pbrt v3 –Path Tracing

Luís Paulo Santos – Dep. de Informática

Universidade do Minho

May, 2022

Path Tracing and Global Illumination Components

Scene Setup

From the e-learning system download the `cornell-T6.pbrt` scene.

Path tracing involves, at each shading point, stochastically selecting the direction upon which the random walk from the camera into the scene (each branch finishing on the light sources) will proceed. Since any direction has a non-zero probability of being selected, most light transport phenomena modelled by geometric optics can be simulated.

Let's play with the path tracer integrator code and try to isolate different light transport phenomena.

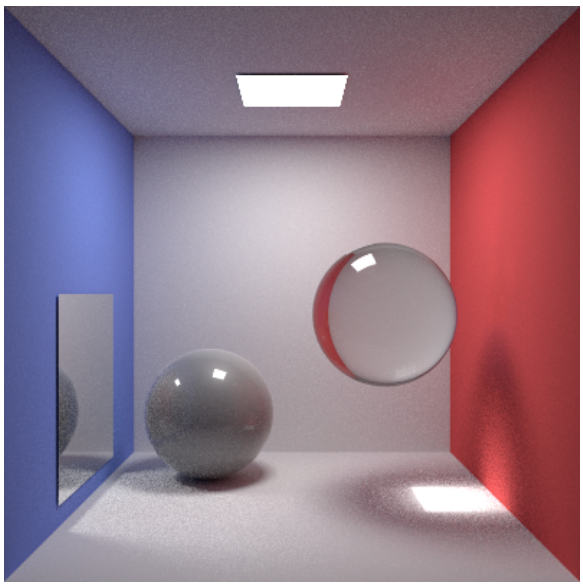
Render the path traced image using the path tracer and a parameterization such that rendering time is maintained at reasonable levels, such as 1024 samples per pixel:

```
Sampler "stratified" "integer xsamples" [32] "integer ysamples" [32]
"bool jitter" ["true"]

Film "image" "string filename" ["cornell_32x32.exr"] "integer
xresolution" [400] "integer yresolution" [400]

Integrator "path"
```

Visualize the output and select a sampling rate (i.e., modify `xsamples` and `ysamples`) that allows for an image with an acceptable noise level, while maintaining the execution time within reasonable limits.



Can you:

1. identify caustics?
2. identify diffuse interreflections and color bleeding?

Path integrator code modification

On the folder `pbrt-v3/src/integrators` copy `path.cpp` to `path-original.cpp`.

Your task throughout this first section of the tutorial is to carefully analyze the path tracer `Li()` method (in `path.cpp`). Note that the method is not recursive, but rather iterative, which will make your task easier. There is a `for` loop which iterates over the depth of the path; the loop counter is the variable `bounces`.

```
for (bounces = 0;  ; ++bounces) {
        .. code ..
}
```

Remember that we are tracing rays from the camera, so `bounce =0` represents the interaction of the primary ray with scene, larger values of `bounce` represent secondary rays at higher depths of the random walk. Start by separating the processing of the first bounce from the `for` loop, such that this ray is treated before the loop and the loop itself starts with `bounce=1`:

```
bounces = 0;
bool stop = false
.. code ..  // primary ray
for (bounces = 1; !stop ; ++bounces) {
        .. code ..
}
```

Note that there are some conditional structures in `code` that depend on the value of `bounces`; you can obviously remove these from the primary ray processing, since you know that `bounces==0`. Also, the `code` includes some `break` statements, which must be changed to `return L` when processing `bounces==0` (this is not inside a loop, such that you can `break` out of it).

Re-render the image to make sure that your code is correct (you should obtain a similar image, differences eventually residing on the different sequence of pseudo-random numbers) – simultaneously write down the rendering time in the table below.

Direct Illumination Only

To simulate direct illumination only, just disable the loop corresponding to `bounces>=1` (for instance, make the condition of the `for` loop equal to `false)`.

Before proceeding look at the full path traced image and try to figure out how will it look like with direct illumination only. Build `pbrt`, change the output filename to `cornell_dir.exr` on the scene description and generate the new image. Visualize the resulting image and compare it to your expectations. Write down the execution time on the table below.

Specular Reflections Only

We will now change the code such that only specular reflections are simulated. Note, however, that we will maintain all light transport paths for secondary bounces. This makes the image much easier to understand. Therefore, within the bounces `for` loop (i.e., `bounces>0`) make sure that you allow the loop to repeat forever (it finishes through `break` statements).

Now you just have to handle the way you light the point intersected by each primary ray, i.e., the part of the code handling `bounces==0`.
Disable direct illumination (only specular reflections will be simulated, remember). Comment the following lines of code:

```
const Distribution1D *distrib = lightDistribution->Lookup(isect.p);

// Sample illumination from lights to find path contribution.
// (But skip this for perfectly specular BSDFs.)
if (isect.bsdf->NumComponents(BxDFType(BSDF_ALL & ~BSDF_SPECULAR)) > 0) {
    ++totalPaths;
    Spectrum Ld = beta * UniformSampleOneLight(isect, scene, arena,
                                        sampler, false, distrib);
    VLOG(2) << "Sampled direct lighting Ld = " << Ld;
    if (Ld.IsBlack()) ++zeroRadiancePaths;
    CHECK_GE(Ld.y(), 0.f);
    L += Ld;
}
```

You must also disable direct viewing of the light source, which is handled by the following lines of code (comment it):

```
// Add emitted light at intersection
// Add emitted light at path vertex or from the environment
if (foundIntersection) {
       L += beta * isect.Le(-ray.d);
       VLOG(2) << "Added Le -> L = " << L;
} else {
       for (const auto &light : scene.infiniteLights)
              L += beta * light->Le(ray);
       VLOG(2) << "Added infinite area lights -> L = " << L;
}
```

When processing the first bounce, the next direction to sample is selected by calling the following function:

```
Spectrum f = isect.bsdf->Sample_f(wo, &wi, sampler.Get2D(), &pdf, BSDF_ALL, &flags);
```

The `BSDF_ALL` flag tells `Sample_f()` that any type of interaction can occur (either reflection or transmission) and any interaction mode can be simulated (specular, diffuse or glossy), as long as they exist in the material's BRDF. By changing the allowed types and modes you can control which phenomena are simulated. The datatype associated with this flag is given by:

```
enum BxDFType {
       BSDF_REFLECTION = 1<<0,
       BSDF_TRANSMISSION = 1<<1,
       BSDF_DIFFUSE = 1<<2,
       BSDF_GLOSSY = 1<<3,
       BSDF_SPECULAR = 1<<4,
       BSDF_ALL_TYPES = BSDF_DIFFUSE | BSDF_GLOSSY | BSDF_SPECULAR,
       BSDF_ALL_REFLECTION = BSDF_REFLECTION |BSDF_ALL_TYPES,
       BSDF_ALL_TRANSMISSION = BSDF_TRANSMISSION | BSDF_ALL_TYPES,
       BSDF_ALL = BSDF_ALL_REFLECTION | BSDF_ALL_TRANSMISSION
};
```

Change `BSDF_ALL` to `(BxDFType)(BSDF_SPECULAR | BSDF_REFLECTION)`.

Before proceeding look at the full path traced image and try to figure out how will it look like with specular reflections only. Build pbrt, change the output filename to `cornell_specrefl.exr` on the scene

description and generate the new image. Visualize the resulting image and compare it to your expectations. Write down the execution time on the table below.

Specular Phenomena

It should be quite obvious now how to change the above flag, such that all types of specular interactions, i.e., reflections and transmissions, are handled. Change the code accordingly.
Before proceeding look at the full path traced image and try to figure out how will it look like with specular interactions only. Build pbrt, change the output filename to `cornell_spec.exr` on the scene description and generate the new image. Visualize the resulting image and compare it to your expectations. Write down the execution time on the table below.

Diffuse Phenomena

Change the above flag, such that all types of diffuse interactions are handled, but specular ones are not.
Before proceeding look at the full path traced image and try to figure out how will it look like with specular interactions only. Build `pbrt`, change the output filename to `cornell_diffuse.exr` on the scene description and generate the new image. Visualize the resulting image and compare it to your expectations. Write down the execution time on the table below.

After rendering all the 5 images complete the table below (including which kind of light paths are actually simulated using the 'L', 'D', 'S', 'E' notation) and compare with the predictions you were asked to do w.r.t. to image appearance and rendering times. Why do you think rendering the diffuse interactions took significantly longer than specular interactions for this particular image?

Configuration	Time (s)	Simulated Paths
All		
Direct Only		
Specular Reflection Only		
Specular Reflection + Transmission		
Diffuse Reflection		

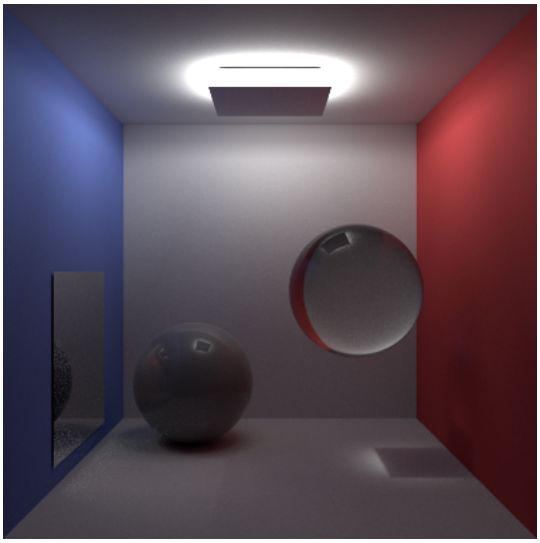
Path Tracing and Hard Direct Illumination

On the folder `pbrt-v3/src/integrators` revert `path-original.cpp` to `path.cpp`.

Uncomment the lines below on the scene description file:

```
# light occluder
Shape "trianglemesh"
          "integer indices" [2 1 0 2 3 1]
          "point P" [-0.3 -0.3 0.89     0.3 -0.3 0.89     -0.3 0.3 0.89
0.3 0.3 0.89 ]
```

A medium to high quality rendering of the Cornell box with this occluder is included below for analysis purposes (spp =2^{18}, T_{render} = 4h48m on 48 cores).



Can you identify indirect shadows?

Render the new scene with following parameters:

```
Sampler "stratified" "integer xsamples" [32] "integer ysamples" [32]
"bool jitter" ["true"]

Film "image" "string filename" ["cornell_occ_32x32.exr"] "integer
xresolution" [400] "integer yresolution" [400]

Integrator "path"
```

Visualize it. Comment on the results? Why is there so much noise?

During the last lecture the bidirectional path tracing method was briefly described and its advantages for difficult lighting conditions were explained. Try it, but reduce the number of pixel samples to `16x16` in order to maintain acceptable rendering times:

```
 Sampler "stratified" "integer xsamples" [16] "integer ysamples" [16]
"bool jitter" ["true"]

Film "image" "string filename" ["cornell_occ_16x16_bdpt.exr"] "integer
xresolution" [400] "integer yresolution" [400]

Integrator "bdpt"
```

Comment on the results.