[CSCI 2240] Assignment 1: Path Tracing (path)

Released: 2/2/2018
Due: 2/16/2018 @ 11:59pm EST

In this assignment, you will implement a physically-based renderer based on the path tracing algorithm. Path tracing is a simple, elegant Monte Carlo sampling approach to solving the rendering equation. Like ray tracing, it produces images by firing rays from the eye/camera into the scene. Unlike basic ray tracing, it generates recursive bounce rays in a physically-accurate manner, making it an unbiased estimator for the rendering equation. Path tracers support a wide variety of interesting visual effects (soft shadows, color bleeding, caustics, etc.), though they make take a long time to converge to a noise-free image.

If you have written a ray tracer in a previous graphics course (e.g. CS 123 here at Brown), you are welcome to extend your ray tracing code for this assignment. If you do not already have a ray tracer, or if you do but you'd rather start from a clean slate, we provide a barebones ray tracer below (see "Resources").

Relevant Reading

- The lecture slides!
- The Global Illumination Compendium
- The Graphics Codex - $10 rendering reference (by Brown alum Morgan McGuire)

Requirements

This assignment is out of **100 points.**

Your path tracer must implement at least the following features:

- Four basic types of BRDFs:
    - Diffuse **(5 points)**
    - Glossy reflections **(5 points)**
    - Mirror reflection **(5 points)**
    - Refraction (with Fresnel reflection) **(5 points)**
- Soft shadows **(10 points)**
    - From sampling area light sources.
- Indirect illumination **(15 points)**
    - Your path tracer must produce color bleeding and caustics.
- Russian roulette path termination **(5 points)**
- Event splitting **(10 points)**

    - At each recursive path tracing step, separately accumulate the contributions of direct lighting and indirect lighting.
    - Be careful not to double-count light sources in both contribution terms!

Your path tracer should take a scene file as input and output an image to disk. To produce output images, you'll need to convert the high-dynamic range radiance values produced by the path tracer into low-dynamic range values that can be written to standard 24-bit RGB image formats such as JPEG or PNG. You'll need a tone-mapping operator for this; a simple global operator such as Equation 3 of this paper is fine (**5 points**).

In addition to your code, you'll also submit images showing off what your path tracer can do **(10 points)**
You must submit images depicting *at least two* different scenes.
These images should demonstrate every feature that your path tracer implements (e.g. you should submit images showing the effects of Russian roulette, next event estimation).
A Cornell Box scene can be a good test-bed for different types of rendering effects; you might consider building and rendering such a scene first.
Submitted images should be easily-viewable low-dynamic range images in e.g. PNG format.
You should try to produce the highest-resolution, highest-sample-count images you can by the submission deadline.

Additionally, you should submit a plaintext README file **(5 points)**
The README file should describe how to run your path tracer (e.g. how to specify different scene file inputs).
This file should list all the features your path tracer implements.
It should also describe what features are demonstrated in the images you've submitted.

Successfully implementing all of the requirements results in a total of **80/100 points** (a grade of B).
To score **100/100** (or more!), you'll need to implement some extra features.

Extra Features

Each of the following features that you implement will earn you extra points. The features are ordered roughly by difficulty of implementation.

- Parallelize your code **(5 points)**
    - The main loop over pixels to render is 'embarrasingly parallel.'
    - Even something as simple as OpenMP's parallel for loop can buy you significant speedups, if applied correctly.
- Attenuate refracted paths **(5 points)**
    - Attenuate the contribution of a refracted light path based on the distance it travels through the refracting object. For example, a path going through a gray glass

crystal ball ends up less bright if it goes a long way through the glass rather than a short distance.

- Stratified sampling **(5 points)**
    - For your path tracer to converge to a relatively noise-free image, it'll need to trace many rays per pixel. Rather than scattering them randomly over the sensor square, try stratified sampling, in which you divide the sensor area into a grid, and then pick one or two or twenty samples uniformly in each grid square.
    - Include at least one image comparing this approach to uniform random ray sampling, and describe this comparison in your README.
- Low discrepancy sampling **(10 points)**
    - The rendering problem that a path tracer tries to solve is an integration problem, and there's nothing that says we have use to random samples to estimate integrals. Rather, any set of samples that are 'well-spaced out' but not perfectly uniform (to avoid aliasing) ought to work. This is the idea behind low-discrepancy sampling (also known as Quasi-Monte Carlo): use an algorithm that deterministically generates random sample points that have the right properties.
    - You can find a more detailed introduction to this concept here.
    - Include at least one image comparing this approach to uniform random ray sampling (or stratified sampling), and describe this comparison in your README.
- BRDF importance sampling **(10 points)**
    - When tracing a recursive ray from a surface intersection, you can choose it randomly from the hemisphere; alternatively, you can choose it in a way that's approximately proportional to the BRDF, and use importance sampling.
    - Include at least one image comparing this approach to uniform hemisphere sampling, and describe this comparison in your README.
- Multiple importance sampling **(10 points)**
    - Next event estimation separates the contributions of direct vs. indirect illumination. Instead, you can use multiple importance sampling (MIS), which provides a general procedure for combining samples drawn using different sampling strategies.
    - Check out the chapter on MIS from Erich Veach's thesis to learn more about how it works.
    - Include at least one image comparing MIS to next event estimation, and describe this comparison in your README.
- Depth of field **(10 points)**
    - Instead of generating rays from a single eye location, scatter the starting location over a lens (e.g. a disk). This produces an effect that mimics camera defocus blur.
    - The scene will be in focus only at the same depth away from the eye/lens as the virtual film plane--rays diverge as they move away from this plane, leading to defocus blur. You can control the location of the focus plane by changing the location of the virtual film plane.

    - Obviously, this is an approximation, and not a physically-based model of how cameras work. If you're interested in how you might do that, check out this paper on putting a simulated camera lens system into your path tracer.
- More advanced BRDFs **(10 points)**
    - There are many other types of BRDFs you could implement to get more interesting material appearance.
    - The Ward anisotropic BRDF and the Cook-Torrance microfacet model are just two possibilities.
- Image-based lighting **(15 points)**
    - Instead of using area lights to illuminate your scene, use a high-dynamic range hemisphere or cube-map as a light source. This allows you mimic real-world lighting environments.
    - Paul Debevec provides several light probe images which you can use for image-based lighting.
- Something else!
    - This list is not meant to be exhaustive--if you've got another advanced feature in mind, go for it! (though you may want to ask a TA / the instructor first if you're concerned about whether the idea is feasible)

Any extra features you implement must be described in your README. You must also submit images showing off each feature.

You can also get extra credit for sharing a scene you've created with the class **(2 points)**. Post the scene file(s) and a rendering of the scene to Piazza. If your path tracer is built on top of your own ray tracer and uses a different scene file format than the one used by the barebones ray tracer we provide, or if you've modified our scene file format in any way, you should also post instructions for how to read your scene file format.

Resources

Here is a barebones ray tracer that you can use to get started:

https://github.com/brown-cs-224/Path-Stencil

This includes all the code needed to load scene files, set up a scene, trace rays through pixels, compute intersections of rays with the scene, and output images.
If you already have your own ray tracer, you may still want to use (a) the scene file parser and (b) the bounding volume hierarchy (for fast ray-scene intersections) provided by this code.

This repository also contains a couple of simple example scenes, each of which contains a single mesh loaded from an .obj file. If you Google around for a bit, you can find a lot more meshes in .obj format to try and render.

Implementation & Debugging Tips

- There are a lot of different probability calculations that go into computing the contribution of a ray to the final image. Make sure you really understand all of this math before you start trying to implement anything. You don't want to get into the situation where your code is producing images that don't quite look right, and all you can do is resort to aimlessly tweaking parts of the code (e.g. fiddling with constants, flipping minus signs) to try and make them look right.
- Don't try to implement all the required features at once. Implement them one by one, thoroughly debugging as you go.
- Path tracers can take a long time to produce an image. The first thing you should do is make sure to compile your code in "Release" mode (or with all compiler optimizations enabled, if you're not using Qt Creator). To speed up your code-test-iterate cycle, you'll want to render low-resolution images (or, small windows cut out of a larger image that focus on some region of interest).
- If you're noticing 'speckling' artifacts, i.e. individual isolated pixels that look incorrect: try using an image editor to identify the coordinates of a problematic pixel (many image editors will display this information somewhere as you mouse over the image). Then, set a breakpoint in your code that only fires on that pixel, and use a debugger to step through and see what is going wrong.

Submission Instructions

Submit your assignment by running `cs224_handin path` from a CS department terminal. You should run the handin script from a directory containing all the files you wish to submit. This directory must include a file named 'README' for the submission to be accepted.

Post-Release Addendum

Various questions have come up about what to implement re: materials/BRDFs and light sources, and how to use the scene file format to do that. Here I'll attempt to clear up any remaining confusion.

<u>Required materials</u>

Your path tracer must support these 4 types of materials:

- Ideal diffuse
- Glossy specular

- Ideal specular (mirror)
- Dielectric refraction (refraction + Fresnel reflection)

You do not need to support any materials beyond this. For example, it's not necessary to handle materials that combine a diffuse and a glossy specular component (or other combinations of material types). You may do so if you wish, but it's not required—in your submission, we only expect to see images depicting the 4 types of materials above.

Getting materials from input files

The simplest way to specify materials for objects in your scenes is to specify per-object material properties in the .xml scene file. The CS 123 scene file specification (included in the Path-Stencil repository) lists these properties and shows how to use them in a scene file.

Our scene files load 'mesh' primitives from .obj files, and .obj files sometimes also come with an associated .mtl file. The .mtl file provides materials for the geometry in the .obj file at the per-face level (i.e. per triangle). If you want, you can have your code use these materials instead, when they are available. **You are not required to do this, though**—using only per-object materials is fine.

The latest revision of the starter code includes some commented-out example code showing how to read both types of materials (per-object and per-face) from a mesh that's been hit by a ray. See the PathTracer::traceRay function in pathtracer.cpp for these examples.

Interpreting materials from input files

The materials parsed from either the scene .xml file or an .mtl file are a single material struct/class that contains coefficients for diffuse, specular, reflective, etc. To convert such a material to one of the 4 required types of materials for this assignment, you can use whatever logic you like. For example, if you encounter a material with nonzero diffuse coefficient but zero for all other coefficients, a sensible thing to do might be to treat that material as an ideal diffuse reflector.

You do not need to worry about making your code handle all possible types of input materials. The only scene files your code needs to run on are the ones you design for it. So, it's perfectly fine to ignore the case when, say, a material has both nonzero diffuse coefficient and nonzero specular coefficient, since you are not required to implement materials that combine those two effects.

Scenes for submission

The assignment spec states that you need to demonstrate your path tracer on at least two different input scenes. It should go without saying, but you cannot use the provided example

scenes for this purpose. The example scenes are provided for debugging/sanity-checking purposes, and they are not good scenes for showing off global illumination effects (since they all depict a single object floating in space!) For one of the two required scenes, we'd recommend building a simple 'Cornell Box' type of scene.

Light sources

The assignment spec states that you need to support area light sources resulting in soft shadows. Any way of creating area light sources is fine. There are two strategies you might consider in particular:

As one option, you can use the area lights provided by the CS 123 scene file format. In the starter code, the Scene class has a method Scene::getLights() which returns a vector of lights. **You are not required to support any other type of light, e.g. point lights**, though the scene file format contains them.

As another option, you can treat objects with emissive materials as area light sources. The best place in the code to look through the scene objects for emissive ones is probably in Scene::parseTree, which constructs a vector of scene objects before building a BVH acceleration structure out of them.

For either option, you'll need to implement a method for sampling a point on an area light source (whether that be a rectangular area light or an emissive triangle mesh).