Final Report

To implement photo-realistic, physics-based rendering of scenes is a complex, difficult task that remains unsolved - it is impossible to accurately emulate the physical transport of light completely. Ray-tracing tackles this problem, simulating rays of light shot out from a camera and tracing the directions they take as they collide, absorb, and reflect / refract into objects. In our introductory ray-tracing project, we implement a form of ray tracing known as Whitted-style, in which we trace singular rays that are shot from the camera, recursively determining shadow, reflection, and refraction rays that are continued by the properties of materials. For our final project, we improve upon this system by implementing path tracing, a technique that involves stochastically tracing multiple rays and averaging their contributions to determine the coloring of a pixel. Our goal is to get an accurate modeling of different materials and their light transport, in order to be able to render a variety of scenes that we control.

$$L(x \to v) = E(x \to v) + \int_{\Omega} f(x, \omega \to v) L(x \leftarrow \omega) \cos(\theta_{\omega})\, d\omega$$

Path tracing is built upon the rendering equation depicted above, for a ray that is shot from an object $\boldsymbol{x}$ towards the camera $\boldsymbol{v}$ (Appendix I). We first calculate the amount of light that $\boldsymbol{x}$ emits towards $\boldsymbol{v}$. Then, we combine that with the integral over the hemisphere of the product of the BSDF (a function that encodes the light transport properties over a material), incoming light towards that object, and a cosine term to attenuate how much light it contributes. To approximate the integral, we use Monte Carlo integration - we stochastically sample the hemisphere of $\boldsymbol{x}$, based on the BSDF function given inputs of incoming ray direction and normal, then average the contributions together to form the equation for our path tracer, noting to only trace a maximum of $\boldsymbol{N}$ rays at each depth level (Appendix II).

To terminate our paths, we employ the technique of dynamic Russian Roulette. We guarantee that each ray is able to bounce for at least 3 times, but past that, we set a probability for a ray to survive - if it fails this random check, we terminate it. For paths that continue, we weigh it more heavily in our eventual contribution to compensate for its rarity. We dynamically choose this probability for each bounce, depending on the possible color contribution, in order to avoid “fireflies” caused by biases in choosing, or rare events of pixels with atypical coloring.

Light sources in our implementation are primarily area lights, providing emissive light over an area. To calculate the contribution of a light, we randomly sample a point on it and divide it by the probability of choosing that point. For scenes with multiple light sources, we also implement a uniform sampling, where on each bounce we only choose one light source and divide by the probability of that choice for the sample to be unbiased, for optimized rendering of scenes.

Finally, we implement a BSDF (Bidirectional Scattering Distribution Function) to attenuate light’s transportation between rays for different materials. For pure diffuse (Lambertian) materials, this is simple - we use an albedo vector divided by pi, and uniformly sample the hemisphere for recursive rays. For more complex materials, which can be a combination of refraction and reflection (specularity), we read from material the probabilities of where further samples can occur (diffuse, refractive, or reflective), and sample one to do, with result divided by the probability so the total expected light should be unbiased.

Implementation Details

For our path tracer, we made the decision to build upon preexisting code from our ray tracer, which already includes a lot of mathematical support for tracing rays and intersections, as well as a decent framework for materials.

First, to implement area lights, we deviated from the light implementation of Whitted-style ray tracing. Instead, we use the emissive term of a material, which denotes the color of the light it produces. We also have another property, called light, that acts as a boolean to denote whether it is a light or not, so that we can parse it into a list for light sampling. Our area lights are usually squares - when we do the light sampling, we can simply use the bounding box of the square in order to find points inside the light. This doesn't have the exact same properties as an actual light - for example, we don't see a "fuzziness" of light around the edges of an area light - but worked well enough for us to illuminate scenes.

When a ray is terminated, we always want to find how much light has emitted towards this ray in order to determine how to color the object (which recursively propagates up). We do this through a helper function **emit()**, which checks if the object we currently are at is a light (if so, we don't want to shoot another ray to sample a light source, since that's double counting). If not, we call the **Material::shade()** function, which we've overridden from Phong shading to sampling a random light point and determining the contribution.

The above functions allow us to compute direct lighting for each point, if we just shoot a ray and call our helper function. To add in the effects of indirect lighting, a bulk of the work is done in **traceRaySecondary()**. In this function, we trace the ray's next intersection, and compute the BSDF depending on the type of material. We then compute the probabilities for continuing based on our dynamic Russian roulette algorithm, recursively shoot a ray if we do, and combine it all together with the rendering equation.

Finally, our **tracePixel()** function is where our path tracing all begins. We choose our ***N***, or the amount of primary rays we want to shoot per pixel, defaulted to 100 (but our renders use 1000 for higher quality and less noise artifacts). Then, we add a small amount of jitter to avoid aliasing artifacts, shoot our rays and average the contributions through Monte Carlo integration.

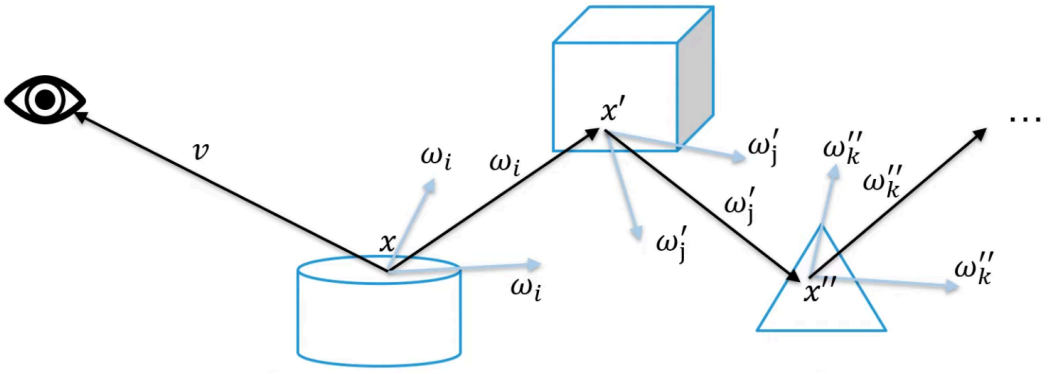
Within the path tracing itself, we do not have any known bugs. One bug comes from our attempt at multithreading the tracer - after tracing a scene, further scenes might crash the program, since it seems like we might not be cleaning threads up properly, so a temporary fix is just to close and re-run the program.

While our path tracer is able to accurately simulate light transport for a large variety of materials, there are some limitations to its effectiveness and accuracy. We have caustic-like artifacts (as seen in our Cornell box scene with refractive sphere), but not fully working, since we were not able to implement bi-directional paths from the light to better emulate refractive materials. We also could have implemented importance / metropolis sampling to more optimally find directions to shoot recursive rays, based on light concentrations or proposal distributions.

To run the path tracer, we have provided a few demo scenes, but you are more than welcome to create your own based on the material properties listed in Appendix III. Running it is just selecting a scene (and potentially a cubemap) - all other selections in the Ray Tracer GUI do not make any difference. Gradeable artifacts can be found in Appendix IV, our presentation.

Appendix

I. Diagram of Light Transport through Path Tracing



II. Expanded Path Tracing Rendering Equation with Monte Carlo Integration

$$L(x \to v) = E_x$$
$$+\frac{1}{N}\sum_{i=1}^{N} f_r\, E_{x'} \cos(\theta_\omega)\frac{1}{p(\omega)}$$
$$+\frac{1}{N}\sum_{i=1}^{N} f_r\, f_r' E_{x''} \cos(\theta_{\omega'}) \cos(\theta_\omega)\frac{1}{p(\omega)p(\omega')}$$
$$+\frac{1}{N}\sum_{i=1}^{N} f_r f_r' f_r''\, E_{x'''}\, cos(\theta_{\omega''})\cos(\theta_{\omega'}) \cos(\theta_\omega)\frac{1}{p(\omega)p(\omega')p(\omega'')}$$
$$+ \ ...$$

III. Special Material Properties:

light(vec3) - a boolean indicator of whether the material is a light or not

emissive(vec3) - the color of the illumination produced by the material

diffuse(vec3) - the albedo for a Lambertian (diffuse) material

reflective(vec3) - the attenuated percentage of light reflected off an object

refractive(vec3) - the attenuated percentage of light refracted into/out of an object

specular(vec3) - a placeholder for probabilities of shooting further rays (x is probability of a diffuse ray, y is probability of a reflective ray, and z is probability of refractive ray)

shininess(float) - an indicator of how rough an object's surface is (0 is shiny, 1 is rough)

index(float) - the index of refraction, but also used for reflection for fresnel

IV. Presentation Link / Gradeable Artifacts:

https://www.canva.com/design/DAFhR7a4Mw4/Ewb7jnHLqEd6mlENYF384Q/edit?utm_content=DAFhR7a4Mw4&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

Note:

Both of us have submitted the eCIS for 10 points of extra credit. We also attended Demo Day on 4/27 for an additional 10 points of extra credit.

Resources Consulted

https://www.youtube.com/watch?v=w36xgaGQYAY

https://blog.demofox.org/2020/06/06/casual-shadertoy-path-tracing-2-image-improvement-and-glossy-reflections/

https://www.scratchapixel.com/lessons/3d-basic-rendering/global-illumination-path-tracing/global-illumination-path-tracing-practical-implementation.html

http://graphics.cs.cmu.edu/courses/15-468/lectures/lecture_11.pdf

https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html#importancesampling materials/returningtothecornellbox

https://pbr-book.org/3ed-2018/Light_Transport_I_Surface_Reflection/Path_Tracing