# CSci 5608: Programming Assignment #3

due: 9pm, Monday, March 23

Rendering requires the computation of many different types of integrals. In this assignment you will take a close look at how **pbrt** computes the radiance reflected from a surface in a certain direction by integrating the surface's reflection function (BRDF) against incoming light from all directions.

Because these sorts of integrals generally don't have analytic solutions, **pbrt** numerically estimates their value by Monte Carlo integration. As you are now well aware, the variance of a Monte Carlo estimator manifests itself as noise in a rendered image. It is therefore very important to the quality of your rendered image to choose samples from the domain of integration such that variance in the integral estimate is minimized. *Importance sampling* is a variance reduction technique that draws samples using a distribution that has the same shape as the function being integrated. This means that we are more likely to choose samples which will make a large contribution to our estimate. In this assignment you will explore various approaches for sampling irradiance on scene objects due to an infinite area light source (environment light) and explain why and when certain approaches are preferable to others.

**Background Reading**

Read Chapters 12 (especially 12.5), 13 (especially 13.3) and 14.1-14.2 (especially 14.2.4) in the **pbrt** book. This assignment will require a solid understanding of importance sampling, so please read these chapters carefully.

*Important Notice:* Older versions of the **pbrt** book have a typo in the multiple importance sampling equation on page 798 (page 676 in the first edition). The code in **core/transport.cpp** is correct, but the equation given in the first edition of the book is wrong. Instead of multiplying both summations by the factor $1/(n_f + n_g)$, the first summation (sum over samples drawn from a PDF from $f$) should be scaled by $1/n_f$. The second summation should be scaled by $1/n_g$.

**Part 1 – Basic Monte Carlo Integration**

The chart below was presented in lecture as an example of how *not* to do importance sampling. Each line of the chart depicts how the choice of a different PDF impacts the calculation, using the Monte Carlo technique, of the integral shown. To answer this question you should first draw a graph that plots the integrand and each PDF over the interval of integration. You can create this diagram any way that you want, including drawing it on paper and scanning it into your computer. Next you are to write a small program that determines, for each choice of PDF, the number of samples necessary to calculate the integral to within 0.008 (last column in the chart). Produce your own table that shows the number of samples that were required in each case. Briefly discuss your results.

The program can be written in any language you like, such as python, Java or C/C++. Please document the build and usage instructions in a file called **`README1`**. If using a compiled language, please include a makefile or explicit command-line instructions to build it (if it's simple enough). Document how to invoke the program as well. Name your program **`part1.*`** where the extension depends on your language.

$$I = \int_0^4 x dx = 8$$

| method | sampling function | variance | samples needed for standard error of 0.008 |
|---|---|---|---|
| importance | (6-x)/16 | 56.8 | 887,500 |
| importance | 1/4 | 21.3 | 332,812 |
| importance | (x+2)/16 | 6.4 | 98,432 |
| importance | x/8 | 0 | 1 |
| | | | |

**Part 2 – Sampling Reflectance Functions**

In this part of the assignment you will produce a visualization that shows how the shape of a reflectance function affects the distribution of samples that should be taken when using that function to model how light reflects from a surface. To accomplish this you will produce a **`.obj`** file that can be read into Blender and that contains the graphics for your visualization.* To help you do this we have created an example file (**`base.obj`**) that contains a surface plane, a surface normal, a lighting vector, and the mirror reflection vector. You can use this file as the starting point for creating your own sampling pattern visualization. If you want to go further, consult the documentation provided for the **`.obj`** file format.

Produce a visualization that shows, for the Phong reflection model, how the sampling pattern for the specular lobe varies with angle of incidence, lobe exponent, and number of samples. Your **`.obj`** visualization file should be generated by a program that you write. It should include the graphics provided in the sample .**`obj`** file (surface plane, surface normal, lighting vector, and mirror reflection vector) and, in addition, a set of vectors that

* To import the **`.obj`** file into blender to visualize, go to File -> Import -> Wavefront (obj). Navigate to your output **`.obj`** file. However, before you finish the import, make sure that the Lines checkbox on the right is checked and that the Forward drop down has "Y Forward" selected. When you do this, it should automatically select "Z Up" in the Up drop down. By default it is set to -Z Forward. If you skip this, the model will be rotated oddly compared with blender's coordinate frame.

depict the direction in which the samples would be taken for the requested incidence angle, lobe exponent, and number of samples. Turn in a **.obj** file for the case where the angle of incidence is 45 degrees, the lobe exponent is 75, and the number of samples is 25.

As in Part 1, the program can be in any language you wish and must include instructions to build and run it (in a file called **README2**) so that we can test it. Please be clear in describing the command-line arguments. Name your program **part2.***.

**Part 3 – Sampling Efficiency**[†]

In this part of the assignment you will evaluate the efficiency of different sampling strategies for Monte-Carlo integration. While there is very little coding in this part of the assignment, you will probably need to script **pbrt** to collect the right data. The data collection itself may take some time, so plan ahead to make sure you have time to complete rendering. The necessary scene files and reference images are available on the course website.

*Sampling and Variance*

Variance of a random variable $X$ is defined as $E[(X - E[X])^2]$. To compute the variance of an image rendered by **pbrt** we can compare it, pixel by pixel, to a reference image that we consider to be ground truth for $E[X]$. We've included three EXR images that will serve as ground truth for this assignment (**occluder_ref.exr**, **occluder_shadow_ref.exr**, and **manykilleroos_ref.exr**). These scenes were rendered with 16384 samples per pixel to ensure that they are very close to the correct image. To measure the variance of any image you render, you can use the **exrdiff** program. This program can be added to your **pbrt** code by placing

**https://github.com/mmp/pbrt-v2/blob/master/src/tools/exrdiff.cpp**

in **pbrt-v3/src/tools/**, and adding:

```
ADD_EXECUTABLE ( exrdiff src/tools/exrdiff.cpp )
TARGET_LINK_LIBRARIES ( exrdiff ${OPENEXR_LIBS} )
ADD_SANITIZERS ( exrdiff )
```

to **pbrt-v3/CMakeLists.txt**, around line 546. After rebuilding, it can be ran with:

```
exrdiff image_ref.exr yourimage.exr
```

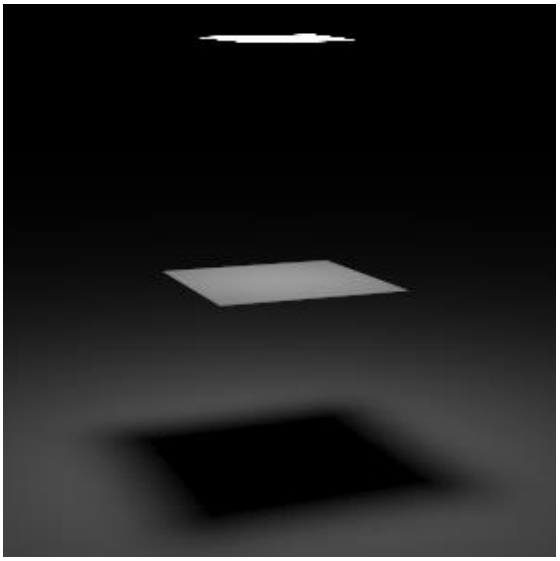
This will print something like:

```
Images differ: 173884 big (64.40%), 179244 small (66.39%)
avg 1 = 0.351801, avg2 = 0.361679 (-2.807667% delta) MSE =
0.126821
```

[†] Part 3 of this homework was adapted from a Stanford CS 348B assignment, taught in 2014 by Matt Pharr.

Look at the code for **`exrdiff`** and convince yourself that the MSE (mean-squared error) value it calculates is equivalent to the variance. You may need to modify **`exrdiff`** to ensure it always prints the MSE value regardless of how close the two images are.

As we increase the number of samples taken, we expect the variance to go down. We will measure this change empirically by modifying the **`pixelsamples`** variable in the scene files. The file **`occluder.pbrt`** contains a scene with a single area-light, a square occluder, and a large ground plane:



Render this scene varying the number of samples from 1 to 64, and compute the variance of each image. You only need to measure powers of 2 since many of the samplers in **`pbrt`** will round this number to a power of 2 anyway.

The scene file is set up to use a low-discrepancy sampler which tries to generate well-distributed non-uniform points such that no two points are close together. Let's compare this to a purely random sampler, by changing “lowdiscrepancy” to “random” in the scene file, and finding the variance for each image.

*Deliverables*

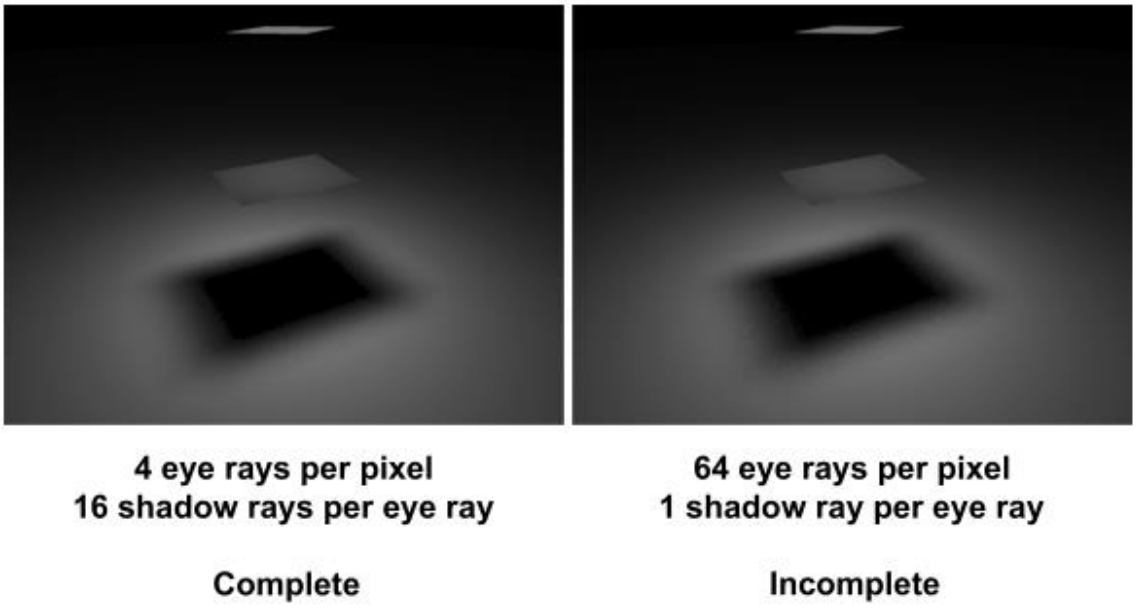
First, submit a graph showing the variance for both the random and low-discrepancy samplers as you increase **`pixelsamples`** from 1 to 64 for the **`occluder.pbrt`** scene. You will probably want to use log scale for both axes since you are only considering powers of 2. You should also answer the following questions:

1. When using Monte-Carlo sampling with uniformly sampled random variables, how does the variance change as a function of the number of samples taken, $N$? Does the data you collected reflect this behavior?

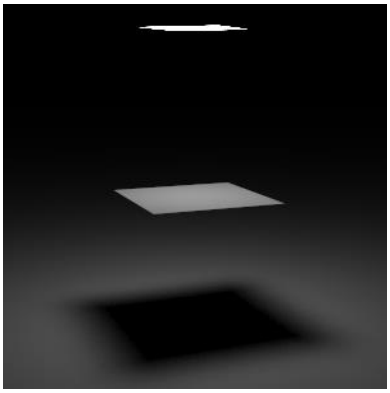
2. How does the low-discrepancy sampler perform in comparison to random sampling? Why does distributing the samples evenly through space change the variance in comparison to random sampling?

*Evaluating Efficiency*

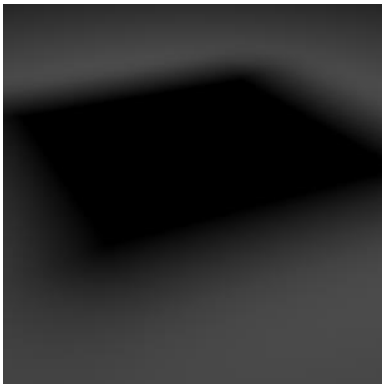
There are many different strategies for sampling the value for a pixel. In one variation, we compute an intersection with the scene and a ray, and then evaluate the radiance along the ray using $N$ samples from each area light. In another variation, rather than taking multiple samples from each light, we take only a single sample ($N = 1$), and compensate by increasing the number of samples taken per pixel.



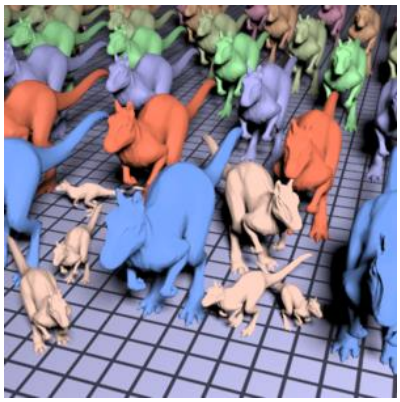

Your task for this step is to compare the efficiency of these approaches on the three scenes we have included: **`occluder.pbrt`**, **`occluder_shadow.pbrt`** (a different view of the first scene), and **`manykilleroos.pbrt`**:



**`occluder.pbrt`**



**`occluder_shadow.pbrt`**



**manykilleroos.pbrt**

You should consider strategies from $N = 1$ sample per light to $N = 64$ samples, again in powers of 2. One strategy is considered more efficient than another if, given a fixed amount of time, it can compute an image with less variance than the image produce by the other strategy (alternatively, given a target variance, it is more efficient if it can compute the image in less time).

Evaluate each strategy (1 to 64 samples per light) by rendering images for a range of pixel samples (1 to 64 samples per pixel). For each image, you should measure the amount of time it takes to render, as well as the variance of the result. To limit the number of scenes you have to render, you can limit yourself to scenes where the [number of light samples] x [number of pixel samples] = 64. You should only time the rendering of the scene, not the setup costs. This is particular important for the **`manykilleroos.pbrt`** scene, which has complex geometry.

*Deliverables*

For each scene, you should construct a scatter plot with time on the x-axis and variance on the y-axis. Each point in the plot should correspond to one of the images you generated (i.e. an image with a specific sampling strategy and number of samples per pixel). Furthermore, you should answer the following questions:

1. For each scene, which approach(s) (e.g 1 light sample per pixel sample, 2 light samples per pixel sample, …) perform(s) most efficiently. Why do you think that is the case? Backup your answer using data from the graphs you have generated.

2. Is the same approach best across all the scenes? Why or why not? Are any approaches particularly bad? Why?

3. What characteristics in the scene make the path tracing ($N = 1$) method more efficient? What characteristics make strategies with a large number of light samples more efficient?

**Submission and Grading**

Please turn in the program and documentation that you wrote for Part 1 (**`part1.*`**, **`README1`** instructions, and **`part1.pdf`**), the program and documentation that you wrote for Part 2 (**`part2.*`**, **`README2`** instructions, and **`part2.pdf`**), the visualization file that you created for Part 2 (**`part2.obj`**), and a document that contains your answers to the questions asked in Part 3 (**`part3.pdf`**). Points for this assignment will be divided up as follows:

30% - Part 1
30% - Part 2
20% - Part 3: Sampling and Variance
20% - Part 3: Evaluating Efficiency