CS224: Interactive Computer Graphics

Photon Mapping

Due: 3/4/10, 11:59 PM

Ray Tracing	15
Photon Emission	25
Russian Roulette	15
Caustics	20
Diffuse interreflection	25
Extra Credit	+25
Total	**100**

This assignment is worth 12% of your final grade.

1 Introduction

In this assignment, we simulate light transport using a Monte Carlo ray tracer with Photon Mapping. The output is an image containing the following lighting effects:

- Mirror reflection
- Refraction
- Caustics
- Soft shadows
- Color bleeding

We use simplified versions of the algorithms described in:

Henrik Wann Jensen. Realistic Image Synthesis Using Photon Mapping. AK Peters. ISBN: 1568811470. July 2001.

Note: There are two parts to the hand-in for this assignment: some math you need to hand in during lecture and a program to submit electronically.

2 Requirements

- Recursive raytracing with reflection and refraction (backward tracing)
- Diffuse and caustic photon tracing (forward tracing) using rejection sampling and Russian roulette
- Combine the photon maps with the backward traced results to estimate the total radiance
- Soft shadows

Support code can be grabbed from: `/course/cs224/asgn/photon`. Be sure to grab all of the subdirectories as well.

There is a `-preview` commandline flag which can be used to view a crude OpenGL preview of a scene. Input parameters to photon mapping such as the number of diffuse photons to shoot, or the KNN gather radius, are all specified in the scenefiles themselves since a lot of these parameters are inherently scene-specific. Code to handle parsing these files and parameters has been provided for you.

We've provided Milton as support code which handles a lot of the aspects of the project which aren't core to the photon mapping algorithm itself. Feel free

to pick and choose what you want to use. The support code has built-in spatial acceleration for shape intersections (so rendering meshes should be fast) and for the K-nearest neighbors (KNN) lookup (PhotonMap).

In Monte Carlo ray tracing it's very easy to have incorrect code and still have the image look pretty good. Be sure to check and double check your probabilities because we'll be looking at them.

3 Overview of the Algorithm

For this assignment, surfaces have emissive properties and reflectivity properties (BSDF). A BSDF (the same thing as a BRDF but defined over the whole sphere) can either be perfectly specular, which will require special consideration, or have some other, arbitrary distribution. This distribution may be constant (perfectly diffuse) or skewed, but the specifics have been abstracted away by a generic BSDF sampling interface that you won't have to worry much about. We ignore participating media and subsurface scattering. Note that under these limitations, the incoming photon direction and type of vertex entirely determine the outgoing direction, and vice versa. This property makes it easy to trace photon paths both forward and backward.

3.1 Paths Simulated Forward from the Light

Direct Caustics: LS^+D
Light focused by specular bounces will collect on a diffuse emitter and be reflected to the eye. This does not account for indirect caustics, for example, the caustic formed beneath a lens after light bounces off a white reflector.

These are traced forward from the light sources towards specular surfaces and are stored in the *caustics photon map*.

Interreflection: $(L(S|D)^*D)^? - LS^*D$
This is the set of all paths that are not direct caustics or direct illumination and terminate on a diffuse bounce. It gives rise to all global illumination effects, including color bleeding, except for reflections, refraction, direct caustics, and direct illumination.

These are traced forward from the light sources and are stored in the *diffuse photon map*.

3.2 Paths Simulated Backward from the Eye

Reflection and Refraction: $(LS^*E)|(DS^*E)$
Note that this also includes the trivial subpath DE, which links the forward-

traced paths above to the backward traced paths.

Direct Illumination: $LD^?E$
Direct specular or diffuse illumination is easy to compute, so we calculate it using direct application of the BRDF, taking shadows into account, just as in a simple ray tracer.

4 Backwards Path Tracing (a.k.a. Ray Tracing)

We're assuming you're familiar with recursive ray tracing. If you're not, read CS123's lecture slides or pick up a copy of Foley et al. Backward tracing computes the paths denoted by $[(LS^*E)|(DS^*E)] \mid (LD^?E)$. This shouldn't take you more than a few hours with the methods `Milton` provides.

Your main implementation will be in the `PhotonMapper` class. `_evaluate` will be called for every pixel in the image, and it is your job to calculate and return the proper radiance value along the given primary ray. Included in the skeleton code we give you is a fully working, albeit basic ray tracer, including direct illumination.

Check out `BSDF::isSpecular()` to differentiate whether a particular `SurfacePoint` lies on a specular or non-specular surface. Also note `SurfacePoint::bsdf`.

5 Forward Path Tracing

When the program runs, it first fills the photon maps by forward tracing photons. For this we need a representation of a photon, or, rather, since Jensen's work does not simulate actual photons but statistical groups of photons, we need a representation for groups of photons. Each "photon" encodes the incident direction, position, and a color value. The color is a statistical measure of the number of photons in the group at certain wavelengths. We use a similar data structure to the one that Jensen provides in this book. Milton abstracts out colors into a `SpectralSampleSet` class, representing any type of wavelength-dependent value, sampled at `n` distinct wavelengths. You can pretty much assume if you want for the purposes of this project that `SpectralSampleSet` is the same as a "color3" with samples taken at canonical R, G, and B wavelengths.

A `Photon` struct is defined for you in the support code; if you change it, you may need to change the given `PhotonMap` implementation which uses the Photon struct.

We provide you the `PhotonMap` structure for speeding up the KNN queries. You can see the methods we've provided for you in PhotonMap.h. Be sure to call `init` after you are done shooting photons. This method will create the Kd-Tree structure. **The queries will not work if you don't do this.** Note: this is

a modified version of the PhotonMap implementation that Jensen gives in his book, extended to support thread-safe KNN queries.

6 Computing the Radiance Estimate

You can ask for the KNN neighbors in the photon map by calling getKNN. On return it will fill the output parameter with the nearest photons and the gather radius. The photon maps are created in the constructor of App. The first parameter is the maximum number of photons to search for. The second parameter is the maximum radius. If the number of photons found is greater than the maximum requested, getKNN will return the smallest radius that contains all the photons. For example, if you asked for 100 photons in a radius of 1 and the KNN structure had 200 photons within a radius of 1, on return, it will return the radius which contains the **closest** 100 photons (i.e .5).

7 Visualizing the Photon Map

The PhotonKdMap structure can be visualized by saving it out to a file. You can save the Photons out to a file at any time by calling the `save` method and passing it the output filename, the camera, and an output Viewport. This should be very useful for debugging. You should see the scene saved out with lots of dots where the photons hit.

8 Test Scenes

We have a few test scenes availible for you that come with the support code distribution. They are in

`/course/cs224/asgn/photon/scenes`

Many are variations of the cornell box. When running the program, you may pass in the name of a scenefile, or leave it blank, and Milton will default to loading a file named `default.js` in the current directory.

For example, if I wanted to run it on the cornell box scene, I would do

```
myprogram /course/cs224/asgn/photon/scenes/cornellBox.js
```

9 Extra Credit

- Tone mapping

- Distributed ray tracing
- Anisotropic reflection
- Participating media
- Sub-surface scattering
- More than three wavelengths
- Shadow photons
- Irradiance caching
- Motion blur
- Cool scenes (provided that you share them)
- Anything else you come up with!

Supersampling isn't extra credit in CS224 :)

10 Handing In

Besides the usual README stuff (compiling instructions, bugs, extra credit, usage, etc. etc.), please identify where your code is for the following:

- Soft shadows
- Photon emission rejection sampling
- Photon bounce Russian roulette

Also give an explanation of why your program does not double-count any photons.

We will be testing your program on several cornell box scenes, in addition to our own, secret test scenes. You're encouraged to make your own and share them with others – scene authors will receive community service points. Everyone benefits from awesome scene files!

Do not hard code any paths in your program! All loading and saving should be based purely on arguments specified on the command line.

`/course/cs224/bin/cs224_handin photon`

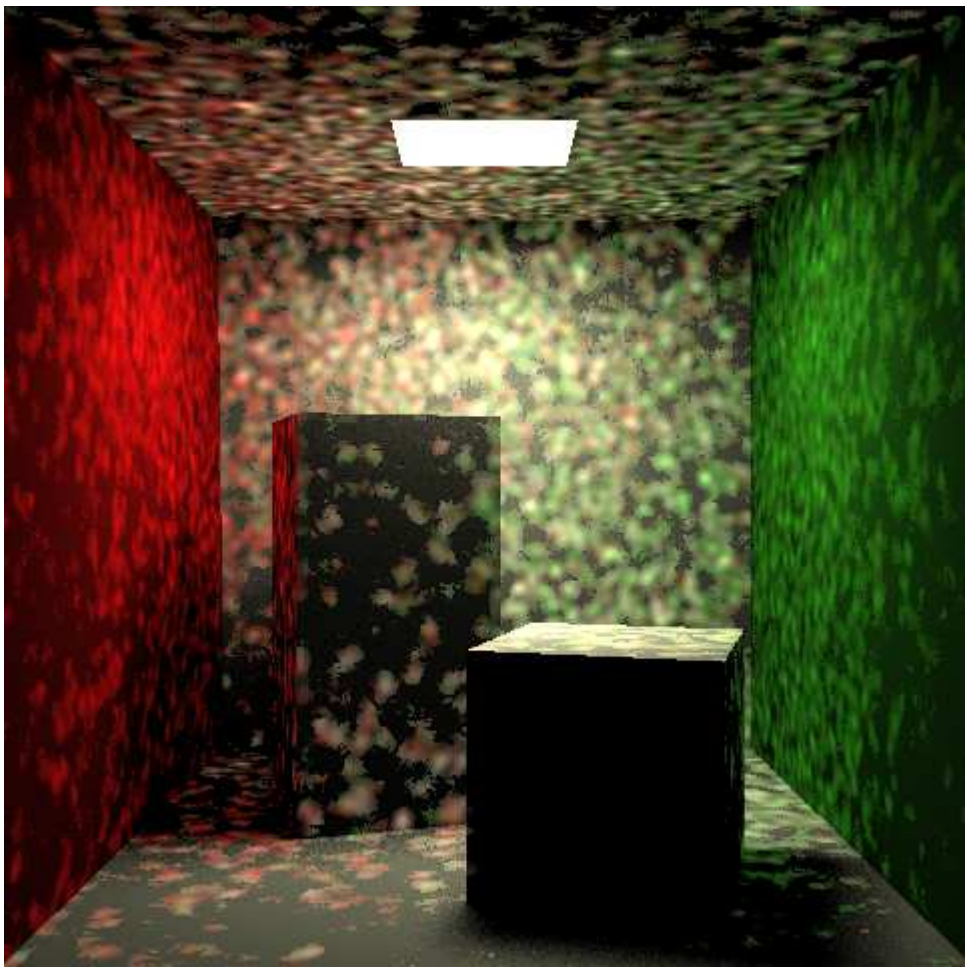
11 Support Code Notes

Just like in RenderWarmup, you'll be compiling an executable and your own renderer plugin that is loaded by Milton dynamically at run-time. Take a look at `plugin/README` for the order that we suggest you to fill things in.

Milton support code documentation can be found at
`http://milton.mjacobs.net/docs/doxygen/index.php`,
and Milton scenefile documentation can be found at
`http://milton.mjacobs.net/docs/scenefile/index.php`.

12 Closing Remarks

"There's a party in my Cornell Box, and you're invited" – Spike



No, this isn't a disco; it comes from a perfectly correct photon mapping implementation. One problem with photon mapping it that to produce good results, its inputs may have to change dramatically from scene to scene. What we're visualizing here (and what you'll undoubtedly run into at some point), is an example of the characteristic blotchiness that occurs during photon mapping when either too few photons or too small of a KNN search radius is used.