CS224: Interactive Computer Graphics

Rendering Warmup

Due: Thursday 2/18, 11:59 PM

1 Introduction

In this warmup assignment, we will explore the basics of physically-based rendering. There are many different rendering algorithms that attempt to solve the rendering equation Spike presented in class and they are usually very effective at accurately reproducing a specific subset of physical phenomenon. In many of these algorithms, there is little attention paid to the physical units of the scene specification[1] and they often use other hacks that have no physical justification.

In this assignment, your job is to create the scene we've specified, paying attention to the physical units and compute the radiance at every pixel seen by the camera. It will be very important that you understand the rendering equation and what value you are trying to compute. The scene we've specified is extremely simple and the focus should be on getting the physically correct values[2].

If you want to review what Spike covered in class, there are a few good sources on the course website (under docs).

2 Scene

The scene consists of a single area light, a perfectly diffuse (lambertian) surface and a pinhole camera. Spike's physical measurements can be found here:

`/course/cs224/data/DiffusePhotos`

2.1 Geometry

The scene has a perfectly lambertian surface lying on the $y = 0$ plane. It is perfectly diffuse and has the BRDF $\rho(\omega^o, \omega^i) = .5/\pi$ for all p on the plane. The normal at all points on the surface is $(0, 1, 0)$.

[1]What does a light that is (240, 220, 220) with brightness "20" mean?

[2]You do not need to implement or design for any functionality that goes beyond rendering this specific scene (handling arbitrary geometry, reading in scene formats, shadows, specular, reflection highlights, etc). The emphasis is entirely on numerically estimating the result of the rendering equation.

2.2 Light

The light in the scene is a single-sided, circular light with radius 1.75 cm. The light is 26.35 cm above the plane with a normal at every point $(0, -1, 0)$. It emits light uniformly in all directions and all points on the surface. It emits energy uniformly in the visible spectrum at 1.2W.

2.3 Camera

The camera is 1m above the surface with a look vector (0, -1, 0) and an up vector (1, 0, 0). A camera model has been provided for you and example usage is given in (`MyRenderer::sample`).

3 Requirements

There will be a written and coding component for this assignment.

In the written component (include in your README), we would like you to:

- Compute the radiance coming from the center of the square going through the center of the film by computing the exact solution from the integral.
- If you had to compute the same integral, but from a point not directly situated under the light (the point (-1, 0, -1), for example), what specifically would have to change in your calculations?
- Comment on how closely your rendered results matched your exact solutions.
- Write down the exact formula you used to estimate the radiance and justify why each term should be there. (i.e. this is the dot product between the light vector and the surface normal).
- In Matlab, plot the radiance at each point of your computed image and compare your radiance distribution to the photograph we've provided. In your README discuss how closely they match.

In the coding component:

- Fill in the function sample in MyRenderer.cpp to estimate the radiance at every pixel by approximating the integral using ray casting. How does the image quality change if you use few samples to estimate the radiance arriving at the surface vs many samples? How many samples do you need before you stop noticing the problems? Can you give any intuition, quantitatively, to explain the relation between variance and number of samples?

4 Support Code

Part of the purpose of this project is to familiarize yourself with the C++ support code we have for rendering. You will be using this same library for Photon Mapping. We're using a new rendering library named Milton written by Travis Fischer for this assignment. We will be using a very small subset of what it provides since a lot of its functionality is tailored towards more complex renderering tasks.

You should make good use of the Milton documentation at `http://milton.mjacobs.net/docs/doxygen/index.php`.

The classes that you will need to use for this assignment are Ray and Vector3 (and possibly Camera and Viewport, though you shouldn't have to touch them). The stencil code will automatically save your radiance values to `out.radiance`. If you would like to save your image (as seen on the canavs), take a look at the `Image` class or you may save your results from the built-in Gui. After 'make'ing the support code we give you and running the resulting executable, you should see a noisy black and white image; the skeleton code is setup by default to set the radiance for a pixel to a random value inbetween 0 and 1. You should change this functionality to instead compute the correct, meaningful radiance given the scene description above.

We have also provided you a matlab function that will read the radiance and produce an (width x height) array. The function is `ReadRadiance` and you should pass it the file.

4.1 stencil

You can find the stencil code for this assignment in `/course/cs224/asgn/RenderWarmup`. We have taken care of going from screen space to world space and looping through all of image pixels. The only function you should have to change is `sample`. The `sample` function is expected to return the radiance at that pixel in the out-parameter 'outSample'. The support code is setup to compile a dynamic plugin and accompanying executable that interfaces with the Milton framework. The (public) source to Milton is available in `/course/cs224/lib/milton/include`. There are some parts of the source that are not currently public (such as its photon mapping implementation).

5 Suggestions

- As you learned in HW2, anytime you take samples nonuniformly in your sample domain, you will need to compensate for this fact. It is not always obvious how to adjust for this and you should be very careful if you are sampling nonuniformly. In many cases they will produce nice looking but incorrect results.
- Always make sure you know what physical value you are computing at each step. Writing down the units will help a lot.

6 Handing In

As usual, include the standard README(compiling instructions, bugs, usage, etc.). Make sure your readme answers the questions in the requirements.

```
/course/cs224/bin/cs224_handin renderWarmup
```