pbrt v3 – Scene Description Language

Luís Paulo Santos – Dep. de Informática

Universidade do Minho

March, 2022

`pbrt` receives as input a text file that describes both the rendering setup and the virtual scene (or world).

Details on the `pbrt` scene description language can be found on https://www.pbrt.org/fileformat-v3. Here we will build a simple scene, known as the Cornell Box, in order to get familiarized with this format. Open a text file named `cornell.pbrt` and follow the steps below.

The scene description file has two different sections. The first one describes the rendering setup, whereas the second describes the virtual scene(referred to as world in the `pbrt` terminology), including, geometry, appearance model and lighting.

Rendering Setup

This includes a series of directives that describe the camera, film, sampling and light transport algorithms to use in rendering the scene.

First let us define the viewing and camera parameters.

The `LookAt` command defines the view transform. It has the following syntax:

```
LookAt ex ey ez lx ly lz ux uy uz
```

Where the `e` point represents the eye position, the `l` point the point where the eye is looking at and the `u` vector defined the up direction. For the Cornell Box let's define these as

```
LookAt 0 -3.4 0       0 1 0       0 0 1
```

Then we have to specify the kind of camera we want to use. To use a perspective camera with 45 degrees field of view add

```
Camera “perspective” “float fov” [45]
```

Let us now define the output. `pbrt` uses a Film component for this, which generates HDR images (OpenEXR (.exr) or PMF (.pmf)) or LDR images (.png or .tga), depending on the suffix of the filename option. Here we'll define the output filename and image resolution:

```
Film “image” “string filename” [“cornell.exr”] “integer
xresolution” [600] “integer yresolution” [600]
```

`pbrt` will distribute rays over the rendering domain (e.g., the image plane) according to some sampling strategy. We have to tell it which strategy to use and this is done by specifying which sampler to use. We will use the stratified sampler, using one sample per pixel (so if our image has 600x600 pixels, then a total of 600x600 primary rays will be shot) and no jittering:

```
Sampler "stratified" "integer xsamples" [1] "integer ysamples"
[1] "bool jitter" ["false"]
```

Finally, let's tell `pbrt` which light model to use to shade every point. These are known as integrators. For now let's use the very simple `directlighting` surface integrator that is distributed with `pbrt`:

```
Integrator "directlighting"
```

OK, that's all for the rendering setup.

World Description

The world definition block starts with the `WorldBegin` directive and finishes with the `WorldEnd` directive; when this is encountered, the `Renderer` takes control and does the required rendering computation. Lights, materials, textures, shapes, and volumetric scattering regions can only be defined inside the world block.

Within this block one can define geometric transformations, materials, textures, geometric primitives (objects) among others.

Geometric transformations apply to entities specified after the transformation. Supported transformations include Identity, Translate, Scale and Rotate among others.

Let's start our Cornell Box world description with:

```
WorldBegin
Identity
```

`Attributes` is a mechanism to save and restore the current graphics state. An `Attribute` block starts with `AttributeBegin` and finishes with `AttributeEnd`. If the graphics state, e.g., the current transformation matrix is modified within an attribute block, it is restored to its previous state when the attribute block finishes.

Any world description must include the definition of the light sources, otherwise there will be no radiant power within the virtual world. For the Cornell Box we will use a diffuse point light source:

```
# lines starting with # are comments
# Point Light Source
AttributeBegin
  LightSource "point"
      "color I" [25 25 25]
      "point from" [0 0 0.9999]
```

```
AttributeEnd
```

We will now define the walls of the Cornell Box. Before defining the geometry, one has to define the material that geometry is built of; let's use the plastic material which is a diffuse material:

```
# walls
AttributeBegin
  # white walls  material
  Material "plastic"
           "color Kd"    [1 1 1]
           "color Ks"    [0.1 0.1 0.1]
           "float roughness" 0.15
```

The walls are squares, each built from two triangles. We will use the `trianglemesh` geometric primitive to build each wall:

```
# back wall
Shape "trianglemesh"
           "integer indices" [0 1 2 2 3 0]
           "point P" [-1 1 -1   -1 1 1   1 1 1   1 1 -1]
# ceiling
Shape "trianglemesh"
           "integer indices" [2 1 0 0 3 2]
           "point P" [-1 1 1   1 1 1   1 -1 1   -1 -1 1]
# floor
Shape "trianglemesh"
           "integer indices" [0 1 2 2 3 0]
           "point P" [-1 1 -1   1 1 -1   1 -1 -1   -1 -1 -1]
```

The left and right walls will have different colors:

```
# red wall material
  Material "plastic"
              "color Kd"    [0.8 0.1 0.1]
             "color Ks"    [0.1 0.1 0.1]
             "float roughness" 0.15
  # left red wall
  Shape "trianglemesh"
           "integer indices" [0 1 2 2 3 0]
           "point P" [-1 -1 -1   -1 -1 1   -1 1 1   -1 1 -1]
  # blue wall material
  Material "plastic"
              "color Kd"    [0.2 0.3 0.8]
             "color Ks"    [0.1 0.1 0.1]
             "float roughness" 0.15
  # right blue wall
  Shape "trianglemesh"
           "integer indices" [2 1 0 0 3 2]
           "point P" [1 -1 -1   1 -1 1   1 1 1   1 1 -1]
```

We can close this attribute block:

```
AttributeEnd
```

Let’s populate the interior of the box with a glass sphere, a metal sphere and a mirror:

```
# glass sphere
AttributeBegin
  Translate -0.45 0 -0.1
  # glass  material
  Material "glass"
           "color Kr" [0.6 0.6 0.6]
           "color Kt" [0.96 0.96 0.96]
           "float index" [1.5]
  Shape "sphere" "float radius" [0.35]
AttributeEnd

# metal sphere
AttributeBegin
  Translate 0.45 0.4 -0.65
  # metal  material
  Material "uber"
           "color Ks" [0.7 0.7 0.7]
           "color Kr" [0.8 0.8 0.8]
           "float roughness" [0.02]
  Shape "sphere" "float radius" [0.35]
AttributeEnd

# mirror
AttributeBegin
  # mirror material
  Material "mirror"
             "color Kr"    [0.9 0.9 0.9]
  # mirror
  Shape "trianglemesh"
           "integer indices" [2 1 0 1 3 2]
           "point P" [0.99 -0.45 0    0.99 0.45 0   0.99 -0.45 -
0.9         0.99 0.45 -0.9]
AttributeEnd
```

And finally terminate the world (NOTE: if you in fact have the power to terminate the world, do not take this literally! What is meant is: terminate the world description):

```
WorldEnd
```

If you run `pbrt` on this scene description you should obtain the `cornell.exr` image file. Just do it:

```
pbrt cornell.pbrt
```

and visualize the results. Remember that to visualize HDR images (such as EXR) you can use several different tools, such as IrfanView for Windows or OpenHDR Viewer online (https://viewer.openhdr.org/).

There are many mechanisms to make your scene more interesting both at the modeling stage and at the rendering one. Let’s add a texture by adding the following statement after the light definition:

```
Texture "checks" "color" "checkerboard"
           "float uscale" [4] "float vscale" [4]
           "color tex1" [0 0 1] "color tex2" [0 1 1]
```

Now change the white walls definition (that is the respective material) such that it uses this texture instead of a constant value for the diffuse reflection coefficient:

```
# white walls  material
Material "plastic"
           "texture Kd" "checks"
           "color Ks"    [0.1 0.1 0.1]
           "float roughness" 0.15
```

More interesting, from the light transport point of view, we can change the light source from a point to an area light. Comment the point light source and add:

```
AttributeBegin
  AreaLightSource "area"
           "color L"    [25 25 25]
           "integer nsamples" [1]
  Shape "trianglemesh"
           "integer indices" [2 1 0 2 3 1]
           "point P" [-0.25 -0.25 0.99     0.25 -0.25 0.99     -
0.25 0.25 0.99     0.25 0.25 0.99]
AttributeEnd
```

Render the new scene (take note on Tabela 1 of how long it takes to execute):

```
pbrt cornell.pbrt
```

Q1 - Is your image noisy? Why?

Change the number of shadow rays that are shot, at each intersection point, towards the light source (**spl** – samples per light) and render the image. On the light source definition make sure:

```
           "integer nsamples" [16]
```

Q2 – How is noise now? What happened to execution time?

Q3 – Can you detect aliasing? Why?

Aliasing can be traded by noise, by doing stochastic sampling of the image plane. Turn jittering on and rerender the image:

```
Sampler "stratified" "integer xsamples" [1] "integer ysamples"
[1] "bool jitter" ["true"]
```

To reduce noise on the image plane you can increase the number of primary rays per pixel to 4 (**spp** – samples per pixel):

```
Sampler "stratified" "integer xsamples" [2] "integer ysamples"
[2] "bool jitter" ["true"]
```

Note that you can reduce **spl** to 4 also on the light source:

```
            "integer nsamples" [4]
```

Rerender the image and comment.

Tabela 1 - Execution Times

cornell.pbrt	
Settings	Time (s)
spp = 1 ; jitter = false ; spl =1	
spp = 1 ; jitter = false ; spl =16	
spp = 1 ; jitter = true ; spl =16	
spp = 4 ; jitter = true ; spl =4	