Scene Conversion for Physically-based Renderers

Luiza Hagemann
Instituto de Informática - UFRGS
Porto Alegre, RS, Brazil
lahagemann@inf.ufrgs.br

Manuel M. Oliveira
Instituto de Informática - UFRGS
Porto Alegre, RS, Brazil
oliveira@inf.ufrgs.br

***Abstract*—The abstract goes here.**

I. Introduction

Ever since the development of modern Computer Graphics, one of the goals researchers aspired to was being able to synthesize images indistinguishable from real photographs. In order to produce physically accurate images, the process of image synthesis - also called **rendering** - simmulates the interaction of light with the representation of a three-dimensional scene.

Physically-Based Rendering (PBR) is a complex process that requires thorough knowledge of optics, material properties, geometry and light propagation.

A. Physically Based Rendering (PBR)

Over the years, PBR became quite popular and was widely incorporated into the entertainment industries. From movies to videogames, from ads to interior design, PBR made it possible for artists to bring their creations and their vision one step closer to reality. Today, we can say that many - if not most - algorithms used in computer animation, geometric modeling and texturing require that their results be passed through some sort of rendering process.

As PBR popularity grew, a brand new market opened up for physically-based (PB) renderers. Following the creation of *PBRT* [1] and the publishing of *"Physically Based Rendering: From Theory to Implementation"* [2], several other research-oriented renderers were created. Among them is *Mitsuba* [3], one of the renderers chosen for this research, which places strong emphasis on experimental rendering techniques.

Following the lead of Pixar's *Renderman*, many commercial and performance-oriented renderers appeared on the market. Focused on animation techniques and visual effects for movies, these renderers provide well-established, stable rendering techniques. These renderers, such as *LuxRender* [4] and *Octane*, are state-of-the-art renderers used by the animation and gaming industries.

Even with different applications, the vast majority of modern PB renderers follows the same general guidelines for defining scene directives and world descriptions. Scene directives establish parameters such as which integration and sampling techniques the renderer must use, the view matrix and other camera properties. World descriptions state which objects compose the scene and which materials must be used to render them. This ensemble of descriptions is what, in PBR, is called a **scene**.

B. Rendering a Scene

Starting with *PBRT*, most PB renderers have used a similar, traditional structure to describe scenes.

These scenes are usually created by a 3D artist, who will use a modeling software (such as Blender, 3D Max or Maya) to draw the objects, choose their materials and then create the object files. These files will then be instanced in the scene file and interpreted by the renderer.

But even with an artists expertise, creating scenes is still a complex process. For instance, scenes created for building overviews and interior design often compile hundreds of 3D models and dozens of customized materials and textures. Each material and texture has to be carefully defined, taking into account the renderer's limitations and particularities.

After a scene is created and rendered, all the hard work invested by the artist is stored, waiting for a possible future use. However, should the artist choose to change renderers, the scene file they created so diligently would have to be rewritten and/or heavily modified.

Converting a scene file from one renderer format to another is very difficult and time consuming. Aside from adapting material and light properties - which can be hard since sometimes renderers don't provide the same features -, the 3D object formats supported may not be the same. For instance, *Mitsuba* supports the Object File Wavefront 3D (.obj) format while *PBRT* does not.

Creating free resources in order to help rendering research, Benedikt Bitterli [5] manually converted 32 scenes for his renderer *Tungsten* [6], later converting them to *PBRT* and *Mitsuba*. According to him, *"this process is time intensive (up to several days per scene)"*.

II. Related Works

III. System Architecture

Our converting pipeline is subdivided into three main states: the **Import Module**, the **Canonical Scene Representation** and the **Conversion Module**, as illustrated in Figure 1. Given an arbitrary input file format, our converter is able to import the scene and transform it into a generic, canonical representation and then export it to different output formats.

Our Proof of Concept encompassed *PBRT* [1], *Mitsuba* [3] and *LuxRender* [4], as these are three of the most popularly used renderers in the community.

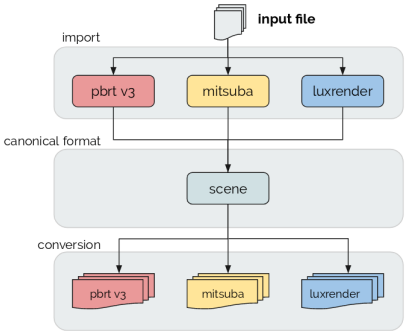

Fig. 1. Illustration of the system pipeline.

A. Import Module

Most physically-based renderers have a similar way of describing a scene. Usually, they divide a scene into two sections: scene-wide rendering options and world block. The former defines overall rendering settings (such as which rendering or sampling technique should be used) while the latter describes the geometry and which materials should be used for rendering.

Our import module specializes in reading and interpreting such scene files. The input file is read, parsed and each directive is loaded into our canonical scene representation. Since each renderer has its own proprietary file format, we have three importing modules: one for each renderer.

PBRT and *LuxRender* file formats are composed of structured text statements defining all scene directives. Given their structure, a Lex/Yacc parser was considered the best choice for these formats. As we intended to keep our system in pure Python, we chose to use PLY [7], a Python implementation of Lex and Yacc.

Mitsuba's file format consists of a XML file. Since there are several XML-parsing libraries for Python that can load the hierarchy into a tree data structure, we didn't think it necessary to create a Lex/Yacc parser. We chose to implement this module using ElementTree [**?**], a XML parsing tool.

B. Canonical Scene Representation

After loading the scene file, the information obtained from them has to be stored somewhere. While most renderers have the same base structure, they differ in which parameters can be used to configure the techniques used during the rendering process.

Renderer directives are usually given in the format of a command, followed by a type and a list of additional parameters. So, for instance, to specify the path integration technique with 8192 samples per pixel in *PBRT* one would write the following directive: *Integrator "path" "integer pixelsamples" [8192]*.

In order to establish a common ground for conversion, we defined a canonical scene representation. This representation can be easily extended incorporate any directives not contemplated in this work.

In this representation, we divide the scene into **scene-wide rendering options** and **world block**. The rendering options are divided into integration technique and sensor options, while the world block is divided into lists of shapes, global emitters and material definitions. This structure is illustrated in Figure 2.

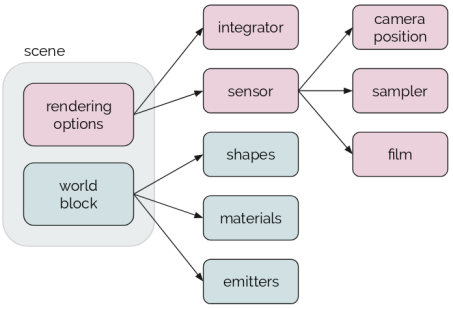

Fig. 2. Illustration of the canonical scene representation.

1) Scene-wide Rendering Options: A set of directives specifying the integration and sampling techniques used for rendering, camera and film properties. These directives are represented in a structure with two fields: a type and a list of parameters.

2) World Block: A set of directives describing the shapes, materials and global emitters present in the scene.

The **shape** directive is represented in a structure with: a type (cube, sphere, ...), an optional area emitter, an optional material reference, an optional transformation matrix and a list of parameters.

The **material** directive is represented in a structure with: a type, an id, an optional texture and a list of parameters.

The **texture** directive is represented in a structure with: a type, an id and a list of parameters.

The **global emitter** directive is represented in a structure with: a type, an optional transformation matrix and a list of parameters.

C. Conversion Module

Subsection text here.

IV. Results and Analysis

Results section.

V. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

REFERENCES

[1] M. Pharr and G. Humphreys, "Pbrt, version 3," 2015. [Online]. Available: https://github.com/mmp/pbrt-v3

[2] ——, *Physically Based Rendering: From Theory to Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[3] W. Jakob, "Mitsuba: Physically based renderer," 2014, http://www.mitsuba-renderer.org/.

[4] J.-P. Grimaldi and T. Vergauwen, "Luxrender v1.6," 2008, http://www.luxrender.net/.

[5] B. Bitterli, "Rendering resources," 2016, https://benedikt-bitterli.me/resources/.

[6] ——, "The tungsten renderer," 2014, https://benedikt-bitterli.me/tungsten.html.

[7] D. Beazley, "Ply (python lex-yacc)," 2001. [Online]. Available: http://www.dabeaz.com/ply/