Photon Mapping

Due: 3/24/05, 11:59 PM
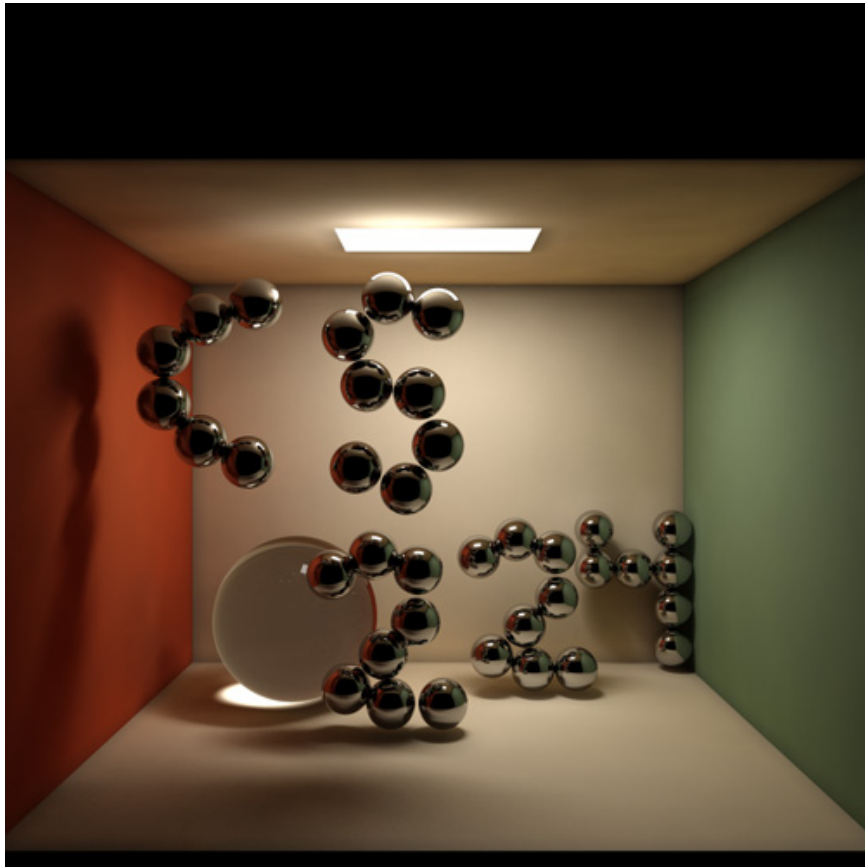
Math Homework	20
Ray Tracing	20
Photon Emission	10
Russian Roulette	10
Caustics	15
Diffuse interreflection	15
Soft Shadows	10
Extra Credit	+25
Total	**100**

This assignment is worth 10% of your final grade.

---

[1]Yes – your results can and should look this good :) (If you render your files for 6 days)

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

- A real-time preview mode[2]. By passing an optional flag on the command line your program should render the scene in 3D giving you a very crude approximation to what the scene will look like.

Your program should take five optional arguments.

`--preview`: The real-time preview mode described above.
`--raytrace`: Ray tracying only (no photons).
`--radius r`: The size of the photon gathering radius.
`--diffuse n`: The number of diffuse photons to store.

---

[2]Check out `Mesh::render` in the sample code for some help

--`caustic n`: The number of caustic photons to store.

We've provide some sample code for loading the scenes and basic intersection – feel free to use it or not.

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 probilities because we'll be looking at them.

Important: Use G3D 6.02 in this assignment. Just change the 6_00 in all of your variables to 6_02.

3 Overview of the Algorithm

For this assignment, we limit ourselves to surfaces where the BRDF is the sum of perfectly diffuse, perfectly reflective specular, perfectly refractive specular, and emissive terms. That is, there is no anisotropy, subsurface scattering, or glossy reflection. We also ignore participating media. 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 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 `G3D` provides.

Your ray tracer should be a `G3D::GApplet` just like your forward tracer. Have `doSimulation()` cast a decent number of rays and store the results in the `G3D::GImage` owned by the `G3D::GApp` subclass. Then `doGraphics()` can draw the buffer with `G3D::Draw::fullScreenImage`. By repeating this process the user can see the results incrementally. The interesting part of the image is usually at the bottom of the image so cast rays from the bottom right up to the top left to decrease debugging time.

Use `G3D::GCamera::worldRay` to generate the initial rays. The methods of `G3D::CollisionDetection`, `G3D::Triangle`, and `G3D::Sphere` will handle your intersections. You'll need the intersection code for forward tracing too, don't forget. There's an intersection example in the sample code, `Mesh::getFirstIntersection()`.

When a ray hits a surface, compute direct diffuse illumination using Lambertian shading. Because we have area light sources, you need to cast multiple shadow rays at the light to approximate the projected area of the light at the point.

Check out `G3D::Vector3::reflectionDirection()` and `G3D::Vector3::refractionDirection()` for all of your reflection and refraction needs.

Even though photon shooting happens before ray tracing, you should code the ray tracer first because it will help you debug your photon tracing.

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 three wavelengths. Jensen uses a balanced kd-tree data structure for his photon map but we'll be using an axis-aligned BSP tree, `G3D::AABSPTree`, which is slightly more efficient. To work with the data structure, your photon representation (call it `Photon`) must define the following:

```
Photon::Photon();
void ::getBounds(const Photon&, G3D::AABox&);
bool ::operator==(const Photon&, const Photon&);
unsigned int ::hashCode(const Photon&);
```

This is done for you in the support code; if you change it, you need to change these methods.

The `G3D::AABSPTree` is designed for objects that may have non-zero volume and extent. For a photon, the bounds are just the photon's position.

Once all of the photons are in the `G3D::AABSPTree`, don't forget to balance it for a big speed improvement when doing lookups later.

Make sure that your emission probabilities and Russian Roulette sampling are correct and clearly commented – you will be graded very critically on those!

Your forward tracer should be a `G3D::GApplet` inside of a `G3D::GApp` subclass that contains the scene and pixel buffer.

6   Computing the Radiance Estimate

Instead of using a constant number of photons when doing the estimate, we take all photons within a constant radius. For example, let $r = 5$cm. Call `G3D::AABSPTree::getIntersectingMembers` with a box that has a width of 10cm and the reject all photons outside of a 5cm radius or any photon that "faces away" from the normal at the surface.

7   Extra Credit

- Tone mapping

- Distributed ray tracing

- Anisotropic reflection

- Participating media

- Sub-surface scattering

- Spherical light sources

- More than three wavelengths

- Rasterize diffuse illumination into lightmaps (a la Quake 3) for real-time rendering

- 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 :)

8 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 the Cornell box, cow scene, and 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. If you put your scenes in a directory called `res` then they will be copied to `distrib` for you.

`/course/cs224/bin/cs224_handin photon`

9   Appendix: Loading the Scenes

There are two different scene formats. The *text* format is easy for you to edit by hand to make your own test cases and see the input data. The *binary* format is more efficient for large scenes.

9.1   Text Format

The file can be read using the `G3D::TextInput` class. C++ and C-style comments and whitespace are ignored. The format is:

numLightMaps = %d
// for each light map:
// Light map number, quoted filename
%d ,"%s",

numBRDFs = %d
// for each BRDF:
// Diffuse * 100, specular, transmissive, emissive
0x%x , 0x%x , 0x%x , 0x%x ,

numTris = %d
// for each triangle:
// Triangle number, light map index, BRDF index,
// vertex 0, light map vertex 0, vertex 1, light map vertex 1, vertex2,
// light map vertex 2
%d , %d , %d ,
(%f , %f , %f), (%f , %f),
(%f , %f , %f), (%f , %f),
(%f , %f , %f), (%f , %f),

numBalls = %d
// for each ball:
// sphere number, brdf, index-of-refraction, center, radius,
%d , %d , %f , (%f , %f , %f), %f ,

numCameras = %d
// for each camera:
// camera number, position, lookAt
%d , (%f ,%f ,%f), (%f , %f , %f),

We avoid the words "triangle" and "sphere" to distinguish between the geometric shape (e.g. `G3D::Sphere`) and the surface type (e.g. `TriMesh::Ball`).

The light map information is used only for the extra credit portion of the assignment. Ball BRDFs are required to have zero emissive and zero diffuse reflectivity and never have light maps.

9.2 Binary Format

The binary format is rougly the same as the text format. The specifics aren't that important because the format isn't

9.3 Units

All positions are in meters. Reflectivities are unitless. The diffuse reflectivity has been premultiplied by 255 to make it fit the range [0, 255] along each channel. Divide this out while loading, and divide by `G3D_PI` (the integral of $\cos(\theta)$ over a hemisphere) to convert the reflectivity to a diffuse BRDF value. Note that constructing a `Color3` from a `Color3uint8` automatically divides by 255.

The specular reflectivity and transmissivity have also been premultiplied by 255 to make them fit the range [0, 255]. This factor must be divided out while loading. The diffuse + specular + transmissive value of the BRDF must not exceed 1.0 along any color channel (this is a physical constraint).

The emissivity is in radiosity units of W/m^2 and has been multiplied by 5. Divide by 5 * `G3D_PI` to get a radiance value.

All of this math is already done for you in the sample code. If you write your own parser, be sure to follow the above steps.

The binary format is already prescaled. In other words, you don't need to do any of the above for the binary files.