# TINGE

# *Task 5*

## Implementing the Rendering Equation

Released: 23 March, Deadline: 3 April

This assignment will take a longer amount of time, compared to the previous assignments owing to the number of reading tasks. As such we have divided the tasks into three major sections. **We request you to work on the tasks collaboratively and complete them before the next meet on 4 April. It would be infeasible to work on all of them during the next meet.**

Before implementing the tasks, it would be a good idea to go through the entire code-base that you have written until now - making sure you understand each and every bit of it. In case you have any trouble understanding some part of the code, do ask your fellow teammates to explain their part of the code and to document it properly.

This assignment will largely involve ***Implementing the Rendering Equation*** and having our path-tracer migrate towards rendering using physically based ideas.

# §1 More Materials

## §1.1 Ideal Material Models

We have already implemented our i) ideal diffuse and ii) ideal emissive materials which i) reflect light in an arbitrary direction after a constant color absorption and emit a constant luminance in all directions, respectively. There are two more:

### §1.1.1 Ideal Metallic

This is characterized by the following:

- Parameters - color, roughness
- $(w_o, w_{reflection})$ pair is governed by the laws of reflection. Now $w_i = w_{reflection}$ or $w_i = w_{diffuse}$ is computed using a roughness percentage $p \quad (0 < p < 1)$ which models how much the reflected ray tends away from the specular reflection direction by probabilistically switching between $w_{reflection}$ and $w_{diffuse}$.
- Fixed wavelengths being absorbed, leading to a constant color factor just like the ideal diffuse class

### §1.1.2 Ideal Transmission

- Parameters - color, index of refraction
- $(w_o, w_i)$ pair is governed by the laws of refraction and the laws of reflection. It turns out that no material is "purely" transmissive. Every material that transmits also reflects to some extent. The percentage is governed by Fresnel's Equations. A derivation of Fresnel's equations can be found here.
- $w_i$ probabilistically switches between $w_{reflection}$ and $w_{refraction}$ based on the reflection coefficient obtained from Fresnel's equations. Unlike the ideal reflection material where the roughness coefficient was a constant, here the reflection coefficient in Fresnel's equation is a function of the orientation of the rays wrt the normal. Schlick's approximation can be used to compute the reflection coefficient.
- Fixed wavelengths being absorbed, leading to a constant color factor just like the ideal diffuse class.

### §1.1.3 Programming Tasks

You would have realised by now that the AbstractMaterial class that we implemented before was rudimentary and needs to be improved to take into account of these new materials.

1. Add a virtual function called `Vec3 sample_wi(Vec3 w_o, Vec3 n)` to the `AbstractMaterial` class which returns the ray direction `w_i` for a given outgoing ray direction `w_o` and normal `n` at the surface of intersection for a given material distribution.
2. Update the `MaterialDiffuse` and `MaterialEmissive` class to implement the function `sample_wi`. **This is very important.** Without this function our renderer would not know how to trace the ray forward from the point of intersection for computing the contribution of indirect lighting.
3. Implement the ideal specular and ideal transmission material models as `MaterialMetallic` and `MaterialTransmissive` classes respectively.

## §1.2 Physical Material Models

What we want you to do by the next meeting is to understand the theory behind physically based rendering so that implementing the physically based Cook-Torrance Model becomes straightforward. Read up on:

1. Principles of Physically Based Rendering https://en.wikipedia.org/wiki/Physically_based_rendering
2. Microfacet Models https://www.pbr-book.org/3ed-2018/Reflection_Models/Microfacet_Models

# §2 The Renderer Class

Recall from Task 3 how the rendering equation and Monte Carlo estimators work. Now that we have an array of material models along with support for random number generaion, it is time to get our hands dirty and implement things.

## §2.1 Implementing it!

1. Create a function called `Vec3 illuminacnce(const IntersectInout surface, int max_depth)` in `renderer.cpp` which returns the illuminance at a surface corresponding to a given $x, w_o, n$. Note that $x, w_o, n$ are all contained in `IntersectInout`.
2. The illuminance function will solve the rendering equation $L_o = L_e + L_{i,reflected}$. It will query the material property $L_e$ from `surface.material->Le(...)` and compute $L_{i,reflected}$ in accordance with the integral representation of the rendering equation. The $w_i$ are sampled using `surface.material->sample_wi(...)` and $f_r(...)$ using `surface.material->Fr(...)`.
3. Update the `Renderer` class to use this function instead of simply setting the normal to be the color of the object while rendering.
4. Since we are now solving the rendering equation using Monte Carlo estimation, the $L_{i,reflected}$ integral estimate will only converge to the actual value of the integral as the number of samples tends to infinity. Rendering just one sample would yield a horrendously noisy image.

   Update the function `Renderer::render` to take in `num_samples` as another input. In the rendering process in the main loop, instead of directly storing the illuminance as the pixel color in the image, accumulate the color obtained from each sample into a `Vec3 accumulated_image[width*height]`. You will have to run the main loop, `num_samples` times to compute each sample.

   After all the samples have been computed finally store the image using $pixel\_color = \frac{accumulated\_color}{num\_samples}$

### §2.2 Sampling Theory

Read up on importance sampling from this great article here. We will be teaching you a short recap of importance sampling in the next meeting.

## §3 Improvements to the Camera

Something that we have been looking forward to for a while is the implementation of Depth of Field. Now that you you have implemented random number generation in our project, you can make use of uniformly sampling a disc to simulate the effect of having an aperture.

This blog details on the implementation of depth of field.

We would have transformed our point camera to a nearly physical lens camera through this implementation.

### §3.1 Programming Tasks

1. Implement depth of field in the `Camera` class by modifying the `generate_ray` function accordingly.
2. Add a function to the `Camera` class called `look_at(Vec3 from, Vec3 at)` which re-orients the frame of the camera such that the camera is placed at `from` and looks towards `at` and sets the focal distance to be `dist(from - at)`.

---

*Good luck on your code! Here's Master of Torture signing off~*

---

## A Teaser for Task 6 (Deadline 11 April)

If any of you are done implementing the tasks in this assignment, feel free to take a brave adventure into the wilderness listed below:

- Dielectrics and Conductors (The Cook-Torrance Model)
- Pure Lights (Directional, Point, Area)
- Importance Sampling Materials
- Using Environment Maps

## What do we have in mind after Assignment 6 till Open House?

### Visual (if you are more inclined towards Physics and Visual Effects)

- Volume rendering and absorption through space
- Image convolution and post-processing effects
- Wavelength based materials, colors and Hero sampling

### Optimizational (if you are more inclined towards Mathematical Optimization)

- AABBs and kd-Trees
- Multiple Importance Sampling
- Bidirectional Path Tracing

### Niceties (if you are more inclined towards Programming and polishing Tinge)

- Loading 3D obj models into our scene
- Saving/Loading our scene to/ from a file
- Implementing a GUI for Tinge