D4.1.2 Technical Specification of the SRA

Project ref. no.	FP7-ICT-2011-7 FP7- 287639
Project acronym	SCENE
Start date of project (dur.)	1 November, 2011 (36 months)
Document due Date :	30 April 2013 (month 18)
Actual date of delivery	22 May 2013
Leader of this deliverable	IVCI
Reply to	haccius@intel-vci.uni-saarland.de
Document status	Final

Deliverable Identification Sheet

Project ref. no.	FP7-ICT-2011-7 FP7- 287639
Project acronym	SCENE
Project full title	Novel Scene representations for richer networked media
Document name	SCENE_D4.1.2_30052013
Security (distribution level)	PU
Contractual date of delivery	Month 18, 30.04.2013
Actual date of delivery	Month 19, 22.05.2013
Deliverable number	D4.1.2
Deliverable name	Technical Specification of the SRA
Type	Report
Status & version	Final
Number of pages	39
WP / Task responsible	IVCI
Author(s)	Thorsten Herfet (IVCI), Victor Matvienko (IVCI), Christopher Haccius (IVCI)
Other contributors	-
Project Officer	Philippe Gelin
Abstract	The technical specification of the Scene Representation Architecture is an important step towards the integration of all SCENE related algorithms and tools into a unified project. This document contains the technical realization of the conceptual description of the Scene Representation Architecture contained in document D4.1.1. Furthermore, it presents an API enabling the requirements to the Scene Representation Architecture identified in document D4.3.1.
Keywords	Scene Representation, API, Technical Specification
Sent to peer reviewer	24 April 2013
Peer review completed	21 May 2013
Circulated to partners	22 May 2013
Read by partners	Yes
Mgt. Board approval	Pending

Table of contents

1 Public Executive Summary

The technical specification of the Scene Representation Architecture is an important step towards the integration of all SCENE related algorithms and tools into a unified project. This document contains the technical realization of the conceptual description of the Scene Representation Architecture contained in document D4.1.1. Furthermore, it presents an API enabling the requirements to the Scene Representation Architecture identified in document D4.3.1.

The document begins with a recap of the conceptual description and the results of a feedback request sent to multiple experts to receive feedback on this conceptual description. This conceptual evaluation is succeeded by a section briefly listing the main requirements from an algorithmic and use case perspective. The main part of this document technically describes the implementation of a SCENE API enabling already defined requirements and allowing future project development and enhancements at the same time.

2 Introduction

The SCENE project develops novel scene representations for digital media, which go beyond the current ability of either sample based or model-based methods to create and deliver richer media experiences to either 2D or 3D platforms. From a technical and research point of view, one challenge of this task is to capture, process and modify 2D and 3D content in a joint, efficient and modular way in order to keep the advantages and the power of existing representation formats and add additional 3D scene features. An architecture which meets these challenges was introduced and discussed in SCENE Deliverable D4.1.1. The technical requirements from an algorithmic view are contained in SCENE Deliverable D4.3.1. It is strongly recommended to be familiar with both documents when dealing with the contents of this document.

This document contains the technical specification of the Scene Representation Architecture as conceptually introduced in D4.1.1. Main challenges here are to provide a flexible and extendable specification for an on-going project with ever changing and ever increasing demands. The proposed solution is an API which provides access to SCENE data and basic computational functionality. As core file format related specifications are depending on multiple demands currently still under development an API provides the most adequate solution to meet these demands. Furthermore, format specific developments like storage, compression or streaming capabilities need to be researched in dedicated future project parts and findings in these areas will help to design and specify a SCENE file format. Up to that point the proposed API is a lot more flexible and extendable compared to a fixed file format, and allows adopting the underlying file format to the future requirements.

The following figure presents a rough conceptual outline of the proposed API as an interface between the scene data and tools that make use of the scene data. An arbitrary number of sources can feed into the scene data, and the sources itself as well as the converters to convert into the scene data format are not part of the technical specification. In a similar fashion, the technical implementation of the interface between API and scene data is not specified, but only requirements are given here. The interface for the tools of the SRA API is specified, such that tools can be implemented which employ scene data independent of the underlying data structure.

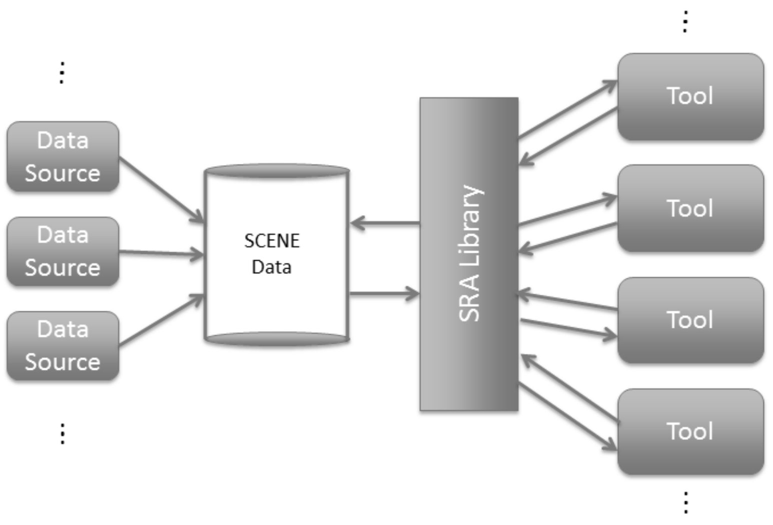


Figure 1: Overview of Structure of Scene Components

This specification comes with a basic implementation of the SRA which also serves as a prototype implementation for further data and module requirements.

3 Structure of the document

This document starts with the conceptual description of the SRA. This description has been proposed in D4.1.1. Feedback requests have been sent to over 60 experts in more than 20 companies. In these requests the experts have been asked for their opinion on the Scene Representation Architecture. The evaluation of this feedback has been included into this document as a motivation for the further specification and implementation of the SRA.

In the following section the Technical requirements motivating the design of the SRA are summarized again. Those technical requirements are fully specified in D.4.2.1.

Afterwards the technical specification of the SRA is given. This technical specification aims to provide an easily extendable framework for future implementations. The core goal is to provide basic functionalities already and provide the necessary functionality for further enhancements according to the demands of other tools. Furthermore, the addition of further functionality is exemplarily shown for the scene renderer which uses the SRA to render scene content. The structure here is a presentation of access to the scene data first and second how tools and modules are used in the context of the SRA.

The example of video rendering shows step-by-step how a communication between renderer and SRA is established and used.

This document concludes with comments on the file structure of the SRA. The proposed API does not enforce any file format or structure. All data formats currently or prospectively required can easily be included. An example implementation illustrates the use of Google Protocol Buffers [1] to store hierarchically structured data.

4 Conceptual Description of SRA

Common video formats have only one graphics layer, which contains the visual information of the video. Post processing steps can only deal with information available in this layer, which is at the same time the same information that is seen by any user. The Scene format's most important innovation is that the user does not see what is captured. This means we can have a lot more information which helps in processing steps but which shall not be seen by the final viewer. A simple example is the following: A moving object shall be motion blurred. In a traditional video the object would be shot with camera parameters such that the object is blurred (disturbed data) post processing of this blurred data is difficult, if not impossible. With the scene format we can capture the moving object in the best possible quality (not blurred) and introduce the blur as an effect. This allows easier post processing / tracking / modification of the object.

To achieve this (and other) innovations, the Scene format is layer based. Conceptually the Scene format is divided into three different layers, a Base Layer, and a Directors' Layer.

4.1 The Base Layer

The Base Layer stores the most basic "physical" information of a scene. As in the real world, the base information is usually undisturbed, means not blurred, everything sharp, full colour space and very densely sampled (or rather continuous) in all dimensions. The Base Layer therefore contains captured data, which can be processed by algorithms already, but where no (meaningful) data shall be discarded. Blurring, clipping, and other effects which remove data irreversibly shall be avoided for this physical data as much as possible. Each captured element is a small part of a large scene, and atomic since either capturing device or algorithm have at some point in time decided that a certain amount of data belongs to the same piece of an object. These atomic scene elements are abbreviated "acels". Conceptually, the Base Layer contains the set of props, stage decor, etc. ("Ausstattung"), which can be used in a movie production.

The Base Layer comprises an arbitrary number of acels of a scene. All data contributing to the Base Layer needs to be located such that it can be non-ambiguously assigned to one scene. Acels in the Base Layer are addressed by unique identifiers. The naming convention is such that acels can be easily added and removed from the Base Layer.

In addition to the “physically” captured data the Base Layer provides functionality to store additional metadata. This metadata can be used to reproduce the processing steps of the acel information (provenance information) and recover the original data (as recorded by an acquisition device). In order to ensure lossless storage the original raw data remains as the basis for a further processing step in the scene information.

4.2 The Scene Layer

The information in the Base Layer might originate from many different sources which are calibrated in themselves but not in a global setting, cover different dimensions and be of very different types. The Scene Layer merges the Base Layer information to a coherent scene. Each acel needs to be defined for all dimensions occurring in a scene, and inter-acels coherencies need to be stored. The Scene Layer conceptually is the stage, where props, decor, etc. form a fully assembled scenario.

Each scene is uniquely identified (e.g. by an ID). All dimensions used in the scene which are required to be a superset of the acel dimensions need to be specified in the header of the scene. Acels are placed in a scene by giving the unique acel identifier and a specific position in all dimensions the scene is defined with. The Scene Layer can transform the whole acel, but not entries of an acel. All kinds of affine transformations (e.g. translation, rotation, sheering, expansion, reflection, …) on arbitrary acel dimensions are allowed.

Acel transitions which belong to the “physically” acquired data are stored in the Base Layer. However, explicit transition or transformation rules are described in the Scene Layer.

Acels can be coherent to other acels. In addition, acels can be likely to be coherent to other acels. Coherencies are managed per dimension and assigned for each pair of acels.

The Scene Layer allows the storage of additional metadata for each Scene element, if available semantic information can be provided for the objects contained in a scene either manually or automatically. In addition, developer’s information can be stored in the Scene Layer to facilitate postproduction.

4.3 The Directors’ Layer

Looking at a scene a viewer has almost unlimited freedom. Since all information is available in the best quality and assembled to a full scene, the viewer himself can take the role of a director and view whatever he wants. The Scene Layer prohibits this, by defining how a user shall interact with the scene. The simplest form of interaction is viewing through a camera. The Directors’ Layer defines a fully specified camera, its parameters, location in space and viewing direction. The Directors’ Layer also defines artistic degradation of data, e.g. by introducing blurs, reducing sharpness, cutting colour channels, etc. Beyond these camera specifications the Directors’ Layer may specify other ways of user interaction, e.g. allowing a certain freedom in the camera movement which the viewer may choose, different viewpoints, content manipulation or content interaction. So conceptually the Directors’ Layer creates a piece of art from a stage by defining what the viewer will see and how the user can interact with the content.

Cameras are defined by a set of parameters, which are set as explicit values. The set of parameters can be differentiated into intrinsic and extrinsic parameters. Cameras used to observe a scene do not become part of that scene, so another camera looking at the position of the first camera does not observe any object there.

By default, no user interaction is allowed. If the director wants to specifically allow user interaction, a rule needs to describe the allowed interaction. Rules may allow any changes to the Scene Layer, e.g. affine transforms on all dimensions of acels or groups of acels. User interaction cannot alter the acels themselves contained in the Base Layer. Therefore, a user may be permitted by rules to change the appearance of a scene, but he may not change the physical content of a scene. A rule specifies a scene, an acel, a dimension and gives the range of allowed interaction.

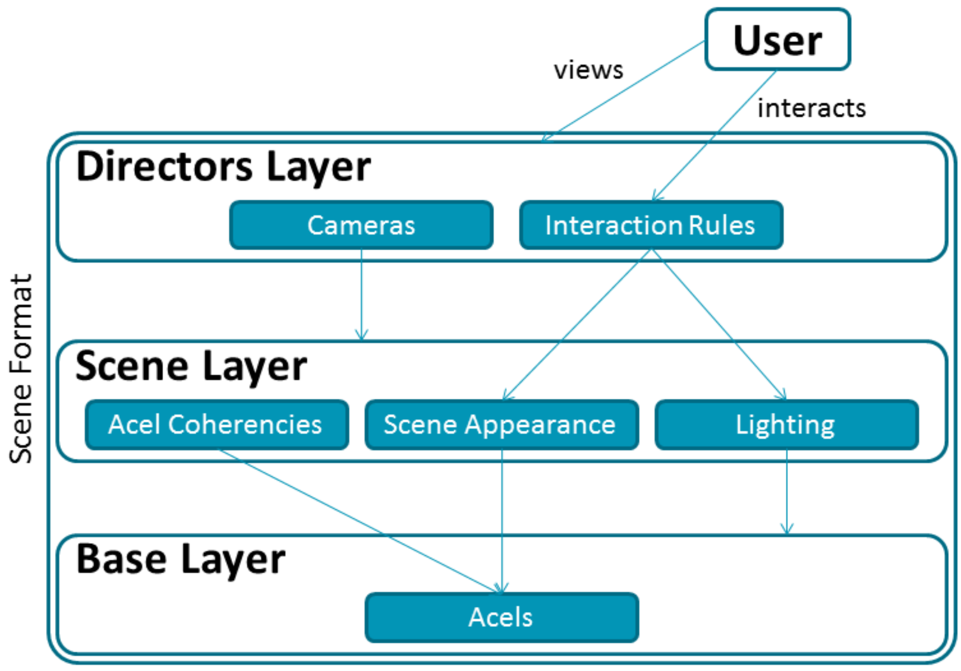


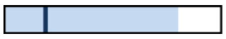
Figure 2: Conceptual Layer Based Architecture of Scene Data

4.4 *Feedback on Scene Concepts*

According to the DOW, feedback requests on the conceptual SRA description have been sent to experts in areas related to the SCENE project. 16 experts have replied and given detailed feedback on the concepts introduced in the D4.1.1. In the following the questionnaire average answers are marked in dark blue (■) and the range of answers is marked in light blue (□).

4.4.1 Historical Motivation

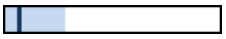
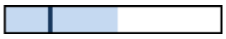
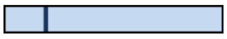
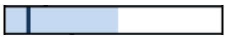
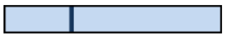
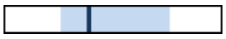
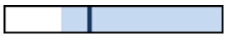
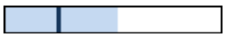
Throughout history the bottleneck of image or movie capturing devices has been the film; in recent times the image sensor. This bottleneck enforced constraints on the optical system. For low light conditions long exposure times or large lenses had to be chosen. The resulting artefacts like motion blur or limited depth of field have formed movies throughout the last century; they even became desired artistic elements and stylistic devices in movie productions. During the last years, new chip technologies have enhanced available image sensors to a level where this physical bottleneck is removed. The amount of light necessary to create an image does not usually dictate camera parameters any more. The Scene Representation Architecture is designed to represent data beyond this bottleneck.

The historical motivation is: clear  implausible

4.4.2 A common representation

When processing multidimensional video data on a computer, a multitude of information sources is required. Video from several sources, camera calibration data, lighting information and spatial knowledge just to name a few. Our proposed architecture unites all the information necessary for a movie scene production in a single representation.

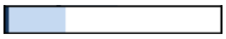
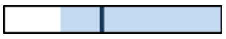
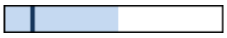
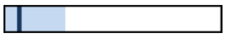
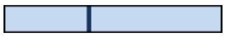
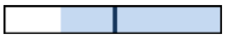
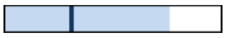
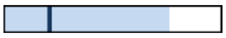
A common representation

1) facilitates post-processing: yes  no
2) enhances the users' media experience: yes  no
3) enables new artistic features: yes  no
4) is relevant for content producers: yes  no
5) Is relevant for content distributors: yes  no
6) Is relevant for content consumers: yes  no
7) Is interesting for your company: yes  no
8) Is interesting for you as a private person: yes  no

4.4.3 Storing undistorted data

When introducing artistic elements like motion blurs or depth of field these effects are traditionally modifying the captured data. Post processing such data is time consuming and difficult. The proposed Scene Representation stores all data in the best available quality, and introduces altering effects in a higher layer, thus preserving all available data for facilitating image and video processing steps.

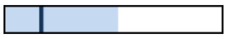
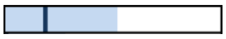
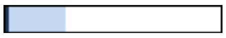
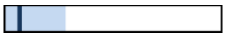
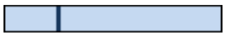
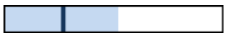
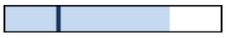
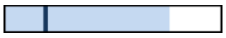
Being able to store undistorted data

1)	facilitates post-processing:	yes		no
2)	enhances the users' media experience:	yes		no
3)	enables new artistic features:	yes		no
4)	is relevant for content producers:	yes		no
5)	Is relevant for content distributors:	yes		no
6)	Is relevant for content consumers:	yes		no
7)	Is interesting for your company:	yes		no
8)	Is interesting for you as a private person:	yes		no

4.4.4 Content Interaction

Image or video content is usually frame-based. The Scene Representation is object based and therefore allows segmented content (segmentation is not restricted to 2D but can have all spatial and temporal dimensions). Knowledge about objects contained in a Scene allows the adjustment of scene objects such as updated product placement, object modification or user interaction.

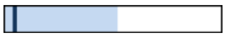
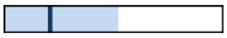
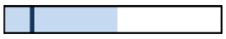
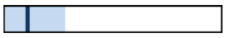
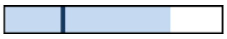
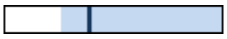
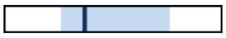
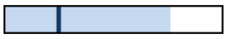
Enabling interaction with scene content

1)	facilitates post-processing:	yes		no
2)	enhances the users' media experience:	yes		no
3)	enables new artistic features:	yes		no
4)	is relevant for content producers:	yes		no
5)	Is relevant for content distributors:	yes		no
6)	Is relevant for content consumers:	yes		no
7)	Is interesting for your company:	yes		no
8)	Is interesting for you as a private person:	yes		no

4.4.5 Unified Representation

Computer generated content and captured video stem from two very different worlds and are processed largely independently in movie productions. The Scene Representation Architecture allows a unified representation of both, computer generated and captured video data, as well as intermediate processing steps, thus merging both worlds in an early stage and facilitating post production ("early fusion").

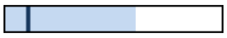
Having a unified representation for computer generated and captured video data

1) facilitates post-processing:	yes		no
2) enhances the users' media experience:	yes		no
3) enables new artistic features:	yes		no
4) is relevant for content producers:	yes		no
5) Is relevant for content distributors:	yes		no
6) Is relevant for content consumers:	yes		no
7) Is interesting for your company:	yes		no
8) Is interesting for you as a private person:	yes		no

4.4.6 Motivation for Layer Based Architecture

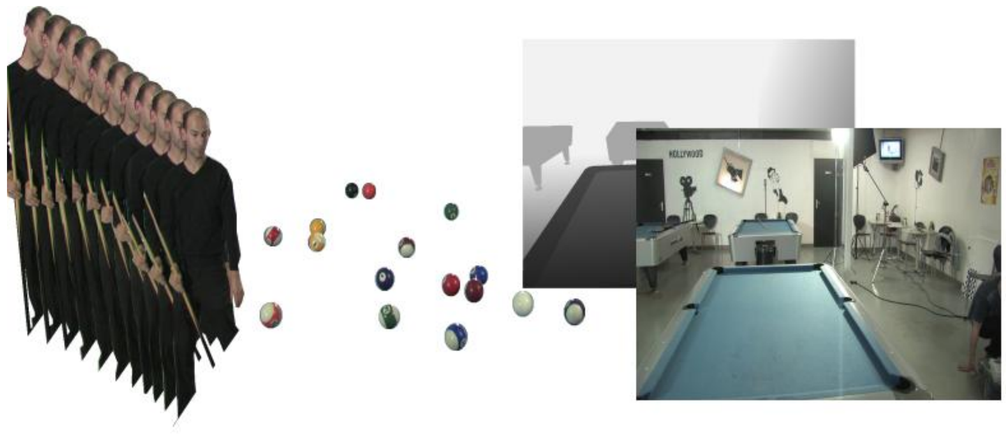
Considering the novelties mentioned above a representation is required, which a) can contain both, computer generated data and captured video data, avoiding a lossy transformation process in between b) can combine different kinds of information in the context of a scene c) can apply the traditionally required effects like blurs without affecting the underlying data Our approach is a layered based architecture, where three layers fulfil the requirements of a), b) and c). We call the layer for a) the Base Layer, the layer for b) the Scene Layer and the layer for c) the Directors' Layer.

The motivation for three layers as described above is:

clear  implausible

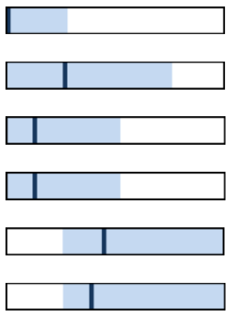
4.4.7 The Base Layer

The Base Layer stores the most basic "physical" information of a scene. As in the real world, the base information is conceptually sharp, of a full color space and quasi-continuous in all dimensions. Blurring, clipping, and other effects which remove data irreversibly are avoided for this physical data as much as possible. Conceptually, the Base Layer contains the set of props, stage decor, etc., which can be used in a movie production. Base Layer elements for the pool scene shown above are:



A Base Layer as described

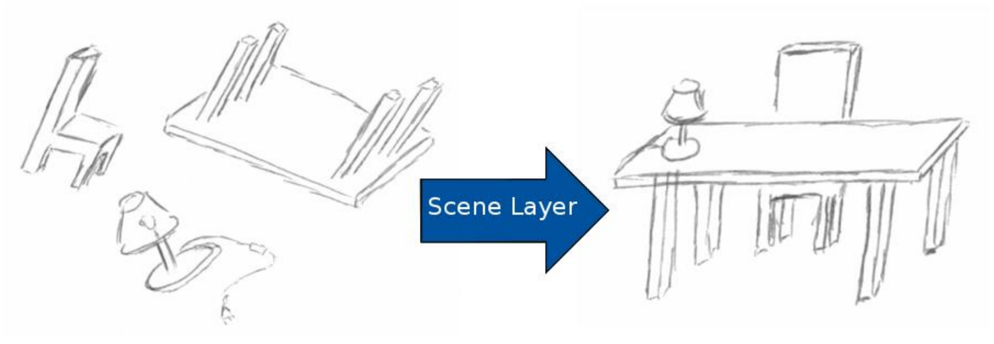
1) facilitates post-processing: yes no
2) enhances the users' media experience: yes no
3) enables new artistic features: yes no
4) is relevant for content producers: yes no
5) Is relevant for content distributors: yes no
6) Is relevant for content consumers: yes no



4.4.8 The Scene Layer

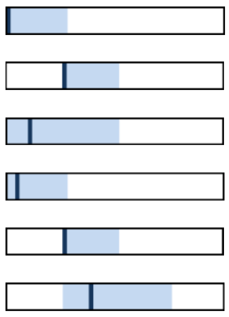
The information in the Base Layer might originate from many different sources which are calibrated in themselves but not in a global setting, cover different dimensions and be of very different types. The Scene Layer complements the Base Layer information to form a coherent scene. The Scene

Layer conceptually is the stage, where props, decor, etc. form a fully assembled scenario.



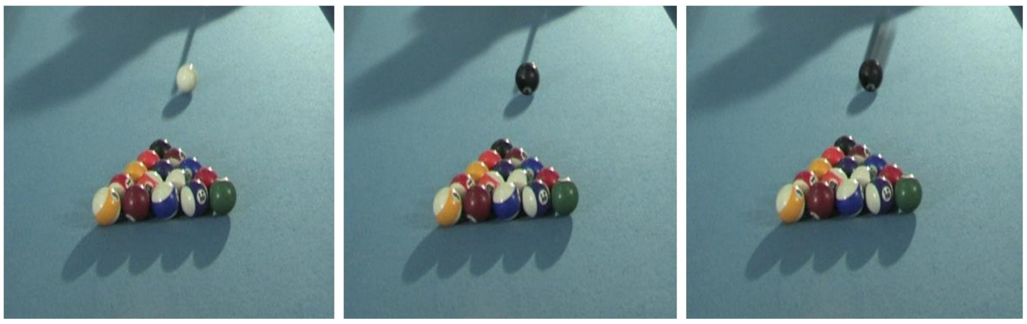
A Scene Layer as described

1) facilitates post-processing:	yes		no
2) enhances the users' media experience:	yes		no
3) enables new artistic features:	yes		no
4) is relevant for content producers:	yes		no
5) Is relevant for content distributors:	yes		no
6) Is relevant for content consumers:	yes		no

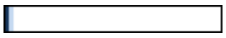
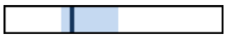
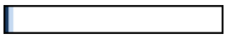
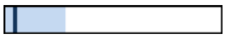
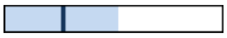
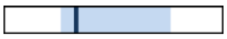


4.4.9 The Directors' Layer

The Directors' Layer includes all artistic elements and defines how a user can interact with the scene. The simplest form of interaction is viewing through a camera. The Directors' Layer specifies camera, its parameters, location in space and viewing direction. The Directors' Layer also defines artistic manipulation of data, e.g. by introducing blurs, reducing sharpness, cutting color channels, etc. Beyond these camera specifications the Directors' Layer may specify other ways of user interaction, e.g. allowing different viewpoints, content manipulation or content interaction. Conceptually the Directors' Layer creates a piece of art from a stage by defining what the viewer will see and how the user can interact with the content.



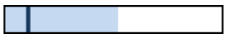
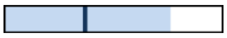
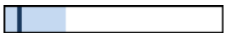
A Directors' Layer as described

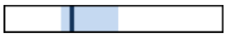
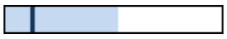
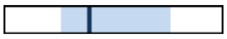
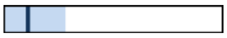
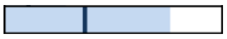
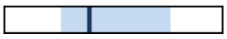
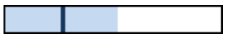
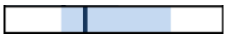
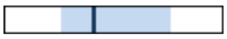
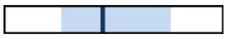
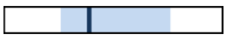
1) facilitates post-processing: yes  no
2) enhances the users' media experience: yes  no
3) enables new artistic features: yes  no
4) is relevant for content producers: yes  no
5) Is relevant for content distributors: yes  no
6) Is relevant for content consumers: yes  no

4.4.10 The example video in SRA

The content of the following video clip [here the first frame is shown] was generated according to the Scene Representation Architecture concepts described here.

Despite the artefacts in this video, do you think, the SRA as a whole

1) is promising: yes  no
2) is necessary: yes  no
3) facilitates post-processing: yes  no

4) enhances the users' media experience:	yes		no
5) enables new artistic features:	yes		no
6) will cost too much effort:	yes		no
7) provides added value to content producers:	yes		no
8) provides added value to content distributors:	yes		no
9) provides added value to content consumers:	yes		no
10) will be used by content producers:	yes		no
11) will be used by content distributors:	yes		no
12) will be used by content consumers:	yes		no
13) will be used by your company:	yes		no
14) will be used by you as a private person:	yes		no

5 Algorithmic Requirements to the SRA

Several algorithmic requirements have been identified for the Scene Representation architecture. These requirements are categorized into three groups: Requirements from application and user side, requirements from algorithmic point of view and requirements coming from the architecture itself. The requirements imposed by the architecture have been described in the previous section already. In the following a brief summary of the requirements for the Scene Representation Architecture from application and user side as well as from an algorithmic point of view are given. For a full description of the Algorithmic Requirements please refer to D4.3.1.

5.1 Requirements from application and user side

One use case addresses the basic functionality for object insertion and interactivity. To realize them there are certain requirements for the Scene format to provide the necessary information or data. To achieve a high level of flexibility and adaptability, the object insertion should be done at a rather late stage of the production and play-out pipeline, e.g. during the play-out in a broadcast center. To achieve this idea the format needs to allow for a late decision concerning the object insertion, i.e. a late selection of the object to be rendered with the scene. Thus, the Scene format shall support the storage of all required information and data to allow for this late insertion.

Virtual studios provide an application use case focused on real time rendering. Virtual studios will use on one hand direct output from capturing devices, and on the other hand processed information included in scene files that will be ready or almost ready for real time rendering.

Cinema production and post-production require a combination of elements generated using very diverse acquisition and modelling methods. Merging these different inputs without compromising on realism or introducing discomfort to the viewer imposes a number of requirements in terms of being able to model the geometry of elements, appearance and lighting or general scene framing.

5.2 Requirements from algorithmic point of view

Polygon meshes are one of the most popular representations used in computer graphics to describe the shape and appearance of objects. As they have become very widely used in the digital media industry and more generally in all fields requiring representation of shape and appearance the Scene Representation Architecture is required to allow storage of polygon meshes.

With regard to data representation, classical computer vision algorithms are based on images. Scene structure properties, such as disparities or depth are therefore usually defined or specified on pixel-level and thus describe pixel-wise relationships. This holds also for extended properties, such as confidence or transparency information. These properties are often stored in maps, which have the same spatial dimensions as the corresponding image. Each entry of a map represents the property of the corresponding pixel in the image. Based on this, one very important requirement for the new scene format is the capability to store these data without loss of quality. More generally, data arrays of pixel properties, such as pixel colour, depth, confidence, transparency etc. need to be allowed in the Scene Representation Architecture.

Algorithms will require camera descriptions in order to be able to relate images to the 3D scene that they represent. The mapping between 3D scene points and 2D image points is represented in the form of a camera model and a set of calibration parameters. The most commonly used representation is the pinhole camera model which models an ideal lens with infinitely small aperture; this is often combined with a distortion function modelling lens aberrations.

To enable different algorithms to properly interface with acels representation it is imperative that acels can be linked to each other. Furthermore, in order to keep the effect of these algorithms consistent with the overall representation, coherence of the connectivity needs to be defined and maintained in the Scene Representation. Here, coherency tables, spatiotemporal connectivity, mesh surface deformations or material connectivity are requirements for the algorithms which need to be represented in the Representation Architecture.

Animation allows the director to alter the geometric appearance of a deformable object under certain constraints. Usually this is done by providing a

3D model of the object that has rigid and flexible parts which can be set to a specific configuration or sequences of configurations using a suitable user interface. Animation data generated in such content provides valuable input to scenes and should therefore be contained in the Scene Representation Architecture as well.

In order to photo-realistically merge objects in a scene extraction of material properties and appearance and the provision of this knowledge in the Scene is fundamental. The scene representation therefore needs to allow storage and retrieval of object properties and appearance.

The following figure summarizes the algorithmic requirements for the Scene Representation Architecture. For a full description and definition of these requirements please refer to Deliverable D4.3.1.

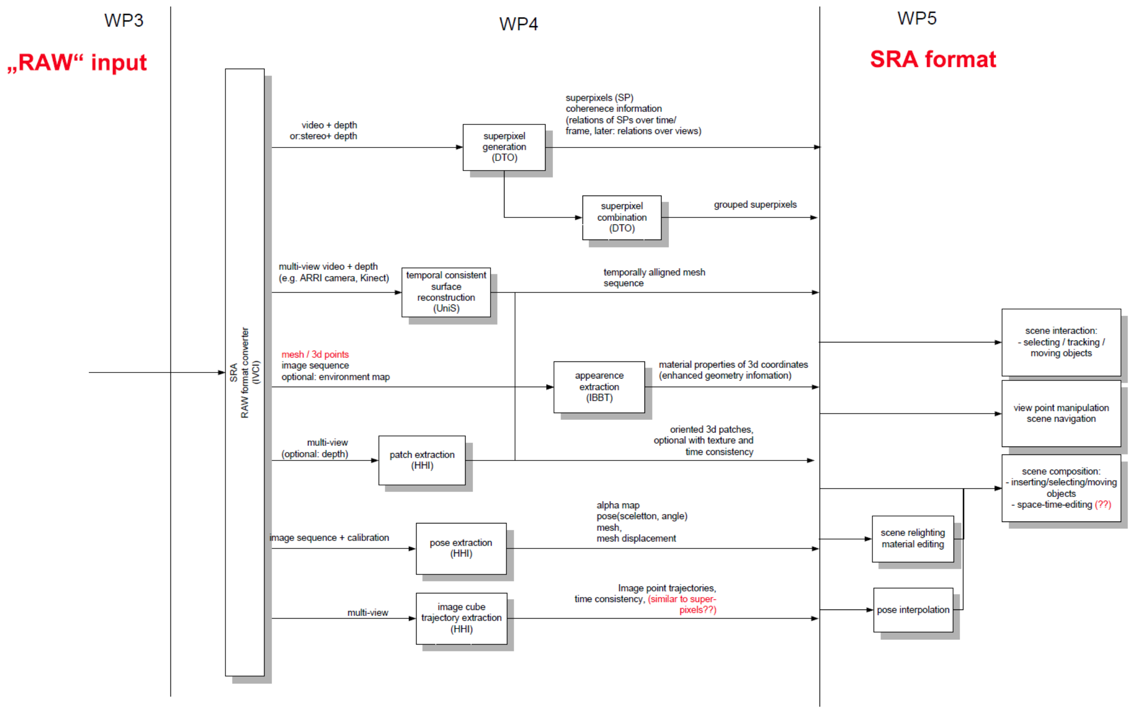

WP3
„RAW“ input
WP4
WP5
SRA format
SRA
RAW format converter
(IVCI)
video + depth
or:stereo+ depth
superpixel generation (DTO)
superpixels (SP)
coherenece information
(relations of SPs over time/
frame, later: relations over views)
superpixel combination (DTO)
grouped superpixels
multi-view video + depth
(e.g. ARRI camera, Kinect)
temporal consistent surface reconstruction (UniS)
temporally alligned mesh sequence
mesh / 3d points
image sequence
optional: environment map
appearence extraction (IBBT)
material properties of 3d coordinates
(enhanced geometry infomation)
multi-view
(optional: depth)
patch extraction (HHI)
oriented 3d patches,
optional with texture and
time consistency
image sequence + calibration
pose extraction (HHI)
alpha map
pose(sceletton, angle)
mesh,
mesh displacement
multi-view
image cube trajectory extraction (HHI)
Image point trajectories,
time consistency, (similar to super-pixels??)
scene interaction:
- selecting / tracking / moving objects
view point manipulation
scene navigation
scene composition:
- inserting/selecting/moving objects
- space-time-editing (??)
scene relighting
material editing
pose interpolation

Figure 3: Algorithmic requirements to the Scene Representation

6 Technical Specification of SRA API

6.1 Motivation

In order to meet the multiple requirements identified in the preceding sections and in the previous documents delivered in the SCENE project (especially D4.1.1 and D4.3.1) a flexible and extendable way of storing data needed to be developed. Especially with multiple tools and algorithms in the project scope still under development this goal of creating an extendable option to store scene data is of huge importance. During the development it was noticed that no fixed file format can be specified that can fulfil requirements which are not yet fully known and still advancing. The implementation of an API solves this dilemma. By designing an API access to scene content can already be specified, allowing applications and algorithms to make use of the scene data. At the same time the underlying data structure can still be enhanced and modified. Even the API is easily extendable by functions exceeding the currently known requirements. A few of the numerous benefits of an API are

a) Extendibility: in the future further functions and necessities can be easily added. If a certain way of accessing Scene data is needed, only a function needs to be added to the API. The underlying file structure is not affected by such enhancements.
b) Flexibility: requirements to the data structure are still open. Questions which will heavily affect the format like compression or streaming are not really in the scope of the SCENE project and will be dealt with in future research. An API allows to exchange the underlying file structure easily without affecting the tools and algorithms which already employ scene data.
c) Creativity: The Scene API is designed to facilitate module contributions and therefore enhance the creativity of its users. Adding new computational modules for further algorithmic processing of scene content or adding new tools with currently unknown requirements is an

easy thing to do. An API therefore boosts the creativity of developers and producers at the same time.

The following figure sketches the SCENE API and presents where further creative input can be implemented.

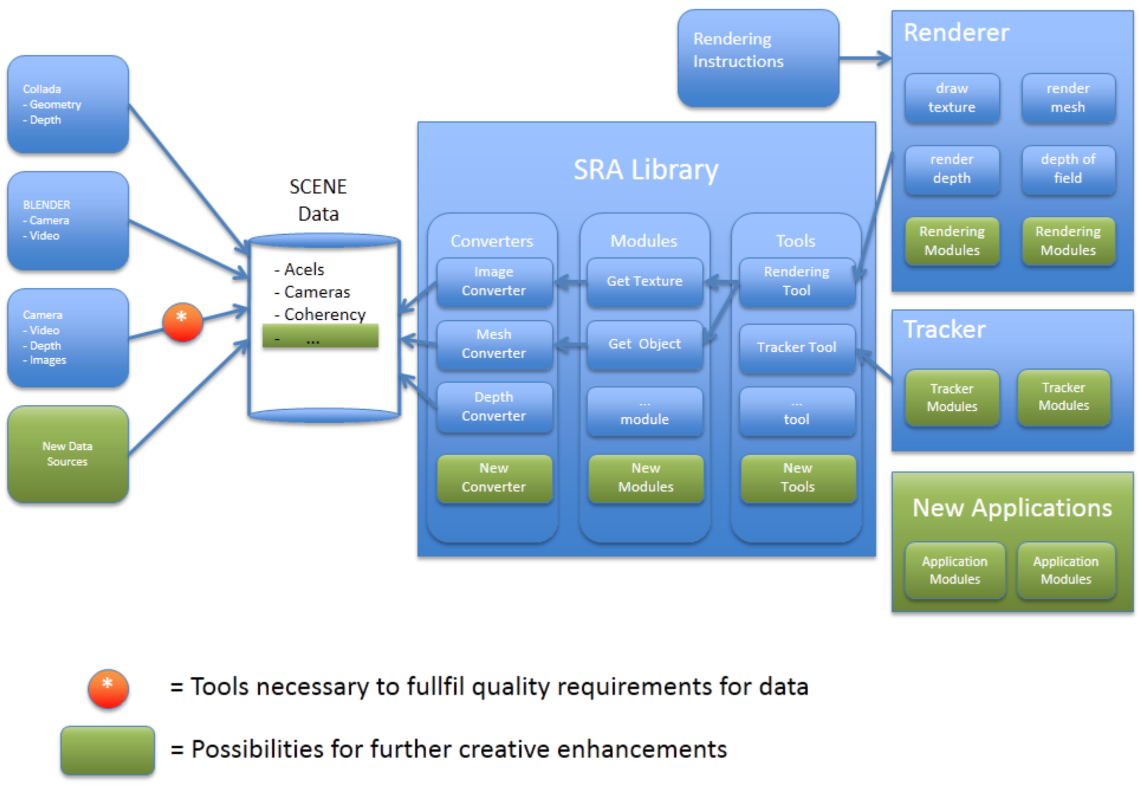


Figure 4: Technical Outline of the SRA Components

6.2 *Basic Access to SCENE Data*

The scene interface defines access to the scene data. The following data classes can be accessed:

1) Scenes
2) Streams
3) Acels
4) Acel Values
5) Acel Lists

6.2.1 Scenes

A scene is a container of all data available in the context of a scene as conceptually defined in section 4 and needed by the technical requirements in section 5. The scene class exposes several functions. These are

- GetGlobals: returns all the acels defined in one scene
- GetStream: return the currently selected stream
- GetStreamCount: retrieve the number of available streams

```
class IScene
{
public:
  /// returns acels defined in for all frames
  virtual PAcel GetGlobals() = 0;

  // get number of available streams
  virtual size_t GetStreamCount() = 0;
  // get new stream with index i
  virtual PStream GetStream(size_t index) = 0;
  virtual ~IScene() {}
};
```

6.2.2 Streams

A stream is a sequence of frames. To use streams two functions are defined

- AdvanceToNextFrame: gets the next stream frame, and returns “false” if end of stream is reached
- GetCurrentFrame: retrieves the acels representing the currently loaded frame

```
class IStream
{
public:
  ///  load next generic stream frame, return false if end of stream,
  /// affects the frame that is returned by GetCurrentFrame
  virtual bool AdvanceToNextFrame()= 0;
  /// get acel representing currently loaded frame
  virtual PAcel GetCurrentFrame()= 0;
  virtual ~IStream() {}
};
```

6.2.3 Acel

An acel is an atomic scene element, and can contain any kind of data. The conceptual and technical description of an acel is given in the previous sections and in previous deliverables. The following functions are exposed for acels:

- ID: is the unique identifier of an acel
- List: a list containing the internal structure of an acel in case such a structure is available
- Property: returns the acel part of the list identified by a property name. If such a property does not exist in the internal acel structure an UnsupportedProperty-Error is thrown
- Value: the value of the acel, that can be converted into a primitive data type (mesh, image, …) depending on the available information and the use

```
class IAcel
{
public:
  // the unique id, equal for the acels with identical content
  virtual oid_t Id()=0;
  // list of child acels
  // the list is owned by the acel
  virtual IAcelList* List() = 0;


  class UnsupportedProperty { };
  //  named child acels, if acel with this name doesn't exist
UnsupportedProperty is thrown
  virtual PAcel Property(string name) = 0;

  // value object, that can be converted to primitive data types
  // object is owned by the acel
  virtual IAcelValue* Value() = 0;
  virtual ~IAcel() {}
};
```

6.2.4 Acel List

An acel list contains multiple acels. This is e.g. used to for multiple acels occurring in a frame, or any other context where many acels are requested. Acel Lists have two can be accessed by

- Count: returns the number of acels in a list
- Get: returns an acel out of the list by ID

```
class IAcelList
{
public:
  /// number of acels in the list
  virtual size_t Count() const = 0;
  /// get acel from the list by index
  virtual PAcel Get(size_t index) = 0;
  virtual ~IAcelList() {}
};
```

6.2.5 Acel Value

The acel value contains the actual acel data. This can be any kind of data, which can be represented by a requested primitive data type. Conversion modules are responsible for the data conversion into a primitive type. If the conversion is possible the data is returned in this primitive data format. If the conversion is not possible, an unsupported type error is returned. The following functions are exposed:

- GetFloat: returns acel data in floating point precision
- GetInt: returns acel data in integer precision
- GetString: returns string representation of acel value
- GetBool: returns Boolean representation of acel value
- GetFloat3: return data as 3D vectors with floating point precision
- GetInt3: return data as 3D vector with integer precision
- GetByteBuffer: return data with arbitrary structure, e.g. as n-dimensional byte vector
- GetFloatBuffer: return data with arbitrary structure, e.g. as n-dimensional vector with floating point precision
- GetIntBuffer: return data with arbitrary structure, e.g. as n-dimensional vector with integer precision

```
class IAcelValue
{
public:
  struct UnsupportedTypeError : public std::exception{};
  // scalars
  virtual float GetFloat() = 0;
  virtual int32_t GetInt() = 0;
  virtual string GetString() = 0;
  virtual bool GetBool() = 0;
```

```
    virtual float3_t GetFloat3() = 0;

    // arrays of common basic types
    virtual byte_buffer_t GetByteBuffer()= 0;
    virtual float_buffer_t GetFloatBuffer()= 0;
    virtual int_buffer_t GetIntBuffer()= 0;
    virtual ~IAcelValue() {}
  };
```

6.3 Scene Tools

The computational modules are ordered into three classes, which are

1) Tools
2) Modules
3) Converters

These three different computational tools are described in the following.

6.3.1 Tools

Tools are providing interfaces for some dedicated functionality. For example, the scene renderer is such a tool, an object tracker could be another one. The Tool-Class exposes a function to create a computation module. If such a module can not be created an unsupported Module Error is thrown. Exposed functionality:

- **CreateModuleXY:** tries to create a computational module for a specific means. For example, the function CreateModuleDepthToMesh can be used to create a module capable of calculating depth information to a mesh, or the module CreateModuleMeshRendering creates a module for returning renderable mesh information. XY here is a placeholder for the specific task a module is created for, i.e. DepthToMesh or MeshRendering in the examples above.

6.3.2 Module

The Module class provides the interface to the computational modules. These modules can execute computations on acels. The following functions are exposed to use module functionality:

- **GetInputXY:** in order to retrieve a specific input this function can be used, where XY specifies the input type, like GetInputTexture or GetInputMesh
- **Execute:** executes the module. The result of this execution is an acel which can be returned to the acel storage or be further processed. In case of missing module information an InputNotSetError is thrown, indicating the missing information.

```
class IModule
{
public:
  // exception raised if one or more inputs are not set
  struct InputNotSetError : public std::exception
  {
    string name;
    InputNotSetError (string name) : name(name) {}
  };

  // execute the module.
  virtual PAcel Execute()  = 0;
  virtual ~IModule() {}
};
```

6.3.3 Converter

Converters are used to convert the representation of an acel into some other representation that is required as module input and load the data into the input of that module. In order to make use of the converters the following module is exposed:

- **Assign:** loads the data from the acel (from disk or input stream) into the associated module input. If the acel cannot be converted to the data structure required by the module input, the exception UnsupportedTypeError is thrown. When data is loaded to the module input its intermediate representation specific to the input type is created. This intermediate representation is used every time this acel is assigned to any input of the same type unless the acel object is destroyed. The return value is false if this intermediate representation was used and true if the object was converted from an acel.

```
  class IConverter
  {
  public:
    /// loads the data from acel into the associated module input
    virtual PConverterCache Assign(PAcel value) = 0;
    virtual ~IConverter() {}
  };
  /// keeps track of module-specific cache objects created from acels
  class IConverterCache
  {
  public:
    // destroy the cache
    virtual void Release();
    virtual ~IConverterCache(){}
};
```

6.4 Access for Applications

Applications access the SCENE API through dedicated interfaces which provide the functionality that the application requires. Exemplarily the renderer interface is specified here.

6.4.1 Renderer Interface

The renderer interface provides the entry point for the renderer implementation and provides a collection of modules for acel processing required by the renderer.

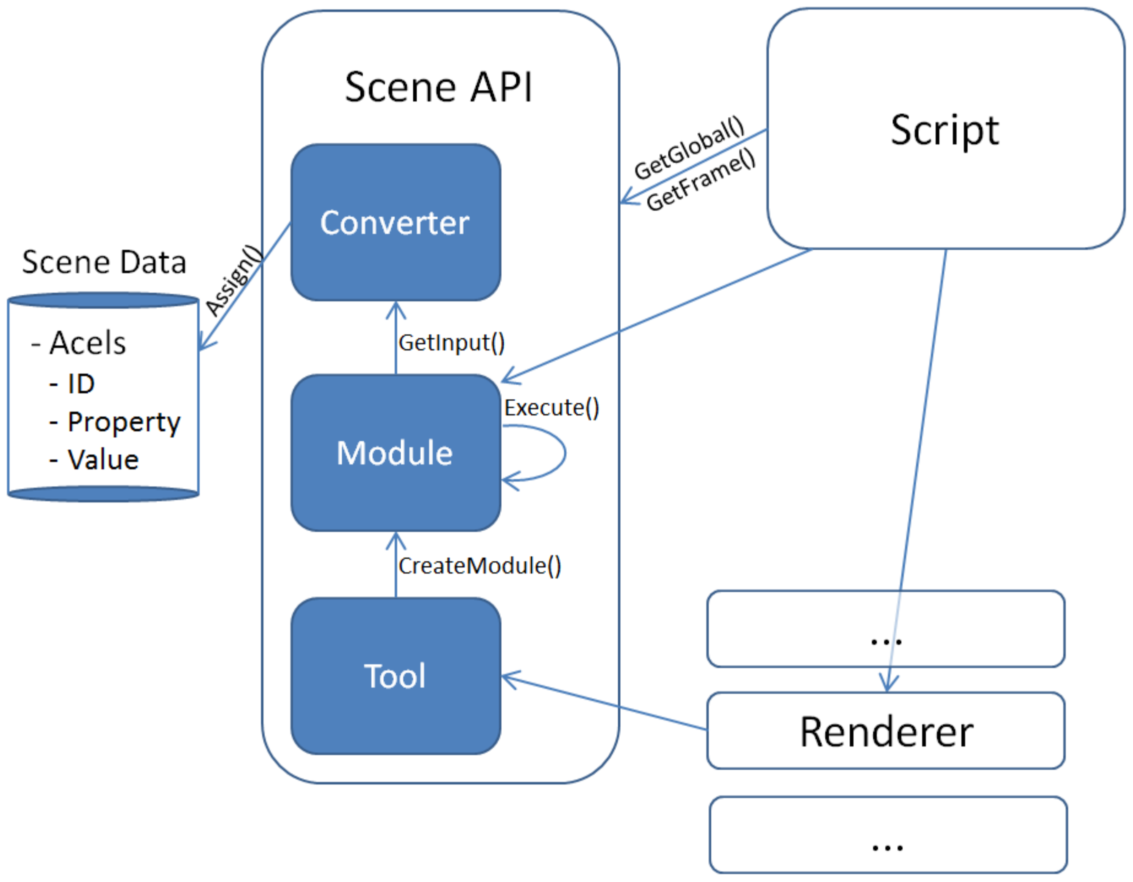


Figure 5: Process Visualization for inter-component communication

7 Examples

7.1 Video Rendering

Exemplarily here is a description of how video rendering with the SCENE API works. For illustration purpose Python-Code snippets are added to the individual steps. The following steps are executed:

1. A script contacts the Scene API to retrieve information on scene content. This request returns some global information of the scene, like what kind of information can be queried from the scene and how that information can be used.

```
filename = "../sample_data/collada.scene"
demo.scene = sra_renderer.get_scene_representation(filename)
```

2. The script choses an available video stream from the scene data.

```
self.stream = self.scene.GetStream(0)
```

3. The script contacts the Renderer that content of the scene file shall be rendered.

```
demo.renderer = sra_renderer.get_proto_renderer(self.scene, '')
```

4. The renderer addresses the SCENE API to create a renderer module. A new module is then created dedicated to rendering SCENE content.

```
demo.mod_render_mesh = self.renderer.CreateModRenderMesh()
demo.mod_camera = self.renderer.CreateModViewPoint()
demo.mod_render_depth = self.renderer.CreateModRenderDepthMap()
```

5. When the modules are created an input port for scene data is returned to the script.

6. The script then accesses the SRA file through the SCENE API and requests SCENE data, this time specific on a frame basis.

```
demo.renderer.BeginFrame()
demo.stream.AdvanceToNextFrame()
```

7. The SCENE API returns the requested frame content of the SRA to the script.

```
stream_frame = demo.stream.GetCurrentFrame()
```

8. The script assigns the frame to the input of the rendering module.

```
demo.mod_render_depth.DepthMap.Assign(stream_frame)
demo.mod_render_depth.Texture.Assign(stream_frame)
```

9. The script asks the rendering modules to execute. The SCENE content delivered to the rendering module is then rendered.

```
demo.mod_render_depth.Execute()
```

8 File Format

For storage of the scene data basically any kind of hierarchically structured data can be used. A set of standard scene elements is proposed as the core of the file format. These elements can be flexibly extended according to the requirements of other scene tools that use the SCENE data. Exemplarily for these tool specific extensions we present structured data required for the renderer here.

Currently a set of converter scripts can convert the traditional inputs to be accessed and used by the scene API. With the help of these scripts even large amounts of data can be automatically converted to the scene format. One of the future efforts will be to allow storing data in the SRA from the SRA API as well, thus allowing tools and algorithms to work with the API only and store processed data in the API again. The set of converters necessary for the integration of different data types can be easily extended, maintained and updated as the requirements to data inputs and outputs evolve.

The entry point for general scene data is called “Global”. This global set of information contains acels of any kind as well as camera information. Further information of global scope can be easily added to the currently defined set. For specific tools this information is extended. Here, the “Streams”-Structure adds information necessary for rendering scene content as a video stream. The following figure represents the structure of data.

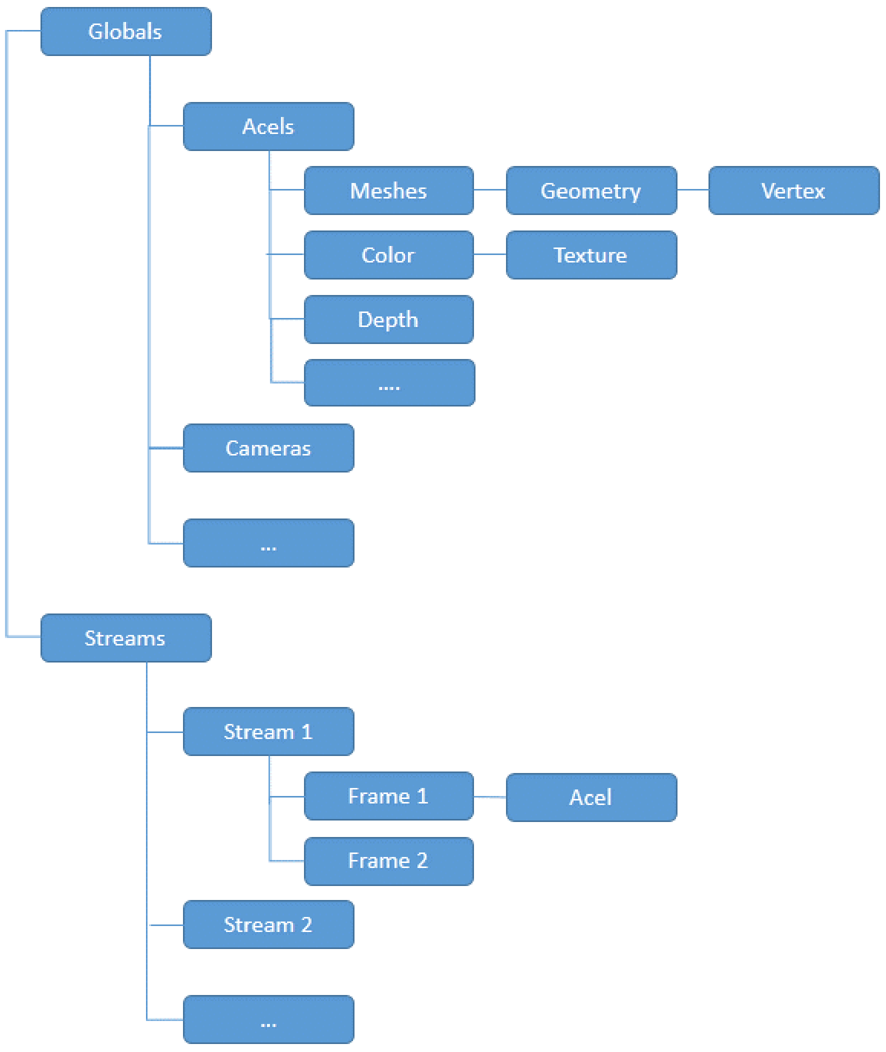


Figure 6: Components of structured data

Access functions to the data need to be designed accompanying the data storage.

While any kind of hierarchical storage can be used as soon as a storage and the corresponding access functions are implemented, the current Scene API is implemented using Protocol Buffers. Protocol Buffers were developed by Google as their internal exchange format for all kinds of data. According to the documentation website “Protocol Buffers are a way of encoding structured data in an efficient yet extensible format.” [1]

While file format and access is on purpose not specified here (see motivation for more flexibility) the following example illustrates how information is currently stored and accessed at the example of an acel.

The following function is used to access acel information from the file. If the decision is made to move acels from Protocol Buffer storage to any other way of storing acel information, only the following header file and the corresponding implementation need to be adjusted. The following two source code pieces exemplarily show the access of acels stored in the Protocol Buffer.

Header File:

```
class XAcel: public IAcel
{
  const pb::Acel& acel;
  std::vector<PAcel> list;
  XAcelList xlist;
  shared_ptr<IAcelValue> PtrValue;
  XAcel(const pb::Acel& a);
public:
  static IAcel* CreateAcel(const pb::Acel& a);
  virtual oid_t Id();
  virtual IAcelList* List();
  virtual PAcel Property( string name );

  virtual IAcelValue* Value();
  virtual ~XAcel() { }

};
```

Class File:

```
sra::oid_t XAcel::Id()
{
  return (sra::oid_t)this;
}

IAcelList* XAcel::List()
{
  if (list.size() == 0)
  {
    for (
      auto p = acel.children().begin();
      p!= acel.children().end(); p++ )
    {
      list.push_back(PAcel(CreateAcel(*p)));
    }

    xlist = XAcelList(&list);
  }
  return &xlist;
}

PAcel XAcel::Property( string name )
{
  for (
    auto p = acel.properties().begin();
    p!= acel.properties().end(); p++ )
    if (p->name() == name)
    {
      return PAcel(CreateAcel(p->value()));
    }
  throw UnsupportedProperty();
}

IAcelValue* XAcel::Value()
{
  if (acel.has_value())
  {
    if (!PtrValue)
      PtrValue = PAcelValue(new XAcelValue(acel.value()));
    return PtrValue.get();
  }
  else
    return &ErrorAcelValue::Instance;
}

sra::IAcel* XAcel::CreateAcel( const pb::Acel& a )
{
  if (a.has_ren_camera())
      new BAcel(&a.ren_camera());
  else if (a.has_ren_depth())
      new BAcel(&a.ren_depth());
  else if (a.has_ren_geometry())
    new BAcel(&a.ren_geometry());
  else if (a.has_ren_rgb())
    new BAcel(&a.ren_rgb());
  else
    return new XAcel(a);
}
```

9 Conclusion

The described Scene API represents the best solution to provide applications and algorithms with the means to access a new kind of data and maintain the flexibility for further developments at the same time. The API can easily be extended to all sides. By adding file readers for additional file formats other formats and data types can easily be integrated into the Scene API. Computational modules and converters can be linked into the Scene API enabling new intermediate processing steps and further required output formats. And finally the functional interface of the API can be extended to meet the requirements of future tool developments.

A basic API implementation provides the basis for further developments. The Scene prototype renderer uses the Scene API for scene data rendering and serves as an example for other tools.

Overall, the API specification presents an entry point for further development and a crucial step to integrate the multiple algorithms and tools that are developed in the context of SCENE.

10 Summary

The technical specification of the Scene Representation Architecture is an important step towards the integration of all SCENE related algorithms and tools into a unified project. It contains the technical realization of the conceptual description of the Scene Representation Architecture contained in document D4.1.1. Furthermore, it enables the requirements for the Scene Representation Architecture identified in document D4.3.1.

The document begins with a recap of the conceptual description and the results of a feedback request sent to multiple experts to receive feedback on this conceptual description. This conceptual evaluation is succeeded by a section briefly listing the main requirements from an algorithmic and use case perspective. The main part of this document technically describes the implementation of a SCENE API enabling already defined requirements and allowing future project development and enhancements at the same time.

11 Bibliography

[1] http://code.google.com/p/protobuf/