pbrt : a Tutorial

Luís Paulo Santos

Departamento de Informática

Universidade do Minho

Janeiro, 2008

Abstract

This document is a hands-on tutorial for using pbrt (PHYSICALLY BASED RAY TRACER) and developing plug-ins for it. Some basic examples are used to illustrate how to achieve this.

This tutorial is neither an in-depth manual of pbrt nor a discussion of the concepts behind plug-ins or pluggable components. The reader is referred to the pbrt book (Pharr & Humphreys, 2004) for further details.

Introduction

pbrt is a ray tracing system with very solid theoretical foundations. It is thus used to illustrate ray tracing and global illumination algorithms throughout this course.

The key characteristic of pbrt architecture is that it is organized as a ray tracing core and a set of components. The core coordinates interactions among components, while the components are responsible for all remaining tasks, such as shooting rays, space traversal, objects intersections, integration, image generation, etc. These components are separate object files, loaded by the core at run time – they are referred to as plug-ins. pbrt can thus be easily extended by writing new plug-ins for the various functional blocks. The core imposes a strict API and enforces a strict interface protocol, thus helping ensuring a clean component design.

The pbrt executable consists of the core code that drives the system's main flow of control, but contains no code related to specific elements, such as objects, light sources or cameras. These are provided as separate components, loaded by the core at run time according to the scene being rendered. There are 13 different types of components, or plug-ins, briefly described by the following table. By writing new plug-ins the renderer can be extended in a straight forward manner.

Type	Description
Shape	
Primitive	
Camera	
Sampler	
Filter	
Film	
ToneMap	

Material	
Texture	
VolumeRegion	
Light	
SurfaceIntegrator	
VolumeIntegrator	

Table 1 - pbrt plug-in types

Tutorial 1 – Rendering and visualization

To follow this small, first tutorial you will need access to pbrt, exrdisplay and exrtotiff. Make sure these are installed in your system. You also need the the cornell box scene and a few High Dynamic Range (HDR) images in .exr format.

Rendering

The scene description is given in a .pbrt file. This includes geometry, materials' properties, light positions and radiant power, etc.

Open a shell, change to the scenes directory and render the cornell box by writing:

```
> pbrt cornell.pbrt
```

You can now visualize the image by writing:

```
> exrdisplay cornell.exr
```

pbrt is a physically based renderer, thus it generates High Dynamic Range images. In practice the pixels' values stored on the output file have floating point values that can range from 0 to any maximum number. To visualize the image using exrdisplay you may need to adjust exposure, which you can do using the top slider on the view window.

The quality of the rendered image depends on many parameters and on the particular choice of algorithms used. The noise you see on the image depends, among others, on the number of samples, i.e. rays, taken per pixel. Increase this number to 16 by changing the "pixelsamples" parameter on the Sampler component as follows:

```
Sampler "bestcandidate" "integer pixelsamples" [16]
```

Render and visualize the new image. You might notice that noise was reduced but, unfortunately, rendering time increased a lot.

If you look carefully to the resulting image you will notice that some things are wrong. For instance, the ceiling is not illuminated by the lamp and the mirror does not project a reflection of the light into the floor. Global illumination is all about selecting and simulating the most relevant paths followed by light. The algorithm used to select these light paths is referred to as the "surface integrator". The integrator used on the previous runs only selects direct illumination paths (from the light source to the objects) and specular paths (from specular surfaces, such as mirrors and glass, to the observer). Other integrators will select additional light paths, but may take longer to execute.

Edit the cornell.pbrt file such that it uses the "path" surface integrator and only shoots 4 primary rays per pixel:

```
Sampler "bestcandidate" "integer pixelsamples" [4]
SurfaceIntegrator "path"
```

Render the image again and visualize it. The first thing you will notice is that the image is full of noise. In fact, path tracing stochastically selects which light paths to trace, does the final result will have variance which human observers perceive as noise. Variance, or noise, can be reduced by increasing the number of samples taken. Unfortunately, rendering time increases linearly with the number of samples but variance reduces with the square of the number of paths. In practice, you'll have to take 4 times more samples to have variance. Do so by changing the Sampler parameter:

```
Sampler "bestcandidate" "integer pixelsamples" [16]
```

Even though noise is still too disturbing you might notice that the ceiling is being lighten by the light source and the mirror is projecting a reflection of the light source onto the floor (this is referred to as a caustic).

To get a "good" image many samples per pixel are required. Path tracing will simulate all light transport phenomena but takes too much time. The irradiance cache is a faster method to simulate diffuse interreflections – this is why the ceiling is illuminated: light from the light source is reflected from the walls to the ceiling and then to the view point. Change the surface integrator by writing:

```
Sampler "bestcandidate" "integer pixelsamples" [4]
SurfaceIntegrator "irradiancecache"
```

Visualization and Tone Mapping

The EXR image is not appropriate for visualization on current displays. The range of luminance values present on these images has no formal maximum value, while displays usually can only display luminances ranging from 0.1 to 100 candelas/m^2.

These high dynamic range images are either visualized using special programs, such as exrtotiff, where you can set the exposure, or they have to be converted to Low Dynamic Range Images. Doing so is the role of tone mapping algorithms.

Tone mapping algorithms strive to maintain local contrast rather than apparent brightness, and usually do so by resorting to models of the Human Visual System. These algorithms can be either global or local. Global algorithms calculate a single mapping factor that is then applied to all pixels, whereas local algorithms calculate a different mapping factor for each pixel based on information about each pixel neighborhood.

pbrt distribution includes 4 different tone mapping components:

- **contrast** – tries to maximize contrast using the notion of Just Noticeable Differences. Is a global operator. Receives as parameter the display adaptation level: "displayadaptationY", with a default value of 50;

- **maxwhite** – global operator that sets the maximum luminance on the image to the display maximum luminance and than scales all pixels by the same factor;
- **highcontrast** – local operator that maximizes contrast on a given neighborhood;
- **nonlinear** – local algorithm that uses an empirical operator based on a S-shaped curve. Accepts the world adaptation level as a parameter (maxY), with a default value of 0.0

Try these operators by using exrtotiff and displaying both the original HDR image (exrdisplay) and the resulting LDR tiff image.

Examples:

```
> exrtotiff –tonemap contrast –param “displayadaptationY” 5.0 StillLife.exr StillLife.tif
> exrtotiff –tonemap maxwhite StillLife.exr StillLife.tif
> exrtotiff –tonemap highcontrast StillLife.exr StillLife.tif
> exrtotiff –tonemap nonlinear –param “maxY” 1.0 StillLife.exr StillLife.tif
```