Scene Graph Adapter: An efficient Architecture to Improve Interoperability between 3D Formats and 3D Application Engines

Rozenn Bouville Berthelot, Jérôme Royan, Thierry Duval, Bruno Arnaldi

▶ To cite this version:

Rozenn Bouville Berthelot, Jérôme Royan, Thierry Duval, Bruno Arnaldi. Scene Graph Adapter: An efficient Architecture to Improve Interoperability between 3D Formats and 3D Application Engines. Web3D 2011 (16th International Conference on 3D Web technology), Jun 2011, Paris, France. pp.21-30. inria-00586161

HAL Id: inria-00586161

https://hal.inria.fr/inria-00586161

Submitted on 6 Apr 2014

Scene Graph Adapter: An efficient Architecture to Improve Interoperability between 3D Formats and 3D Applications Engines

Rozenn Bouville Berthelot[*] Orange Labs and IRISA, Rennes, France
Jérôme Royan[†] Orange Labs France
Thierry Duval[‡] IRISA, Rennes, France
Bruno Arnaldi[§] IRISA, Rennes, France

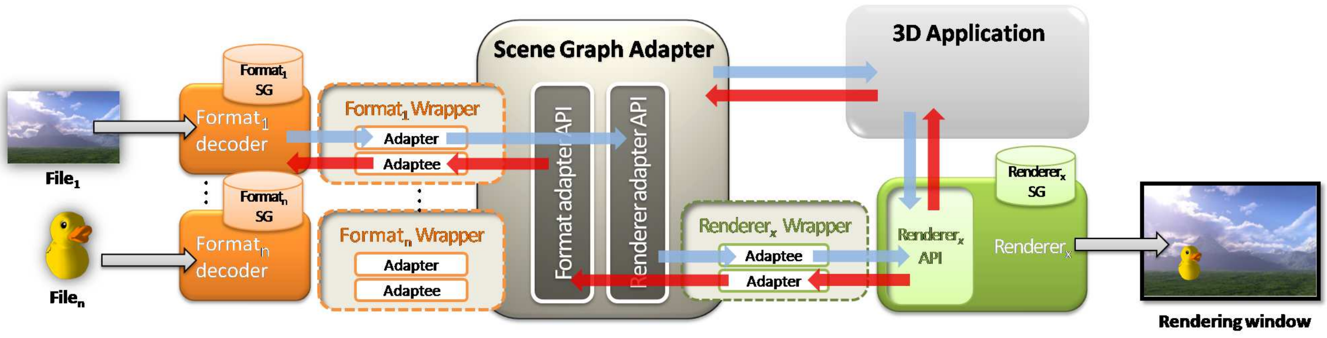


Figure 1: *Our architecture allows the loading of any 3D graphics format simultaneously in any available rendering engine. The scene graph adapter is an interface that adapts a scene graph (SG) of a given format into a renderer scene graph and which also allows the rendering part to request this scene graph.*

Abstract

We present a generic architecture which enables the loading of several 3D formats in most 3D application engines. The purpose of this work is to solve the interoperability issue among 3D formats raised by the multiplicity of formats. We propose a programming interface, the Scene Graph Adapter, that allows mixing several 3D formats in a single application without transcoding. The Scene Graph Adapter is composed of two API; a first one that wraps each 3D format and a second one that wraps rendering engines. Wrappers created thanks to these API can be reused in other applications based on the Scene Graph Adapter. The paper introduces our approach and gives an implementation example.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual reality; D.2.12 [Software Engineering]: Interoperability—Data mapping; D.2.13 [Software Engineering]: Reusable Software—Reusable libraries

Keywords: interoperability, 3D format, software architecture, rendering, virtual worlds, web3D

1 Introduction

In 1994, the VRML was conceived at the first annual World Wide Web Conference in Geneva, with the goal to provide a language for describing multi-participant interactive simulations, virtual worlds networked via the global Internet and hyperlinked with the World Wide Web. This great and pioneering initiative was and is successful in the industrial domain, but did not reach the success as hoped in the field of applications for the public at large. This problem has been extensively discussed in the Web3D community. Its lack of extensibility and the fact that 3D tools and modelers did not provide a good VRML export were certainly majors issues. However, the main reason of its qualified success was probably the fact that VRML appeared much before processors and networks could support the graphics that the technology enabled, as noticed by Pesce [Pesce 1994] and Ortiz [Ortiz Jr. 2010]. Indeed the VRML was essentially used to visualize simple 3D content on the Web. Nonetheless the VRML has served as a basis to develop many 3D formats and platforms for virtual worlds during the last decade, increasing the options to create 3D services within virtual worlds. Unfortunately, this proliferation of virtual world platforms raises a huge interoperability issue: 500 are currently online and more than 900 are forecast for 2012. The various developments concerning Web3D do not respect one of the main and fundamental characteristic of the Web: its uniqueness. This increase of 3D formats, protocols, browsers for virtual worlds generates the following consequences.

First of all, we can maintain that the proliferation of non interoperable platforms for virtual worlds does not encourage industry to invest in virtual worlds. Although the number of users in virtual worlds is estimated at over a billion, a dream come true for a marketer, the fragmentation of users among a large number of virtual worlds disrupts the principle of the current Web 2.0 where investments in a Web service target all users who have access to the Web. In other words, developing a service for a virtual world will target only the users of this virtual world. The ratio between investing in the development of a service in virtual worlds (designing, modeling and scripting) and the number of targeted users cannot be compared to what currently exists on the Web. Unfortunately, this ratio often makes the difference when decision-makers have to choose between improving their current web services or investing in virtual worlds. When we know that the emergence of the web is directly linked to its

[*]e-mail: rozenn.bouville@orange-ftgroup.com
[†]e-mail:jerome.royan@orange-ftgroup.com
[‡]e-mail:thierry.duval@irisa.fr
[§]e-mail:bruno.arnaldi@irisa.fr

adoption by the industry, we can conclude that interoperability between virtual worlds is a main issue for the emergence of Web3D.

On the other hand, consortiums committed to the creation and deployment of standards, whether open, royalty-free, ISO or not, provide outstanding efforts of specification to make virtual worlds consistent with each other. To be successful, however, a standard has to be adopted by a great majority of the community. To achieve this, several criteria can improve the promotion of 3D standards such as:

- offering specific functionality required by the community,
- providing a way to create content, such as a dedicated authoring tool, or export modules for authoring tools that are widely used on the market,
- providing a way to visualize the coded content, thanks to a dedicated viewer, thanks to a transcoding module to fit the content to renderers that are not directly compliant with the standard, or finally, thanks to a loader that can be easily plugged in to existing platforms for virtual worlds.

In this article, we focus on the latter item, namely the solutions for using 3D content encoded in a given standard within most virtual world platforms. The development of a virtual world platform around a viewer dedicated to a specific standard is not currently widely used. Indeed, virtual worlds developers require not only a 3D viewer, but a complete platform embedding a set of engines for rendering, physics simulation and networked synchronization. Concerning the rendering engine, they generally opt to develop it themselves, or reuse "generic" engines (Unity, Ogre3D, Irrlicht, CryEngine), instead of using a viewer dedicated to a specific standard. Thus, a good way to import standardized content within a virtual world is to transcode it to a compliant format. Unfortunately, these widely used solutions can become restrictive, as the transcoding tools from a standard to a format compliant with an engine have to be developed, and transcoding generally results in the lost or the modification of data such as interactions, physics parameters, texture coordinates and so on. Finally, to provide developers of virtual worlds with a software component that allows them to directly load and run a standardized content appears to be an interesting solution. Except that the architecture of virtual worlds platforms must be well-suited to reduce integration costs. In conclusion, no real solution exists today to easily integrate standardized content within virtual worlds platforms, which could curb the adoption of 3D standards.

In the remainder of this paper, we aim to propose a new architecture for Web3D platforms providing an easy way to integrate any 3D standard and mix them in the same 3D scene. To achieve this goal, in Section 2, we describe related works concerning the solutions improving the interoperability for Web3D. Then we present in Section 3 a global overview of the proposed architecture which components will be detailed in Section 4. Afterwards, we propose in Section 5 an example of implementation and show the first results. Finally, we discuss the efficiency and usability of the proposed architecture and finally conclude in Section 6.

2 Related work

VRML was the first attempt to put 3D contents in a web browser and people who were working on that subject had great expectations as told in [Lea et al. 1996]. This first initiative did not have the expected success as explained in section 1. Indeed, today, 3D technology is used online but not in a web browser, most often through applications dedicated to games and virtual worlds. Indeed, general web browsers cannot natively run complex 3D content. There are a lot of available solutions, a complete overview of these solutions can be found in [Behr et al. 2009]. Most of them create a new format that can be run on a web browser through plugins; among them the most recent are Universal 3D [Standard ECMA-363 2007], 3DMLW [3D Technologies R&D 2009] and 3DXML [Sons et al. 2010]. The problem with these solutions is that users prefer not to use plugins because they find them inconvenient to install.

The WebGL [Khronos Group 2011] solution does not rely on a plugin. It is a standard managed by the Khronos Group that enables declarations and interpretations of OpenGL ES 2.0 instructions directly in the javascript part of an HTML page. These instructions are then run using hardware capabilities of the client machine. This represents a huge step toward enabling a widespread use of 3D on the web. Latest versions of web browsers like Google Chrome included WebGL before the final specification was issued. It shows how much it is expected by the web community.

Further work has been done with the X3DOM project [Behr et al. 2010]. The latest HTML5 specification mentions X3D as the reference format for 3D contents in XHTML document and X3DOM which is based on HTML5 enables the loading of X3D files in any web browser that supports WebGL without plugin installation. Through their open source framework and runtime, they overcome WebGL limitations by providing an adaptation of the rendering technique (native X3D, WebGL based or Flash).

The aforementioned solutions are very efficient and will surely increase the amount of 3D content on the web but they do not solve the problem of reusing existing content. Creating 3D content is indeed expensive and time-consuming, and the amount of online 3D applications such as games or metaverses is increasing every year. The only solution is to bring interoperability between 3D formats, as stressed by Polys [Polys et al. 2008] it is a "crucial requirement for success".

Interoperability has been studied in various fields of computer science and we use Haslhofer's classification of metadata interoperability presented in [Haslhofer and Klas 2010] to identify interoperability solutions. There are 3 categories of solutions:

1. the model agreement,
2. the metamodel agreement,
3. the model reconciliation.

The model agreement consists in establishing a standard by means of consensus building. It is an intuitive, technically effective and economically well-recognized way to achieve interoperability. To be efficient, it requires the acknowledgment of an institution (e.g. W3C, ISO, . . .). It would have been a good solution for 3D formats, nevertheless it is hard to apply considering the present situation. There are indeed more than 50 3D formats and although some of them are more or less abandoned, new ones continue to appear. The need to bring new features that are not present in existing formats is the reason why new formats are still emerging. Thus, establishing a standard will imply keeping those features to have a real compromise. WebGL seems to be a good alternative but it relies on OpenGL ES 2.0, which is entirely shader-based. It means that we have no real scene graph and that all vertex attributes of a mesh need to be declared before being used in a vertex shader. To use this standard with existing 3D formats we have no other choice than to resort to transcoding, but this technique often leads to a loss of functionality.

The metamodel agreement consists in agreeing on a common metamodel. For all existing models, instances of each model feature are created in the metamodel as if they were translated in the metamodel. Metamodel agreement implicitly enables interoperability by creating correspondences between the features of an existing model and those of the common model. If we apply this solution to 3D formats it means to create a metaformat that includes every feature of every 3D format. This solution is as efficient as the model agreement but it requires it to be comprehensive and extensible to be widely adopted. However this solution applied to 3D formats seems hardly feasible since there exist too many 3D formats and new ones will continue to appear. This makes this solution non-sustainable because it is not extensible. To our knowledge, there is no such solution for 3D formats.

The last interoperability solution is the model reconciliation. It achieves interoperability by reconciling heterogeneities among formats. This is the most complex way to achieve interoperability because heterogeneities can be of multiple types in a model. However, as for 3D formats, there are many similarities between them making reconciling heterogeneities easier than in other domains. Besides it is an extensible solution so that it is suitable for 3D formats. Some solutions already exist for 3D formats like MPEG-V (MPEG for virtual worlds) [Gelissen 2008] that aims at standardizing intermediate formats and protocols to exchange information between virtual worlds. Another similar initiative is the VWRAP project [Bell et al. 2010] but this solution only works on a set of virtual worlds similar to Second Life (i.e. spatially partitioned into regions hosted by simulation servers, with agents representing users and user-controlled via client application). Fang et al. [Fang and Cai 2009] propose a solution based on RESTful web services.
The PLUG solution [Hu and Jiang 2008] presents a single universal client for all online virtual environments. This client can retrieve several virtual worlds from hosting sites. Virtual environments are run on this hosting site and retrieved by the client using image streaming techniques. In the work from Soto and Allongue [Soto and Allongue 2002], the focus is laid on reusability of virtual worlds entities. They use multiagent concepts and learning techniques to enable the transfer of an entity from a virtual world to another.

From our point of view and as regards reusability as well as keeping heterogeneities of 3D formats, the model reconciliation is the best solution. Thus we propose to design a generic API that interfaces a format and a 3D application. It is based on similarities between 3D formats and 3D application components (rendering engine, physics engine, etc.). This API enables the adaptation of a format scene graph into a renderer scene graph. Our goal is to provide the use of any available 3D format without functionality loss whatever is the targeted viewer.

3 Overview of our architecture

In this chapter we present our architecture for building 3D applications that use several 3D formats in most rendering engines. It is based on an API, the Scene Graph Adapter, which aims at interfacing communication between 3D application inputs (e.g. 3D files) and output (e.g. the interactive visualization window).

3.1 Concepts and prerequisites

When creating a 3D application we have technical and functional requirements that lead us to the choice of a rendering engine. But rendering engines support a limited amount of input formats that makes the choice even more difficult when we also have formats requirements. These requirements could be the need to reuse existing 3D contents or to work with specific modeling tools which formats are not supported. Even if 3D formats and rendering engines have different usages and functionalities, they nonetheless have many similarities. Most of them are indeed based on a scene graph data structure. They also use similar ways to model and organize data in the scene graph. For example they have a similar shape description structure with a separation between geometry and appearance. They also use similar modeling transformations as well as similar primitive shapes. For 3D formats, this can be explained by the fact that since the concept of scene graphs was introduced by Open Inventor [Strauss and Carey 1992] in 1992, no other concept as efficient and flexible as this one has been proposed. It indeed reflects the underlying rendering pipeline.
The explanation for rendering engines is more obvious. In fact they all rely on one or both of the existing low-level API: OpenGL and DirectX. Thus, their evolution follows those API improvements as well as GPUs improvements. It was for example the case when GPUs enabled shader programming.

In [Hinrichs 2000] and later in [Döllner and Hinrichs 2002], Döllner and al. have proposed a generalized scene graph structure based on these similarities with a view to improve the rendering process. Steinicke et al. [Steinicke et al. 2005] propose a generic virtual reality software system based on Döllner's work in which rendering can be performed by several low-level rendering APIs. We have used those previous works to design our API, the Scene Graph Adapter API.

3.2 Overall architecture

The figure 1 depicts our architecture. The purpose of the Scene Graph Adapter API is to enable communication between an input scene graph of a given 3D format and the output scene graph of a rendering component (rendering engine, physics engine, behavior engine, etc.) in a 3D application. This application can be a game, a plugin, a GUI for a 3D display and so on. We call the first scene graph the *format scene graph* and the second scene graph the *engine scene graph*. In this paper, we will only use a rendering engine, this is why the *engine scene graph* is called the *renderer scene graph*.

To complete our architecture we need components that load, decode and adapt a *format scene graph* using the Scene Graph Adapter API on the input side. Similarly, on the rendering side of our architecture, we need a component that adapts instructions from the Scene Graph Adapter API into instructions of the renderer API to build the *renderer scene graph*. We call these two components the *format wrapper* and the *renderer wrapper* respectively. There is only one format wrapper for every file of a given format as well as one renderer wrapper per rendering engine.

Format wrappers do not depend on an application. They can be reused in any application providing that it relies on the Scene Graph Adapter API. Yet a *Format Wrapper* depends on a Format Decoder. Format Decoders are composed of existing tools like a parser or a loader that help at developing a *format wrapper*. Every 3D format indeed comes with at least a viewer so that this part should not be made from scratch. Besides as every 3D format possess specific features, reusing these pre-existing tools helps at keeping them up to date. The Format Decoder decodes and parses the input file and builds an internal scene graph data structure of the *format scene graph*. Depending on the used decoder, this component may include an update message handler that receives event update messages and updates relevant nodes accordingly. If it is not the case, then it must be implemented in the *format wrapper*. The *format wrapper* then uses this representation to adapt it using methods from the Scene Graph Adapter. Our architecture allows to mix several *format wrappers* within a single application in order to mix different input formats.

Renderer wrappers do not depend on an application either and can be reused in any application based on the Scene Graph Adapter. A *renderer wrapper* relies on the rendering engine API in the same way a *format wrapper* relies on a Format Decoder. It uses methods from the Scene Graph Adapter API and adapts them using the rendering engine API to build and update the *renderer scene graph*. As explained in 3.1, the choice of a rendering engine is crucial when designing a 3D application. Therefore it is important to enable the use of most of the available rendering engines.

3.3 Benefits

Our architecture addresses the reusability issue and achieves the rendering of any 3D format without functionality loss. It has several benefits:

- First, it fully supports all the scene-graph-based 3D formats without rendering limitation, assuming that we have the appropriate *format wrapper*. It makes it possible to use old 3D models without transcoding and functionality loss. Thus it allows a more efficient collaborative work. Research teams can share their resources without being hampered by compatibility problems. Furthermore, depending on the requirements of an application, it is possible to use the most appropriate format without rendering engine restriction. It also makes easier the porting of a format into rendering engines; this can help to promote a new format.

- Second, it works with any rendering engine without input format restriction. Once a *renderer wrapper* is available, it gives access to every available *format wrappers*. An application can use the most appropriate rendering engine without being hampered by format compatibility. Designers team can use any modeling tool and export their 3D models without compatibility problems. It is also possible to directly import a modeler native format in the application while avoiding transcoding drawbacks.

- Third, it makes it possible to mix and reuse *format wrappers* and *renderer wrappers* as required by application. It allows to mix 3D formats and their functionalities. We can for example load a Collada model with physics properties in an X3D world and explore it using X3D's navigation and interaction features. Besides a wrapper can be reused in any application that is based on the Scene Graph Adapter API. It allows development teams to share their wrappers hence facilitating components reuse and teams cooperation. In addition, more and more 3D application use third-party rendering components instead of creating new ones. There are physics engines (Havok [1] , PhysX [2] , Open Dynamics [3] , etc.), character animation engines (Granny [4] , Havok Animation [5] , etc.) or AI components (Kynapse [6]). Thus, a wrapper for those components can be reused in many applications.

[1]http://www.havok.com/index.php?page=havok-physics

[2]http://www.nvidia.com/object/physx_new.html

[3]http://www.ode.org/

4 The Scene Graph Adapter

After having introduced the basic architecture of the Scene Graph Adapter, this section illustrates how the different components are working and how they interact together. The Scene Graph Adapter allows the rendering of multiple 3D formats simultaneously in a single view but also manages interactions and animations declared in the input files. To achieve this, the Scene Graph Adapter API provides a two-way communication between format wrapper and renderer wrapper.

4.1 The Scene Graph Adapter principle

Let Z be the set of all scene-graph-based 3D formats. A 3D content is defined by the couple (f, z) where f is a file and z is a 3D format with $z \in Z$. A 3D application includes several rendering components that we call engines (rendering engine, physics engine, behavior engine, artificial intelligence engine, etc.). Let E be the set of all engines. An engine can be characterized by the couple (e, t) where $e \in E$ and t is its type (rendering, physics, behavior, etc.).

Besides N is the set of nodes in a *format scene graph* and N' is the set of nodes in an *engine scene graph*. Then the triplet (f, z, n_i) with $n_i \in N$ characterizes the node with index i in the *format scene graph* of the file f encoded in the format z. Likewise the triplet (e, t, n'_i) with $n'_i \in N'$ is the node with index i in the *engine scene graph* of the engine e of type t. Within each scene graph a node has a unique index.

The Scene Graph Adapter allows to retrieve:

1. a *format scene graph* node given a *renderer scene graph* node.
2. *engine scene graph* nodes given a *format scene graph* node.

For a node n'_i of an *engine scene graph*, the Scene Graph Adapter returns $\emptyset$ or a node of a *format scene graph*. Let F' be the retrieve function:

$$F'(e, t, n'_i) = \emptyset \text{ or } (f, z, n_i)$$

Likewise, let n_i be a node in a *format scene graph*, the Scene Graph Adapter returns $\emptyset$ or a set of *engine scene graphs* nodes. Let F be the retrieve function:

$$F(f, z, n_i) = \{(e_j, t_j, n'_i) \mid F'(e_j, t_j, n'_i) = (f, z, n_i)\}$$

In other words, a node from a *format scene graph* can match none, one or several nodes in the *engine scene graphs* of an application. For example, that node can match one node in the rendering engine and another node in the physics engine of the application. Whereas, in the case of a LOD node, in the *format scene graph*, not all the level nodes will be present in the *renderer scene graph*. Only the selected level node matches a node in the *renderer scene graph*. On the other hand, a node from an *engine scene graph* can match none or a single node in a *format scene graph*. Some nodes in an *engine scene graph* do not match any node from an input file, such as the root node of the *renderer scene graph*.

[4]http://www.radgametools.com/granny.html

[5]http://www.havok.com/index.php?page=havok-animation

[6]http://usa.autodesk.com/

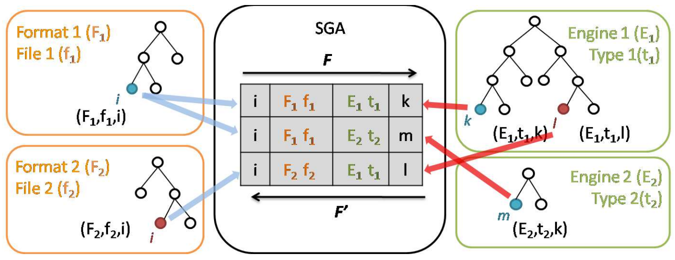


Figure 2: *The Scene Graph Adapter (SGA) principle: to each node of a* ***format scene graph****, the SGA gives the corresponding node in every* ***engine scene graph*** *of the application. Similarly, to each node of an* ***engine scene graph****, the SGA retrieves the corresponding node in the* ***format scene graph****.*

Figure 2 shows an illustration of this formalism. The Scene Graph Adapter executes this task while the application is running. It keeps those informations in a map which evolves depending on the rendering context (elapsed time, camera position, etc.).

4.2 The Scene Graph Adapter API

As shown in figure 1, the Scene Graph Adapter API is composed of two distinct API:

- the Format Adapter API,
- the Renderer Adapter API.

The Format Adapter API provides methods to update and query a *format scene graph*. These methods are called by the application or by *engine wrappers* and are implemented in a *format wrapper*. Implementation of these methods depends on the format and relies on the loader used in the Format Decoder. The Format Adapter API methods can be classified by:

- *scene graph loading*: loadFile, loadNode, etc.
- *frame events*: setTime, setViewpoint, etc.
- *user events*: onClick, onDrag, onRollOver, onKeyPressed, etc.
- etc.

The Renderer Adapter API is composed of methods to add, remove, update and query the *renderer scene graph*. They are called directly by the application or by a *format wrapper* and are implemented in a *renderer wrapper* using the chosen rendering engine API. The Renderer Adapter API methods can be classified by:

- *environment settings*: setViewpoint, setViewport, setBackground, setFog, etc.
- *camera settings*: createCamera, setCamera, etc.
- *navigation settings*: setNavigation, etc.
- *ligths settings* : setDiffuseLight, setAmbientLight, etc.
- *scene graph settings* : createNode, deleteNode, attachNode, etc.
- *geometry settings* : createSphere, createBox, createIndexedFaceSet, etc.
- *transform settings* : createTransformNode, setTransformNode, etc.
- *material settings* : createMaterial, setMaterial, etc.
- *texture settings* : createTexture, setTexture, etc.
- etc.

The Scene Graph Adapter API also provides tools that help at building new wrappers and new applications based on our architecture. First, the Scene Graph Adapter API maintains a node indexer. This indexer keeps informations about all the nodes of the *renderer scene graph* to identify them and retrieve the corresponding node in the *format scene graph* while the application is running. Moreover, in order to avoid naming conflicts in the *renderer scene graph*, the Scene Graph Adapter API provides a naming method that delivers an identifier for each node of the *renderer scene graph*. This identifier is then used as the key in the node index. An example will be given in section 5 to illustrate all these functionalities.

4.3 The wrappers

To create an application based on the Scene Graph Adapter API, once a rendering engine is chosen, all that needs to be done is to use an existing *renderer wrapper* for the chosen engine or build a new one. At this point, the application can load any 3D format provided that there exists an appropriate *format wrapper*. We have divided wrappers role in two tasks: an adapter task and an adaptee task. We use those terms with reference to the adapter pattern GoF 139 [Gamma E. et al. 1995]. In this section we explain for each type of wrappers the adapter and the adaptee tasks.

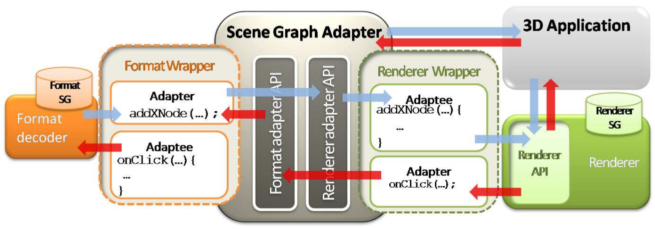


Figure 3: *The two tasks of wrappers: the adapter task and the adaptee task. Wrappers communicate from an adapter to an adaptee.*

Format wrappers

To build a *format wrapper*, the *format scene graph* is parsed by the Format Decoder and methods from the Renderer Adapter API are called to build a *renderer scene graph* based on this *format scene graph*. It is the **adapter task** of a *format wrapper*. On the other hand, methods from the Format Adapter API are implemented using the scene graph representation used by the Format Decoder. This is the **adaptee task** of a *format wrapper*.

Renderer wrappers

Building a *renderer wrapper* means implementing methods of the Renderer Adapter API using the chosen rendering engine API. This forms the **adaptee task** of a *renderer wrapper*. Besides, an action on the *renderer scene graph* that needs to query the original scene graph (i.e a *format scene graph*) must be implemented using the Format Adapter API, this is the **adapter task** of a *renderer wrapper*.

Figure 3 gives an example of these tasks for a *format wrapper* and a *renderer wrapper*. Wrappers communicate from an adapter to an adaptee.

Since almost all 3D graphics formats offer the possibility to modify the scene (e.g. switch node, scripts or level of detail), the Scene Graph Adapter provides two exchange modes between wrappers as illustrated in figure 4: a push mode and a pull mode. In push mode, the whole *format scene graph* is sent to the *renderer wrapper* which loads it on-the-fly. For example, if the *format scene graph* has a level of detail (LOD) node, the complete LOD information is sent to the rendering engine which manages LOD

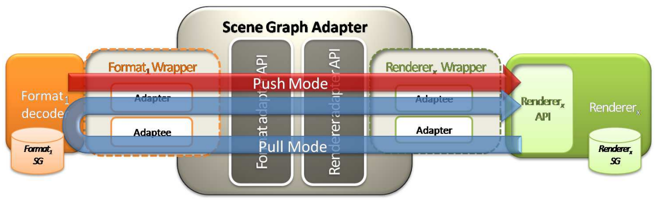


Figure 4: *The two exchange modes provided by the Scene Graph Adapter between a* ***format scene graph*** *and a* ***renderer scene graph****: a pull mode and a push mode.*

selection. Whereas in pull mode, upon request from the renderer, a subgraph of the *format scene graph* is sent to the *renderer wrapper*. Through that mode, LODs informations are kept by the *format wrapper* and the rendering engine is only aware of the LOD that is currently selected. Upon a change of camera position, the *renderer wrapper* sends the new position to the *format wrapper* which computes LOD selection and sends the new level nodes to the *renderer wrapper* if required. Actually we can provide different implementations of the same *format wrapper* depending on how it must interact with the *renderer wrapper*.

4.4 Integration in a 3D application

We have already explained how components are working and how they interact at runtime. We will now present how to integrate this architecture in a 3D application. The application is in charge of configuring the rendering engine and also of sending input files url to *format wrappers*. Figure 5 shows the messages exchanged by the different components of the architecture when the application is started. First, the application registers all the available *format wrappers* (message 1), then it configures the rendering engine (messages 2 to 4) and eventually it sends an url to the Scene Graph Adapter (message 5), which loads the appropriate *format wrapper* (message 6). The *format wrapper* then begins the adaptation of the *format scene graph* into the *renderer scene graph* (messages 7 to 12). The last message, number 13, initializes the animations described in the loaded file. This is carried out during the static rendering phase because the event system must be initialized as soon as possible so that animation can be performed as described in the input file when the rendering loop starts.

Figure 6 gives an illustration of how our architecture works for dynamic rendering. It shows how animations are managed by a *format wrapper* in order to run the animation described by the following X3D excerpt:

```
<Scene>
 <Transform translation='3␣3␣3' scale='5␣5␣5'>
   <Shape>
         <Sphere/>
          <Appearance>
                <Material DEF='MAT'/>
          </Appearance>
        </Shape>
 </Transform>
 <TimeSensor DEF='TIMER' loop='true'
      cycleInterval='5' />
 <ROUTE fromNode='TIMER' fromField='
      fraction_changed' toNode='MAT' toField='
      transparency' />
</Scene>
```

The rendering loop is controlled by the renderer (message 1). On each loop, an update is made of frame parameters (message 2). This update contains the elapsed time in our example. The renderer then sends an update to the Scene Graph Adapter (message 3) which informs every *format wrappers* (messages 4 to 5). The time update is transmitted to the Format Decoder (message 6) which is in charge of informing every time listener in the *format scene graph* such as TimeSensors. The *format wrapper* is informed of the TimeSensor update (message 7) and updates all its listeners (message 8). In our sample code, the route node sets the sphere transparency as a listener of TimeSensor fraction changes. The transparency field of the material node is updated with the new value and this update is transmitted to the renderer (messages 9 to 10).

Note that if the application needs to import another format, no changes or adaptations need to be done on the rendering part of the architecture. Nevertheless, changing the rendering engine impacts the application. Indeed the application is in charge of configuring the rendering engine and of the loading of its assets: the application controls rendering parameters such as the size of the rendering window for example.

5 Implementation and results

In this section, we will present the realized implementation of our architecture.

5.1 Implementation

We have realized an implementation of our architecture in C++ to prove the concept of our solution. We try to design an as generic as possible API to fit every 3D format as well as every engine scene graph. We choose Ogre 3D [7] as a rendering engine because it is open-source, it has a well-documented API and a large user community. Then we use our architecture to load X3D and Collada files in Ogre. We choose X3D because it is the most complete royalty-free open standard; it includes indeed many features that other formats only partially proposed. As for Collada, it seems to us to be a good example since it is also royalty-free and widely adopted since its release.

In order to realize the X3D Wrapper, our first task was to choose the tools that we wish to use in the X3D decoder. We use CyberX3D for C++ [8] as the parser and the decoder. The X3D Wrapper relies on CyberX3D's scene graph representation. It uses this representation to traverse the scene graph and call the appropriate methods from the Renderer Adapter API. Besides, in order to be able to render animations described in a X3D file, we need an event library. CyberX3D does not propose an event handler, that is why we realize this part in the *format wrapper*. We chose Boost Signals 2 [9] to implement this task. During the parsing of the *format scene graph*, we create new signals instances for every sensor and interpolator in the graph. Later, when route nodes are parsed, listeners are attached to the previously created signals. Let us consider this sample route to clarify this process:

```
<ROUTE fromNode='TIMER' fromField='
     fraction_changed' toNode='MAT' toField='
     transparency' />
```

When this route is parsed, a listener is attached to the TIMER signals. Then, at runtime, when a time update is emitted by the renderer, it is retrieved by the X3D wrapper which uses the X3D

[7]http://www.ogre3d.org/
[8]http://www.cybergarage.org/twiki/bin/view/Main/CyberX3DForCC
[9]http://www.boost.org/

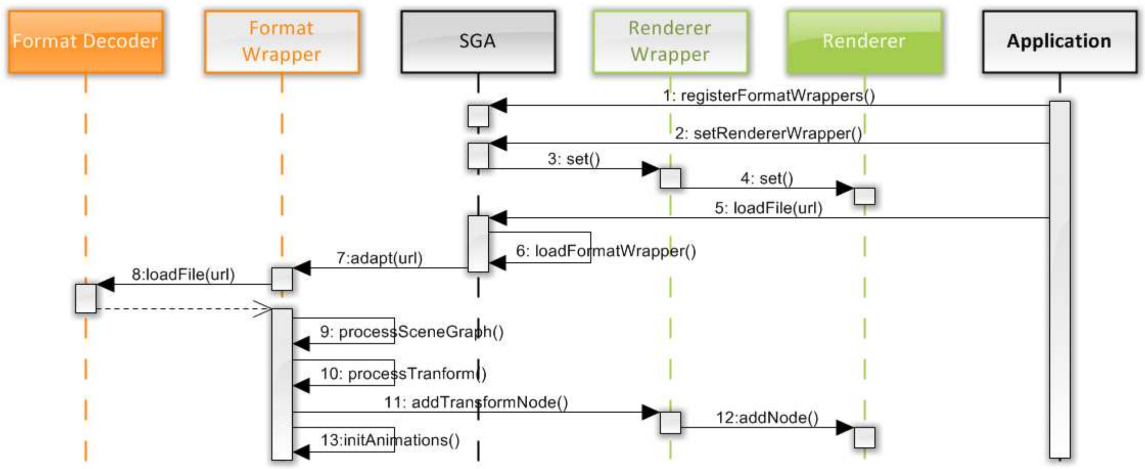


Figure 5: *Sample sequences of static rendering through the Scene Graph Adapter (SGA) architecture.*

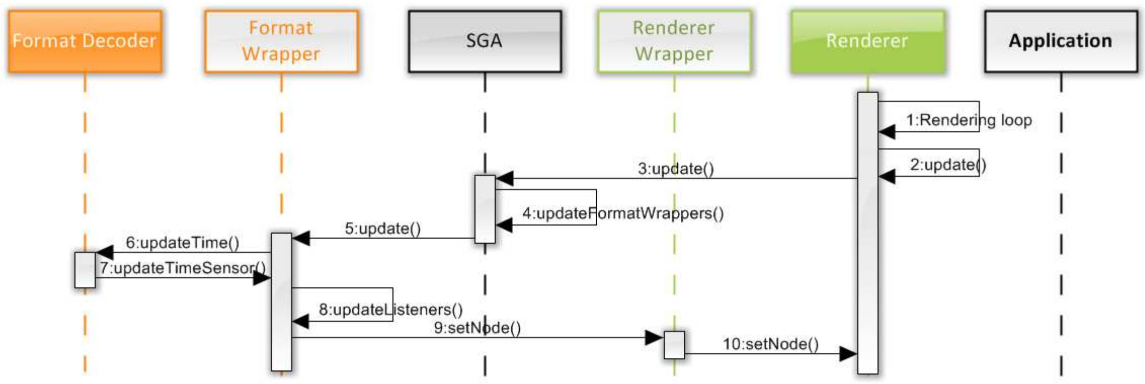


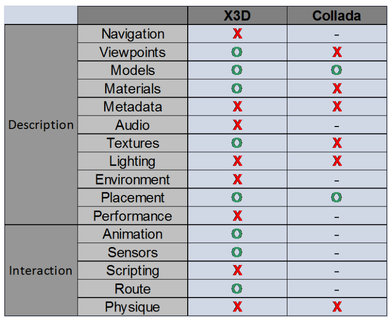
Figure 6: *Sample sequences of dynamic rendering through the Scene Graph Adapter (SGA) architecture.*

Decoder to decode a time update result. The time update triggers an update of all TimeSensors in the *format scene graph*. Then, as defined in the route node, the transparency field of the material node is updated. This is done thanks to the listener attached to the Timer node. An illustration of this case is shown in figure 6.

To create the Collada Wrapper we use FCollada[10] as a parser. The Collada wrapper uses FCollada scene graph representation to render Collada files using the Renderer Adapter API methods.

Creating the Ogre Wrapper was a simple task. We use Ogre API to implement all methods of the Renderer Adapter API. We have developed a sample application to test our architecture. The application is in charge of setting Ogre configuration and of sending input files url to the Scene Graph Adapter. The Scene Graph Adapter loads the appropriate *format wrapper* for each files. *Format wrappers* exchange data with the Ogre Wrapper to render the content of their input file. Figure 7 illustrates the implementation and figure 8 gives wrappers implementation details. In figure 9 we show the implementation state of the two wrappers.

[10]http://www.feelingsoftware.com/3D-collada-tools/collada-tools.html



		X3D	Collada
Description	Navigation	X	-
	Viewpoints	O	X
	Models	O	O
	Materials	O	X
	Metadata	X	X
	Audio	X	-
	Textures	O	X
	Lighting	X	X
	Environment	X	-
	Placement	O	O
	Performance	X	-
Interaction	Animation	O	-
	Sensors	O	-
	Scripting	X	-
	Route	O	-
	Physique	X	X

Figure 9: *Features implemented in the X3D wrapper and the Collada wrapper. The X symbol indicates that this feature has not been implemented yet, the O symbol shows features already available and the - symbol stands for features not available in the format.*

5.2 Generalization

At this state of development we can make new *format wrappers* to load other formats in Ogre without modifying the Ogre Wrapper. Similarly we can use the available *format wrappers* (X3D and Collada) in another application based on another rendering engine

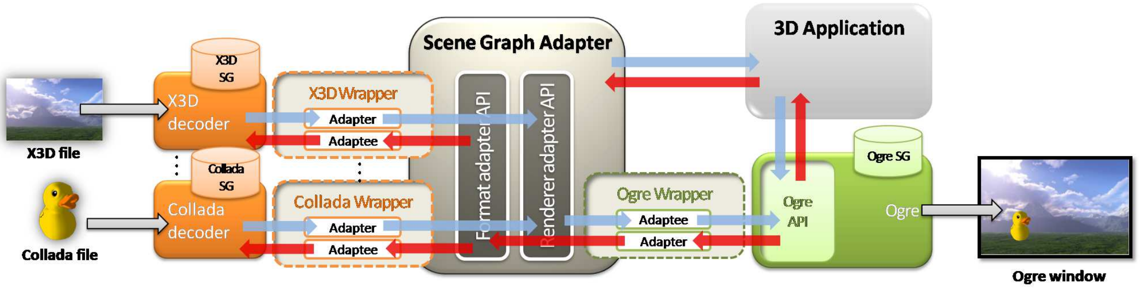
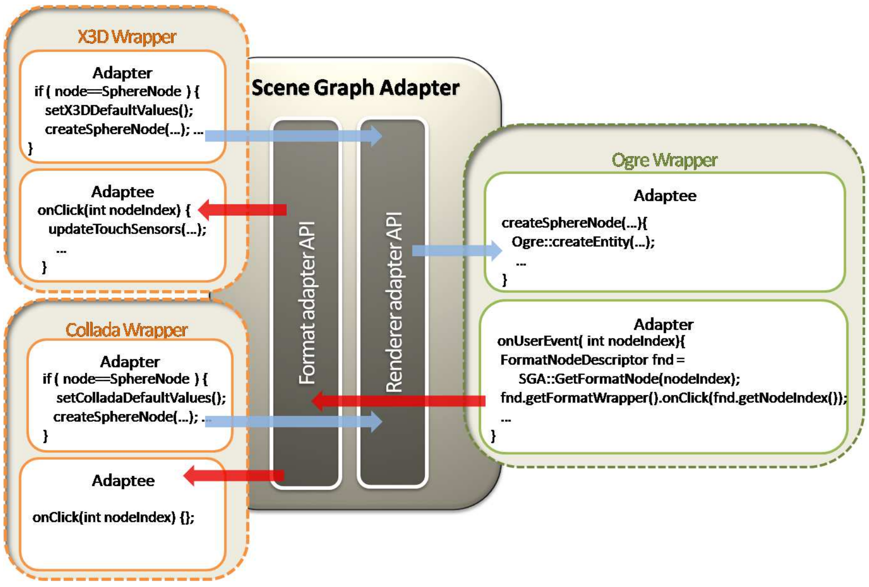


Figure 7: *An implementation of the Scene Graph Adapter that loads X3D and Collada files and render them in Ogre rendering engine.*



```
Adapter
if ( node==SphereNode ) {
  setX3DDefaultValues();
  createSphereNode(...); ...
}
```

```
Adaptee
onClick(int nodeIndex) {
  updateTouchSensors(...);
  ...
}
```

```
Adapter
if ( node==SphereNode ) {
  setColladaDefaultValues();
  createSphereNode(...); ...
}
```

```
Adaptee

onClick(int nodeIndex) {};
```

```
Adaptee
createSphereNode(...){
  Ogre::createEntity(...);
  ...
}
```

```
Adapter
onUserEvent( int nodeIndex){
FormatNodeDescriptor fnd =
    SGA::GetFormatNode(nodeIndex);
fnd.getFormatWrapper().onClick(fnd.getNodeIndex());
...
}
```

Figure 8: *Implementation details of the wrappers.*

providing that firstly the application use the Scene Graph Adapter and that secondly an *engine wrapper* has been implemented for the rendering engine. It is also possible to make another X3D Wrapper that uses another parser, for example we can use Xiot[11] instead of CyberX3D. One more possibility is to change the wrapper implementation. As explained in section 4.3 the Scene Graph Adapter provides two exchange modes between format wrappers and renderer wrappers. Indeed, we can take advantage of the rendering engine capacities and transmit more information to the *renderer scene graph*. We can for example delegate LOD management to the rendering engine and change the X3D wrapper implementation in order to send all levels of detail to the *renderer wrapper*.
We are also able to be compliant with new standards like WebGL. Several WebGL API already exist (SpiderGL[12], GLGE[13], CopperLicht[14], etc.) on which a WebGL Wrapper can rely. The main difficulty consists in the C++ to Javascript communication, WebGL and all the aforementioned API are indeed javascript based. This point needs to be investigated.

[11]`http://forge.collaviz.org/community/xiot`
[12]`http://www.spidergl.org/`
[13]`http://www.glge.org/`
[14]`http://www.ambiera.com/copperlicht/`

5.3 Discussion

The key feature of the presented approach consists in enabling the rendering of any scene-graph-based 3D format in most of available rendering engines. We choose this approach in order to fulfill the interoperability issue among 3D formats without transcoding. However one drawback of this approach is the need of a wrapper instance for each input file. It increases the amount of memory used by an application based on the Scene Graph Adapter. A Format Wrapper indeed keeps an internal representation of the *format scene graph* at runtime. Nevertheless we have tested our sample application with two running modes:

1. a normal mode; i.e keeping internal representation as required by the Scene Graph Adapter architecture.

2. a lightweight mode; i.e deleting all *format wrappers* instances before rendering

We estimate the amount of memory used by the normal mode (1) against the lightweight mode (2) of +0,5% for Collada files and of +1,2% for X3D files. We use X3D files and Collada files of equivalent size to make this estimation. We give a result for each file type because the memory use of a *format wrapper* depends on the parser used in the Format Decoder; it is indeed this component that creates the internal scene graph data structure used by *format wrappers*. Besides we need to extend our implementation to fully evaluate our architecture.

6 Conclusion and Future Work

In this paper we have presented an approach to enable the loading of 3D files of different 3D formats in a single viewer using any available rendering engine. Our solution aims at developing components that can be reused in several applications and that can be mixed in order to fit any 3D application requirements. Our goal is to propose a solution to the current multiplicity of 3D formats as well as rendering possibilities and among them the WebGL standard. Indeed we think that the future of 3D on the web does not only depend on standards establishment but also in reusing existing contents and in increasing access to these contents. The Scene Graph Adapter tries to solve this problem.
We plan to extend our architecture to other engines and mainly to a physics engine. We also want to investigate a WebGL-based rendering to make our solution compliant with this standard. On the input part of our architecture, we would like to enable inlines of files inside files of different formats. For example, create a X3D file with an inline node that linked a Collada file and being able to render it. Future work also includes handling interactions between files of different formats. Indeed it would be interesting to describe behaviors with X3D interpolators for objects encoded in Collada.

References

3D TECHNOLOGIES R&D. 2009. 3DMLW. http://www.3dmlw.com.

BEHR, J., ESCHLER, P., JUNG, Y., AND ZÖLLNER, M. 2009. X3dom: a dom-based html5/x3d integration model. In *Proceedings of the 14th International Conference on 3D Web Technology*, 127–135.

BEHR, J., JUNG, Y., KEIL, J., DREVENSEK, T., ZOELLNER, M., ESCHLER, P., AND FELLNER, D. 2010. A scalable architecture for the html5/x3d integration model x3dom. In *Proceedings of the 15th International Conference on Web 3D Technology*, 185–194.

BELL, J., DINOVA, M., AND LEVINE, D. 2010. Vwrap for virtual worlds interoperability. *IEEE Internet Computing 14*, 1, 73–77.

DÖLLNER, J., AND HINRICHS, K. 2002. A generic rendering system. *IEEE Transactions on Visualization and Computer Graphics*, 99–118.

FANG, Z.-C., AND CAI, H. 2009. Building interoperable 3d virtual world platforms with restful web services. In *Proceedings of the 2009 Congress on Services - I*, IEEE Computer Society, 70–77.

GAMMA E., HELM R., JOHNSON R., AND VLISSIDES J. 1995. *Design patterns: Elements of reusable Object-Oriented Software*. Addison-Wesley.

GELISSEN, J. H. 2008. Iso/iec jtc 1/sc 29/wg 11/n9901.

HASLHOFER, B., AND KLAS, W. 2010. A survey of techniques for achieving metadata interoperability. *ACM Comput. Surv. 42*, 2, 1–37.

HINRICHS, J. D. 2000. A generalized scene graph. *Vision, modeling, and visualization 2000: proceedings: November 22-24, 2000, Saarbrücken, Germany*, 247.

HU, S.-Y., AND JIANG, J.-R. 2008. Plug: Virtual worlds for millions of people. In *Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, IEEE Computer Society, 787–792.

KHRONOS GROUP, 2011. WebGL. http://www.khronos.org/webgl/.

LEA, R., MATSUDA, K., AND MIYASHITA, K. 1996. *Java for 3D and VRML Worlds*. New Riders Publishing Thousand Oaks, CA, USA.

ORTIZ JR., S. 2010. Is 3D Finally Ready for the Web ? *Computer 43*, 1, 14–16.

PESCE, M. 1994. Mark pesce, tony parisi – vrml / vrml 97 / x3d 10th anniversary – 3d test panorama of web 3d technologies – real time 3d and interactive media.

POLYS, N. F., BRUTZMAN, D., STEED, A., AND BEHR, J. 2008. Future standards for immersive vr: Report on the ieee virtual reality 2007 workshop. *IEEE Comput. Graph. Appl. 28*, 2, 94–99.

SONS, K., KLEIN, F., RUBINSTEIN, D., BYELOZYOROV, S., AND SLUSALLEK, P. 2010. Xml3d: interactive 3d graphics for the web. In *Proceedings of the 15th International Conference on Web 3D Technology*, ACM, 175–184.

SOTO, M., AND ALLONGUE, S. 2002. Modeling methods for reusable and interoperable virtual entities in multimedia virtual worlds. *Multimedia Tools and Applications 16*, 1 (Jan), 161–177.

STANDARD ECMA-363. 2007. Universal3D. http://www.ecma-international.org/publications/standards/Ecma-363.htm.

STEINICKE, F., ROPINSKI, T., AND HINRICHS, K. 2005. A generic virtual reality software system's architecture and application. In *Proceedings of the 2005 international conference on Augmented tele-existence*, ACM, ICAT '05, 220–227.

STRAUSS, P. S., AND CAREY, R. 1992. An object-oriented 3d graphics toolkit. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, ACM, vol. 26, ACM, 341–349.