D5.1.1 Demonstrator for basic patch rendering

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 04 2013
Actual date of delivery	20 05 2013
Leader of this deliverable	BS
Reply to	jmontesa@brainstorm.es
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_D5.1.1_30042013
Security (distribution level)	PU
Contractual date of delivery	Month 18, 30.04.2013
Actual date of delivery	Month 19, 20.05.2013
Deliverable number	D5.1.1
Deliverable name	Demonstrator for basic patch rendering
Type	Report and Demonstrator
Status & version	Final
Number of pages	24
WP / Task responsible	IVCI
Author(s)	Javier Montesa (BS), Victor Matvienko (IVCI), Christopher Haccius (IVCI), Ingo Feldmann (HHI), Sammy Rogmans (iMINDS)
Other contributors	BS, HHI, iMINDS
Project Officer	Philippe.GELIN@ec.europa.eu
Abstract	On the basis of the designed SRA architecture a new file format has been defined which will allow files containing three-dimensional models. A parsing library and some example files have also been created by IVCI that will allow a common framework when loading SRA models. This deliverable describes two software modules that IVCI and BS have developed based on these tools: • IVCI has developed an acel renderer that is able to render acel data coming from SRA files • BS has prepared its commercial engine, eStudio, in order to import SRA files content.
Keywords	SRA LIBRARY PATCH RENDERER eSTUDIO PLUGIN
Sent to peer reviewer	DTO
Peer review completed	YES

Circulated to partners	BS, HHI, iMINDS, IVCI, DTO
Read by partners	BS, HHI, iMINDS, IVCI, DTO
Mgt. Board approval	Pending

Table of contents

1 Public Executive Summary

This deliverable is related to WP5T1 “Scene Rendering” by describing the developments made by IVCI regarding acel rendering, and those made by BS regarding the preparation of eStudio in order to import SRA content.

Scene data is stored in the Scene Representation, which can be accessed using the Scene API. This API is defined in deliverable D4.1.2. A first implementation provides data for the Scene renderer, which can render acel data coming from the API into a video stream. In the current prototype implementation acels can be basic scene data in texture plus depth or mesh format. Similar to the Scene API the renderer follows a modular approach which provides basic acel rendering capabilities and a few effects from computational videography, but allows easy integration of further rendering modules. Currently the renderer can render texture plus mesh or texture plus depth data to a video stream. Simple effects like shadowing are implemented as well. A well-defined renderer interface allows the integration of further rendering parts. The renderer is available as a simple Windows installer and can be used through Python scripts.

On the eStudio side, in order to facilitate the integration of the SRA loading plugin, several modifications have been done on this professional tool which will allow creating new plugins in order to import new file formats directly into the graphics engine architecture. This document explains these modifications and how they will allow the development of the SRA file loader as a particular case of model import plug-in once the SRA API is ready, based on the specification that just became available in D4.1.2.

2 Introduction

Different capturing methods are currently being developed in the project, which will provide their results based on the Scene Representation Architecture. The works described in this deliverable aim at displaying these contents, integrated with other synthetic and real elements, on the two graphics engines involved in the project. BS's eStudio is oriented to real time virtual scenography while IVCI's graphics engine is oriented to render high quality cinematic productions using all the new possibilities enabled in the Scene project.

In the Scene project novel scene representations with additional data and new data structures are made available in the Scene Representation. The Scene renderer is one fundamental tool to visualize this data. The renderer prototype which is described in this document presents the first layout of the Scene renderer and basic functionality. Acel data can be rendered with simple effects and shown as a video stream.

The renderer is designed in a way to allow easy integration of future rendering modules which employ developments in computational videography to provide realistic rendering of the scene content.

Regarding eStudio, it allows for loading the most common model file formats, although it is a fairly open graphics engine, the most common way to import proprietary or new model formats is to convert them into the tool internal representation format. In the Scene project, a new and open loading system has been designed that will allow for fast and easier implementation of new file format loaders.

This document will explain how the engine has been modified in order to make it easier to develop these new file format model loaders, how the SRA loader will be implemented and the expected final result once it is finished.

3 Structure of the document

Section 4 describes the acel renderer developments and their results. Section 5 describes the developments related to the SRA loader for eStudio, the expected results of the integration of SRA models in virtual studios with a capture of the talent, and the resulting graphical user interface.

4 Acel rendering

This section describes a renderer for basic acel data. Acels are the primitive scene elements, which are defined in Document D4.1.1. Document D4.1.2 defines how these acels can be accessed by tools, for example a rendering tool. The renderer makes use of the Scene API to access stored scene data and renders the contained data.

4.1 Data Elements for Scene Rendering

Scene Deliverable D4.1.2 introduces the data elements contained in the Scene Representation. The basic content can be easily extended by data requirements of certain applications. One of these applications with additional requirements is the Scene renderer. For rendering a texture video structure is added which can contain either a video stream or a number of images. In addition, depth data can be provided in form of depth maps. Multiple objects specified by their geometry data for rendering can be included, and a camera to view the scene is provided. The structure is outlined in the following figure.

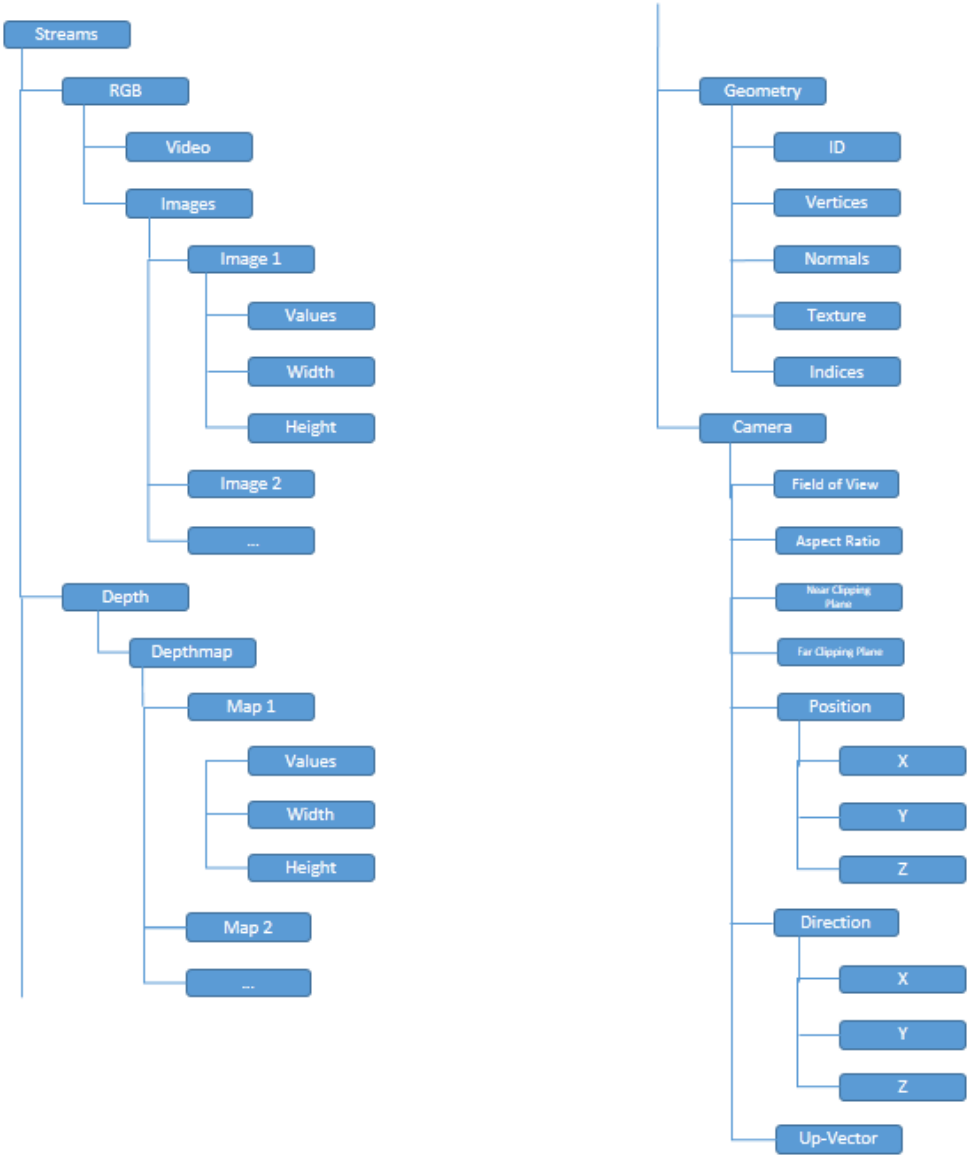

Figure 1. Data structure requirements for renderer.

4.1.1 RGB Information

RGB information contains the colour or texture information of a scene. This information is contained in either a colour video or in individual image files. In case of image files the pixel values are given for each image as well as the horizontal and vertical dimensions. An arbitrary number of images can be present in a texture acel making up a stream of colour information.

4.1.2 Depth Information

Depth information is structured like colour image information, if it is stored as a depth map. In this case an arbitrary number of depth maps can be stored, containing the depth values as well as vertical and horizontal dimensions of the map.

4.1.3 Geometry Information

Geometry information is uniquely identified by an ID and contains vertices, normal, texture coordinates and indices. Together with camera information this information can be used to generate the depth for a certain perspective.

4.1.4 Camera Information

For rendering scene content the camera is essential. Required camera information includes the horizontal or vertical field of view and optionally the aspect ratio between horizontal and vertical field of view. The viewing distance is limited by the distance to the near clipping plane and far clipping plane. The camera has a camera position which is given in 3D spatial coordinates, and a viewing direction also given as a 3D vector. Furthermore, the camera has an up-vector fixing the rotation of the camera in 3D.

4.2 Rendering Modules

The renderer exposes public rendering modules such that the renderer can be used by external applications. Currently three rendering modules are implemented, which are

- A Depth Map Rendering Module: a module to render scene information with spatial information from a depth map
- A Mesh Rendering Module: a module to render scene information with spatial information from a mesh
- A Camera Module: a module that manages camera information for rendering

Modules provide different public functions for usage. In the following the rendering modules are explained in detail.

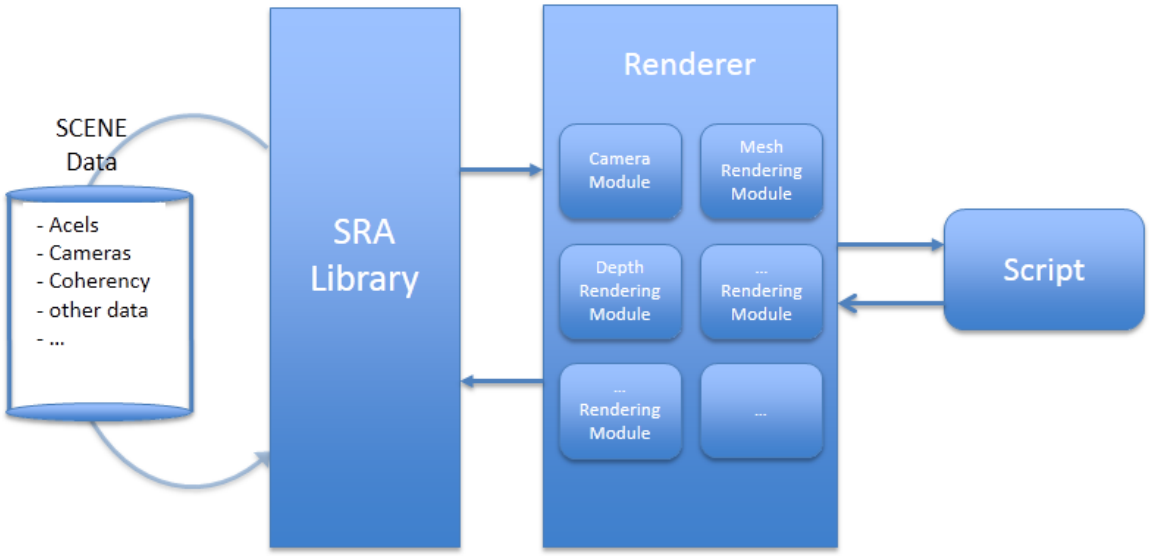

Figure 2. High-level structure of renderer in Scene.

4.2.1 Depth Map Rendering Module

The depth map rendering module performs the following functions:

- Transform: retrieves spatial transform information for depth map rendering using a converter, that provides arbitrary acel data in form of a depth map
- Texture: retrieves texture information for depth map rendering using a converter
- DepthMap: retrieves depth information for depth map rendering using a converter
- CameraSettings: retrieves the camera parameters for the depth information using a converter
- Execute: executes the module for a given acel input

These functions provided by the depth map rendering module can be used to render a scene object of which the geometry information can be made available as a depth map. For rendering this geometry data the spatial transform information is required, which describes how the geometry information is spatially located and transformed. In addition to the transformed depth map information texture information can be applied to the geometry data. This texture is retrieved using the Scene API and integrated into the rendering process using the same depth map rendering module. As a final piece of information a camera is defined. For the camera the necessary calibration parameters are specified and the camera is made available for rendering. As soon as “execute” is called, the depth map rendering module renders the information provided before graphically.

4.2.2 Mesh Rendering Module

The mesh rendering module provides the following functions:

- Transform: retrieves spatial transform information for mesh rendering using a converter

- Texture: retrieves texture information for mesh rendering using a converter
- Mesh: retrieves the mesh for mesh rendering using a converter
- Execute: executes the module for a given acel input

Similar to the depth map rendering module the functions provided by the mesh rendering module can be used to render geometry information available in mesh format. The mesh information can be spatially transformed and a texture can be applied to the mesh. All information is made available through the Scene API in the requested formats by using the converters which can provide these formats. When calling the “execute” function information provided prior to the mesh rendering module is rendered graphically.

4.2.3 Camera Module

The camera module provides the following functions:

- CameraSettings: retrieves camera settings using a converter
- Execute: executes the module for a given acel input

The camera module describes functions for changing the camera settings. New camera settings can be loaded into the module from a Scene file using the Scene API. They are applied for rendering by calling the “execute” function.

4.3 The Renderer Interface

The renderer itself provides an interface to make use of the renderer using external scripting languages.

The *get_proto_renderer*-function is the entry point to the renderer implementation. The renderer which is returned provides access to the rendering modules described above. Once the renderer is created the modules can be created and used in further processing steps. If a module does not exist or support the request an “UnsupportedModuleError” is thrown.

In addition to the modules defined in the previous section, the renderer provides the following functions:

- Closing
- BeginFrame
- EndFrame

4.4 Example for Video Rendering

The current prototype renderer is available as a Windows installer and can be employed using the Python scripting language.

Here is a description of how video rendering with the Scene API works. For illustratative purposes Python-Code snippets are added to the individual steps. 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 so that the 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. This module in the Scene API provides a unique interface for the the renderer in the Scene API, adding more functionality that is only required by the renderer (such as providing the scene content frame-based). 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()
```

4.5 Rendering

Currently the renderer can render depth coming from mesh data as well as depth maps from depth capturing devices plus textures.

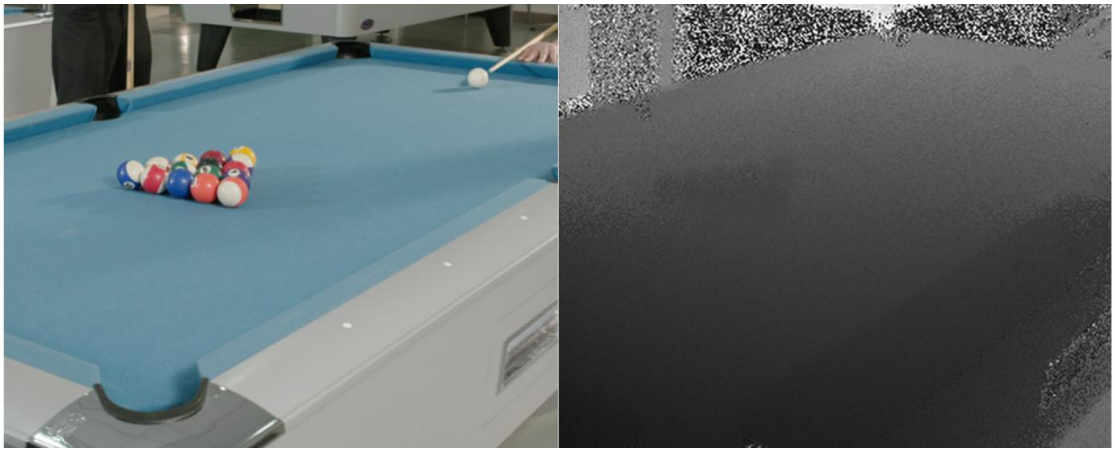

Figure 3. Video plus Depth data.

Texture data can be retrieved from the SRA as either individual images or a video stream. Depth map information can be retrieved from the SRA as individual frames with floating point precision (OpenEXR format). For camera and geometry information the Collada format which can be exported from multiple tools like Blender is preferred.

Figure 4. Rendered video and depth data.

Noisy depth data has shown to introduce very severe artefacts in the rendering results. Best results can currently be achieved with synthetic data, which is also used to show the functionality of the renderer. Data pre-processing algorithms to achieve a suitable rendering quality need to be developed in the Scene project for better rendering results. Such algorithms are currently developed in WP3 and WP4. WP3 enhances the quality of captured data to a certain degree. Further information that can enhance the rendering quality is added in WP4.

Figure 5. Synthetic Video plus depth render.

5 Integration of SRA models in real time virtual sets

eStudio has been designed trying to make it as open as possible. It has a plug-in system that allows the plug-in developer to access a vast number of internal functions and also the whole user interface can be accessed not only by hand but also by means of Python scripts or from C or C++ compiled modules.

The plug-in system uses different types of modules depending on the desired purpose. Some examples for such module categories are database accessing modules, video input or output

plugins depending on the video card or frame grabber, input device plugins, camera tracking plugins, etc...
This open system includes almost every module in the system but the file loading system. eStudio is able to load the most common file formats but in order to make it capable to load new ones it is necessary to modify the application itself.
The need to support an SRA directly in the graphics engine has been shown, however, that it is necessary to re-design a new type of plugins that should allow loading 3D models coming from new or proprietary file formats, even those created by eStudio users if they have the ability to develop their own loader plug-ins.

In order to include tridimensional models inside the eStudio scene graph, it is needed to create buffers of vertices, normals and texture coordinates, all these actions can be done from the application kernel but not from its plug-ins. Already existing types of plugins have full access to every element in the application interface and also to deeper functions, but there is no means to create geometries or access their information.

In order to grant this access two different plug-in modes have been designed:

1. *Based on eStudio GUI.*
   By giving access to geometry buffers in the graphics user interface, any plug-in or python script can create objects automatically.
2. *Based on modification on the loading system.*
   Instead of the hard code different types of files that the graphics engine is able to read, the file loading system will now be module based, so when trying to open a new file, and depending on its extension, the corresponding dynamic library, for example libSRA.dll, will be used to load it. A new API that gives access to all the geometry buffers, materials, textures, etc... has been prepared in order for these libraries to be able to create the same structures that were created previously by the application kernel when loading an object

5.1 eStudio GUI and plug-in system

eStudio design is based on different main elements:

- A **Kernel** responsible for the whole system functionality, status and real time rendering.
- A **Graphical User Interface** that communicates every single user interaction to the kernel by means of python sentences.
- A **plug-in system** that has access to every interactive item in the GUI plus other deeper functions intended to be used just by software modules.

- A **Python interpreter** that allows executing code at loading time, by frame or under certain triggered events.

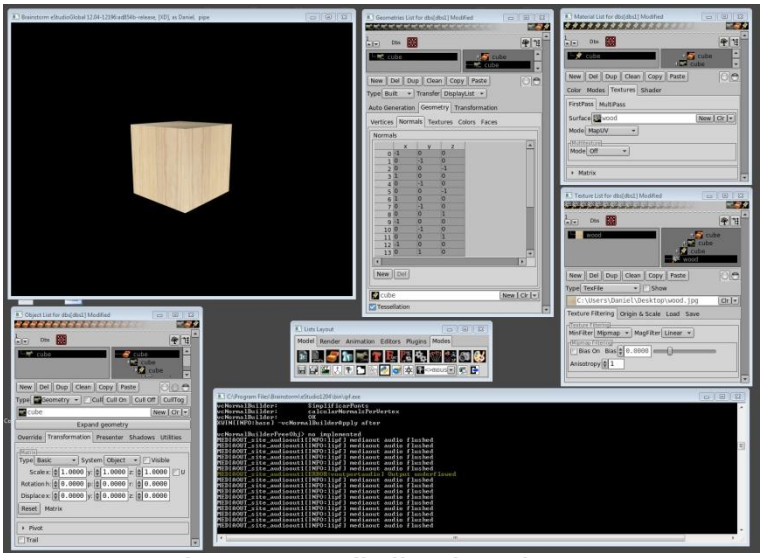

Figure 6. eStudio lists based GUI.

Based on this environment, the fastest way to allow creating objects has been to extend the GUI functionality for the user so it is possible now to create new objects directly from the interface.

Of course it would not make much sense for a user to create complex objects based on this interface, but since every single element in the user interface is accessible from Python scripts and from compiled plug-ins, it becomes possible now to program Python or C++ modules that after reading a file can create the corresponding model inside the eStudio architecture.

Figure 7. Main interface gives access to all the application lists.

eStudio GUI and its internal functionality is organized in lists of items. Every single parameter is part of an item inside one of the application lists:

- Cameras
- Lights sources
- Objects
- Math primitives
- Geometries
- Texts
- Particles

- Materials
- Textures
- Sounds
- Timers
- Trajectories
- Events
- Python scripts
- Plugins

Loading a new object, usually involves the creation of items in three different lists; Geometries, Materials and Textures. Plug-ins were already able to create and modify items in the Materials and Textures lists, but geometries where holding data buffers that were not accessible outside the kernel. A new type of geometry type (**Built Geometry**) has been created now that allows the user creating and accessing all these buffers of vertices, normals, texture coordinates and faces.

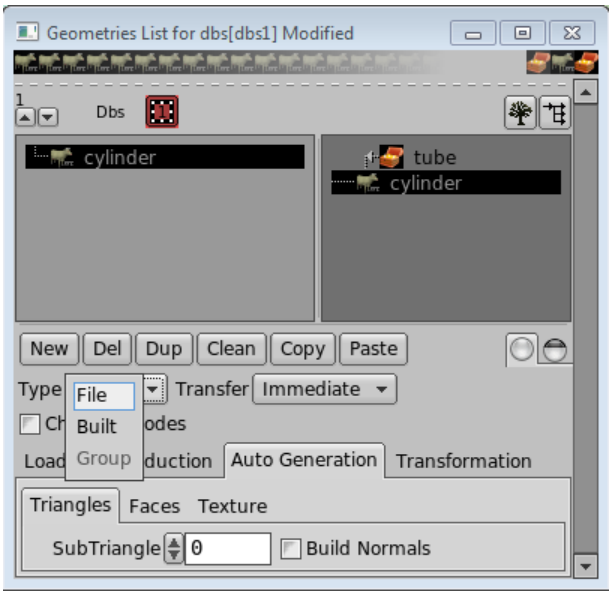

Figure 8. New type "Built" of geometry created

The new type "**Built**" exposes in its interface the different geometry buffers that now can be accessed directly in the GUI:

- Vertices
- Normals
- Texture coordinates
- Per vertex colors
- Faces

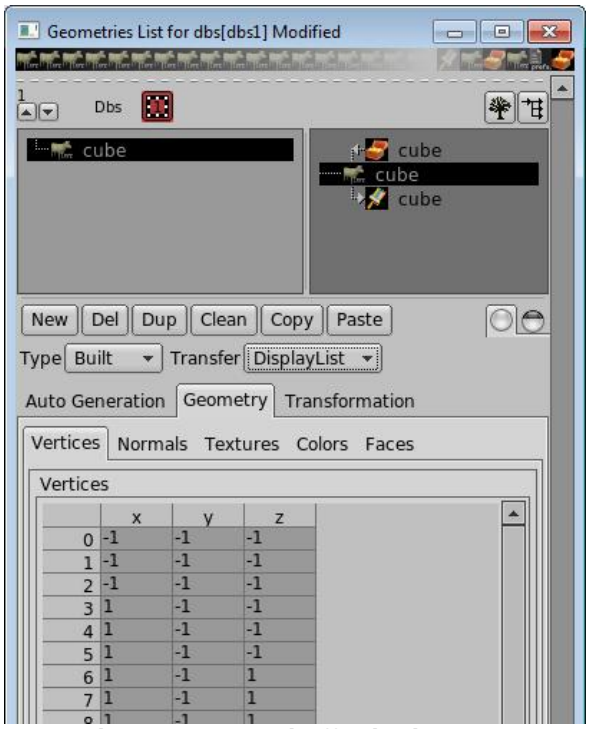

Figure 9. Vertex buffer in the GUI.

By double clicking any vertex component, it can be edited. The render window updates it in real time.

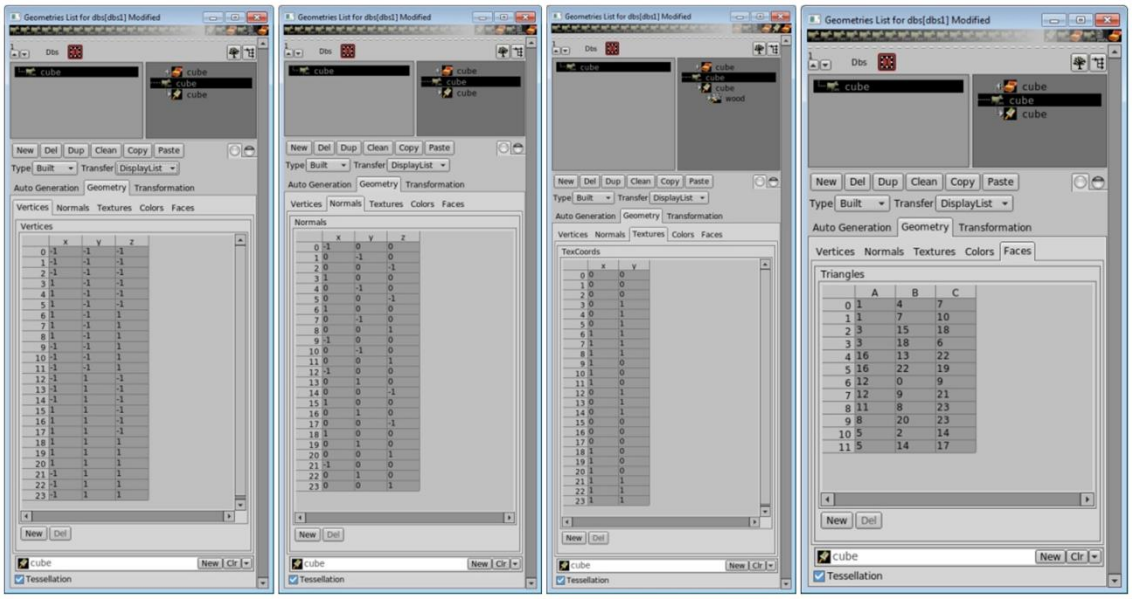

Figure 10. Vertex, normals, texture coordinates and faces of cube created by means of the GUI.

These are the Python functions that allow creating the different geometry buffers:

- Vertex buffer:
  itemset("cube", "GEO_VERTEX", [[x0,y0,z0], …… ,[xn, yn, zn]])

- Normal buffer:
  itemset("cube", " GEO_NORMAL", [[x0,y0,z0], …… ,[xn, yn, zn]])

- Texture corrdinates buffer:
  itemset("cube", "GEO_TEX", [[u0, v0], ……. [un, vn]])

- Faces buffer:
  itemset("cube", "GEO_TRIS", [[pi, pj, pk], ……. [pl, pm, pn]])

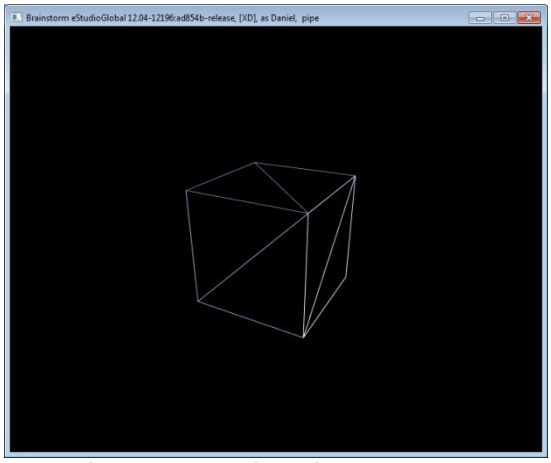

Figure 11. Basic cube geometry.

A complete python script capable to create a built geometry cube is shown here:

```
itemnew("<>tex", "wood",
    "TEX_FILE", "C:\\wood.jpg")

itemnew("<>mat", "cube",
    "MAT_COLOR", vector4(1, 1, 1, 1),
    "MAT_SHADE_MODEL", "gouraud",
    "MAT_SELSURF", "wood",
    "MAT_TEXSUP_MODE", "MapUV")

itemnew("<>geo", "cube",
    "GEO_TYPE", "Built",
    "GEO_TRANSFER_MODE", "DisplayList",
    "GEO_MAT", "cube")

itemset("cube", "GEO_VERTEX", [[-1, -1, -1], [-1, -1, -1], [-1, -1, -1], [1, -1, -1], [1, -1, -1], [1, -1, -1], [1, -1, 1], [1, -1, 1], [1, -1, 1], [-1, -1, 1], [-1, -1, 1], [-1, -1, 1], [-1, 1, -1], [-1, 1, -1], [-1, 1, -1], [1, 1, -1], [1, 1, -1], [1, 1, -1], [1, 1, 1], [1, 1, 1], [1, 1, 1], [-1, 1, 1], [-1, 1, 1], [-1, 1, 1]])

itemset("cube", "GEO_NORMAL", [[-1, 0, 0], [0, -1, 0], [0, 0, -1], [1, 0, 0], [0, -1, 0], [0, 0, -1], [1, 0, 0], [0, -1, 0], [0, 0, 1], [-1, 0, 0], [0, -1, 0], [0, 0, 1], [-1, 0, 0], [0, 1, 0], [0, 0, -1], [1, 0, 0], [0, 1, 0], [0, 0, -1], [1, 0, 0], [0, 1, 0], [0, 0, 1], [-1, 0, 0], [0, 1, 0], [0, 0, 1]])

itemset("cube", "GEO_TEX", [[0, 0], [0, 0], [0, 0], [0, 1], [0, 1], [0, 1], [1, 1], [1, 1], [1, 1], [1, 0], [1, 0], [1, 0], [0, 1], [0, 1], [0, 1], [0, 0], [0, 0], [0, 0], [1, 0], [1, 0], [1, 0], [1, 1], [1, 1], [1, 1]])

itemset("cube", "GEO_TRIS", [[1, 4, 7], [1, 7, 10], [3, 15, 18], [3, 18, 6], [16, 13, 22], [16, 22, 19], [12, 0, 9], [12, 9, 21], [11, 8, 23], [8, 20, 23], [5, 2, 14], [5, 14, 17]])

itemnew("<>obj", "cube",
    "OBJ_TYPE", "Geometry",
    "OBJ_GEOMETRY", "cube")
```

The C API, used in compiled plugins, is very similar to the Python one but much faster as it passes the float buffers directly:

- Vertex buffer:
  float vertices[] = {x0, y0, z0, x1, y1, ….., yn, zn}
  editemSetEditorFromArray(pItem->geoBell, geoVertexBell, vertices, sizeof(vertices));

- Normals buffer:
  float normals[] = {x0, y0, z0, x1, y1, ….., yn, zn}
  editemSetEditorFromArray(pItem->geoBell, geoNormalBell, normals, sizeof(normals));

- Texture corrdinates buffer:
  float tex[] = {u0, v0, u1, v1, ….., un, vn}
  editemSetEditorFromArray(pItem->geoBell, geoTexBell, tex, sizeof(tex));

- Faces buffer:
  float tris[] = {n1, n2, ….. nm}
  editemSetEditorFromArray(pItem->geoBell, geoTrisBell, normals, sizeof(tris));

A part of the different geometry buffers, the user can select the material to be applied on the geometry from the list of materials, that does not need any change in order for its configuration be accessible:

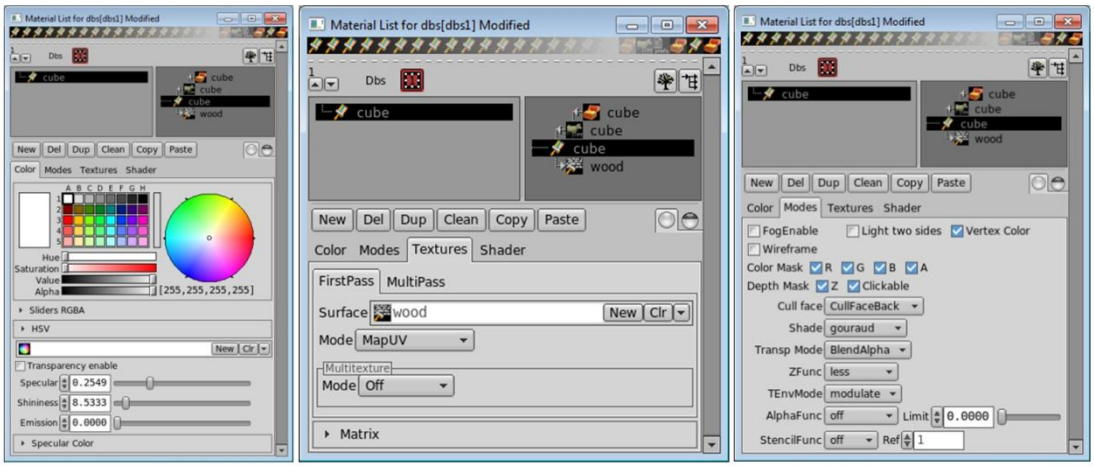

Figure 12. Material parameters in its list interface.

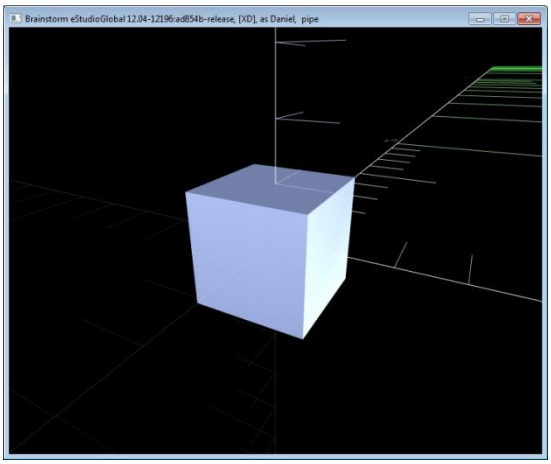

Figure 13. Blue material applied on the geometry.

The material in turn references a texture to be applied following the texture coordinates. Again, all the parameters that define the texture can be accessed directly by the user, by Python Scripts or by plug-ins.

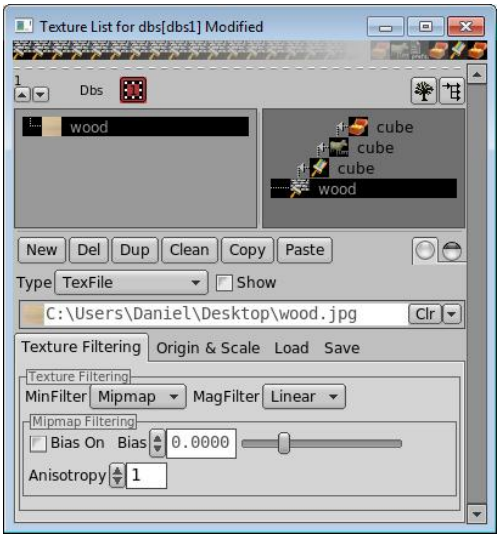

Figure 14. Parameters that define the texture.

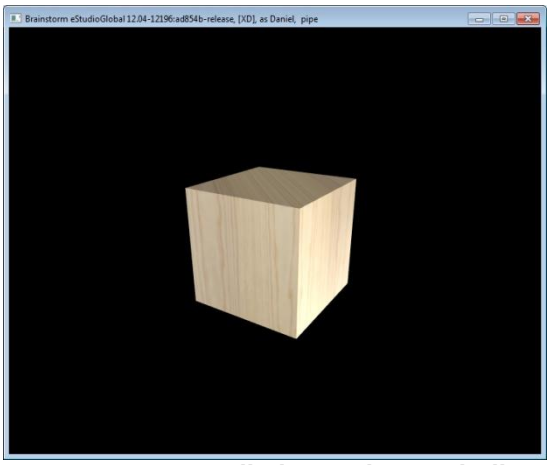

Figure 15. Texture applied over the GUI built cube.

In order to place the geometry in the scene and make it visible, it must be referenced by an item in the list of objects, an object. As part of allowing the geometry to be displayed, an object item allows positioning the geometry not only in the space but also in the system scene graph.

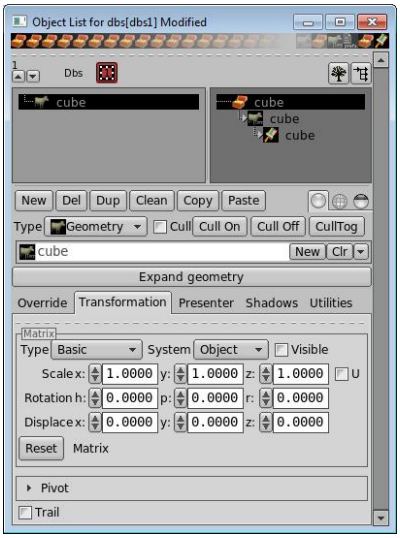

Figure 16. Object item referencing the cube geometry.

Once the system is able to load and build new objects in the scene, they can be integrated in already existing scenarios with video captured talent images.

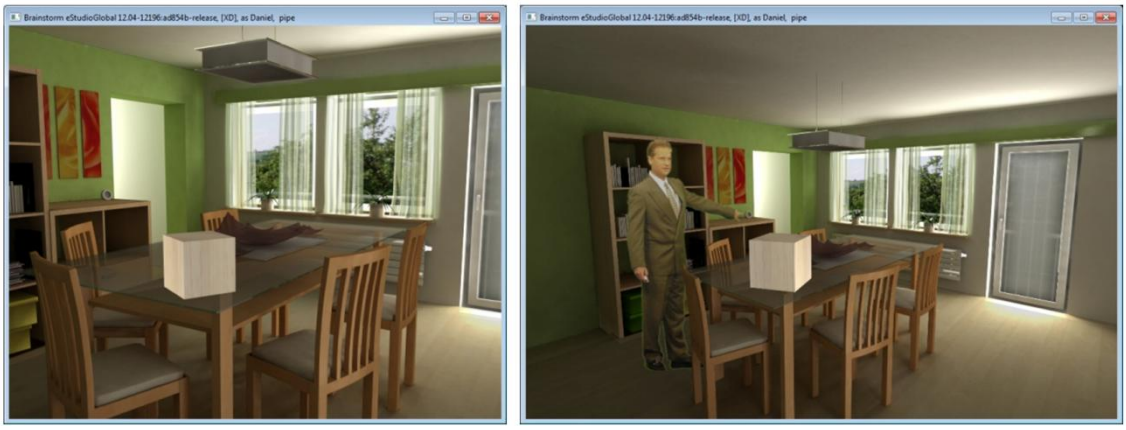

Figure 17. Final integration of a built geometry in existing scenarios.

5.2 New eStudio file loading system

Granting access to the geometries buffers in the eStudio GUI has been very convenient not only because it has been relatively easy to implement but also because plugins based on this system are very fast to test and debug directly over the application GUI.

This system lacks, nevertheless, some interesting properties:

- It is needed to load a plug-in before being able to load the corresponding type of file.
- It is not as fast as the native file loading system.

But a part of this method, a new system has been also created that will overcome these two drawbacks. This new approach has led to a complete refactoring of the file loading system, which, until now was able to load four different types of models, 3DS, OBJ, DEM and M3B. eStudio users usually convert their models to M3B, the eStudio native format or directly export the models from 3DStudio or Maya as M3B files by means of the BS exporter plug-ins.

This new file loading system objectives are to be fast, extensible, open so expert users can create their own loading modules, and of course user friendly. It is module based instead of hard coding each different file type. For this reason a new API was necessary in order to allow modules to translate file contents into data inside the graphics engine kernel. Below there is an explanation of this system and its API.

eStudio started its development on Irix platforms and therefore it was based on OpenGL and Performer, a high level 3D graphics API that was original created based on OpenGL by Silicon Graphics. When the company started to port its products to Linux, and Windows graphics stations a problem was found related to the Performer library. Performer became available for Linux systems but Silicon Graphics never released a Windows version. BS decided then to develop its own high level library that in many cases remains similar to Performer. The building geometries functions are an example of these similarities.

File loading modules are implemented as dynamic libraries that should follow two main directions:

- Their file name must be of this form libXXX.dll.
  In the sra case it will be ***libSRA.dll***.
- They must implement the following symbol:
  extern m3Node* m3dLoadFile (const char *fileName);

Instead of passing the buffers directly, in this case, the API allows to build every face in the geometry and then let the system optimize it by removing redundant information. Once the geometry has been completely built and optimized the module returns a data structure that holds the loaded module.

The API is C based and provides the following functions:

A builder struct allow creating new geometry, the responsible function to start a builder instance
m3dBuilder * m3dNewBldr(void);

Before starting the creation of any geometry, the builder must be initialized:

void m3dInitBldr(void);

After a geometry has been built and before building a new one, the builder must be informed:
void m3dResetBldrGeometry(void);

Geometries can be named:
void m3dSelectBldrName(void *name);

The responsible function to create and return a new m3dGeom is m3dNewGeom It will return a m3dGeom with room enough for numElement points:

m3dGeom * m3dNewGeom(uint numElements);

m3dGeom fields can be accessed directly, the main fields used here are:

- The number of vertices in the geometry: **m3dGeom.numVerts**
- The type of primitives forming the geometry: m3dGeom**.primtype M3GS_POLYS**
- The buffer of vertices: **m3dGeom.coords[]**
- The normal buffer: **m3dGeom.norms[]**
- The texture coordinates buffer: **m3dGeom.texCoords[]**

If needed, the geometry size can be extended with:
void m3dResizeGeom(m3dGeom geom, uint numElements);

As geometries primitives are defined, they are sent to the sytem by means of this function;
void m3dAddBldrGeom(m3dGeom, uint numElements);

Once all the data has been sent to the builder, it can optimize it and return the resulting scene graph node:

m3Node*m3dBuild(void);

Conclusion

This document adds a description to the prototypes delivered as software components. The Scene renderer is a first renderer to render scene data provided by the SRA API. It can renderer spatial video information coming from video plus depth and mesh information, and therefore shows how scene data from different sources can be seamlessly integrated First effects like shadowcasting are already implemented, and the structure for further enhancements is given. While the current prototype is a first hint at the possible advances it also presents requirements to data quality and calibration which needs to be fulfilled to achieve hiqh quality results.

The main reason to make eStudio compatible with SRA contents is to make it capable to load and render some of the Scene capturing technologies results, those that are real time compatible. Then, as eStudio is a commercial product used by a considerable number of clients, integrating these capabilities in it will automatically make the Scene technology available to those clients that could require it and that already own a virtual set and some of the equipment required to get the best results out of the Scene technology.

In short, this integration could spread the use of some of the project technologies to a number of TV broadcasters that have already shown their interest in them.