Tutorial 4
OSPRay – Path Tracing

Iluminação e Visualização II

Luís Paulo Santos, Maio 2021

Throughout this tutorial you will develop a path tracer.

Introduction

The distributed ray tracer developed during the previous tutorial will now be extended to a path tracer. Essentially, when a diffuse material is intersected stochastic hemisphere sampling will be performed.

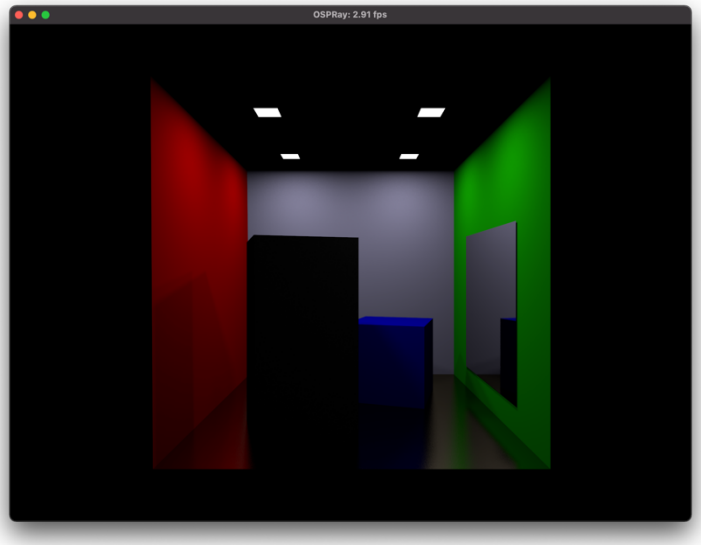
Edit the `CMakeLists.txt` file in `ospray-vi2/src/rtlibrary` and make sure that you use the `MyRenderer-T4.ih` and `MyRenderer-T4.ispc` files for this project:

```
ispc_target_add_sources(ospray_module_rtlibrary
  ModuleInit.cpp
  render/MyRenderer.h
  render/MyRenderer.cpp
  render/MyRenderer.ih
  render/MyRenderer.ispc
  render/MyMaterial.h
  render/MyMaterial.cpp
  render/MyMaterial.ih
  render/MyMaterial.ispc
  render/MyRenderer-T4.ih
  render/MyRenderer-T4.ispc
)
```


Universidade do Minho

Also make sure that in `MyRenderer.cpp` the `commit()` method is setting the lights for the Cornell box scene. Build the renderer and run it:

```
>./cornell_VI2
```



This renderer is still the same as developed during Tutorial 3. Intersections with:

- **diffuse** materials result on the end of the light transport simulation, other than an extra hop towards light sources (direct lighting);
- **specular** materials result on shooting an additional secondary specular ray, if the maximum depth has not been reached yet.

Diffuse Interreflections

Consider the `secondary_PT` function. You will find the following two lines of code:

```
do_specular = true;
SPECULAR_PROB = 1.f;
```

These determine that only the specular component of the BRDF is considered, with a probability of 1. We want to add the diffuse component. This function will stochastically select whether to follow the specular or the diffuse path!

Delete or comment the two lines above. You will see that the code now is:

```
float SPECULAR_PROB=0.5f;

bool do_specular;
float r = frandom(rng);
do_specular = (r < SPECULAR_PROB ? true : false);
```

Note that `do_specular` will be true with probability `SPECULAR_PROB`. If you look below at the code you will see that this variable selects whether the SPECULAR OR DIFFUSE path is followed:

```
if (do_specular) { // specular interaction
  Ray specRay;

  ... ...
  vec3f specularC = recursive_PT (self, model, specRay, depth, rng);
  color = color + Ks * specularC * dot(dg.Ns, Rs);
}
else {        // diffuse interaction
  Ray diffRay;

  ... ...
  vec3f diffC = recursive_PT (self, model, diffRay, depth, rng);
  color = color + Kd * diffC * pi;
}
return (color);
```

The big difference to the code we had before is that upon an intersection with a diffuse material a path will continue by shooting a new ray. Diffuse interreflections are now possible and this will change completely the feel and look of our rendered images.

Let us now focus on the diffuse interaction. We need a diffuse ray and a direction for it. OSPRay utility code includes a function to generate a direction over the hemisphere, whose probability is cosine weighted ($p(\omega) = \cos\theta/\pi$) :

```
Ray diffRay;

// sampling the hemisphere
// get a random direction distributed over the hemisphere
// this direction is local, i.e. is centered around the Z axis
const vec2f s = make_vec2f(frandom(rng), frandom(rng));
const vec3f local_diff_dir = cosineSampleHemisphere(s);
```

Let's rotate the generated direction from the Z axis to the intersection shading normal and get the ray:

```
const vec3f diff_dir = frame(dg.Ns) * local_diff_dir;
setRay(diffRay, dg.P, diff_dir, dg.epsilon, inf, ray.time);
```

Let's call our recursive function to get the radiance travelling along this ray:

```
vec3f diffC = recursive_PT (self, model, diffRay, depth, rng);
```

Finally let's make color equal to the corresponding contribution:

```
color = color + Kd * diffC * pi ;
```

Remember that the rendering equation tells us that the differential radiance due to a differential direction (such as $\omega = \texttt{diff\_dir}$) is $f_r(V \leftrightarrow \omega) * L(\leftarrow \omega) * \cos(N, \omega)/p(\omega)$. However, in the above equation you cannot see the cosine term and there is a multiplication by π! Can you explain why? What is the value of $p(\omega)$?

We are mostly done. There is still a problem, however. Remember that in stochastic Monte Carlo integration, **every time a probabilistic decision is made the result has to be divided by the probability of the decision!**

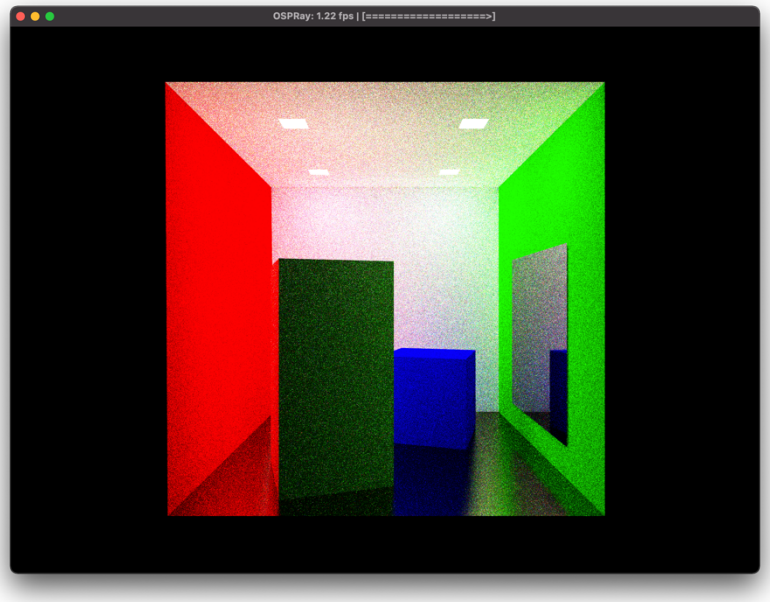
Above we decided whether to follow a specular or a diffuse path with probability `SPECULAR_PROB` for the former and `(1-SPECULAR_PROB)` for the latter. The respective estimates have to be divided by these probabilities. On the appropriate places of your code do:

```
specularC = specularC / SPECULAR_PROB;
```

and

```
diffC = diffC / (1.f - SPECULAR_PROB);
```

Build the renderer and run the code.



There are several observations that can be done on the previous image:

- the ceiling is no longer black! When a primary ray intersects the ceiling it finds a diffuse material. Previously our path would end up here (other than the shadow rays shot towards the light sources, which would contribute 0 due to the cosine being equal to 0). Now our path tracer stochastically selects a direction to shoot a secondary ray. Therefore, the algorithm is able to gather secondary reflected radiance onto diffuse surfaces: this is referred to as **diffuse interreflections** and the corresponding Heckbert notation is `LDD+E`;
- there is lots of noise in the image. This is because the direction to shoot secondary diffuse rays is stochastically selected over the hemisphere. Even though noise should reduce as more frames get rendered (remember, this is **progressive rendering**) the truth is that the random number generator we are using -- `frandom` -- is not that good and after a few frames it starts repeating random numbers and therefore the image no longer improves. A better random number generator would help a lot;
- **color bleeding**, a phenomenon associated with diffuse interreflections is clearly visible. Notice how the ceiling is reddish to the left, nearer to the red wall, and greenish to the right, nearer to the green wall. Also the left face of the tall white block looks red. This is because these surfaces are receiving more light that has been reflected by the red or green wall, respectively. This diffusely reflected light contains more red (or green) radiance and thus tints the surfaces it illuminates with that color – this is color bleeding, as if the color bled from one surface into another.

Setting the selection probabilities

Fixing the specular versus diffuse paths selection probability *a priori* might not be the most intelligent option. What if the value of `Kd` is much larger than `Ks` or *vice-versa?* What if one of these values is `{0., 0., 0.}`? Can you think on a smarter way of stochastically selecting whether to follow the specular or the diffuse path?

Uncomment the following code in `secondary_PT`, and carefully analyze what it does:

```
float normalizing_constant = (length(Kd)+length(Ks));
normalizing_constant = (normalizing_constant <= 1.e-5 ? 1.f : normalizing_constant);
SPECULAR_PROB=length(Ks) / normalizing_constant;
```

Russian roulette

Look at the termination condition in `recursive_PT` (i.e., the condition to call `secondary_PT`). It is completely deterministic (`(depth+1) < self->super.maxDepth`), which results on a biased result: the generated image will not converge to the physically correct result because paths longer than `self->super.maxDepth` are not possible:

```
if ((depth+1) < self->super.maxDepth) {
      vec3f secondary = secondary_PT (self, model, rng, dg, ray, Kd, Ks, ns, depth+1);
      color = color + secondary;
    }
```

Now consider the code under the Russian roulette option. Whether or not to continue a given path is stochastically decided, which means that paths of any length can be followed, resulting on a non-biased (but noisier) image.

Comment the code above and uncomment the code below, rebuild and run the program:

```
    const float continue_p = 0.5f;
    if (frandom(rng) <= continue_p) {
      vec3f secondary = secondary_PT (self, model, rng, dg, ray, Kd, Ks, ns, depth+1);
       secondary = secondary / continue_p;
      color = color + secondary;
    }
```

By trying with very different values of the continuation probability (`const float continue_p = 0.5f;`) comment what happens to the execution time and apparent convergence rate. Try values such as `0.01f`, `0.1f` and `0.9f`.

That's all, folks!