D4.1.3 SRA Parser and Validation

Project ref. no.	FP7-ICT-2011-7 GA-287639
Project Acronym	SCENE
Start date of project (dur.)	1 November, 2011 (36 months)
Document due Date:	31/07/2014 (M33)
Actual date of delivery	28/07/2014 (M33)
Leader of this deliverable	IVCI
Reply to	haccius@intel-vci.uni-saarland.de
Document status	Final

Version	Date	Description
1	11/07/14	Version for peer review
2	18/07/14	Rework after peer review.- Version sent to Coordinator
3	28/07/14	Final pdf. submitted

SEVENTH FRAMEWORK PROGRAMME

Deliverable Identification Sheet

Project ref. no.	FP7-ICT-2011-7 GA-287639
Project acronym	SCENE
Project full title	Novel Scene representations for richer networked media
Document name	SCENE_D4.1.3_13062014
Security (dissemination level)	PU
Contractual date of delivery	Month 33, 31.07.2014
Actual date of delivery	Month 33, 28/07/2014
Deliverable number	D4.1.3
Deliverable name	SRA Parser and Validation
Type	D
Status & version	Final
Number of pages	17
WP / Task responsible	IVCI
Author(s)	Javier Montesa, Pablo Arias, Wolfram Putzke-Röming, Jörn Jachalsky, Andrej Schewzow, Ingo Feldmann, Johannes Furch, Sammy Rogmans, Nick Michiels, Jean-Yves Guillemaut, Martin Klaudiny, Thorsten Herfet, Victor Matvienko, Christopher Haccius
Other contributors	
Project Officer	Philippe Gelin
Abstract	This document describes the software deliverable of the SRA parser and validation. The software deliverable includes the SRA API, a package enabling the use of the underlying SRA structure as well as test data, validation tests and usage examples.
Keywords	Scene Representation Architecture, API, Parser, Persistence
Sent to peer reviewer	11.07.2014
Peer review completed	*15.07.2014*
Circulated to partners	18/07/2014
Read by partners	Via plone
Mgt. Board approval	pending

Table of contents

1. Public executive summary

Deliverable 4.1.3 comprises a software implementation of the Scene Representation Architecture (SRA) and a description of this software. It implements the conceptual outline of the SRA from D4.1.1 and the technical specification of the SRA in D4.1.2. The SRA is a core development connecting data acquisition in SCENE with data processing and data rendering algorithms. Besides a description of the delivered software this document also provides usage instructions and tests for validation.

In the SCENE project, multidimensional data is acquired by advanced hardware like the Motion Scene Camera and different sensor setups as outlined in D3.3.3. Sophisticated algorithms as presented in D4.2.2 and D4.3.3 need to access this data, and finally the data is visualized using different scene rendering schemes. The SRA serves as the SCENE data exchange format, providing both, captured and generated data, as well as metadata and annotations to algorithms and visualization tools. Data is made available in the most flexible way, allowing access to arbitrary parts in arbitrary formats contained in the Scene Representation.

The Software is C++ source code with CMake files for facilitated installation. It has been tested on both Windows 7 and Ubuntu.

2. Introduction

Purpose of this Document

This document contributes to the SCENE objective O4.

> ***O4:*** *(Research and Develop) A video scene representation that can combine image based and computer graphic information with metadata to deliver an inherent 3D, spatio-temporally consistent worldview.*

The DoW describes the general task of WP4 as follows:

> *The goal is to bridge the gap between image-based formats (texture plus depth, multi-view, etc.) and object-based formats that are computer graphics oriented (polygon meshes, voxel based representations, etc.). The new format will be based on 4D patch-like super-structures, which are parameterized in the 4D spatio-temporal domain. Additional parameters, such as 4d spatio-temporal patch shapes, patch neighbourhoods, etc. will create a powerful scene description that is suitable for future immersive visual media content production. The new scene format implicitly includes state of the art scene representations, such as texture plus depth, voxels or meshes. Beyond this, it allows additional powerful spatio-temporal scene parameterizations, which allow immersive interaction and navigation as well as manipulation and rendering without any semantic knowledge about objects or masks in the image domain.*

More specifically,

in WP4T1 SCENE dissolves the limits between sample-based (video, depth sensors) and object-. shape-based (computer graphics) representations. Since real-time extraction of object information from scenes is still in its infancy while the real-time generation of patches from triangular or polygonal meshes is state of the art the Scene Representation Architecture (SRA) will comprise the following components:

- *The **SRA-base layer** will be a patch assembly. Patches are characterized by their position in 4-dimensional space, their shape, orientation, color and orientation and their transparency. The SRA base layer enables a smooth merging of camera- and computer-generated content and a scalable rendering across a variety of devices and scene complexities. It does, however, lack information on spatio-temporal coherence and on more advanced material characteristics like reflectivity.*
- *The **SRA-scene layer** will add advanced information about the patches as far as they are available. For computer-generated scenes characteristics like lighting, materials (reflectivity, shaders etc.) will be added. For camera-generated scenes spatio-temporal coherence information derived in WP3T3 and WP4T2/3 will be added together with extracted material reflectance WP4T4 to improve the co-existence of natural and synthetic scene components beyond the state of the art.*
  The coherence layer requires a unique identification of the patches in the base layer. Consequently an unambiguous way of addressing patches will be developed.
- *Finally, the **SRA-director layer** will provide information required to render the final scene in the way the director foresaw. Ingredients are view point, exposure times, apertures / depth of field, camera motion etc.*

This document describes the software demonstrator which presents a parser and validation for the above mentioned Scene Representation Architecture. The parser exposes an interface to the user which allows adding many standard file types to the SRA, most importantly all file types which have been agreed as exchange types in the SCENE project. Moreover, it allows reading these file types for usage by further processing or visualization applications. A description and usage explanation of the accompanying demo datasets is given as well.

Document Structure

Section 3 describes the software implementation of the Scene Representation Architecture. The SRA consists of a backend implementation providing a persistency structure and a frontend implementation exposing functionality to users.
Section 4 gives details about data sets generated in the SCENE content and available for tests with the SRA. Instructions how to use the SRA with the aforementioned datasets and descriptions of the included validating tests are given in Section 5. Two examples illustrating the use of the SRA backend and frontend are explained in Section 6. Finally a conclusion is drawn in Section 7.

Related Documents

This document and the software deliverable are based on the description of the SRA contained in D4.1.1 and the technical specification of the SRA as described in D4.1.2. Core requirements of the SRA are to structure and persist data produced by the various acquisition methods evaluated and used in SCENE, as explained in D3.3.3, with the Motion Scene Camera presented in D3.2.3 as the central acquisition hardware.
The SRA provides an interface for Scene analysis and post-processing algorithms described in D4.3.3, as data from the SRA is an input to those algorithms and output data is written back to the SRA. Finally, D5.1.3 describes renderers capable of visualizing data in the SRA format.

3. Software Description

The Scene Representation Architecture combines a data structure and the necessary procedural interface to access it. The SRA demonstrator shows different small use cases of how to

- write data to the SRA,
- access data in the SRA, and
- visualize data from the SRA.

The software is implemented in C++, and was tested under Windows and Ubuntu operating systems. For Ubuntu, CMake files exist which facilitate the make process. The SRA was successfully build as Visual Studio and Eclipse project, allowing programmers a wide choice of programming interfaces.

The SRA is structured into two major parts, the backend which provides a data structure, and a frontend to access the backend data structures.

SRA Backend

The backend provides all the necessary structures to store any kind of data relevant to SCENE representations. Core to a Scene is its structure in the SRA. The Scene contains Acels, which we call the Atomic Scene Elements, the elements a scene is composed of. Atomic here is the smallest structure that is applicable in a given context. This means, that a whole image (which might, as a frame, be the smallest structure in a video) is an acel, but at the same time its individual pixels are acels in image processing context. Therefore, in the underlying structure, acels, are again compositions of further Acels, Values, and Properties. Values can be any data or data array. Properties contain a descriptive name and further Acels assigning values to the property. This structure is presented in Figure 1. For a full description of the conceptual meaning behind the different layers represented in a Scene and Acels as the basis structures please refer to Documents D4.1.1 and D4.1.2.

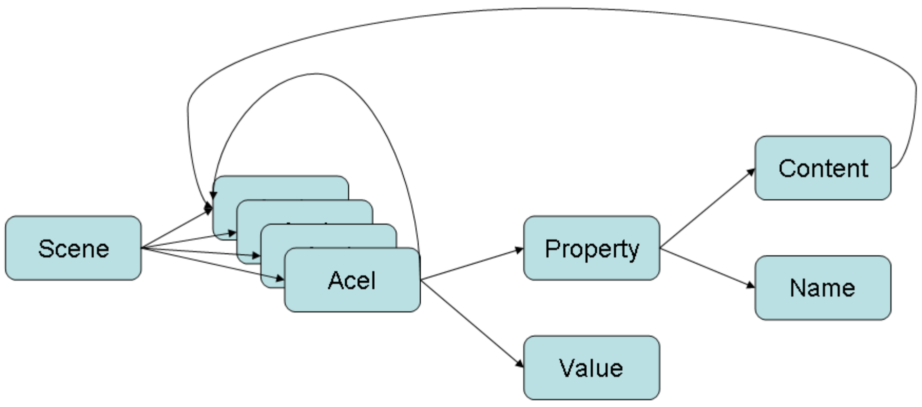

Figure 1: SRA Sructure of Acels

Behind the backend can be any kind of mechanism providing persistency. Currently the Google Prototype Buffers are implemented, but it can be extended to any other kind of structure which supports element structuring as displayed in Figure 1.

The Scene Backend is accessed through the sra_scene.h header file. This function exposes the following public functions to work on Acels:

- **virtual oid_t Id()**: the unique id, equal for the acels with identical content

- **virtual bool SetId(oid_t id)**: sets acel id, returns false if the ID is already defined
- **virtual IAcelList* ChildAcels()**: gets the list of child acels
- **virtual std::shared_ptr<IAcel> Property(string name)**: retrieves property with given name
- **virtual std::shared_ptr<IAcel> AddProperty(string name, oid_t id)**: adds a property with given name and unique ID
- **virtual bool DeleteProperty(string name)**: delete a property with the given name
- **virtual std::shared_ptr<IAcel> AddChild(oid_t id)**: add a child acel with given ID virtual
- **bool DeleteChildId(oid_t id)**: delete child Acel by ID
- **virtual bool DeleteChild(size_t index)**: delete child Acel by index in list of child Acels.
- **virtual IAcelValue* Value()**: retrieve the value of an Acel
- **virtual IAcelValue* SetValue()**: set the value of an Acel
- **virtual void ClearValue()**: remove the value of an Acel
- **virtual ~IAcel()**: destructor for Acel

SRA Frontend

The SRA frontend structures and facilitates the use of the SRA backend. Since elements in the SRA are ID based, the frontend offers an ID management system. This way, the user can add Scene content in the form of Acels without worrying about IDs. Furthermore, it exposes a couple of easy-to-use functions to the user to write and retrieve various data types to and from the SRA. The file sra.h exposes the following functions to the user:

- **Scene(std::string sceneFile)**: start a new or open an existing Scene with the given filename. *sceneFile* is the parameter containing the filename.
- **virtual ~Scene()**: desctructor for Scene
- **int writeScene(std::string sceneFile)**: write a Scene to file, where *sceneFile* identifies the file name
- **bool removeAcel(int id)**: remove an Acel with given *id* from Scene
- **int addMesh(std::string filename, meshType::type type, int parentId, int **ids)**: add a mesh (from an arbitrary format file) to an existing scene. *fileName* specifies the input file, *type* gives the data type (OBJ, STY, ...) and *parentId* gives the ID of the parent acel to the new mesh. *ids* is the returned list of all child acels which are added in the process, as each mesh contained in a file is stored as an individual acel.
- **aiMesh *getMesh(int id)**: retrieve a mesh from a Scene. Mesh is returned as an ASSIMP Mesh structure
- **int addMap(std::string filename, mapType::type type, int parentId)**: add a Map (image, depth) to an existing scene
- **CImg<unsigned char> *getMap(int id)**: retrieven a Map from a Scene. Map is returned as CImg structure
- **int addFilePath(std::string filename, int parentId)**: add a file path linking an external file to a Scene.
- **std::string getFilePath(int id)**: retrieve the path to a file from a Scene.
- **int addBinaryData(std::string filename, int parentId)**: add a binary file to a Scene.
- **int getBinaryData(int id, char **buffer)**: retrieve a binary file from a Scene
- **int addCameraProjectionMatrix(const std::string cameraName, const sra::float_buffer_t& projectionMatrix, const std::vector<std::string>& correspondingStreams, const int parentId)**: add a camera projection matrix to a Scene.
- **bool getCameraProjectionMatrix(const int id, std::string& cameraName, sra::float_buffer_t& projectionMatrix, std::vector<std::string>& correspondingStreams)**: retrieve a camera projection matrix from a Scene.

4. Description of accompanying Data Sets

Numerous sample data sets were produced in the context of the SCENE project. For the datasets introduced below samples are included in the *sample-data*-folder accompanying the deliverable. For access to the whole data please contact the consortium (http://3d-scene.eu/).

Paris Pool Data Set

The Paris Data Set is one of the earlier data sets produced by the SCENE consortium. It includes bowling and pool scenes, taken from different cameras. The raw footage of the take is in the *m2t* format which is the Blu-ray Disc Audio-Video (BDAV) MPEG-2 Transport Stream (M2TS) container format. The first take has a total number of 5,561 frames at a resolution of 1440x1080 pixels each, stored in the *png* format. A sample frame along with its corresponding depth map is shown in Figure 2. The left image is the original frame and the one on the right shows the depth map.

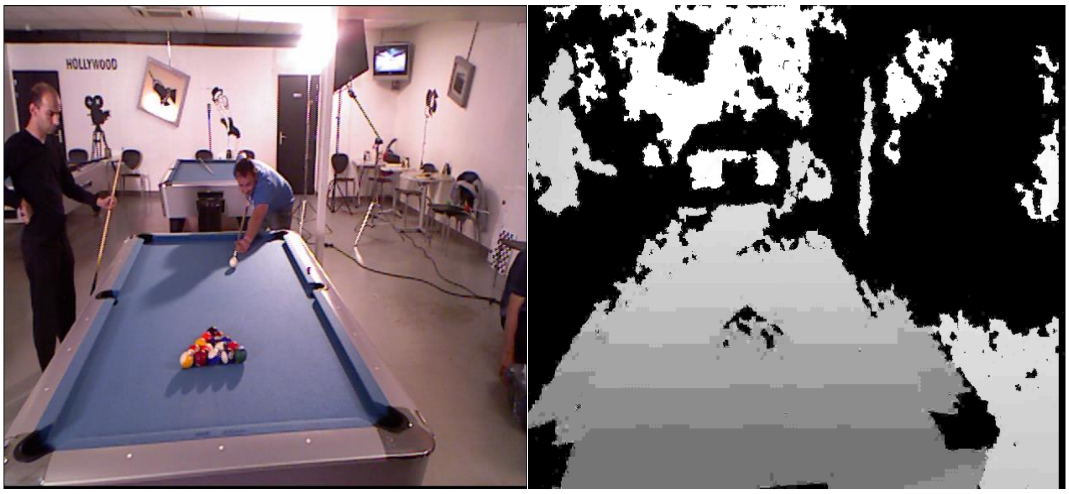

Figure 2: Sample of Paris Pool Data

Another 2000 frames of this scenario have been captured with an early stage Motion Scene Camera and a second Alexa on a stereo rig. Its raw data (*tiff* format) from the left and right cameras are included in their native resolution of 2048x1536. This data is accompanied by depth maps captured by a time of flight sensor. However, while the time of flight sensor was at a later stage of the SCENE project integrated into the camera to shoot through the same optical system, this is not yet the case for the available depth data. The setup at the capture site is shown in Figure 3. Calibration data for the cameras is available in the provided data set.

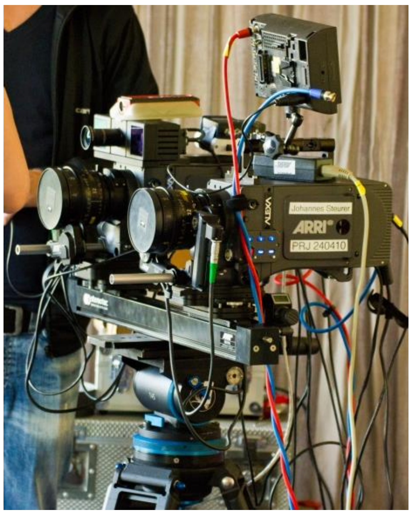

Figure 3: Early State Motion Scene Camera Setup

Paris Bowling Bar Data Set

This data set shows a man standing at the bar table in an Bowling alley. This data set has been shot using 8 calibrated cameras. The calibration information of the cameras has been included as part of the data set. The data from each of these cameras are stored in *bmp* format having a resolution of 1920x1080 pixels per frame. Depth information for each frame is calculated and stored in *exr* format. A sample of the images from two different cameras (Cam1 and Cam7) is shown in Figure 4.

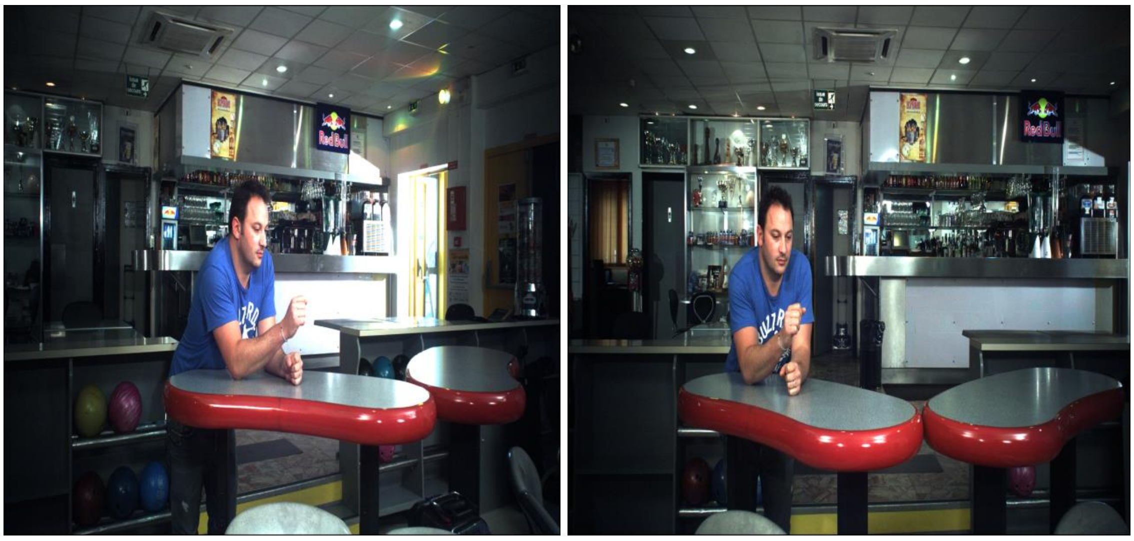

Figure 4: Sample of Paris Bar Table Data Set

Mannequin Data Set

The Mannequin Data Set shows a bevy of mannequins with a female actor interacting with the scenario. The data set includes data captured with the Motion Scene Camera extended by an ARRI Alexa on a stereo rig. The data from both, left and right camera, is included. The data consists of RGB frames stored in the logC wide gamut color space. This color space can be converted to linear color space using a LUT. There are a total of 408 frames each having a resolution of 1920x1080.

The depth maps captured by the TOF camera have a native resolution of 325x198 pixels. They are stored in 16bit *tiff* format. An example of the original image and its corresponding depth map is displayed in Figure 5.

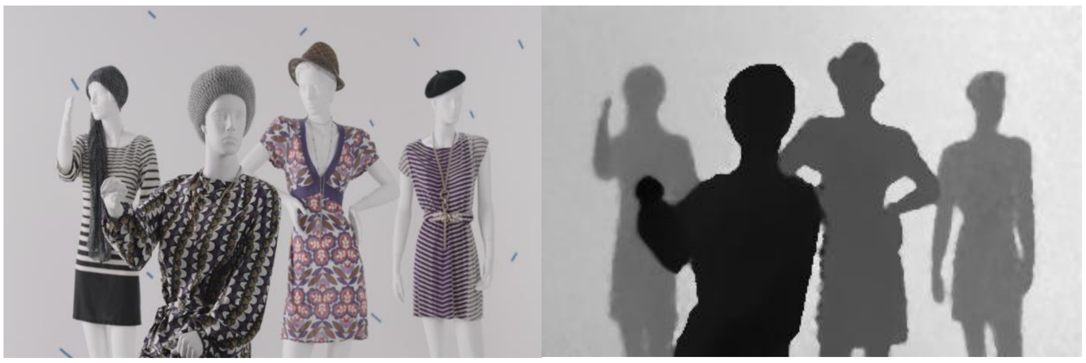

Figure 5: Sample of Mannequin Data Set

Laura Data Set

The Laura Data Set display a female actor in front of a green screen with simple motion. The data set contains RGB Images with a resolution of 1920x1080 calibrated according to ITU Rec. 709. Spherical depth maps which have been registered to the RGB image. Each depth map has a resolution of 352x198 pixels and has lens distortions corrected. The aligned and optically undistorted depth is part of the data set, as well as the raw spherical depth maps. Furthermore, upscaled HD depth maps are included, with a resolution of 1920x1080 generated using the joint-bilateral NAFDU algorithm. Figure 6 shows an example of the data set: the actress in the studio and the corresponding depth map.

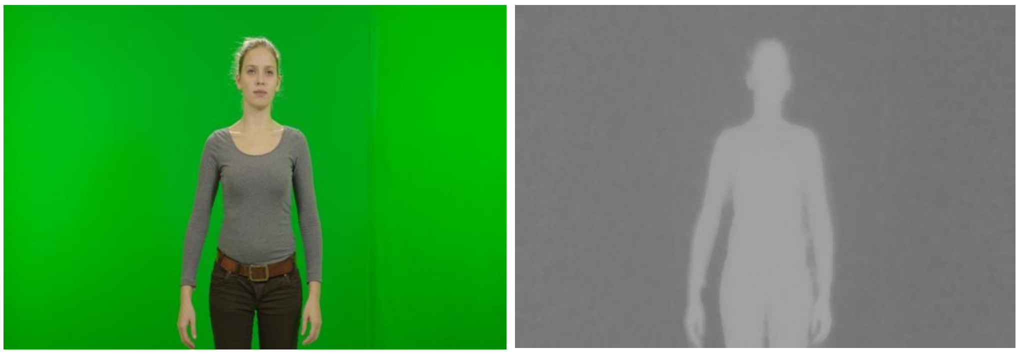

Figure 6: Sample of Laura Data Set

5. Usage and Validation

To work with the SRA frontend the sra.h header file needs to be included. This file is located in *./sra/scene-persistence/sra.h*, The functionality is described in Section 3.1. When working on the SRA backend, the file sra-scene.h needs to be included. This file is located in *./sra/scene-core/sra-api/sra-scene.h*, and its core functionality is also described in Section 3.2.

In the parent SRA folder a CMakeLists file is contained. Using CMake the whole SRA including unit tests can be build. These unit tests validate different use cases of the SRA.

Load / Write Camera Projection

This unit test has two files, *LoadCameraProjectionTest* and *WriteCameraProjectionTest*. It showcases how camera projection matrices can be stored in the SRA and retrieved from the SRA. A camera projection matrix is a 3x3 matrix of integer or double values. Two examples demonstrate a simple and a more complex use of writing such a matrix to the SRA in a structured form.

Load / Write EXR Stream

This unit test consists of the files *LoadEXRStreamTest* and *WriteEXRStreamTest*. It demonstrates the use of EXR streams in the SRA. The EXR format allows storing multi-channel maps, which becomes relevant when storing RGB plus depth information in a single frame or high precision depth data. For EXR data OpenCV is required.

Load / Write Stream

Compared to the above mentioned use case this example demonstrates the storage of arbitrary consecutive files in a stream. It consists of two files, *LoadStreamTest* and *WriteStreamTest*. Streams extend the SRA structure by a common format to represent consecutive data, e.g. frames in a video.

Load / Write Meshes

This example is contained in the file *PersistanceAPI-Test*. It demonstrates the storage of meshes and retrieving whole meshes or individual vertices from a mesh. Meshes are returned from the SRA as ASSIMP Meshes, thus requiring the ASSIMP Library.

Load / Write Maps

Loading and writing maps, like images or depth maps, is demonstrated in this test. It is contained in *PersistenceAPI-Test* and showcases storage of whole maps, their retrieval and the access of individual channels or pixels. Image access requires the OpenCV library.

Load / Write File Paths

Writing file paths to the SRA is a common way to include objects into a Scene, without restructuring a file to the SRA. A demonstrating example is included in *PersistenAPI-Test*, which shows how file paths can be inserted into and read from a Scene.

Load / Write Binary Data

Speed, size and portability are the major reasons for including binary data into a Scene. Such data is not further structured, that means, if an image is stored as binary data, individual pixels cannot be accessed, or if a mesh is stored as binary data its vertices cannot be referenced individually. An example showcasing the insertion and extraction of binary data to and from a scene is shown in *PersistenceAPI-Test* as well.

Visual Validation

Scenes can be visualized by different tools implemented in the SCENE project. One such application is the Scene Renderer, which is fully described and provided as software in D5.1.3.

6. Examples

Working with the Frontend

The following example illustrates the use of the SRA frontend. In this example, the user will add an .obj file containing several meshes to a scene. This requires the ASSIMP library to be installed.

In his source, the user needs to add the SRA frontend and a filetype specification:

```
#include <scene-persistence/sra.h>
#include <scene-persistence/backend/filetypes.h>
```

He can than create a scene or load an existing scene file. This is done automatically, depending on the existence of the named file:

```
string sceneName = "myScene.sra";
Scene myScene(sceneName);
```

Any new scene objects need to be added to a parent Acel. If no parent Acel is specified, or the specified Acel does not exist, the new content is added to the top Scene level. Here, we add the contents of file “test.obj” to the parent acel with id 10. The file type needs to be specified with the input; this information is coming from defined types in “backend/filetypes.h”.
The function returns an integer value with the number of meshes added to the scene. This number corresponds to the number of meshes contained in the input file. Additionally, a pointer to a list of ids is returned, allowing the user to access the newly added meshes in the scene structure.

```
string meshName = "test.obj";
int parentID = 10;
int numOfMeshes = myScene.addMesh(meshName, meshType::OBJ, parentID,
&ids);
```

The mesh, all faces and vertices, normal vectors and texture coordinates are now stored as individually structured acels in the scene file and can be individually accessed and processed.

Meshes as a whole are returned as aiMeshes, the native structure of the ASSIMP library. To do this, the id of a mesh in the Scene file has to be passed to a function asking for the mesh. The following query returns the first of the meshes added above to the Scene file.

```
aiMesh *myMesh = myScene.getMesh(ids[0]);
```

Working with the backend

Programmers can access the backend directly. In this case, they need to take care of id management, creation of scene files and parsing of content themselves. To work with the backend, the backend headers need to be included in the source code:

```
#include <scene-core/sra-api/sra_scene.h>
```

```
#include <scene-core/sra_implementation.h>
```

In the backend, a Scene is constructed as a PScene object:

```
PScene sc = sra::make_scene_representation(".");
```

The programmer needs to choose an ID that is currently unused:

```
sra::oid_t id = 2;
```

And can than add any integers, strings, vectors or other standard data types to a child acel created with this id:

```
PAcel acelVal = globals->AddChild(id);
acelVal->SetValue()->SetFloat(50.3);

id++;
PAcel acelVal = globals->AddChild(id);
acelVal->SetValue()->SetString(“test”);
```

To retrieve a value from the Scene structure, the programmer needs to know the ID and the type of the data he wants to access. The following line retrieves the float value stored in the lines above:

```
double out = globals->GetId(2)->Value()->GetFloat();
```

7. Conclusion

This document describes the software deliverable D4.1.3. It concludes the development of the SRA as described in the DoW of the SCENE project. However, during the project the SRA has been developed to provide a structure appreciated for merging content from multiple sources with meta-information describing whole scenes. Due to that the development of the SRA has by far not come to an end: it just reached a state where it becomes useful. Future usage in different projects and for other goals will require extensions of the SRA, and many important aspects like compression or streaming of the SRA have not yet been touched. While this document describes a working software solution and presents a successful evaluation and usage of this package, we are still looking forward to the SRA being further maintained, extended and used by both academic and industry.

8. Laboratory Validation of Prototypes

Assessment / Validation measure	Level / Threshold	Comments and conclusions
Store Data in SRA	*The SRA includes a number of file parsers to store different file formats in SRA. Parsing those files can either be successful or unsuccessful.*	*Tests as described in Section 5 have shown that many map types (all those supported by OpenCV) and many mesh types (all those supported by ASSIMP) as well as any form of data included as file path or binary data is supported.*
Retrieve Data from SRA	*The SRA should return data stored in the SRA for further processing. Returning this data can be either successful or unsuccessful.*	*Tests as described in Section 5 have shown that maps and meshes can be returned as types defined by the respective libraries. File paths and binary data of arbitrary formats can be returned. Furthermore, individual mesh vertices and vectors as well as pixels in meshes can be accessed.*
Visualize Data from SRA	*Data in the SRA format can be visualized using the Scene Renderer. Understanding an SRA file can either be successful or unsuccessful, visual evaluation of the renderer is part of the evaluation of D5.1.3.*	*As described in D5.1.3 data in SRA format can be read and visualized by the Scene Renderer, thus providing a validation for the content of a Scene file.*