Late Breaking Results: Enabling interoperability between 3D formats through a generic architecture

Rozenn Bouville Berthelot* Orange Labs and IRISA, Rennes, France

Jérôme Royan† Orange Labs France

Thierry Duval‡ IRISA, Rennes, France

Bruno Arnaldi§ IRISA, Rennes, France

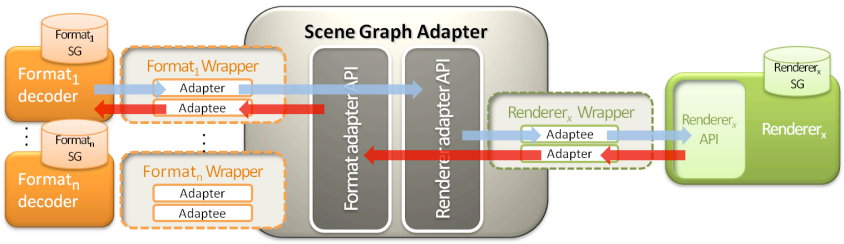

Figure 1: Our architecture allows the loading of any 3D graphics format simultaneously in any available rendering engine. The scene graph adapter is an interface that adapts a scene graph (SG) of a given format into a renderer scene graph and which also allows the rendering part to request this scene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

ABSTRACT

The ever growing number of 3D formats and rendering engines inevitably leads to compatibility problems, which makes difficult the use of any 3D format into a rendering engine and nearly impossible the interoperability between them. We thus propose a generic architecture able to decode and mix 3D formats whatever the rendering engine used by Virtual Reality platforms. This architecture is based on a scene graph adapter which relies on two APIs to handle each 3D format, wrap each rendering engine and allow them to communicate. Thanks to this flexible architecture, format and renderer wrappers are both reusable.

Index Terms: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual reality; D.2.12 [SoftwareEngineering]: Interoperability—Data mapping; D.2.13 [SoftwareEngineering]: Reusable Software—Reusable libraires

1 INTRODUCTION

Interoperability in computer science has always been a challenging problem. In this paper we will focus on the problem of 3D content interoperability. As stressed by Polys in [4], interoperability is a "crucial requirement for success" of Virtual Reality (VR) systems. Creating 3D content is indeed expensive and time-consuming and slows the widespread of VR. However there exist more than 50 3D graphics formats (including proprietary formats) and although some of them are more or less abandoned, new ones continue to appear such as XML3D [5] which has been released this year. These formats are of different types and we propose to classify them as: a) 3D modeller formats (3ds Max, Maya, ...) b) 3D renderer formats (Ogre mesh, Unity, ...) c) exchange formats (Autodesk's fbx, Collada, ...) d) scene description formats (VRML, X3D, ...).

In the following we will focus on the problem of generating a 3D scene composed of several 3D contents defined in different 3D formats. Our goal is to propose an architecture that can perform this task without transcoding these contents whatever their formats and the targeted viewer (for example loading Collada and X3D objects in the same viewer directly).

As it is noticed in [2], there exists so far no other solution to this problem than to resort to format conversion. Each rendering engine supports a limited amount of input formats. Contents that are not supported must be converted but this process often leads to a loss of functionalities. Actually the reason why new formats are still emerging is to bring new features that do not exist in other formats, which means that content degradation must be avoided as much as possible. Mixing several 3D formats allows functionality mixing e.g. loading a Collada object with physics properties in an X3D scene to examine it using X3D's navigation and interaction functionalities. The possibility to mix 3D formats also allows a more efficient collaborative work. Research teams can share their resources without being hampered by compatibility problems.

On the other hand we observe that there are many similarities in 3D formats. Most of them are based on a scene graph data structure. They also use similar ways to model objects and to organise data in the scene graph (similar shapes description with a separation between geometry and appearance, similar modeling transformation, similar grouping node, similar primitive shapes, ...). Likewise we observe that most rendering engines also use scene graph representations even if it is aimed at different usages and even if they have different functionalities.

These observations lead us to the conclusion that it must be possible to describe a generic scene graph interface allowing cooperation between a given 3D format scene graph and the renderer scene graph.

2 THE GENERIC ARCHITECTURE WE PROPOSE

We propose a scene graph adapter interface to be used in a flexible architecture that allows simultaneous rendering of multiple 3D formats with any rendering engine. This architecture is to be used in an application that will not be described in this paper. As shown in figure 1, our architecture is composed of 4 components: a) format decoders, b) *format wrappers*, c) the *renderer wrapper*, d) the rendering engine.

The format decoders parse input files of a given format and also manage scene graph updates as well as format specific processes. They are specific to each 3D format and can integrate any useful

*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

20th International Conference on Artificial Reality and Telexistence (ICAT2010), 1-3 December 2010, Adelaide, Australia
ISBN: 978-4-904490-03-7 C3450 ©2010 VRSJ

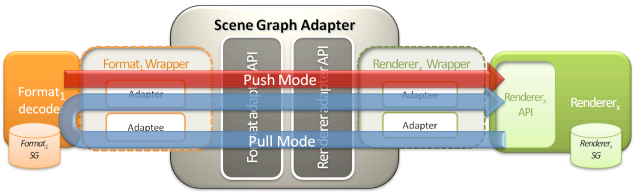

Figure 2: The 2 modes for accessing scene graphs information: a pull mode and a push mode.

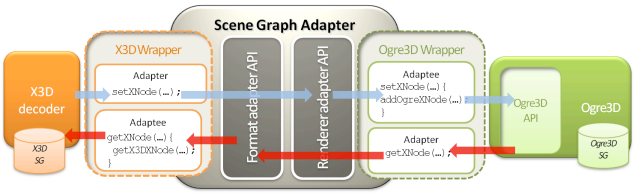

Figure 3: An implementation example with X3D as an input format and Ogre3D as the rendering engine. This use case requires the implementation of 2 elements: the *X3D wrapper* and the *Ogre3D wrapper*.

format tools. Each one keeps an internal representation of the scene graph, we call this representation "format scene graph".
The rendering engine can be any rendering engine available (Ogre3D, Unity, Open Scene Graph, ...). It keeps its own scene graph that we call "renderer scene graph".
Wrappers part is to convert a format scene graph to a renderer scene graph using the scene graph adapter. To achieve this, the scene graph adapter provides 2 APIs: a format adapter API and a renderer format API. The *format wrappers* and the *renderer wrapper* must also achieve 2 differents tasks: an adapter task and an adaptee task. We use those terms with reference to the adapter pattern GoF 139 [1] which concept is close to our architecture. The adapter task consists in calling methods of the scene graph adapter. The adaptee task consists in implementing methods of the scene graph adapter using an external API. More precisely, during the format scene graph evaluation process, the format wrapper adapter calls methods from the renderer adapter API. The implementation of those methods is then delegated to the renderer wrapper adaptee using the rendering engine API. On the rendering side of the architecture, the renderer wrapper adapter relays request from the rendering engine using methods of the format adapter API and those methods are implemented in the format wrapper adaptee. The *format wrappers* and the *renderer wrapper* interchange scene graph nodes information from an adapter to an adaptee.
This architecture is highly flexible since: (a) we can import any 3D format providing that we have developed the appropriate *format wrapper*, (b) we can use any rendering engine providing that we have developed the appropriate *renderer wrapper*, (c) *format wrappers* and *renderer wrappers* can be reused and combined at will. The design of the scene graph adapter is based on the observed similarities in 3D formats and 3D rendering scene graph. The basic components of a scene graph were first described in [6]. Several authors have proposed a generalized scene graph structure (as in [3] for example) with a view to improving the rendering process. We have used these previous works to design our interface; it provides methods to add, remove, update and interrogate nodes. Therefore our interface also provides methods to manage an index for each node.

Since almost all 3D graphics formats share the possibility to modify the scene (e.g. switch node, scripts or level of detail), the scene graph adapter provides 2 exchange modes as illustrated in 2: a push mode and a pull mode. In push mode, the whole format scene graph is sent to the *renderer wrapper* which decodes 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 selection. 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 information are kept by the format scene graph and the rendering engine is only aware of the LOD that is currently rendered. On a change of level, the rendering engine requests the appropriate node of the scene graph format.
Through this architecture we can for example render an X3D file in Ogre 3D (see figure 3). We first chose an X3D parser to use it in the X3D decoder (e.g. CyberX3D[1]). Then we implemented the *X3D wrapper* that meets the given format scene graph specifications and uses the call methods from the renderer adapter API. Finally we implemented the *Ogre3D wrapper* using Ogre3D API. We can extend our implementation with other formats by developing other *format wrappers* while keeping the rendering part of our application. We can also change the rendering engine by developing a new *renderer wrapper* that complies with the scene graph adapter interface and reuse the format wrappers.

3 Conclusion

We have proposed a generic architecture that allows combining different 3D objects described in different 3D formats in a unique rendering window. Moreover we do not depend on a given rendering engine. Our architecture is highly flexible as it fits to any 3D format and any rendering engine based on scene graphs. It can be extended at will to meet the requirements of future 3D formats and rendering engines. Our approach has several benefits:

1) it supports every 3D formats and their functionalities without any rendering restrictions,

2) it works with any rendering engine without any format restrictions,

3) it makes it possible to reuse and mix these components as required by the application.

Up to now we have worked on the proof of concept of this architecture. We have achieved a first implementation dealing with X3D and Collada and using the Ogre3D rendering engine. Then we plan to improve our architecture by allowing 3D objects that are described in different formats to interact together. We also consider extending our concept with a physics engine and a collaborative engine in order to apply our architecture to virtual worlds.

References

[1] Gamma E., Helm R., Johnson R., and Vlissides J. *Design patterns: Elements of reusable Object-Oriented Software*.

[2] S. Havemann and D. W. Fellner. Seven research challenges of generalized 3d documents. *Computer Graphics and Applications, IEEE*, 27(3):70–76, 2007.

[3] J. D. Hinrichs. A generalized scene graph. *Vision, modeling, and visualization 2000: proceedings: November 22-24, 2000, Saarbrücken, Germany*, page 247, 2000.

[4] N. F. Polys, D. Brutzman, A. Steed, and J. Behr. Future standards for immersive vr: Report on the ieee virtual reality 2007 workshop. *IEEE Comput. Graph. Appl.*, 28(2):94–99, 2008.

[5] K. Sons, F. Klein, D. Rubinstein, S. Byelozyorov, and P. Slusallek. XML3D: interactive 3D graphics for the web. In *Proceedings of the 15th International Conference on Web 3D Technology*, pages 175–184. ACM, 2010.

[6] P. Strauss and R. Carey. An object-oriented 3D graphics toolkit. *ACM SIGGRAPH Computer Graphics*, 26(2):341–349, 1992.

[1] http://www.cybergarage.org/twiki/bin/view/Main/CyberX3DForCC