

Project - Pathtracer

Introduction

In this lab we will implement and improve a simple path-tracer. Path-tracing is a way of rendering global illumination images that has recently become quite popular in off-line renderers. You will start from some simple code that renders an image with direct lighting only. From there we will add some basic functionality and then you will do one larger assignment (suggestions at the end) on your own.

Note: This is the first time we do this tutorial and there are bound to be some things that will have to be improved. Keep a lookout on the webpage for updates, do it at your own risk and have fun!

The rendering Equation

We will begin by describing the basic math behind what we are going to do, but very briefly. Refer to lecture notes, your textbook (chapters 7 and 9) and the internet for details. We start out with *The Rendering Equation*. This equation is actually all we have to solve to create glorious photo-realistic images:

$$L_o(\mathbf{p}, \omega) = L_e(\mathbf{p}, \omega) + \int_{\Omega} f(\mathbf{p}, \omega, \omega') L_i(\mathbf{p}, \omega') \cos(\mathbf{n}, \omega') d\omega'$$

It says that the *radiance* (Watt per unit solid angle and unit area) L_o that leaves a point \mathbf{p} in the direction ω can be found from the:

- **Emitted Light**, $L_e(\mathbf{p}, \omega)$ – If the surface at point \mathbf{p} emits radiance in direction ω
- **BRDF**, $f(\mathbf{p}, \omega, \omega')$ - This is a function that describes how the surface reflects light.
- **Incoming radiance**, $L_i(\mathbf{p}, \omega')$ - This is the incoming light from direction ω' , which we typically find by evaluating $L_o(\mathbf{p}', -\omega')$ again at the first intersection point (\mathbf{p}') in that direction.
- **Cosine term**, $\cos(\mathbf{n}, \omega')$ - This will attenuate the incoming light based on the angle between light and normal.

This equation is recursive (since L_i depends on L_o). That is, the outgoing light from any point depends on the incoming light from all directions in that point, which is the outgoing light from a lot of surrounding points, which in turn depend on the incoming light at those points, and so on.

There is usually no analytical solution to the rendering equation. We can however estimate a solution using what is called *Monte Carlo integration*:

$$L_o(\mathbf{p}, \omega) \approx L_e(\mathbf{p}, \omega) + \frac{1}{2\pi N} \sum_{i=0}^N f(\mathbf{p}, \omega, \omega_i) L_i(\mathbf{p}, \omega_i) \cos(\mathbf{n}, \omega_i)$$

and that is exactly what we are going to do. The directions ω_i above are N random directions uniformly distributed over the hemisphere. We will discuss better sampling later.

The BRDF and Materials

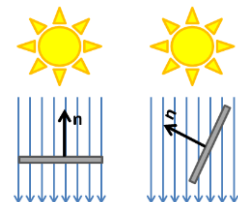
In our discussion and code we will talk about BRDFs and Materials. The *Bidirectional Reflectance Distribution Function* (BRDF) is a four-dimensional function that says how much of the energy that

comes from direction ω will be reflected in some direction ω' . The BRDF is *reciprocal*, i.e. $f(\mathbf{p}, \omega, \omega') = f(\mathbf{p}, \omega', \omega)$ and *energy-conserving*. This second property means that unless energy is absorbed and turned into heat, the total energy reflected will equal the total incoming energy over the hemisphere.

In this tutorial, a Material can “contain” one BRDF or a combination of several BRDFs, and various settings that describe the material (e.g. reflectance, index-of-refraction and so on...).

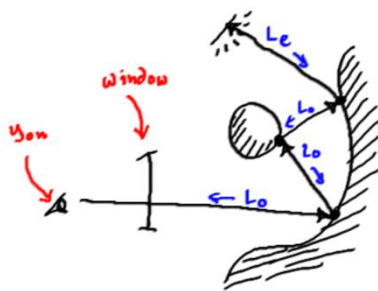
The Cosine Term

The BRDF describes how a specific surface reflects light, and can be expressed in tons of different ways. It is important to understand that the *cosine term* in the equations above is *not* a part of the BRDF. To understand this term it is important to realize that we are evaluating the reflected radiance from an *infinitesimal surface* (not a point), and this surface will *receive* different amounts of energy depending on its orientation towards the incoming light.



Path-tracing

Let's move on to something more practical and find out how we can use all this to generate pretty images. We will use a method called path-tracing to solve the equation above and in this section we will briefly cover how that works, before we go on to actually implement one.



For starters, each frame we will shoot one ray per pixel (the ray starts at the camera and goes “through” the pixel and into the scene). We find the first intersection-point with the scene and there we want to estimate the outgoing radiance.

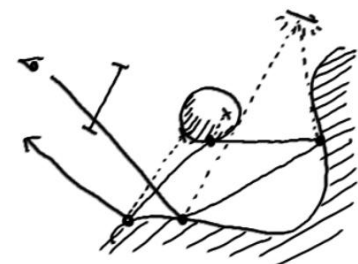
We do so by shooting *one single ray* in a perfectly random direction to estimate the incoming radiance. At the first intersection point for that ray, we again shoot a single ray and so on, forming a *path* that goes on until there is no intersection (or we terminate on some other

criteria). At each vertex of the path we will evaluate the emitted light, the BRDF and the cosine term to obtain the radiance leaving the point. When the recursion has terminated and we have obtained L_o for the first ray shot, we record this radiance in the framebuffer.

This is all we need to do to create a mathematically sound estimation of the true global illumination image we sought! It's probably all black though, unless we were lucky and hit an emitting surface with some rays in which case it might be black with a little bit of noise. But if we keep running the algorithm, and accumulate the incoming radiance in each pixel (and divide by the number of samples) we would eventually end up with a correct image. This will take more computing time than is left for this course though, so let's speed things up a bit.

Direct Illumination

The first step is to separate *indirect* and *direct* illumination. At each vertex of the path we will first shoot a *shadow-ray* to each of the light-sources in the scene. If there is no geometry blocking the light we will multiply the radiance from the light with the BRDF and cosine term and *then* add the incoming indirect light by shooting a single ray as before. For this to be correct, we just have to make sure that the surfaces we have chosen to be light-sources do not contribute emitted radiance if sampled by an indirect ray.

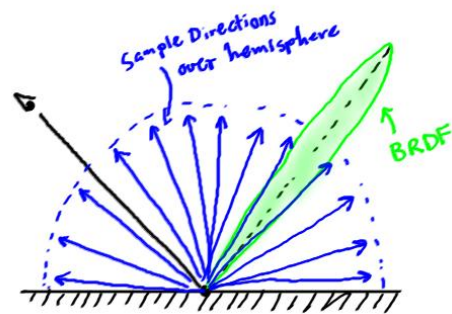


There! That's a path-tracer and it generates correct images. It's still painfully slow if any of our materials have a narrow brdf though. We will remedy this with what is called *importance sampling*.

Importance sampling

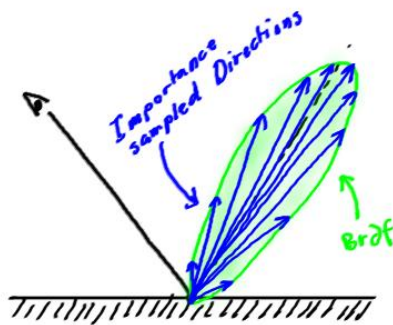
The problem is illustrated in the image to the right. Say that our BRDF represents a near mirror reflection. That means that all sample directions that are not close to the perfect specular reflection direction will contribute almost nothing.

With importance sampling we will instead pick samples from a distribution that looks more like the BRDF we are trying to sample, for which we know the *Probability Density Function* (PDF), $p(\omega_i)$.



$$L_o(\mathbf{p}, \omega) \approx L_e(\mathbf{p}, \omega) + \frac{1}{N} \sum_{i=0}^N \frac{f(\mathbf{p}, \omega, \omega_i) L_i(\mathbf{p}, \omega_i) \cos(\mathbf{n}, \omega_i)}{p(\omega_i)}$$

This new equation is no different from the previous one, except that we now allow for varying PDFs (Before, the PDF was a constant $\frac{1}{2\pi}$ since we sampled uniformly on the hemisphere). We will now pick samples with a probability that is much higher where the BRDF is large. We then have to divide the resulting radiance of each sample by the PDF in the sample direction to account for this. So we will have more samples where the BRDF is high, but if we happen to pick a sample where the PDF is low, its contribution will be increased.



Note that we rarely can find a sampling scheme that exactly matches the BRDF and that the sampling scheme must be carefully chosen so that it will not occasionally sample directions with very low probability where the BRDF is not correspondingly low. If we do, then we can get very strong samples in some random pixels that take a very long time to converge.

Let's look at a simple (perhaps the simplest) example. Say that our surface has a gray perfectly diffuse material. The surface absorbs 30% of the incoming light and reflects the rest. A diffuse material reflects equally in all directions.

$$\mathbf{color} = (0.7, 0.7, 0.7)$$

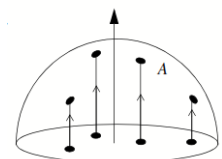
$$f(\mathbf{p}, \omega, \omega_i) = \frac{1}{\pi} \mathbf{color}$$

Well... If it reflects equally in all directions, wouldn't sampling the hemisphere uniformly be the best choice? No because what we are sampling is not just an integral over the BRDF. We are trying to find a solution to the integral:

$$\int_{\Omega} f(\mathbf{p}, \omega, \omega') L_i(\mathbf{p}, \omega') \cos(\mathbf{n}, \omega') d\omega'$$

So ultimately we would like to importance sample with a sample distribution that matches *this* integral as well as possible. But the incoming light is rarely known in advance, and is expensive to evaluate. We can however importance sample on the cosine-term.

In fact, there is a simple way to choose a direction with a pdf that is $p(\omega_i) = \frac{\cos(\mathbf{n}, \omega_i)}{\pi}$. All you have to do is to generate points uniformly on a disc, then project these points on the hemisphere. You can find a number of useful sampling methods with PDFs and other good stuff in the *Global Illumination Compendium*:



<http://people.cs.kuleuven.be/~philip.dutre/GI/TotalCompendium.pdf>

Now let's use this:

$$\frac{1}{N} \sum_{i=0}^N \frac{f(\mathbf{p}, \omega, \omega_i) L_i(\mathbf{p}, \omega_i) \cos(\mathbf{n}, \omega_i)}{p(\omega_i)} = \frac{1}{N} \sum_{i=0}^N \frac{\frac{1}{\pi} \mathbf{color} L_i(\mathbf{p}, \omega_i) \cos(\mathbf{n}, \omega_i)}{\frac{\cos(\mathbf{n}, \omega_i)}{\pi}} = \frac{1}{N} \sum_{i=0}^N \mathbf{color} L_i(\mathbf{p}, \omega_i)$$

How about that? Just by changing the directions you sample, the expression becomes a lot simpler *and* you get a better result.

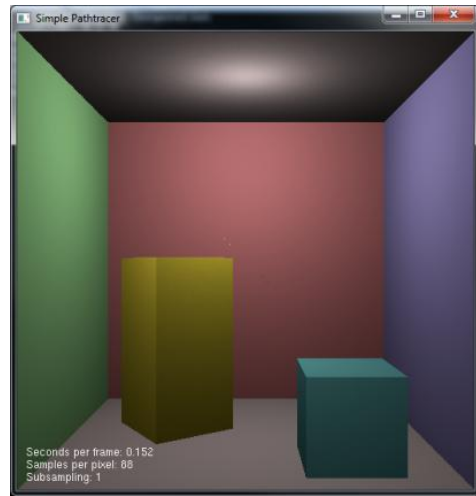
Implementing a basic path-tracer

That's enough theory for now. Open the Visual Studio solution file ("Pathtracer.sln"), make sure you have "Release" build activated and run the program. You should see something like this:

Yikes! That's not pretty at all. Well, don't worry. You will have many opportunities to improve on this image. But first, let's examine what the program does.

Open the file `pathtracer_main.cpp`. This should look quite familiar by now. It is a simple OpenGL program that loads an OBJ model (in the main function towards the end) and then starts a GLUT main loop.

Every time the `display()` function runs, it will trace one ray per pixel (by calling `g_pathtracer.tracePrimaryRays()`). This is where all the interesting stuff happens. When this is done, the `PathTracer` object will contain an updated framebuffer and that will be copied to a texture and rendered as a full-screen quad in the window.



You will spend very little time in this file, but look it through and make sure you understand what goes on. Then look at the very top where you can find these lines

```
#ifdef _DEBUG
int g_subsample = 8;
#else
int g_subsample = 1;
#endif
```

This is a variable you may want to change at times. The `g_subsample` variable lets you raytrace a smaller image than the size of the window while showing a magnified version. This is very useful while debugging or just trying something new. Set `g_subsample` to 2 for now. You can change these values in-game by pressing 's' or 'S'.

Pathtracer::tracePrimaryRays()

Now take a look at this function to make sure you understand it. It takes the currently selected camera and generates one ray for each pixel. It then calls `Scene::intersect` for this ray. This method will find the first intersection between the ray and the scene and fills in an `Intersection` object. This object contains the position, normal and a pointer to the `Material` of the surface that was intersected. Or, the method returns false if there was no intersection.

Then, if there was an intersection, we need to evaluate the radiance that reaches us from that point. This is calculated in `Pathtracer::Li`. If there was no intersection `Pathtracer::Lenvironment` is called instead, which currently just returns a constant radiance.

Assignment:

Let's do a little something to improve the quality of the image already. The code currently samples the lower left corner of the pixel every single iteration. Modify this so that it chooses a random position within the pixel instead, to get some nice super-sample antialiasing.

Tip: There is a `randf()` helper function in the file `MCSampling.h` which returns a random float value between 0.0 and 1.0

Cameras and lights

A quick intermission about cameras and lights. The pathtracer reads OBJ files, which look just as they always have, but the corresponding MTL files look nothing like real MTL files. Take a look at `cornell.mtl`. You need not care about the material definitions just yet, but note that we can define lights and cameras in these files which our program will respect.

You can have any number of lights and any number of cameras. The lights are all used at once, and you can change the chosen camera with the 'c' key. Give it a try. The arguments for lights and cameras should be self-explanatory, but if not, ask an assistant.

Pathtracer::Li()

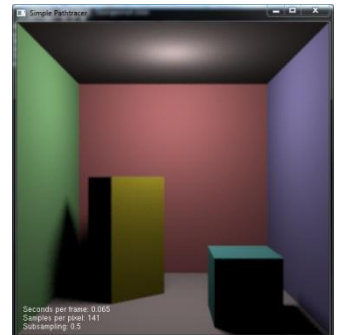
This is where all the interesting stuff happens. Or doesn't happen as it stands, but you will soon take care of that. The current code just iterates over all lights and chooses one position on each light to sample. It then evaluates $f(\mathbf{p}, \omega, \omega_i) L_i(\mathbf{p}, \omega_i) \cos(\mathbf{n}, \omega_i)$, where L_i is the radiance from the light, and returns the sum of all light contributions.

Assignment:

The least thing we can do is to include light visibility so we get some shadows. Create a shadow-ray and intersect it with the scene to see if the light should contribute or not. When you are done you should see a lovely soft shadow like the image to the right.

Tip #1: The Scene class has an `intersectP()` method that may suit your needs. What is the difference between this and the `intersect()` method?

Tip #2: There is a value `PT_EPSILON` defined. This might come in handy here.



Well, that was easy. Now for the tricky part. The current program is not a path-tracer. Because it doesn't trace a path. It's time to rewrite it so that it does not only look at the direct lighting but also shoots a ray to gather indirect illumination.

We could (and you may) implement this as a recursive algorithm, but in practice this is often a lot slower so in the pseudo code below we will instead suggest an iterative approach. Read the pseudo code below and make sure you understand it, then roll up your sleeves and implement it in C++:

```
L                <- (0.0, 0.0, 0.0)
pathThroughput   <- (1.0, 1.0, 1.0)
currentRay       <- primary ray
isect            <- primary intersection
for bounces = 0 to PT_MAX_BOUNCES
{
    // Direct illumination
    for each Light
        sample a point on light
        L += pathThroughput * direct illumination from light if visible

    // Sample an outgoing direction, the brdf and pdf
    (wo, brdf, pdf) <- material.sample_f(wi)

    cosineterm = abs(dot(wo, isect.normal))

    // You may want to make sure pdf cannot be 0 before this
```

```

pathThroughput = pathThroughput * (brdf * cosineterm)/pdf

// If pathThroughput is too small there is no need to continue
if max(pathThroughput) < PT_EPSILON return L

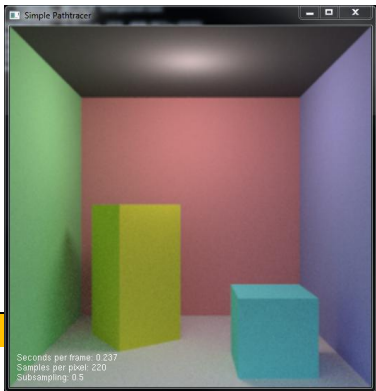
// Create next ray on path
currentRay <- new ray from intersection point in outgoing direction

// Offset the ray slightly to avoid self-intersection
currentRay.o += PT_EPSILON * isect.normal

// Trace the new ray
isect <- m_scene.intersect(currentRay)

if no intersection
    return L + pathThroughput * Lenvironment(currentRay)
}

```



Allright. If you get it right, you should have something like the image to the left. Isn't life better with Global Illumination? Now we will add some importance sampling. In your pathtracer, the direction in which to sample indirect illumination is chosen by the `Material::sample_f()` method.

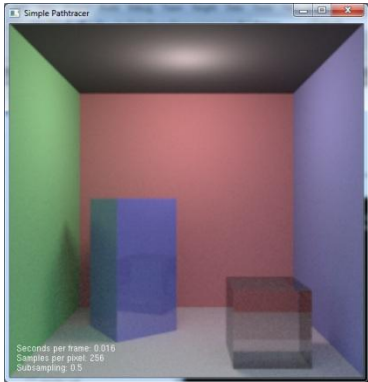
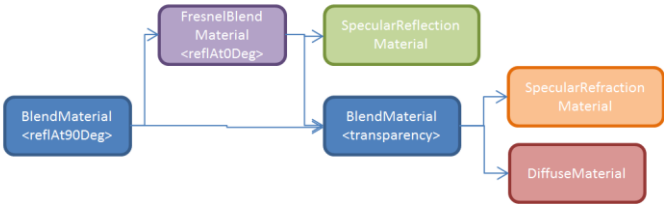
Assignment:
 These cubes all have a simple diffuse material so far. Change the `DiffuseMaterial::sample_f()` method so that it importance samples the cosine term.

Tip: In `MCSampling.h` we have already provided a `cosineSampleHemisphere()` method for you.

Now compare the non-importance sampled image and the importance sampled after the same number of samples per pixel (you can set the `MAX_SAMPLES_PER_PIXEL` pixel in `pathtracer_main.cpp`).

Materials

Take a look at `Material.h`. Besides the `DiffuseMaterial` with which you are already acquainted there are a number of others. When we read an OBJ/MTL file from disc we actually build a little material tree that looks as to the right.



Now uncomment the additional arguments to the materials in `cornell.mtl`. If everything is perfect, you should end up with something like the image to the left.

But you will probably have problems with the refractive box. Consider the offset you do when shooting your new indirect illumination rays. What happens when it refracts?

Once it actually works well, go ahead and play with the material settings and have fun for exactly one hour. Then choose a project from below and show your work to an assistant.

Project

Now that you know the ins and outs of the path-tracer, choose a project from one of the suggested options below, or if you want to do something else suggest it to an assistant. The suggestions are roughly listed in order of expected difficulty. Before you start on a project, talk to an assistant to get some tips.

1. **Textures** – Just as in realtime rendering, offline renderers frequently use textures to increase the level of detail in a scene. Add support for diffuse textures to the code. The current program already contains code that reads texture coordinates and textures (there is a file called “cornell_textured.obj” which has a texture on the floor for testing), but you will have to make sure it ends up on screen.

Optional: You will be able to get even more interesting images if you allow for other types of textures as well. Try to implement support for normal-maps or specular-maps or transparency-maps or whatever tickles your fancy.

2. **BVH** – Examine the Scene class. You will find that the triangles are all stored in one list and that when intersect is called, we will simply iterate through this list of triangles and intersect them one by one to find the closest intersection. This works decently for the simple scene we have looked at so far which only has 32 triangles, but for anything more complex it will be extremely slow.

Implement an acceleration structure (a fairly simple, still powerful suggestion is an AABB tree). You should at least be able to get below 4 seconds per frame for the scene “cornellbottle2.obj” on the lab room machines.

Optional: The acceleration structure is not the only thing that is slow in this path-tracer. Try to profile the code and make it as fast as you possibly can.

3. **HDRI lighting** – Currently when a ray misses all geometry, the environment will return a single constant radiance. You can get much more interesting lighting of an outdoor scene if you instead fetch radiance from an HDRI environment.

Either load and use an HDRI environment map or (trickier) implement a Sun&Sky model, for example this one:

<http://www.cs.utah.edu/~shirley/papers/sunsky/sunsky.pdf>

The FreeImage library that is used for reading textures in the current code can also read .exr or .hdr high dynamic range images.

Optional: If you use a tricky HDRI environment map you may find that it takes a long time for your render to converge. This can be greatly improved if you treat your environment as a light instead and importance sample the direction you sample.

4. **Microfacet BSDF** – The BRDFs available in the pathtracer so far are quite simple. You can represent much more realistic looking materials if you add a BRDF that supports glossy reflections (not perfect mirror reflections). One good paper that presents a physically plausible Microfacet based BRDF that also handles rough refractions is:

<http://www.graphics.cornell.edu/~bjw/microfacetbsdf.pdf>

Extend the path-tracer to use one of the models presented in this paper instead of our perfect specular reflection BRDF.

Optional: Also allow rough refractions.