this document
This document explains how the RAGE project defines, develops, distributes and maintains a series of applied gaming software assets that it aims to make available. It describes a high-level methodology and infrastructure that are needed to support the work in the project as well as after the project has ended. This deliverable is an intermediate document, expressly included to inform the first project review. The final version (D1.1) will be available in Month 29.
.

1.1 Identifying the problem space of asset creation, delivery and maintenance

The asset repository system will be a central component of the RAGE Ecosystem portal. Essentially the repository system is composed of two separate repositories, both compliant with the same asset representation model, but addressing different user groups:

- The development repository
  This is a platform to be used for the development and maintenance of assets, in the following referred to as “the repository”. It is equipped with tools and services for asset creation and maintenance.
- The delivery repository
  This is the main asset publication platform (the ecosystem’s “digital library”), which allows game developers to find and download certified assets that comply with well-defined quality requirements.

The transition of assets from the development repository to the delivery repository is subjected to a well-defined workflow, which aims to preserve required quality standards. The repositories store the separate constituents of the assets and their metadata, and allow for creating suitable packages for distribution.

Figure 1 depicts the two repositories along with the three main user roles: 1) asset developer, 2) game developer (typically an end-user), and 3) repository manager:

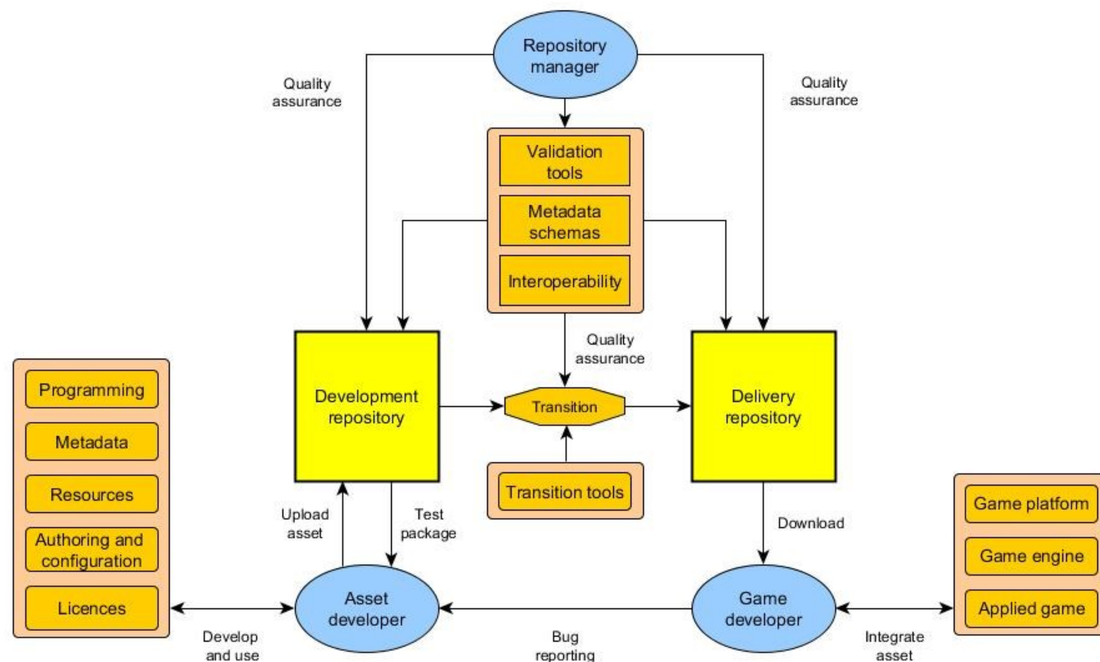

Figure 1. The asset problem space

1) The asset developer
The asset developer develops the asset code, and collects or creates additional artefacts, such as manuals, tutorials, licences and tools, and uploads these to the development repository. For coding, the asset developer can use his or her own development platform and decide to use a separate online software community platform (e.g. Github) for software management and maintenance. Metadata of the asset will be created in the RAGE development repository, which includes tools for this. In all cases the asset developer should comply with general requirements regarding asset development, testing, the provision of metadata and the interoperability of the assets across multiple game platforms. Although the publication of an asset in the development repository goes with some basic checks and balances, the assets need not include fully operational production versions but could also include early prototypes, truncated versions or untested versions. For this, the metadata will include a special maturity tag to indicate the developmental status of the asset. The publication of preliminary asset versions allows for the early involvement of stakeholders, e.g. other asset developers and game studios, who could be engaged in testing and reviewing from the start. The transition of an asset from the development repository to the delivery repository makes more stringent requirements in order to preserve the assets' quality. This transition will go with several validation checks, e.g. validation of the metadata, requirements with respect to interoperability, the completeness of the package, and test scenarios and reports. Technically the transition is automated via an OAI-PMH harvesting process. Only validated assets are admitted to the delivery repository. The asset developer is considered the legal owner of the asset (although the owner may be a different individual), who is responsible for copyright clearance and the asset´s life-cycle management, including bug fixing, patches and release management.

2) The game developer
The validated assets in the delivery repository will be primarily used by game developers wanting to add functionality to their applied game under construction. Next to game developers, researchers, publishers and educators may be interested in assets and visit the repository. Some assets may also be usable outside games e.g. for gamification of a community platform. The repositories offer search functionality, which uses the assets' relevant set of metadata allowing for appropriate filtering. Assets

should include documentation and instructions about their functioning, and should go with tools, code or guides for their installation and integration with existing game platforms. The delivery repository assists in creating a package that can be downloaded, unpacked and included in the game developer's software project. The asset may come with specific configuration tools that would allow the game developer to adjust the asset´s behaviours, and - when appropriate - with specific authoring tools to specify relevant content. Importantly, the game developers should be able to provide feedback, to report bugs and to receive responses to these.

3) The repository manager
The repository manager is responsible for the overall functioning of the repositories. This role includes the provision and maintenance of generic tools and the associated instructions and procedures of quality assurance, e.g. testing protocols, metadata schemes, validation aids. The repository manager role may readily be decomposed into various subordinate roles such as system administrator, tool developer and quality assurance manager.

1.2 Challenges and principles

Although the RAGE project aims to support and enhance the quality of applied games it does not particularly focus on full applied game development. Instead RAGE focuses on the creation of software modules (assets) that could be incorporated within such games. Even though some existing communities offer a wide variety of game objects to be used by game developers, e.g. the Unity Asset Store, or the Unreal Marketplace, these objects are mostly media objects (content) and to a lesser extent include game software for enhanced functionality. The RAGE project is unique in that 1) it focuses on software assets rather than media assets, 2) it supports the integration of assets in multiple game platforms, and 3) it allows for linking multiple assets together for enhanced functionality (e.g. combining player data collection and game adaptation). This means that RAGE cannot directly adopt and reuse existing approaches and methodologies used by game engine vendors and game development companies. The ambitions of RAGE go with a set of technical and methodological challenges. Below we will briefly explain some of the main challenges that RAGE will be facing and formulate respective starting points for dealing with those.

- Involving stakeholders
  RAGE has identified stakeholder involvement as a key success factor. In order to create an appropriate information base for asset development and the associated technologies a primary needs assessment among (European) game studios will be carried out. ***Principle 1:*** *RAGE will collect stakeholder information to base its actions.*

- Dealing with diverse game platforms and programming languages
  As the game industry uses a wide variety of programming languages and game platforms, RAGE should take into account the compatibility of the assets with these different technical contexts as much as possible. Nevertheless, for practical reasons and efficiency reasons RAGE will not be able to cover a wide variety of programming code bases and game platforms.
  Principle 2: *RAGE needs to confine its asset development to a limited set of code bases, while it still should support as many common game platforms as possible.*

- Defining an asset system architecture
  An overall asset system architecture is needed for realising a consistent set of assets, with efficient and reliable mechanisms for data storage, data exchange and interfacing with game engines. It should also account for data exchange between assets mutually, which allows for grouping assets together into more complex aggregates. The architecture should be robust over extending the set of assets with new assets. In addition it would require a systematic approach to nomenclature of variable names, objects and methods to be used across the whole set of assets.
  Principle 3: *RAGE requires a coherent asset system architecture that enables the well-coordinated, efficient and reliable functioning of assets.*

- Conforming to open interoperability standards
  In contrast with the domain of (e-)learning, which exhibits a culture of open standards, the domain of gaming rarely conforms to open standards. Instead game developers either provide ad hoc, local solutions that neglect existing standards, or they conform to the proprietary standards that are enforced by game console manufacturers.
  Principle 4: *RAGE needs to devise an operability framework that enhances compliance with open standards, particularly in the domains of learning and teaching.*

- Making available the assets in a searchable collection
  The asset repositories are needed to store and manage access to the asset packages. It should offer a dedicated search facility that allows potential end-users to identify appropriate assets and download these. This requires a shared metadata format: the assets need to contain a descriptive set of metadata, to be covered by a generic schema that applies for all assets. Also the schema should contain information about software dependencies, that is, explaining what additional software versions are required for correct functioning.
  Principle 5: *RAGE will develop a shared metadata format.*

- Defining packaging requirements
  An asset package should meet general requirements for packaging. The packaging should be compliant with the packaging formats that are supported by the repositories.
  Principle 6: *RAGE will specify packaging conditions and comply with these.*

- Allowing for different asset development environments
  Since asset development is supposed to be covered by diverse independent contributors, ultimately by wider community populating the RAGE Ecosystem, it would be unrealistic to enforce the use of one preferred development environment. Instead, asset developers should be allowed to use their preferred tools and platforms as much as possible.
  Principle 7: *The RAGE asset development approach will not enforce a particular integrated development environment (IDE).*

- Allowing for different asset programming methodologies
  Likewise, it would be unrealistic to enforce a fixed asset programming methodology and associated workflow. Instead, contributors should be able to work according to their preferred software development approach as much as possible, provided that they meet a set of general quality standards (e.g. metadata, documentation, testing, ownership, etc.). Although asset developers will have to comply with some architectural and interoperability principles, and quality assurance requirements, the process of asset development should be as less restrictive as possible.
  Principle 8: *RAGE will allow asset developers to use their preferred software development methodology, and will devise a quality assurance framework to warrant the quality of assets.*

- Managing and maintaining the collection of assets
  Lifecycle management of assets requires mechanisms for collecting user feedback and reporting bugs and the associated workflow regarding asset improvement, bug fixing and release management.
  Principle 9: *RAGE needs to accommodate the workflow and processes for managing and maintaining assets across their life cycle.*

- Providing asset tools
  RAGE will accommodate stakeholders with systems and tools for creating and handling assets.
  Principle 10: *RAGE will reuse existing tools and systems as much as possible and only create new tools when alternatives are unfavourable.*

- Open source licences
  For achieving maximum involvement of the game development community RAGE will pursue an open source policy, but aims at the same time at preserving opportunities for future commercial business models.
  Principle 11: *RAGE will select open source licences that preserve ample flexibility for future RAGE business models.*

1.3 Structure of this document

To a great extent the challenges and principles identified above are reflected in the structure of this document.

Chapter 2 provides and explanation of the main concepts used in this document.

Chapter 3 deals with collecting baseline end-user information, which is guided by principle 1. The section explains the needs assessment that was carried out among 21 European game studios. In structured interviews with key personnel of the studios we collected information about the ways that game studios view and deal with emerging technologies.

Chapter 4 describes and substantiates what programming languages RAGE will favour for its asset development (cf. principle 2).

Chapter 5 describes the overall asset system architecture required for realising a consistent set of interoperating assets that can be linked with selected game engines (cf. principle 3).

Chapter 6 explains the approach to complying with open standards and specifications (cf. principle 4).

Chapter 7 describes the approach toward asset metadata definitions (principle 5).

Chapter 8 presents the asset repository infrastructure (various of the principles).

Chapter 9 covers the packaging format required for publishing assets in the RAGE repositories (principle 6).

Chapter 10 synthesises the overall methodology and tools needed for asset development and maintenance (cf. principles 7, 8, 9 and 10).

Chapter 11 describes the outcomes of the software licences analysis (principle 11).

2 MAIN CONCEPTS

An asset in the domain of IT is generally defined as a collection of related artefacts that provides a solution to a problem (Ackerman, et al., 2008). Some asset definitions are restricted to content and/or media rather than software. For instance, Niekerk (2006) distinguishes three major groups of “digital assets”: textual assets (digital assets), images (media assets), and multimedia assets (a combination of different content forms). The IMS content packaging information model (IMS-CP, 2004) likewise uses the word asset to explain the term resources: “the resources described in the manifest are assets such as Web pages, media files, text files, assessment objects or other pieces of data in file form”.

IBM´s Reusable Asset Specification (Ackerman, et al., 2008) uses a high level definition of an asset as “a collection of related artefacts that provides a solution to a problem”, allowing it to package together as an asset almost anything: Models, Design documents, Patterns, Web services, Frameworks, Components, Requirements documents, Test plans, Test scripts, Deployment descriptors, Model templates, UML profiles, Domain specific languages, etc.

W3C, while specifying the Asset Description Metadata Schema (Asset Description Metadata Schema), defines an asset as an abstract entity that reflects some intellectual content (Dekkers, 2013). An asset differs from an asset distribution, which is typically a downloadable computer file (but in principle it could also be a paper document or API response) that implements the intellectual content of an Asset.

In the domain of gaming the term asset is often used for media files to be incorporated in a game. The Intel® XDK HTML5 Cross-platform Development Tool (https://software.intel.com/en-us/html5/tools) offers an asset manager for game development in conjunction with several game platforms. Here assets are considered game objects, to be included in a project (audio/visual). Likewise, the Unity Asset Store (https://www.assetstore.unity3d.com), which is a prosumer-based community and market for game assets, is dominated by such media assets, which are to be imported and used in the Unity game engine. Although originally game developers would reserve the term “asset” for such media files rather than for software artefacts, the Unity Asset Store increasingly includes software assets, e.g. code for physics, special effects, controller software, GUI software, AI, and maze generation among many other things.

In RAGE we will follow up on the widest possible definition of an asset and define some specific characteristics that aim to enhance the assets functioning and interoperability.

2.1 Terminology

Asset
In the RAGE project assets are advanced game technology modules (software), enriched and transformed to support applied games development. RAGE Assets are self-contained solutions that demonstrate economic value potential, based on advanced technologies related to computer games, and intended to be reused in a variety of game platforms and scenarios. Each RAGE asset includes one or more software components working together on a dedicated task. It can work together with other assets, target game platforms and external services/software. Assets may include manuals, documentation, scientific data, examples of use, demos, content authoring tools and a wide range of additional resources. This definition is consistent with the definition in IBM’s Reusable Asset Specification (Ackerman, et al., 2008) and the W3C ADMS Working Group (Dekkers, 2013). It should be noted that the term “asset” is used in this document as a technical (computer science) concept. As game assets are usually understood by the industry as media assets rather than software assets, the branding of RAGE assets may be based on a less ambiguous term, e.g. component, module, or service.

Asset software component
Asset software component is the functional part of the asset that is to be integrated in the game. On the client side the asset software is made available either as a binary file or as a source

code file that needs to be compiled/interpreted as part of the game. Server-side solutions will use web services instead (REST).

Asset metadata
Asset metadata refer to machine readable information such as keywords and semantic information that can be used by the repository's search engine and information that is needed for running the asset software in an operational environment, e.g. on a game platform. It includes version info and info about dependencies of other software.

Asset metadata schema
The asset metadata schema defines the overall structure of the asset metadata and is used as the reference for interpretation and validation.

Asset package (asset distribution)
The asset package is generated by the repository for wrapping the asset software and its metadata, supplemented with any artefact necessary for deployment/installation and use (e.g. installation guide, installation test suite, tutorial, examples, scientific data, authoring tools for adding content to the asset) into a single file for transportation (viz. downloading).

Game engine
A game engine is a high level specialised software system designed for the creation and development of video games. Game engines usually abstract the target gaming platform and are used to create games for video game consoles, mobile devices and personal computers, for which they produce executables (or they may even integrate with the respective app stores).

Game development platform
Game development platforms show considerable overlap with game engines. In general, game development platforms are software development kits (SDK); they are more modular than game engines and they offer a lower level of technical abstraction. Typical constructs in a game development platform are close to the concepts computed by the machine (e.g. polygon, model, etc.) than in typical game engines (e.g. character, object, game level). Similarly, while engines typically incorporate authoring tools or can be configured with more abstract scripting languages, game development platforms typically integrate directly with common software development tools (e.g. Eclipse, Visual Studio, etc.).

Repository
A repository is a structured storage facility for digital resources. The main function of the RAGE asset repositories is to facilitate the distribution and exchange of the assets.

3 BASIC NEEDS ASSESSMENT

For identifying the needs and expectations of European game studios the RAGE project has carried out a number of in-depth interviews in the first 4 months of the project. The main goal of this study is to understand the game studios´ practices and their views, strategies and expectations with regard to emerging technologies. The survey aims to clarify how game studios - as part of technology-driven creative industries - balance production routines and innovative approaches. The next sections summarise the method and outcomes of this survey. This study and its outcomes were presented on the GALA conference 2015 in Rome (Saveski et al., 2015), see Annex 1 for more details.

3.1 Method

Multiple influential factors were taken into account. For this purpose the problem space was divided across the following endogenous and exogenous dimensions:

- Business contexts
- Technological diversity
- Technology standards
- Pedagogical practices
- Emerging technologies and methods
- Information and knowledge resources

21 game studios from 10 European countries participated in structured interviews of maximum one hour. All respondents had senior positions in the studios, either as CEO, creative director, owner, programmer, producer or sales manager. The interview was guided by 35 questions (both closed questions and open questions) covering the respective dimensions.

3.2 Results

Generally, the game studios are highly interested in pedagogical models, product validation, newly emerging technologies and wide range of knowledge resources. With respect to customers, the games, tools and infrastructure, the game studios display a large diversity. Major platforms are targeted, including mobile platforms. Many studios develop browser games. Exceptional is the predominance of the Unity game engine. Game consoles are avoided by most of the studios as platforms for serious games.

Experiential learning was reported as the most popular pedagogical approach used in the games. Natural feedback and debriefing (after the game) were ranked as the most important pedagogical strategies, whereas guidance and instruction during the game are rarely used. It remains unclear what game studios actually know about pedagogical strategies. More detailed probing is needed into the operational significance of the studios' pedagogical approaches. Most game studios claim to test their games' effectiveness for learning. Some even use randomised controlled trials for collecting evidence. But the depth and validity of the approaches that the studios claim to apply could not be established in this survey. Game studios assigned highest priority to information that enhances their business opportunities. They are eager to check information about mobile technologies, game technologies, game design, best practices and channels that point them at potential new personnel. A variety of external repositories are already accessed to support the work. Game studios designated interoperability as a major issue and many experienced practical difficulties when connecting their games to other systems. Only a small number of studios referred to main Technology Enhanced Learning (TEL) standards that might help to resolve the issues. It may suggest that the game studios have only limited awareness of these standards or that they do not consider (TEL) interoperability as an urgent topic. With respect to the importance of emerging technologies highest rankings were given to learning analytics, real time tracking of learning progress, adaptive gameplay and game evaluation, respectively. They look for added value in terms of better games or commercial potential, and at the same time fear complex and cumbersome implementation.

Stakeholder involvement will be continued throughout the lifetime of the RAGE project. In the course of the project stakeholders will be involved in asset development (WP2, WP3), game development (WP4), applied game usage (WP5), Ecosystem usability (WP6), Business modelling (WP7), evaluation (WP8) and dissemination (WP9).

4 RESTRICTING THE TECHNOLOGY RANGE

In accordance with principle 2 RAGE has to identify primary technologies (e.g. programming languages, game platforms) that should be conformed to, particularly for client-side assets. The challenge is to restrict asset development to a few primary code bases that still would allow to reach out to a maximum number of game developers and their platforms. For this purpose a survey has been conducted during the first four months of the project that identifies topical trends in applied game development and the key platforms and programming languages that are currently being used. The results of this study have been used for decision making by RAGE's Executive Management Board about which technologies to support. A detailed report of this study is included in annex 2.

4.1 Method

The technology analysis is based on multiple sources. First, the 4 game studios within the RAGE project were involved in a consequential role. They provided detailed information about their technological infrastructure and conditions of work. Second, technical information from the needs interviews (chapter 3) was used to estimate the wider popularity of game technologies. Third, the authors of the study retrieved relevant knowledge and anecdotal evidence from their professional contacts with the industry. Fourth, the authors made use of what little statistic information could be found. As a final step the outcomes and alternatives were intensively discussed with the game companies that participate in RAGE, which eventually led to a consensus about which technologies to prioritise.

4.2 Results

The 4 game studios within RAGE appeared to use a wide variety of game platforms (e.g. Marmalade, Unity, Cocos2d, Xamarin, OpenSimulator, MonkeyX, Source Engine, Playcanvas) and programming languages (C, C++, C#, Java, GoLang, Flash, Javascript, Assembler). They also address a wide variety of target runtime platforms (Android, iOS, desktop, Browser, Windows Phone, Xbox One, PS3, PS4, WiiU). An even greater technological variety was observed in the needs interviews among a wider circle of game studios (cf. chapter 3). However, as all game studios are using multiple platforms and code bases, some of the technologies stand out as being widespread and highly popular, e.g. the Unity3D game engine, Android, programming languages C++ and C#. Additional sources to estimate the penetration of game platforms and programming languages were used to check for consistency of the overall picture (e.g. from the 2015 Global Game Jam). Four scenarios were proposed: 1) to restrict ourselves to a reduced set of highly popular game platforms, 2) to select few code-bases for most popular programming languages, 3) to allow for different assets targeting different platforms, 4) to select a single code base to be translated or wrapped to different platforms. These options were elaborated and discussed, while balancing off the technical limitations, the feasibility of realising tangible results and the opportunities for achieving substantial impact in the game market.

4.3 Conclusion

It was generally confirmed that the RAGE Ecosystem should be a platform-independent arena, where multiple assets target multiple platforms without specific restrictions. For technical reasons, code translation or wrapping (option 4) was removed as an inappropriate approach. The option of allowing different assets targeting different platforms (option 3) is assumed a weak approach as it would conflict with asset interoperability. It was unanimously concluded that selecting few code-bases for most popular programming languages – relevant for games - would be the most favourable strategy, even though it would require a dual or triple coding regime.

As a balance, RAGE aims to deliver technology assets that are independent of specific game engines, but that can be easily integrated with such engines with a relatively small effort. This requires targeting specific languages, and also requires paying special attention to reducing potential issues affecting portability to different engines (e.g. avoiding direct interference with DirectX controls or engine-specific APIs).

For selecting the most appropriate set of programming languages, the discussion used the following line of argumentation

- The programming languages should allow to create modules with clear interfaces and APIs that can then be integrated in development projects using popular game engines.
- Most games and game engines identified during our surveys are based on C++, C#, Java, Objective-C or HTML5/JavaScript.
- C++ has been a dominant game programming language but is more complex than C# or Java and often relies on platform-dependent constructs.
- Game companies within RAGE favour either C++ or C#.
- HTML5/JavaScript is a *de facto* standard for new browser-based games.
- TypeScript can be used to improve quality of JavaScript as it is typed, contains concepts like Interfaces and Classes and translates into valid JavaScript during its 'compilation'.
- The needs survey among external European game studios reveals dominance of C#.
- Major game engines or multiplatform tools (e.g Unity/Xamarin) use C# as their core language.
- The growth of C# has accelerated significantly since 2014, when Microsoft released .NET core framework as open source, supporting any operating system and pushing the multiplatform nature of C#.
- In 2015, C# matched C++ in the Redmonk programming languages ranking[1] (both are ranked 5th).

The following technology proposal was derived, which was then unanimously accepted as a strategy by the Executive Management Board of RAGE (June 2015).

1. By default, all RAGE assets will be developed supporting two core code bases: C# (for Desktop and mobile games) and HTML5/JavaScript (for browser games).
2. As general guidelines, C# code should be as interoperable as possible (avoiding platform-specific libraries), and HTML5/JavaScript code should be written using TypeScript.
3. Some specific assets may deviate, understanding that not all assets make sense in all target platforms.
4. Before the end of the project, RAGE aims to create additional proof of concept implementations of specific assets in Java, and C++.

This way RAGE is assumed to deliver assets that are suitable for the widest possible range of game engines and target platforms. It enables RAGE to deliver technology assets that are not dependent on specific game engines, but that can be easily integrated with such engines with a relatively small effort.

[1] http://redmonk.com/sogrady/2015/01/14/language-rankings-1-15/

5 ASSET SYSTEM ARCHITECTURE

An overall asset system architecture is needed for realising a consistent set of assets, with clear feasible and reliable mechanisms for data exchange and integration with game engines and for data exchange between assets mutually, which allows for grouping assets together into more complex aggregates (cf. principle 3). A concise description of the asset architecture will be given in the sections below.

5.1 Architecture requirements

Based on RAGE´s Description of Action the following basic requirements have been identified:

- The architecture and integration points should be as simple as possible
- Avoiding platform and hardware dependencies as much as possible
- Maximum portability between game engines
- Maximum portability between programming languages
- Consistent coding style and documentation within each asset
- Minimum dependencies on external software frameworks
- Conformance with (subsets of) existing standards in order to minimise overheads

5.2 The RAGE Component-Based System Architecture

Given the nature of the assets' software as reusable software components, RAGE adopts a component-based software architecture (e.g. Mahmood, Lai, & Kim, 2007). It addresses both the level of interaction of assets with the outside world and the internal workings of an asset.

The purpose of the RAGE asset architecture is to allow for the easy integration of reusable software modules across a multitude of different game engines and platforms. The architecture combines a service-oriented architecture (SOA) using web-services for communication with remote agents via the http protocol (e.g. REST) and a client-side framework for components that are to be fully integrated into the game engine (Van der Vegt et al., 2016a; Van der Vegt et al., 2016b). Such a hybrid approach is needed to bypass the limitations of both SOA and client-side solutions. Generally, SOA offers several advantages such as decoupling from implementation details and a high degree of reusability of services. On the downside, SOA may reduce system performance due to frequent network calls and additional overheads, and it offers limited customisation and configuration of services by service consumers. Moreover, the player is supposed to be permanently online, which need not always be the case. The RAGE client architecture relies on a limited set of well-established software patterns and coding practices aimed at decoupling abstraction from its implementation. It avoids dependencies on external software frameworks and minimizes code that may hinder integration with game engine code. However, if client-side assets require extensive processing, remote services may be a better option to avoid poor game performance on the local machine, e.g. reduced frame rate or reduced game responsiveness. Client assets will primarily be developed in C# and JavaScript/TypeScript as to comply with most popular development platforms. Even so, proofs of concept specifically designed within RAGE are available in C++ and Java (Van der Vegt et al., 2016a). Figure 2 sketches the basic structure of the RAGE architecture (Westera et al., 2016).

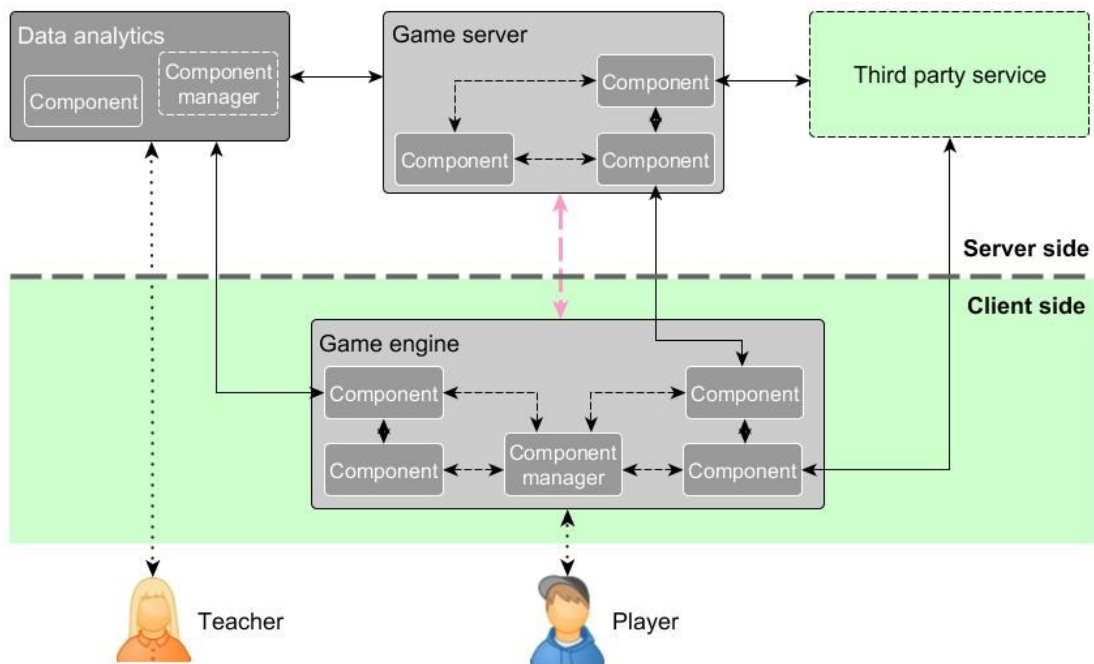

Figure 2. Outline of the RAGE asset-based architecture and its diverse data communication routes.

In figure 2, the game engine and the eventual game server are in pale grey. Dark rectangles represent diverse RAGE software assets (components). The RAGE data analytics server is a central server-side system that aggregates interaction data from the population of players to be inspected by teachers and educators. This server handles authentication and authorisation, and acts as a gateway proxy server to underlying assets and services. It includes a component (asset) manager for the registration of web services. Client-side RAGE assets are coordinated by a local component (asset) manager. RAGE assets (components), either client-side or server-side, can directly interact with each other. Server-side assets can be included in the game server or use a separate server. Third-party web services can be called by both client-side and server-side assets.

5.3 RAGE assets

Figure 3 sketches the general lay-out of a RAGE asset.

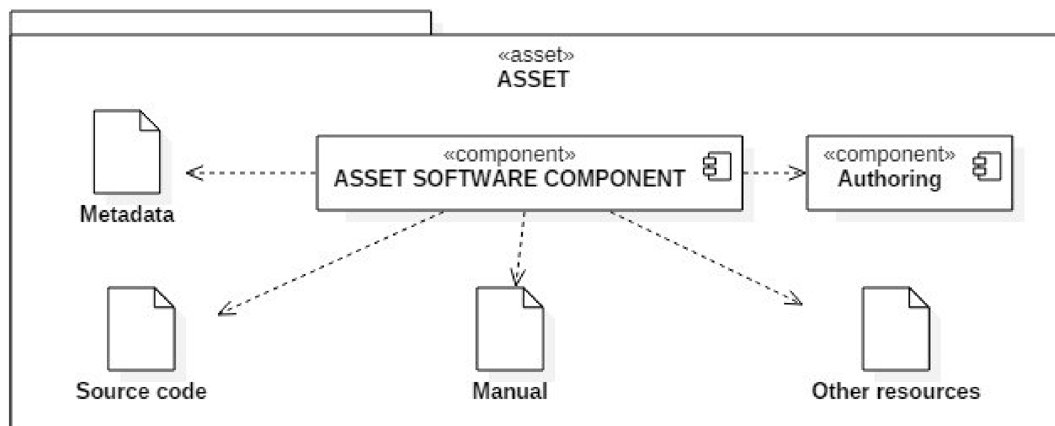

Figure 3. Exemplary lay-out of a RAGE Asset

RAGE Assets contain a software component that may either be a compiled version or a source code file. Also the asset contains machine-readable metadata (that cover version info, dependences info, programming language used, etc.) and various additional artefacts that are not to be compiled (tutorials, manuals, a licence, a configuration tool, an authoring tool and other resources). While the constituents of the asset are stored separately in the 0, the asset can be packaged for distribution. The package may be platform-dependent to allow inclusion in a specific game platform.
Note: 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. We will first detail the approach to client-side assets.

5.3.1 The internal structure of a client-side Asset Software Component

The asset software component can be further decomposed. Figure 4 displays the internal structure of a client-side asset software component. Details are in two scientific papers, provided in Annex 3 and 4.

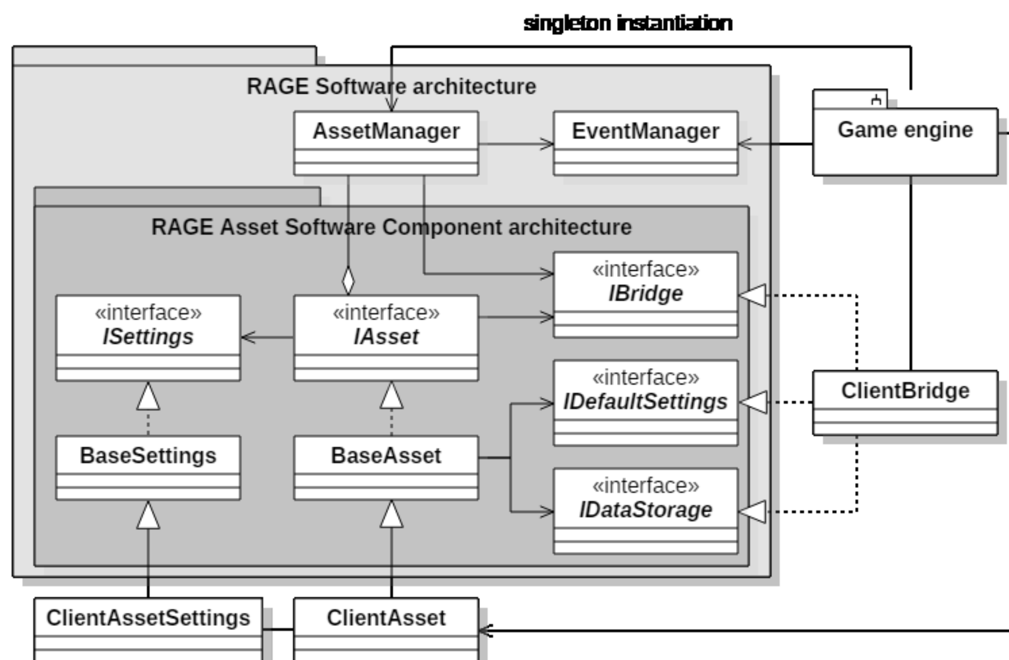

Figure 4. Internal structure of the asset software component (Van der Vegt et al., 2016a)

The asset software component is composed of a set of interfaces and classes for the required functionality and basic capabilities such as storing configuration data (settings), localization data (string translation tables), displaying version information and dependency information (which allows an asset to determine its own version and its dependency on other assets' versions; an asset can be queried for it). While most metadata are read-only, the asset configuration data may be changed to adjust the asset's functioning.

Coordination between multiple assets is covered by the Asset Manager, which covers the registration of asset software components. It exposes methods to query this registration in order for asset software components to find other components and link to these. For avoiding duplicate code in each asset software component the Asset Manager will also be the coordinating agent for basic services that are relevant for all assets, such as collecting game heartbeat, and the name and type of the game engine. It centralises the code that is commonly used by all assets and requires only a single interaction point with the outside game engine. These data could be broadcast to asset software components that have subscribed themselves for this event.

For the communications with the outside world (e.g. the game engine) a number of standard software patterns and coding practices will be used. We distinguish between the following cases:

- Communication through event subscription
  The event subscription pattern typically supports an 1,N type of communication (broadcasting). Asset software components could offer event subscription to be used by other asset components or the game engine.
- Communication through game engine methods
  For this (the asset software component calling a game engine method) the bridge software pattern will be used, which is platform dependent code exposing an interface. By opting for interfaces that implement polymorphism the asset may identify and select a suitable interface and use its methods or properties to get the pursued game data.
- Communication via the Asset Manager
- Communication through (web) services, e.g. REST.
  For making an actual web-service call a component may need a platform/game engine-dependent adapter.

Once registered at the Asset Manager, an asset could use the Asset Manager's set of commonly used methods and events in order to minimize the game engine – asset interaction points.

5.3.2 Technical validation of the client-side asset architecture

The RAGE client architecture has been successfully validated by implementing and testing a basic software asset in four major programming languages (C#, C++, Java and Typescript/JavaScript, respectively) (Van der Vegt et al., 2016a). The validation tests involved the creation of the asset manager as a singleton, the identification of assets and the bridge code and its associated communications with the game engine and web services, among other things. Tested implementations of the basic asset in all four programming languages can be found on GitHub (https://github.com/rageappliedgame).

Also, portability of assets has been demonstrated and documented (Van der Vegt et al., 2016b). To this end, a RAGE-compliant C# software component providing a difficulty adaptation routine (the HAT asset, task 3.4) 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.

5.4 Server-side architecture

As described above, different assets may need to communicate with additional centralised components, which reside in a separate server. Some of these assets may incorporate a server-side component, while other assets may be directly designed to reside on a server, instead of being integrated in a game platform.

This requires expanding the architecture to contemplate both client-to-server communication (for assets with a server-side component) and server-to-server communication (for information exchange across server-side assets).

At the core of the server-side architecture, is the basic principle of Authorization and Authentication. Any data exchange between a game component and a web server, or between two web-servers, should be subjected to an Authentication and Authorization procedure. It therefore makes sense that all these communications go through a specific RAGE asset, labelled A2 (Authentication and Authorization). Even when the exchanged data is not sensible, it still makes sense to channel the communication through this central asset, which would verify that the request is, in fact, authorized. This is similar to the role that the Asset Manager plays on the client side, and provides an overall structure for the server-side architecture.

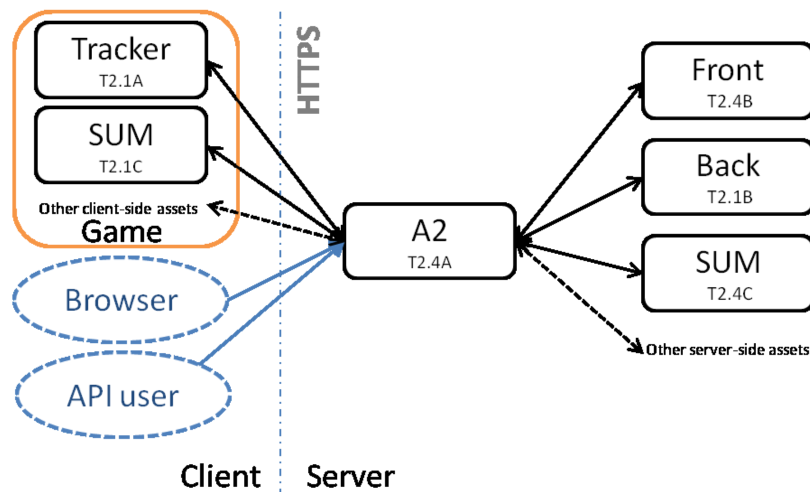

Figure 5. Overall server-side architecture. All remote communications necessarily pass through the A2 asset.

Another key asset in the server-side infrastructure is the Simple User Model asset (SUM), which provides a general service for assets to store relevant data as simple key-value pairs. This allows assets to store information for later retrieval, either from the same asset or another asset (provided that all Authentication and Authorization parameters allow it). The next two subsections provide additional details about these assets.

5.4.1 A2: Authentication and Authorization Asset

Authentication and authorization are important concerns for multiple RAGE assets. Having a single server entry-point (this asset: A2) that can perform authentication and authorization on behalf of all other assets, provides single-sign-on capabilities to those assets, and drastically reduces duplicated effort. A2 proxies all authenticated, authorized requests to the corresponding server-side assets.

For clients of server-side assets that are proxied behind A2, single-sign-on is completely transparent. Additionally, since A2 effectively becomes a single point of contact, client configuration is drastically simplified: only A2's location login and credentials need to be stored, instead of each individual server-side asset's location and login credentials.

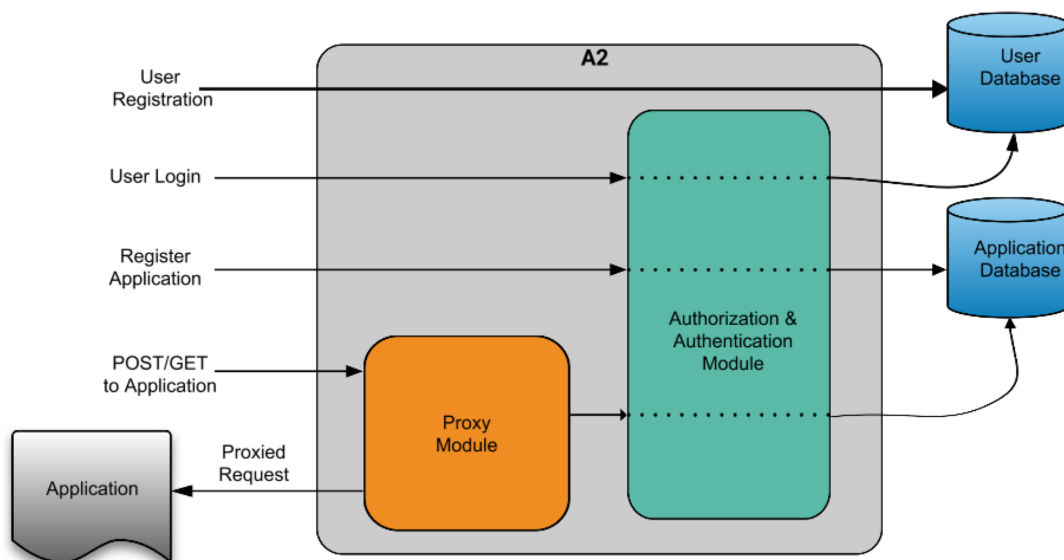

Figure 6. Internal structure of the A2 asset.

5.4.2 Simple User Model (SUM) and cross-asset communication

Server-side assets can programmatically (via API) register themselves and configure what they consider to be authorized via customizable roles. This allows each asset to provide services to other assets through specific APIs, always under the control of the A2 asset.

As part of the registration process, it is possible to add, query, modify and delete resources (URLs to be proxied to registered server-side assets), users, and roles; and to restrict access to URLs to users with specific roles, including the option to allow or deny unauthenticated access.

However, it will be common that different assets would require the storage and retrieval of data, perhaps asynchronously. The Simple User Model, provides both client and server-side storage for key-value pairs for other assets. This can be used to store asset settings, communicate assets with each other, or connect players with each other.

6 OPEN INTEROPERABILITY STANDARDS

6.1 Goals

This chapter relates to task T1.2 in the Grant Agreement, Specifying the interoperability requirements of assets. The target outcomes for this are to produce substantiated guidance on the formats and standards that are to be used in RAGE in three ways:

1. To link assets so that they can share data when they are deployed together in a game
2. To connect or integrate assets with external systems, e.g. VLEs

6.2 Concepts

It is worth noting that in the Grant Agreement the distinction between interoperability and integration is blurred. These are, however, different processes and should be distinguished:

- Integration concerns the way that a component is adsorbed and works in a system.
- Interoperability concerns communication between components, which may be in separate systems.

Hence target outcome 1 is concerned with both interoperability (communication between assets) and integration (the use of multiple assets in a single game). It is the achievement of asset interoperability through the work reported in this deliverable that it becomes possible to achieve integration in particular games which use RAGE assets.
Target outcome 2, despite making use of the word integration, is better characterised as concerning interoperability.

6.3 Mandation framework

As identified in the needs analysis reported in section 2 (Saveski et al., 2015+ see annex 2) game studios across Europe showed little awareness and interest of interoperability, and revealed little practical work being done. In response to the lack of adoption of existing standards a bottom-up case-based methodology was devised. This was accompanied by the development of a three level mandation framework to be adopted by the project: 1) obligatory, 2) desirable, and 3) for reference.

6.4 Target outcome 1: Strategy for inter-asset interoperability

There are many RAGE assets, which interact in many ways, and which use a wide range of terms. We are therefore focusing on a bottom-up case-based approach to inter-asset communication. The cases addressed in this work are the 8 games that are developed using RAGE assets for the 6 pilots. Cross-WP interoperability working groups were established charged with finding practical solutions to the potential interoperability issues that have been identified. The work will be intensified from summer 2016 to accompany game development and asset deployment. It will leverage existing standards wherever possible. A shared nomenclature of methods and variables will be developed to support inter-asset communication. The WP1 team has analysed the use of assets in the games which are in development in the first year of the project and has gathered information about the inputs and outputs to the assets, and the dependencies which these create. Initial results indicate that a number of assets do not raise evident interoperability issues, because they operate on the client side in accordance with the RAGE asset architecture, and do not involve extensive exchange of information. Still priority is being given to ensuring that the nomenclature used is consistent across these assets, and is documented.

6.5 Target outcome 2: Strategy for interoperability with external systems

At the time of writing, RAGE has established 6 pilots representing 8 games. There has, as yet, been no requirement from developers for interoperability at the level of platform. Nor has there been any demand for integration with institutional systems from the organisations that are hosting pilots. However, given the growing importance of VLEs and associated data analytics at the level of institutional management and strategy, work will be carried out in interoperability

with VLEs, which covers issues such as provisioning. As regards VLEs, an initial pilot has been planned to integrate RAGE games with Moodle, hosted at RAGE partner University of Bolton, potentially by making use of the IMS LTI specification. This demonstrator will raise awareness of the potential of this functionality, given that here has, as yet, been no requirement from developers or pilot hosts for interoperability at the level of platform.

In addition to this work, a more radical response to the problem is to sidestep the existing infrastructure of VLEs and corporate training platforms by building on emerging work with the Experience API (xAPI) specification (ADL 2016). The use of xAPI enables RAGE to cover the case of a more loosely coupled infrastructure, where provisioning is not an issue, but the reporting of results to a central application is a requirement. The use of xAPI has two more advantages in this respect. Firstly, it leverages work being done in RAGE to provide xAPI assets for games development. Secondly, it has the potential to be applied in a wide range of contexts, beyond the confines of VLEs and corporate training platforms, while maintaining interoperability with those legacy systems. From the evidence of the RAGE pilots to date, it is likely that the applied games supported by RAGE will often be used in combination with an ad hoc collection of tools, rather than being integrated in an LMS. With xAPI any data can be collected and shared from a game, so long as the appropriate RAGE assets have been included, and this is the case for all RAGE pilots currently under development. An xAPI Learning Record Store (LRS), enables data to be gathered from the learners activities, and made available for analysis and subsequent presentation on a dashboard. This may be sufficient interoperability for many contexts of use, but in cases where it is not, the LRS can be connected into institutional systems, providing the necessary flexibility. It is therefore planned to manage RAGE interoperability with external systems through the use of the xAPI specification and an LRS, leveraging the work being done on such systems in work package 2. To facilitate this strategy it is –again- important to achieve alignment in the nomenclature used in collecting data from games. In xAPI this is achieved through recipes. A significant achievement of RAGE has been work led by partner UCM which has resulted in the publication of a Serious Games Vocabulary and associated xAPI recipes, with the support of ADL (see http://xapi.vocab.pub/datasets/seriousgames/). The vocabulary and recipes will be supported by the RAGE asset infrastructure, and further developed within the project and the wider community.

7 DESIGNING A SHARED METADATA SET

Each asset should go with appropriate metadata (cf. principle 5) that supports a diversity of functions, e.g. accommodating search in the asset repository, allowing for version checks of the software environment, checking for other assets that are needed, ownership and licensing information.

In order to support a wide set of services through the software repository and other related tools and, in parallel, to be close to the specified domain of reusable gaming components (RAGE software assets), the RAGE metadata model is focussed on the following main aspects:

- **Technical** – how the RAGE asset might be used by game developers. We follow the RAGE asset model which describes assets as software components according to software engineering standards.
- **Contextual classification** – here we focus primarily on the pedagogical, educational and game characteristics, whilst leaving space for further characteristics.
- **Usage** – not restricted to how to install and configure the software, but also providing additional artefacts such as training materials, tutorials, educational goals, etc.
- **Intellectual Property Rights** – to enable various business models proposed by RAGE project to be implemented.

Whilst designing the model, we broadly adhered to the following general metadata design principles:

- **Reusability** – reusing where possible existing metadata models, standards and available taxonomies and ontologies. We reuse parts of the RAS and ADMS metadata fields as well as parts of the LOM taxonomy.
- **Flexibility** – to easily facilitate extension of the metadata description of an asset with additional features and characteristics.
- **Simplicity** – we define most of the fields as not mandatory in order to easy the efforts of asset developers.

The RAGE asset metadata model reuses and extends the specifications of RAS and ADMS. We have chosen this approach to use a core subset of RAS and extend it with elements from ADMS, IEEE LOM and metadata related to the applied games domain. We could not use LOM directly, as it does not provide features for describing compound software objects, and their technical field lacks many important features related to software engineering such solution, implementation, tests, etc. The most comprehensive and close to our needs RAS model is too general and complex, has slow adoption and is difficult for our users. It is also not consistent with current software engineering practices (as it supports the “waterfall” model for software development). The ADMS does not provide support for external artefacts and is - just as LOM - lacking software engineering features.
Although the internal representation of the RAGE metadata in the asset repository is in RDF, we have chosen to use XML as a manifest file format (a special file that contains information about the files packaged in a RAGE asset package) and an XML schema for the model for the following reasons:

- If the manifest file in the asset package is in RDF, it is difficult to validate it.
- The URIs of the asset and artefacts are automatically generated after the asset is ingested in the repository so we cannot use them in the manifest beforehand.

Thus, we have chosen to use an XML schema for validation of the manifest files and to follow the approach used by the Europeana[2] project for representing RDF in XML. This approach provides for automatic validation of manifest files and easy transformation from XML to RDF and vice versa.
The RAGE Metadata Model defines the format of asset metadata as an XML schema, which will be implemented in all tools that require the processing of these metadata, e.g. a package metadata editor, the asset repository, asset installation widgets, etc.

The XML schema represented as a UML class diagram is displayed in Figure 7.

[2] http://www.europeana.eu/portal/

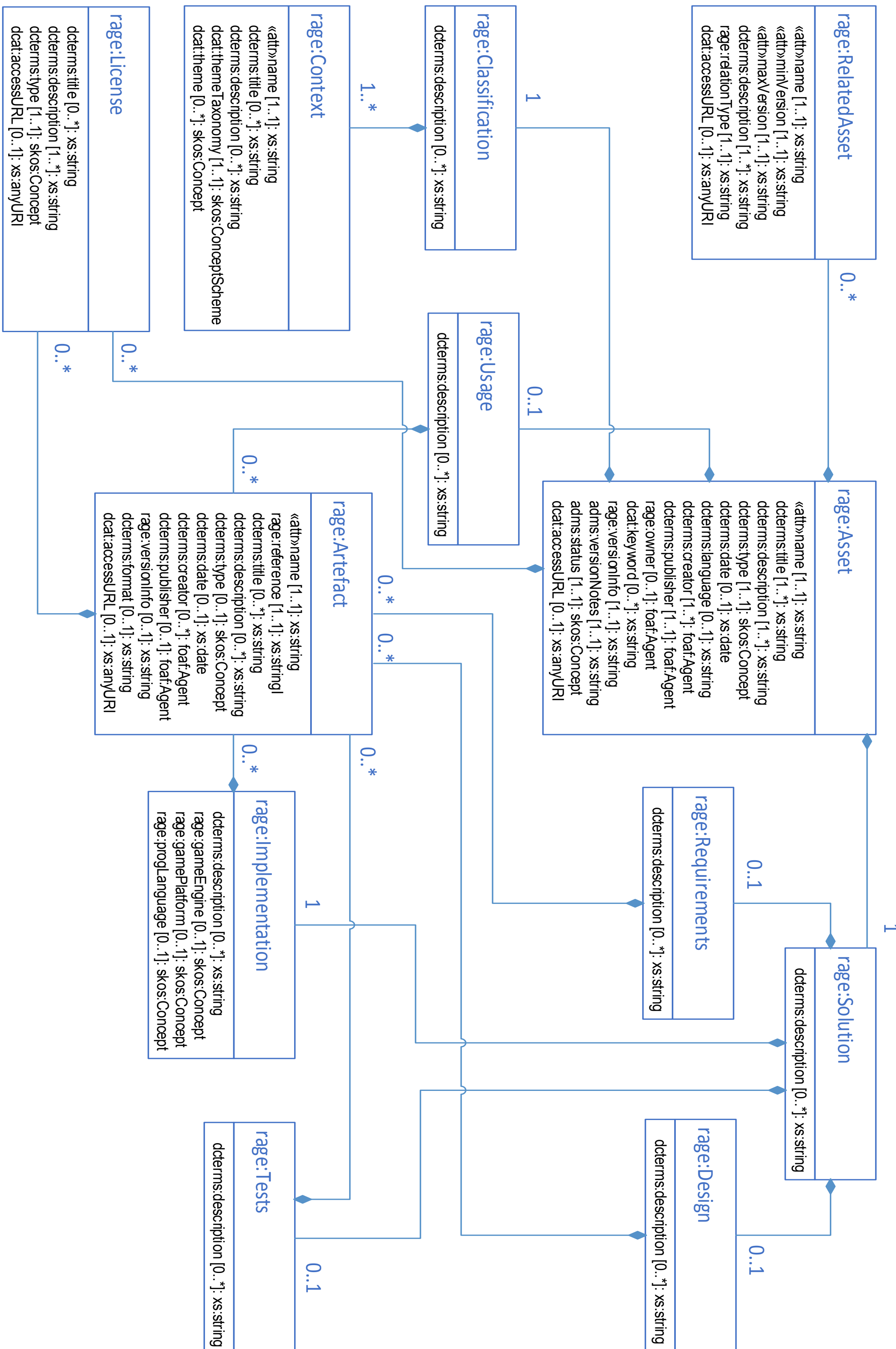

Figure 7. The RAGE metadata schema

Figure 7 can be interpreted as follows:

- A UML class represents a complex XML element. For example, “Asset” is a complex XML element that has XML attributes and/or child elements.
- Within a UML class definition, the prefix «attr» denotes that the corresponding field is an XML attribute (and not a child element).
- Within a UML class definition, a field without «attr» prefix denotes a child element nested inside the element represented by the UML class definition. The field represents either a new definition of a simple element or a reference to an element (either simple or complex) defined in a referenced schema (e.g., dcterms).
- Composition connection denotes a parent-child relationship between two complex elements defined in the RAGE Metadata Schema (RMS).

Table 1 provides a textual description of the XML elements of the RAGE metadata schema.

Table 1. Description of RAGE metadata schema elements

Asset	A self-contained solution that demonstrates economic value potential, based on advanced technologies related to computer games, and intended to be reused or repurposed in a variety of game platforms and scenarios.
Classification	Includes a set of descriptors for classifying the asset as well as a description of the context(s) for which the asset is relevant.
Solution	Describes the artefacts of the asset.
Usage	Contains information for installing, customizing, and using the asset.
Related Assets	Describes the asset’s relationship to other assets.
Artefact	Any physical element of an asset corresponding to a file on a file system. Artefacts can include also version and license information.
Requirements	Contains artefacts that specify the asset requirements such as models, use-cases, or diagrams.
Design	Contains artefacts that specify the asset design such as diagrams, models, interface specifications, etc.
Implementation	Has a collection of artefacts that identify the binary and other files that provide the implementation.
Tests	Contains artefacts (models, diagrams, artefacts, and so on) that are intended to describe the testing of the asset such as testing procedures, concerns and test units.
License	Contains conditions or restrictions that apply to the use of an asset or artefact, e.g. whether it is in the public domain, or that some restrictions apply such as attribution being required, or that it can only be used for non-commercial purposes, etc.
Context	Defines a conceptual frame, which helps explain the meaning of other elements in the asset.
Custom Metadata	An element that serves as an extension point for adding custom metadata as name-value pairs.
Agent	Describes a person or organization that is a contributor (creator, publisher, and owner) of an asset or artefact.
Concept Scheme	A vocabulary, thesaurus or taxonomy used for organizing concepts.
Concept	Represents a particular concept within a vocabulary, thesaurus or taxonomy.

Further details of the RAGE metadata model are in Annexes 5, 6 and 7.

8 ASSET REPOSITORY INFRASTRUCTURE

The RAGE assets will be made available through an open repository system. Moreover, the RAGE repository system will be extended with editing and management tools that allow third parties to define and create their own software assets and make these available to the wider community. To this end, the repository will be embedded within a social platform along with a digital library and archive of applied gaming resources (the ecosystem portal) in order to establish a main applied-gaming entry point for a variety of stakeholders including teachers, students, researchers, and, significantly, game developers. The integration of the asset repository infrastructure and the ecosystem portal is grounded in the shared metadata model and a metadata harvesting process carried out by the ecosystem.

This chapter focuses specifically on the RAGE asset software repository architecture. We provide details of the back-end repository system architecture and corresponding front-end tools. Details about the implementation of the backend software repository architecture are presented in Asset repository infrastructure document of Annex 8.

The RAGE asset software repository is at the core of the asset development infrastructure. It is used to store and manage access to:

- Reusable game assets
- Artefacts – resources within game assets
- Metadata for game assets and for artefacts
- Relationships between assets – dependencies, related assets, etc.

The Asset software repositories leverage the discovery, development reuse and repurpose of game assets and artefacts. They help both game asset developers and consumers in all the activities related to the game asset lifecycle.

For the distribution of given asset, an asset package is created, which is an archive file containing at least one manifest file, either in XML (eXtensible Markup Language) or JSON (JavaScript Object Notation) file format, and the required artefacts. The manifest file should validate against the RAGE metadata model.

Importanttly, in the following the focus will be on the development repository (cf. figure 1, chapter 1). The main functions of the RAGE Asset software repository are as follows:

- Search for assets and artefacts in the repository
- Find related assets/artefacts
- Browse assets/artefacts
- Create, update, publish, delete assets/artefacts
- Download assets/artefacts
- Versioning support
- Source code import from GitHub - using the GitHub API (GitHub API, 2016)
- Integration with IDEs
- Harvesting of repositories for game assets and metadata using the Open Archives Initiative - Protocol for Metadata Harvesting (OAI-PMH)
- Review and rate assets/artefacts
- Subscription to assets

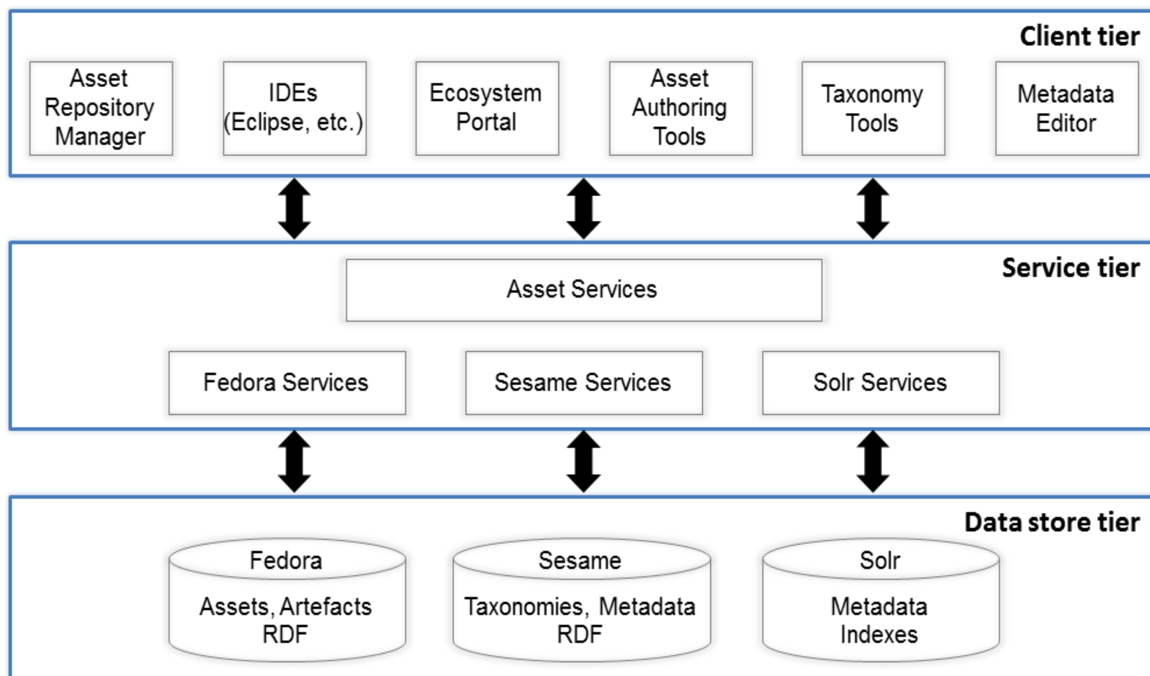

Figure 8. Asset Repository Architecture

The asset repository infrastructure comprises three tiers (Figure 8): client, service and data store tiers.

Based on the functionality exposed by the services, they can be grouped as:

- **Asset Access Services** defining an open interface for accessing assets within the RAGE Asset Repository. The access operations include retrieving asset packages and metadata, search and browse for assets using keywords and metadata fields. The search interface provides both full-text search and semantic search. Full-text search enables performing of natural language queries using keywords and phrases occurring in any of indexed asset's metadata elements. The semantic search is using SPARQL for querying on asset metadata and SKOS taxonomies data represented as RDF triples.
- **Asset Management Services** defining an open interface for administering assets, including creating, modifying, and deleting. These services interact with the underling services providing an abstract level of the operations and thus hiding the complexity about the internal formats, protocols and procedures for storing an asset in the Asset Repository.
- **Taxonomy Services** defining an open interface for managing classification taxonomies and controlled vocabularies used in RAGE Asset Metadata Model to classify and describe an asset in educational and gaming contexts. For representation and storage the Asset Repository adopts Simple Knowledge Organisation System standard.
- **Authentication and Authorization Services** – provide an extensible and configurable access best suiting organisational needs. These services are based on Fedora 4 Authentication and Authorization framework and use OpenId authentication provider and mapping to one or more preconfigured roles, respectively.

The technical core is based on Fedora, Sesame and Solr. These are briefly explained below:

- Fedora provides a general RESTful HTTP API for accessing repository resources through HTTP methods. It supports OAI-PMH [20] re-quests on content and metadata in the repository.
- Sesame offers a RESTful HTTP interface supporting the SPARQL Protocol for RDF. It is a superset of the SPARQL and supports communication for Update operations and the Graph Store HTTP Protocol [21].

- Apache Solr exposes Lucene's Java API as REST-like API's which can be called over HTTP. The RESTful endpoints allow CRUD style op-erations to be performed on the repository resources.

8.1 Front-end tools to access the software asset repository

8.1.1 RAGE Taxonomy tools

The taxonomies that are used in RAGE's asset metadata schema represent hierarchies of concepts and controlled vocabularies. They are used to provide prefixed sets of values for some metadata elements. The front-end taxonomy tools in RAGE are actually a set of three tools – **Taxonomy Viewer**, **Taxonomy Selector** and **Taxonomy Editor**, which are all part of the development repository. They are generic tools written in JavaScript that share a common core and can be embedded in a web page.
The design principles of the taxonomy tools are: (1) minimalistic user interface and set of features; (2) consistent visual layout; (3) multiplatform support through HTML5 and JavaScript; and (4) multilingual interface and taxonomy content.
The taxonomy tools are part of the client tier and they access the repository through the asset services of the services tier, which in turn communicates with the Fedora, Sesame and Solr services.

The RAGE Taxonomy Viewer is used to browse taxonomies. It provides several layouts and interface languages. The RAGE Taxonomy Selector is used to pick concepts from taxonomies. This tool supports the functioning of other front-end tools, such as the metadata editor, asset configuration tools and various authoring tools. Finally, the RAGE Taxonomy Editor, is used to edit taxonomies. It supports structural changes to a taxonomy (i.e. adding, deleting and reallocating concept nodes around the hierarchy) and content changes (i.e. changing concept names and concept translation in several languages). More details are in Annex 7.

8.1.2 RAGE metadata editor

Another front-end tool is the RAGE **Metadata Editor**. It provides functionality to edit the metadata associated with an asset or another RAGE metadata entity. The editor hides the internal metadata complexity and constructs a flexible dynamic interface.

When a metadata file is loaded by the editor, it extracts the schema definition of the metadata and builds the user interface. The hierarchy of metadata is represented as nested blocks the nesting of which goes as deep as it is defined by the schema. When the interface is constructed, the editor extracts the actual metadata and populates them in the interface. This may recursively trigger a partial reconstruction of the interface for nodes of cardinality higher than 1. After the metadata are edited by the user, the editor generates an XML representation of the metadata and sends it back to the server. More details are in Annex 7.

9 RAGE ASSET PACKAGING

In accordance with principle 6 an asset package should conform to a general format for packaging. To support the storage, search, deployment and usage of RAGE assets, a mechanism is needed for packaging the asset.

An Asset is packaged into an Asset Package, which includes the asset code (the asset software component) combined with additional resources such as meta-data, background information, manual, installation test suite, configuration/authoring tools, tutorials, scientific data or tool with auxiliary functionalities. Packaging is the process of combining all files of an asset into an easy to find and easy to download entity. Essential for this is to add metadata describing the asset, i.e. its dependencies and content in such way that it can be retrieved easily. A RAGE Asset Package is distinct from and should not be confused with notations of a package as a reusable software library (e.g., jar, dll, and other binaries) or an installation package. A RAGE **Asset Package** is primary unit of **distribution** to be deployed in the **RAGE Asset Repository**. A game developer should be able to (1) search the Repository for an asset, (2) download it as a Package, (3) extract the asset code and other artefacts from the Package, and (4) integrate/install the asset software.

Every **Package** should have **Metadata**. Such metadata is necessary to facilitate:

- Overall quality assurance of an asset
  - To check whether the package contains necessary artefacts (e.g., API documentation) and to facilitate a proper use of a RAGE Asset Instance.
- Proper deployment and search of assets in the Repository
  - Is the package in a proper format that is required for deployment?
  - Does the asset code support the testing?
  - Does the asset code have dependencies on other assets?
  - Are any necessary legal contracts, licenses, and so forth in place to allow the use the asset?
  - Does the package contain relevant metadata to help a game developer finding the asset in the repository?

An XML Schema (described in DefaultProfile.xsd) in accordance with the RAGE Metadata model has been developed. It defines a standardized machine-readable syntax for creating Package Metadata. The Package Metadata will be validated against the Package Schema to verify compliance with RAGE distribution standards.

As a unit of distribution a **RAGE Asset Package** is a Package Interchange File (PIF) – a ZIP archive, containing the manifest file, metadata and artefacts (see Figure 9).

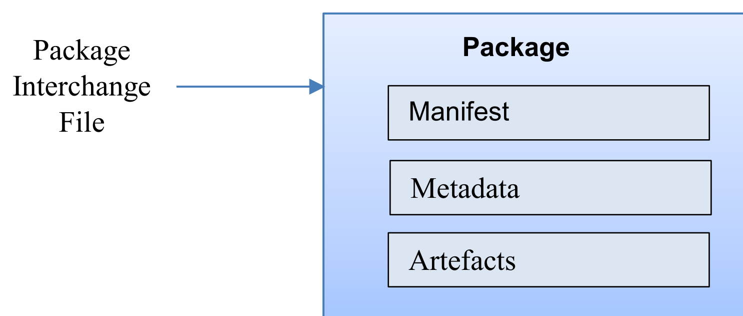

Figure 9. The Asset Package

All details related to the RAGE Asset package format and its implementation are in the Asset package format description of Annex 9.

10 ASSET DEVELOPMENT AND MAINTENANCE

In this section we will describe how assets are to be developed and maintained. While priority was given to technical architecture and its implementation, the elaboration of the asset development and maintenance methodology and the associated tools is work in progress, which will be carried out in close collaboration with WP2 and WP3: a cross-WP working group is preparing the approach. Still an outline can be presented here.

The methodology requires a generic description that should be neutral towards different software development environments (principle 7), neutral towards different asset programming methodologies (principle 8) and should cover the workflow across the entire asset lifetime (principle 9).

For being able to ultimately involve a wider community of software developers in asset creation, asset developers should be free to use their preferred tools, platforms and software development methodologies as much as possible (cf. principle 7). Although RAGE has restricted its scope to C# and TypeScript, the overall architecture would still allow asset developers to use any programming language and associated tools, provided that they comply with some basic requirements as to preserve the integrity of the overall asset framework.

The RAGE Asset Development Methodology is aimed at explaining all activities and operations needed for developing a valid RAGE Asset package. Main users of the RAGE Asset Development Methodology are the Asset (package) developers.

The RAGE Software Asset Repository is the core of the asset development infrastructure and contains all the needed tools to support Asset package developers. The main goal of this repository is to facilitate the process of RAGE Asset development, exchange, reuse and sharing. So, while we are developing the asset development infrastructure and documenting how best to use it, we will also co-develop the overall methodology for asset (package) development. This means that the asset development methodology should stay away from the detailed processes of asset design and programming, but instead should define *clear transition points and benchmarks* for assessing the (intermediate and final) products. Therefore RAGE has to put a set of quality assurance mechanisms in place. Before elaborating on this, the scope and starting points of such quality assurance system should be clarified.

1. Addressing external developers
   As we ultimately want the wider external community of software developers to contribute to the development of assets our approach is not restricted to RAGE partners but is targeting the wider community. Hence the guiding question is: what should any asset developer comply with?
2. Covering both asset development and asset maintenance
   Quality assurance needs to cover the whole asset lifecycle (principle 9). End-user feedback loops should be enabled to allow for bug reporting, bug fixing, continued testing and release management.
3. Open source rather than closed source
   In principle RAGE will opt for open source software, be it that all options for a profitable business models and sustained exploitation should remain open. Hence, the software licenses to be selected should allow for eventually charging asset users (game studios) with a fee (licensing proposal to be presented in chapter 11 of this deliverable).
4. Public rather than private development
   In accordance with the starting point of community involvement it is proposed to pursue an open development policy, which means that intermediate source code will be made publicly available. Still this could (or should) go with appropriate release and patch management.
5. Self-sustainable quality assurance
   Any quality control mechanism should be self-sustainable, largely in accordance with the self-organisation capabilities of the community and avoiding centralised governance and overheads.

6. Minimum set of quality assurance requirements
   RAGE should focus on a limited number of key quality issues and avoid procedures and details that would add limited value but require high overheads. This is meant to avoid repulsion and dropout among potential asset developers.

10.1 Asset quality assurance

Whatever software development methodology is used, whether agile, waterfall or scrum-based, the silent assumption is that each methodology would have a limited set of **transition points** yielding software artefacts of different stages of completeness and maturity, that can be assessed for their quality. This suggests to specify a set of maturity values potentially ranging from an early prototype to a fully tested and approved version, e.g. compliant with the latest operating system and hardware. For each maturity level a specific set of assessment criteria should be defined as much as a set of associated verification methods and tools. In addition, the maturity levels could reflect the actual operational state of the asset across multiple dimensions (e.g. tested/not tested, successfully implemented in game engine X, well-documented/not yet properly documented).

Asset quality assurance can be viewed from the following perspectives:

- Software quality
  How mature are the software components with all relevant software documentation, guides and artefacts in relation to software quality requirements;
- RAGE asset package quality
  How mature and well described is the software asset package with all needed and mandatory metadata descriptions and conforming to RAGE asset packaging requirements;
- Interoperability quality
  What different reference implementations in different game engine(s) for demonstrating interoperability were reported and demonstrated.

With respect to maturity Quality Assurance can be defined at three levels:

- Basic level
  We only assure, that the Asset Package is valid against the Asset packaging and Asset metadata models.
- Middle level
  In addition to basic level, we check available artefacts inside the Asset package, and test how well these satisfy the stated requirements and test cases for each software component inside the asset. Here we can have formal checks based on some general criteria, and more elaborated checks based on testing the software against preliminary defined testing scenarios.
- Higher level
  To demonstrate that the asset can be integrated at least in two different games (game engines) – here we can rely on external sources confirming this integration, or to check the integration by following user guides and configuration examples provided in the asset.

10.2 Asset tools and infrastructure

Beside the quality assurance approach this methodology also specifies what systems and tools for creating and handling assets will be provided by RAGE, and how these will be used in the process of developing RAGE asset packages. We will reuse existing tools and systems as much as possible and only create new tools when alternatives are unfavourable (cf. principle 10). An overview of all the tools and the overall infrastructure components is given below.

10.2.1 Coding tool

RAGE will not enforce a specific software development environment. Yet for starting the asset development we will make available a dummy asset (template) along with a project file that can be imported in selected programming tools.

10.2.2 Metadata tools

The following tools are anticipated:

- Asset metadata editor
- Asset metadata validator
- Taxonomy Viewer/Selector/Editor
- Metadata extractor (automated metadata generation)

In order to allow asset developers to configure and use their preferred metadata tools we will also make available the XML schema files for asset metadata and asset package metadata, which are needed for configuration of metadata tools.

- Asset metadata XML schema file
- Industry XSD schemas definitions
- Metadata styling CSS files
- Local JSON-LD formatted taxonomies

10.2.3 Configuration and authoring tool

Asset configuration refers to setting simple working conditions of the assets, e.g. the URL of a web service to be called by the asset's core functionality component. Authoring is technically closely related to configuration, but is more oriented towards manipulating the (educational) working or content of an asset. Authoring may require more complex editing operations than configuration. The provision of configuration editors and authoring tools can be considered a coding task that is at the discretion of the asset developer (principles 7 and 8). However, RAGE will make available a generic editor that can be configured by the asset developer to function as an asset configuration editor, or even as a simple asset authoring tool.

10.2.4 Source code management tool

Management of source code is required during both asset development and asset maintenance, covering supporting tools for code distribution, code review, issue tracking, revision control, release management and some more things. Currently GitHub is the predominant web-based repository service for source code management, allowing software developers to undertake distributed code development and maintenance.

Although services such as GitHub include a repository of software, this repository should not be confused with the anticipated RAGE repository. While the former's main role is to support the processes of software development and maintenance, the latter's role is to deliver ready-to-use software products to end-users. Yet, two factors may invoke some overlap here. First, the primary end-users in RAGE happen to be game developers, which are software developers by definition. Second, many software development approaches are based on the early and persistent involvement of end-users. To some extent this would remove the artificial distinction between development and exploitation, or between supplier and end user. Still, the RAGE repository should primarily be a publication platform targeting end-user communities – also including educators and trainers - within the context of the RAGE Ecosystem. For code management RAGE will use GitHub, in particular open GitHub repositories rather than private ones, as to involve the community of stakeholders as much as possible in the process of asset development. Ready-to-use asset versions would then be packaged and transferred to the RAGE repository for distribution. In sum:

- Open GitHub repositories, featuring
    - Code management
    - Code review
    - Bug tracking and debugging
    - Revision control
    - Release management (also sub-releases)

10.2.5 Asset packaging, storage and validation tool

For distributing releases the asset source code is transferred from Github and packaged along with additional resources. The packaging is supported by the Asset Manager Tool at repository level. Before acceptance of a package by the RAGE repository, various checks are carried out

to verify the correctness of the uploaded packages, e.g. the completeness of the package, mandatory metadata, conflicting variable names, quality assurance data, etc..

10.2.6 Asset distribution and support tool

Asset packages are made available in the RAGE asset repository. The RAGE (delivery) repository allows for downloading the package and allows for community services such as adding user reviews, user rating, discussion, bug posting, messages, and integration support (covered by WP6).

10.2.7 Installation and integration tools

RAGE will make available a widget for guided installation and integration with selected game engines. Also snippets with developing platform dependent code will be considered.

- Installation widget
- Platform dependent code snippets

11 ASSET SOFTWARE LICENSES

In order to protect the rights of RAGE software assets an internal working group has explored the licensing options, while taking into account principle 11 (cf. section 1,2): RAGE will select open source licenses that preserves ample flexibility for future RAGE business models. The working group started with a requirements analysis based on the guidelines and requirements outlined in the H2020 Work Programme 2014-2015, the Model Grant Agreement, the RAGE Consortium Agreement, and the RAGE Description of Action. This resulted in the following RAGE assets licensing requirements, summed up by principle 11 above:

- RAGE assets should be granted an Open Source (OS) license (formal, contractual, CA requirement)
- It should be possible to combine RAGE assets with third party assets into aggregated assets or libraries, without such third party assets being ‘contaminated’ by the RAGE OS license (not to impede future re-use and re-purposing of RAGE assets)
- It should be possible for partners to bring in existing assets (‘background’ or ‘prior art’) into the project without such background becoming ‘contaminated’ by the RAGE OS license or the source code becoming public upon publication under the RAGE project (not to impede the use of background from RAGE partners or third parties)
- It should be possible to change the license type after the completion of the project when required by the post-RAGE project business model (in view of the business model-under-development by WP7)

After investigating established software licenses, the working group long-listed five licenses. Upon closer analysis, three were shortlisted: Apache2.0, MIT, and BSD-2.
All three licenses allow for charging a fee for deploying the assets, and - after (minor) modifications - to grant a closed source license (optional, in view of the future business model). However, this holds only under the conditions that any background used in creating the asset:

1) is not subjected to any viral licenses,
2) does not prohibit a commercial model (e.g. some open source licenses do),
3) does not include third party software with licenses that prohibit a commercial model.

The Apache 2.0 license was eventually ranked first by the working group as it is the only of the three shortlisted licenses that protects against third parties claiming a patent on the RAGE software assets (a practice increasingly common in the US, not in Europe). If this would occur the Apache license invalidates the license and thereby invalidates the patent.
As indicated earlier however, adopting the Apache license implies that no royalties can be charged for background brought in by RAGE partners. Hence, when adopting the Apache 2.0 license, partners bringing in background should waive any such royalty claims. Likewise this restriction applies to third party software that is used as background.
Before putting up the proposal for decision making, the working group sought legal validation for the analysis and resulting proposal from Legaltree, a Netherlands-based law firm specializing in software licensing. Legaltree confirmed the analysis and resulting recommendation of the working group, upon which it was brought up for tiered decision making: a first proposal was put before the Executive Management Board (EMB) early September 2015, and upon acceptance by the EMB, to the Strategic Management Board at its meeting mid September. The following verbatim proposal was accepted unanimously (SMB minutes September 15):

RAGE partners will grant the Apache 2.0 license to the software assets they develop under RAGE; partners accept that no royalties will be charged for any background/prior art that they or third parties bring in; and any background/prior art with a viral license can only be brought in when it does not (potentially) conflict with the RAGE business model. Exceptions, for example because of background brought in with another license, will be looked into on a case-to-case basis.

Once formalized by the SMB, the license text and a brief guide on how to grant the license to the RAGE software assets was included in the online RAGE Handbook, and granting the license was included in the QA-checklist for assets by WP2 and WP3.
The detailed licenses analysis report is in Annex 10.

12 LIST OF ANNEXES

Annex 1 Needs interviews

Preprint of:
Saveski, G. L., Westera, W., Yuan, L., Hollins, P., Fernández Manjón, B., Moreno Ger, P., & Stefanov, K. (2015, 9-11 December). What serious game studios want from ICT research: identifying developers' needs. In: A. De Gloria and R. Veltkamp (Eds.), Proceedings of the GALA 2015 Conference, December 7-8, Rome, Italy, LNCS 9599, pp. 1–10. DOI: 10.1007/978-3-319-40216-1_4

Annex 2 RAGE Technology Report

RAGE internal report

Annex 3 RAGE Asset System Architecture

Preprint of:
Van der Vegt, W., Westera, W., Nyamsuren, E., Georgiev, A. and Martínez Ortiz, I. (2016) "RAGE Architecture for Reusable Serious Gaming Technology Components", *International Journal of Computer Games Technology*, advance online publication. doi:10.1155/2016/5680526. Retrieved from http://www.hindawi.com/journals/ijcgt/2016/5680526/.

Annex 4 RAGE reusable game software components and their integration into serious game engines

Preprint of:
Van der Vegt, W., Nyamsuren, E. & Westera, W. (2016). RAGE Reusable Game Software Components and Their Integration into Serious Game Engines. In: Georgia M. Kapitsaki and Eduardo Santana de Almeida (Eds.), Bridging with Social-Awareness, 15th International Conference, ICSR 2016, Limassol, Cyprus, June 5-7, 2016, Proceedings, Lecture Notes in Computer Science, Volume 9679 2016, pp. 165-180.

Annex 5 RAGE Vocabulary and metadata categorisation

RAGE internal report

Annex 6 RAGE Metadata Model

RAGE internal report

Annex 7 RAGE Software Asset Model and Metadata Model

Preprint of:
Georgiev, A., Grigorov, A., Bontchev, B., Boytchev, P., Stefanov, K., Bahreini, K., Nyamsuren, E., Van der Vegt, W., Westera, W., Prada, R., Hollins, P. and Moreno Ger, P. (2016). The RAGE Software Asset Model and Metadata Model. Accepted for the international Joint Conference on Serious Games 2016, September 26-27, Brisbane, Australia.

Annex 8 RAGE Asset Repository Infrastructure

RAGE internal report

Annex 9 RAGE Asset Package

RAGE internal report

Annex 10 Software licencing

RAGE internal report

13 REFERENCES

Ackerman, L., Elder, P., Busch, C., Lopez-Mancisidor, A., Kimura, J., & Balaji, R. S. (2008). Strategic Reuse with Asset-Based Development. IBM. Retrieved from http://www.redbooks.ibm.com/redbooks/pdfs/sg247529.pdf

ADL (2016). Experience API v 1.0.2. Available at: https://github.com/adlnet/xAPI-Spec/blob/master/xAPI.md [Accessed May 28, 2016].

Dekkers, M. (2013). Asset Description Metadata Schema (ADMS). W3C Working Group. Retrieved from http://www.w3.org/TR/vocab-adms/

IMS-CP, 2004, http://www.imsglobal.org/content/packaging/cpv1p1p4/imscp_infov1p1p4.html

Mahmood, S., Lai, R., & Kim, Y. S. (2007). Survey of component-based software development. IET software, 1(2), 57-66.

Niekerk, A.J. van (2006) The Strategic Management of Media Assets; A Methodological Approach. Allied Academies, New Orleans Congress, 2006

Saveski, G. L., Westera, W., Yuan, L., Hollins, P., Fernández Manjón, B., Moreno Ger, P., & Stefanov, K. (2015, 9-11 December). What serious game studios want from ICT research: identifying developers' needs. In: A. De Gloria and R. Veltkamp (Eds.), Proceedings of the GALA 2015 Conference, December 7-8, Rome, Italy, LNCS 9599, pp. 1–10. DOI: 10.1007/978-3-319-40216-1_4

Van der Vegt, W., Nyamsuren, E. & Westera, W. (2016a). RAGE Reusable Game Software Components and Their Integration into Serious Game Engines. In: Georgia M. Kapitsaki and Eduardo Santana de Almeida (Eds.), Bridging with Social-Awareness, 15th International Conference, ICSR 2016, Limassol, Cyprus, June 5-7, 2016, Proceedings, Lecture Notes in Computer Science, Volume 9679 2016, pp. 165-180.

Van der Vegt, W., Westera, W., Nyamsuren, E., Georgiev, A. and Martínez Ortiz, I. (2016b) "RAGE Architecture for Reusable Serious Gaming Technology Components", International Journal of Computer Games Technology, advance online publication. doi:10.1155/2016/5680526. Retrieved from http://www.hindawi.com/journals/ijcgt/2016/5680526/.

Westera, W., Van der Vegt, W., Bahreini, K., Dascalu, M. and Van Lankveld, G. (2016). Software Components for Serious Game Development. Paper contribution to the 10th European Conference of Game-Based Learning, October 6-7, 2016, Paisley, Scotland.