pbrt v3 – Whitted Integrator

Luís Paulo Santos – Dep. de Informática

Universidade do Minho

March, 2022

Throughout this tutorial let's use the Cornell Box scene developed on the previous tutorial. It is available on the e-learning platform together with this tutorial. Make sure you are using the Point Light source.

Sampling

`pbrt` provides a series of samplers that allow controlling how the multidimensional radiance signal that we want to approximate is sampled. This includes obviously sampling the image plane.

The number of pixels the final image will have is controlled by the `Film` component[1]. How many primary rays are generated and how are these distributed over the image plane, results from a combination of the `Camera`, `Film` and `Sampler` components.

One of such samplers is the "`stratified`", which allows direct control over the number of samples (primary rays) per pixel (spp). Check that your scene description header includes the following line:

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

You might also want to control the output filename to facilitate results comparisons. Do so by changing the Film component filename parameter:

```
Film "image" "string filename" ["cornell_strat1.exr"]
```

Run `pbrt` for the different `spp` and fill the table below:

spp	xsamples	ysamples	filename	Time (sec)
1	1	1	cornell_strat1	
4	2	2	cornell_strat4	
16	4	4	cornell_strat16	

What do you conclude regarding both image quality and execution time? What is the relationship between the number of samples per pixel and the execution time (i.e., the latter grows linearly/logarithmically /exponentially with the former)?

[1] Remember that a full description of the scene description language and, consequently, of available `Films`, `Samplers`, `Cameras`, `Integrators`, etc. is available at https://www.pbrt.org/fileformat-v3.

Shading

It is the `Integrator` that tells `pbrt` how to shade a point on the surface of an object.

Throughout this tutorial we will develop an integrator, which will, progressively, become equivalent to the `whitted` surface integrator distributed with `pbrt`.

In the course website you can find the files `VI-Tutorial2.cpp` and `VI-Tutorial2.h`, which you have to add to your folder `pbrt-v3/src/integrators`.

Depending on the operating system and project management system you are using (`Windows`, `MacOS`, `Linux`, `make`, `CMake`, `MSVS`, `xcode`, etc.) you have to add these 2 files to your project, under the `integrators` tab.

Opening either `VI-Tutorial2.cpp` or `VI-Tutorial2.h`, you'll verify that it defines a new class `VITutorial2Integrator`, which is the integrator we will be working on.

The new integrator has to be included in the `pbrt` library and a relation ship has to be established between the integrator name used in the scenes' descriptions (`VITutorial2`) and the new class `VITutorial2Integrator`. This is done in the file `api.cpp` belonging to pbrt's core (`pbrt-v3/src/core`). Open this file and make the two modifications indicated below:

1. where it reads:
   ```
   #include "integrators/whitted.h"
   ```

   change it to:
   ```
   #include "integrators/whitted.h"
   #include "integrators/VI-Tutorial2.h"
   ```

2. where it reads:
   ```
       if (IntegratorName == "whitted")
           integrator    =    CreateWhittedIntegrator(IntegratorParams,
   sampler, camera);
   ```

   change it to:
   ```
       if (IntegratorName == "whitted")
           integrator    =    CreateWhittedIntegrator(IntegratorParams,
   sampler, camera);
       else if (IntegratorName == "VITutorial2")
           integrator   =  CreateVITutorial2Integrator(IntegratorParams,
   sampler, camera);
   ```

Rebuild `pbrt`.

Make sure that your `cornell.pbrt` scene uses the new integrator (remove or comment out any other integrator):

```
Integrator " VITutorial2"
```

In order to fully understand how `pbrt` code is organized it is recommended that you read section 1.3 of the book (https://www.pbr-book.org/3ed-2018/Introduction/pbrt_System_Overview) , in particular subsection 1.3.5, which described the `whitted` integrator in detail; this is, however, not strictly required since this tutorial tries to give you enough information to understand the process.

The Integrator `Li()` method is called for each ray. This method has to evaluate the radiance associated with that ray:

```
Spectrum VITutorial2Integrator::Li(const RayDifferential &ray,
                                   const Scene &scene, Sampler &sampler,
                                   MemoryArena &arena, int depth) {
```

The first step is to initialize a Spectrum variable (holding float values for R, G and B) to accumulate radiance as it is computed through the method:

```
        Spectrum L(0.);
```

The next step is to find the first intersection of the ray with the shapes in the scene. The `Scene::Intersect()` method takes a ray and returns a Boolean value indicating whether it intersected a shape. For rays where an intersection was found, it initializes the provided `SurfaceInteraction` with geometric information about the intersection; otherwise it returns the computed radiance (which will be 0 at this stage).

```
        // Find closest ray intersection
        SurfaceInteraction isect;
        if (!scene.Intersect(ray, &isect)) {
            return L;
        }
```

Below you will find several different suggestions for this integrator, which will incrementally lead you to the full Whitted ray tracing version.

Please note that the code being presented here IS ALREADY INCLUDED in the file you downloaded. You just have to comment and uncomment the correct code blocks.

DEPTH RENDERER

You'll see that radiance is being computed as

```
    // DEPTH RENDERER
    Float nDepth = ray.tMax / 50.;
    L = (nDepth, nDepth, nDepth);
```

where `tMax` is the ray length, that is the depth of the intersected point. Run and visualize the output image. This is a depth map and a form of shading points with pseudo-color.

FLAT BRDF RENDERER

Comment the 2 lines above that are only used to the depth renderer.

There is some data that is required by all renderers that take into account the geometry, the relevant directions and the material's appearance. These include the normal, the outgoing direction (symmetric to the incident ray) and the BRDF. Include these by commenting out the following code:

```
// Compute the normal and the outgoing direction vectors
// Note that the latter is just the symmetric of the incident ray
const Normal3f &n = isect.shading.n
Vector3f wo = isect.wo;
// Compute scattering functions for surface interaction
// isect.bsdf() will give you access to the BRDF after the
// following call:
isect.ComputeScatteringFunctions(ray, arena);
```

We will now use the value of the BRDF to shade each point. Note that light sources are not taken into consideration.

The BRDF has to be evaluated for a pair of directions, which usually are the incident direction (`wi`) corresponding to the direction to the light source, and the outgoing direction (`wo`) corresponding to the direction of the observer. In this case there is no `wi` defined, since we are not considering any light source. We will use the normal (`n` , computed above) as the second direction. We have, however, to make sure that the normal and the outgoing direction vectors point to the same side of the surface. Therefore, if the cosine between them is less than 0, then we flip the normal.

Uncomment the following lines of code:

```
// FLAT BRDF Renderer
Float n_wo_cosine;
Spectrum f ;
n_wo_cosine = Dot (wo, (Vector3f)n);
if (n_wo_cosine <0.)
   f = isect.bsdf->f(wo, (Vector3f)(-n));
else
   f = isect.bsdf->f(wo, (Vector3f)n);
if (!f.IsBlack())
    L += f ;
```

Run and visualize the output image.

NO SHADOWS RENDERER

Comment the above segment of code (marked with `// FLAT BRDF Renderer`)

Accounting for the light sources without evaluating their visibility (i.e., no shadows) can be achieved by summing their unoccluded contributions. Uncomment the following code (do not forget to comment the code corresponding to the DEPTH renderer above):

```
// Add contribution of each light source
for (const auto &light : scene.lights) {
    Vector3f wi;
    Float pdf;
    VisibilityTester visibility;
    // sample the light source, i.e. select a direction (wi)
    // for a ray which points towards the light source
    Spectrum Li =
      light->Sample_Li(isect, sampler.Get2D(), &wi, &pdf, &vis);
    if (Li.IsBlack() || pdf == 0) continue;
    // Get  the BRDF for the in and out directions pair
    Spectrum f = isect.bsdf->f(wo, wi);
    // Now add the light source contribution (Li) WITHOUT verifying
    // whether there are object in the middle (NO SHADOWS)
    L += f * Li * AbsDot(wi, n) / pdf;
}
```

Render the cornell box and verify that the objects are now lighten, but there are no shadows.

SHADOWS RENDERER

Shadows can be accounted for by testing their visibility, i.e., by shooting shadows rays (uncomment only the red highlighted text):

```
// Add contribution of each light source
for (const auto &light : scene.lights) {
    Vector3f wi;
    Float pdf;
    VisibilityTester visibility;
    // sample the light source, i.e. select a direction (wi)
    // for a ray which points towards the light source
    Spectrum Li =
      light->Sample_Li(isect, sampler.Get2D(), &wi, &pdf, &vis);
    if (Li.IsBlack() || pdf == 0) continue;
    // Get  the BRDF for the in and out directions pair
    Spectrum f = isect.bsdf->f(wo, wi);
    // Now add the light source contribution (Li) BUT verifying
    // whether there are object in the middle (SHADOWS)
    if (!f.IsBlack() && vis.Unoccluded(scene))
        L += f * Li * AbsDot(wi, n) / pdf;
}
```

Render the cornell box and verify that the objects are now lighten, and there are shadows. Do you understand what happened?

WHITTED RAY TRACING RENDERER

Finally, in order to have a full Whitted shader you need to add reflection and transmission secondary rays. Do so by uncommenting the code segment below in the end of the `Li()` method, just before the return statement:

```
// Full Whitted ray tracing with reflections and transmissions
if (depth + 1 < maxDepth) {
    // Trace rays for specular reflection and refraction
    L += SpecularReflect(ray, isect, scene, sampler, arena, depth);
    L += SpecularTransmit(ray, isect, scene, sampler, arena, depth);
}
```

Render the cornell box and verify that there are shadows, specular reflection and transmission.

Make sure you understand what tree of rays was shot (spawned) for the different types of lighting phenomena in the image.