# Graphics Final Report—Pathtracer

Kevin Song and Aditya Durvasula

May 5, 2018

## 1 Basic of Path Tracing

Recall that the ultimate goal of almost all raytracing techniques is to simulate the rendering equation, which is given by

$$L_{out}(\theta_r, \phi_r) = \int_{\theta_i} \int_{\phi_i} f_r(\theta_r, \phi_r, \theta_i, \phi_i) L_{in}(\theta_i, \phi_i) \cos(\theta_i) \tag{1}$$

where $f_r$ is the BRDF, encoding the properties of the material.

If we could find the solution for this equation directly, most of the rendering problem would be solved. The problem with finding a solution is that the values for $L_{in}$ rely on the amount of light coming off of the surrounding materials. To find the amount of light coming off of the surrounding materials, we need to solve the rendering equation for those surfaces. Unfortunately, this means solving even more copies of the rendering equation, and. . . .

The Whitted illumination model used in the "standard" raytracer solves this by assuming point lights and direct illumination, i.e. only light that arrives directly from the light contributes to $L_{in}$. While this is fast, it removes things such as soft shadows, reflections, and specular highlights. While some of these can be added back in, a truly globally correct illumnation model can only be implemented with better calculations of the $L_{in}$ term.

*Path tracing* attempts to solve the $L_{in}$ term in a global manner: it attempts to integrate over the actual $L_{in}$ function by taking samples and averaging their contribution. In essence, we make the following approximation:

$$\int_{v \in \text{hemisphere}} L_{in}(v) \approx \frac{1}{n} \sum_{i=0}^{n} L_{in}(v_i) \tag{2}$$

where the $v_i$ are sampled randomly in the hemisphere.

We do a similar sampling trick to get soft shadows: instead of firing a single shadow ray from the intersection towards a light, we randomly sample points on the light, then fire many shadow rays from the intersection to the light. The actual value of the shadow at that point is the average of the shadow samples.

## 2 Implementation

Our implementation uses a naïve MC sampler to integrate the rendering equation: a pair of random numbers on $[0, 1)$ is turned into a random vector on the unit sphere. This vector is then used to sample the hemisphere and the contributions are averaged.

Interestingly, when we first implemented this, GLM's `sphericalRand()` was proving to be a bottleneck. We implemented the sampler ourselves, only to find that over 90% of the program's runtime was being spent in calls to the C++ random device. Given that this implementation uses an LCG, which is generally considered the gold standard of fast, medicore-quality randomness, this was very surprising. We eventually implemented our own fast, multithreaded XorShift32 generator, which knocked random vector generation down to 60% of the program's runtime.

Our soft shadow implementation is similar to what is found in the extra credit for this project–a certain number of samples are taken on the light, and shadow rays are fired from the intersection to the light.

Other acceleration tricks in this code include spatial data structures (a semirandomized kd-tree), parallelization with OpenMP's dynamic scheduling, and an adaptive sampling scheme which takes fewer samples when the contribution of the samples is lower.

Area lights are specified to the program in one of two ways: either by using a CMake flag which forces all lights to behave as area lights with radius 1, or by using a new area_lights specification in a ray file, which allows selective area lighting.