Assignment 3

Task 1: Direct lighting

Fill in the function `estimate_direct_lighting` in ***pathtracer.cpp***.

This function is called by `trace_ray` in order to get an estimate of the direct lighting at a point after a ray hit. At a high level, it should sum over every light source in the scene, taking samples from the surface of each light, computing the incoming radiance from those sampled directions, then converting those to outgoing radiance using the BSDF at the surface.

The starter code at the top of the function uses the input intersection normal to calculate a local coordinate space for the object hit point. In this frame of reference, the normal to the surface is $(0, 0, 1)$ (so $\mathbf{z}$ is "up" and $\mathbf{x}$, $\mathbf{y}$ are two other arbitrary perpendicular directions). We compute this transform because the BSDF functions you will define later expect the normal to be $(0, 0, 1)$.

Additionally, in this coordinate system the cosine of the angle between a direction vector `w_in` and the normal vector `Vector3D(0,0,1)` is simply `w_in.z`. (This is clear if you remember that the cosine of the angle between two unit vectors is equal to their dot product.)

We store the transformation from local object space to world space in the matrix `o2w`. This matrix is orthonormal, thus `o2w.T()` is both its transpose and its inverse -- we store this world to object transform as `w2o`.

```
Matrix3x3 o2w;
make_coord_space(o2w, isect.n);
Matrix3x3 w2o = o2w.T();
```

For your later convenience, we store a copy of the hit point in its own variable:

```
const Vector3D& hit_p = r.o + r.d * isect.t;
```

Finally, we calculate the outgoing direction `w_out` in the local object frame. Remember that this should be opposite to the direction that the ray was traveling.

```
const Vector3D& w_out = w2o * (-r.d);
```

You need to implement the following steps:

1. Loop over every `SceneLight` in the `PathTracer`'s vector of light pointers `scene->lights`.
2. For each light, check if it is a delta light. If so, you only need to ask the light for 1 sample (since all samples would be the same). Else you should ask for `ns_area_light` samples in a loop.
3. To get each sample, call the `SceneLight::sample_L()` function. This function requests the ray hit point `hit_p` and returns both the incoming radiance as a `Spectrum` as well as 3 values by pointer. The values returned by pointer are
    - a probabilistically sampled `wi` unit vector giving the direction from `hit_p` to the light source,
    - a `distToLight` float giving the distance from `hit_p` to the light source, and
    - a `pdf` float giving the probability density function evaluated at the returned `wi` direction.

4. The **`wi`** direction returned by **`sample_L`** is in world space. In order to pass it to the BSDF, you need to compute it in object space by calculating **`Vector3D w_in = w2o * wi`**.

5. Check if the **`z`** coordinate of **`w_in`** is negative -- if so, you can continue the loop since you know the sampled light point lies behind the surface.

6. Cast a shadow ray from the intersection point towards the light, testing against the **`bvh`** to see if it intersects the scene anywhere. Two subleties:

    - You should set the ray's **`max_t`** to be the distance to the light, since we don't care about intersections behidn the light source.
    - You should offset the origin of the ray from the hit point by the tiny global constant **`EPS_D`** (i.e., add **`EPS_D * wi`** to **`hit_p`** to get the shadow ray's origin). If you don't do this, the ray will frequently intersect the same triangle at the same spot again because of floating point imprecision.

7. If the ray does not intersect the scene, you can calculate the BSDF value at **`w_out`** and **`w_in`**. Accumulate the result (with the correct multiplicative factors) inside **`L_out`**.

Factors to remember when summing up the radiance values in **`L_out`**:

- **`sample_L`** returns an incoming radiance. To properly multiply this by the BSDF, you need to convert it to irradiance using a cosine term.
- You are summing up some number of samples per light. Make sure you average them correctly.
- The **`sample_L`** routine samples from a light. It does this using some probability distribution which can bias the sample towards certain parts of the light -- you need to account for this fact by using the returned **`pdf`** value when summing radiance values.

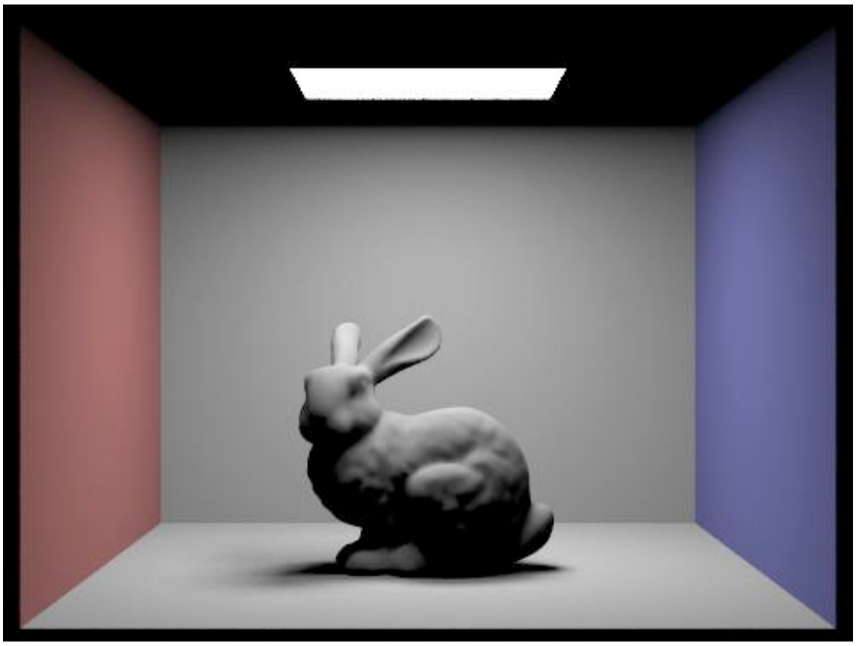
Now you should be able to render nice direct lighting effects such as area light shadows and ambient occlusion, albeit without full global illumination. However, you will only be able to render files with Lambertian diffuse BSDFs, as we have not yet implemented any of the other BSDFs.

At this point, here are the results of the commands

```
./pathtracer -t 8 -s 64 -l 32 -m 6 -f dragon_64_32.png -r 480 480 ../dae/sky/dragon.dae
```

And

```
./pathtracer -t 8 -s 64 -l 32 -m 6 -f bunny_64_32.png -r 480 360 ../dae/sky/CBbunny.dae
```



Task 2: Indirect lighting

In this task, you need to fill in the function `estimate_indirect_lighting` in ***pathtracer.cpp***.

This function is called by `trace_ray` in order to get an estimate of the indirect lighting at a point after a ray hit. At a high level, it should take one sample from the BSDF at the hit point and recursively trace a ray in that sample direction.

We provide with the same setup code as in the direct lighting function:

```
Matrix3x3 o2w;
make_coord_space(o2w, isect.n);
Matrix3x3 w2o = o2w.T();

Vector3D hit_p = r.o + r.d * isect.t;
Vector3D w_out = w2o * (-r.d);
```

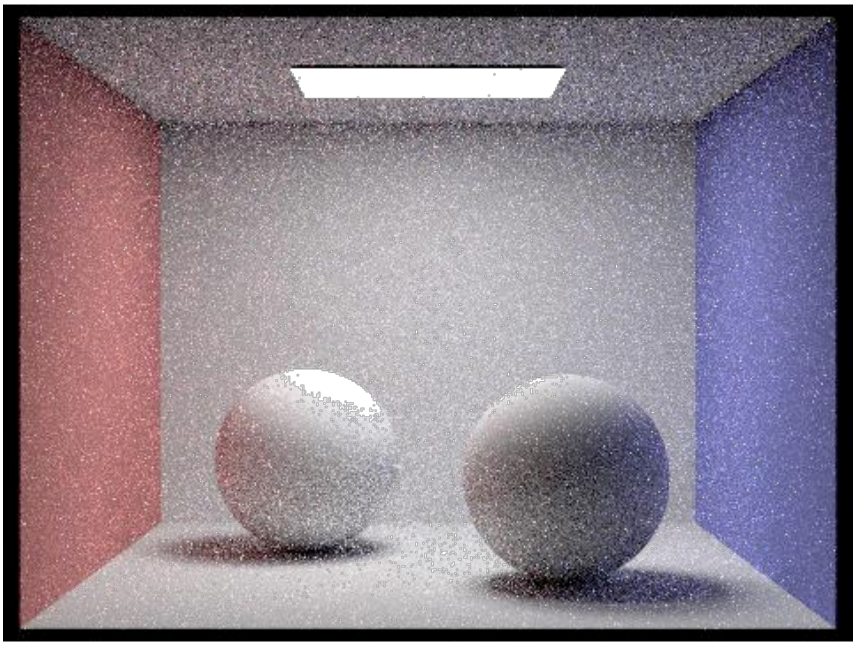
As in part 3, `w2o` transforms world space vectors into a local coordinate frame where the normal is $(0, 0, 1)$, the unit vector in the z direction. Thus `w_out` is the outgoing radiance direction in this local frame.

You need to implement the following steps:

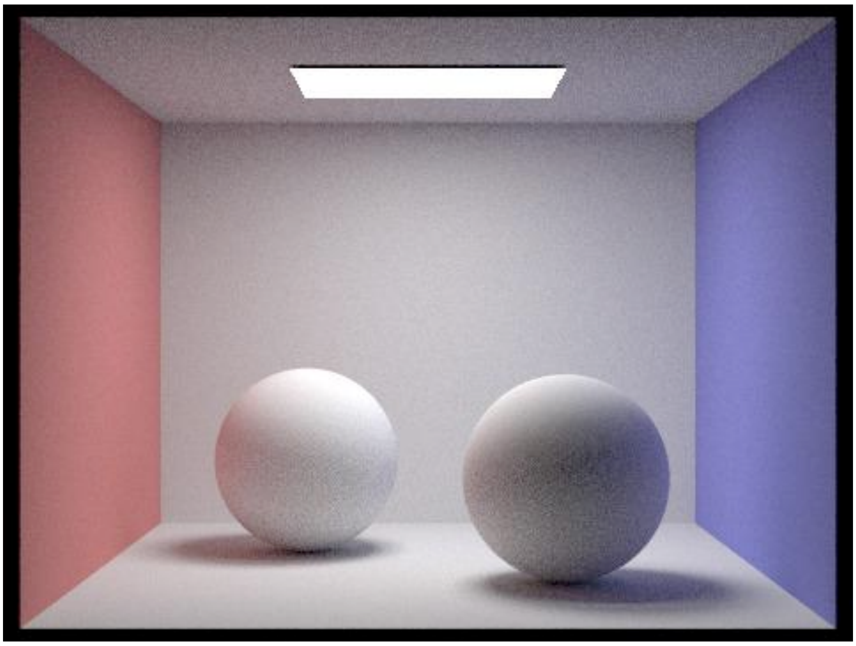
1. Take a sample from the surface BSDF at the intersection point using `isect.bsdf->sample_f()`. This function requests the outgoing radiance direction `w_out` and returns the BSDF value as a `Spectrum` as well as 2 values by pointer. The values returned by pointer are
    - the probabilistically sampled `w_in` unit vector giving the incoming radiance direction (note that unlike the direction returned by `sample_L`, this `w_in` vector is in the ***object*** coordinate frame!) and
    - a `pdf` float giving the probability density function evaluated at the return `w_in` direction.
2. Use Russian roulette to decide whether to randomly terminate the ray. We recommend basing this on the reflectance of the BSDF `Spectrum`, calculated with the spectrum's `illum()` method -- the lower the reflectance, the more likely you should be to terminate the ray and return an empty `Spectrum`. You can use the `coin_flip` method from ***random_util.h*** to generate randomness. Remember that you need to divide by one minus the Russian roulette probability (i.e., the probability of ***not*** terminating) when you return a radiance value. **Note: you may want to multiply `illum()` by some large constant (>10) and then add it by some constant (e.g. 0.05) to make sure that rays are not terminated too early, resulting in high noise.**
3. If not terminating, create a ray that starts from `EPS_D` away from the hit point and travels in the direction of `o2w * w_in` (the incoming radiance direction converted to world coordinates). This ray should have depth `r.depth - 1`. (Note that for the ray depth cutoff to work, you should initialize your camera rays to have depth `max_ray_depth`.)
4. Use `trace_ray` to recursively trace this ray, getting an approximation for its incoming radiance. The `includeLe` parameter should be set to `isect.bsdf->is_delta()` since emission was not included in the direct lighting calculation for delta BSDFs.
5. Convert this incoming radiance into an outgoing radiance estimator by scaling by the BSDF and a cosine factor and dividing by the BSDF `pdf` and one minus the Russian roulette termination probability.

You still can only render diffuse BSDFs until completing Part 5, however, you should now see some nice color bleeding in Lambertian scenes. You should be able to correctly render images such as

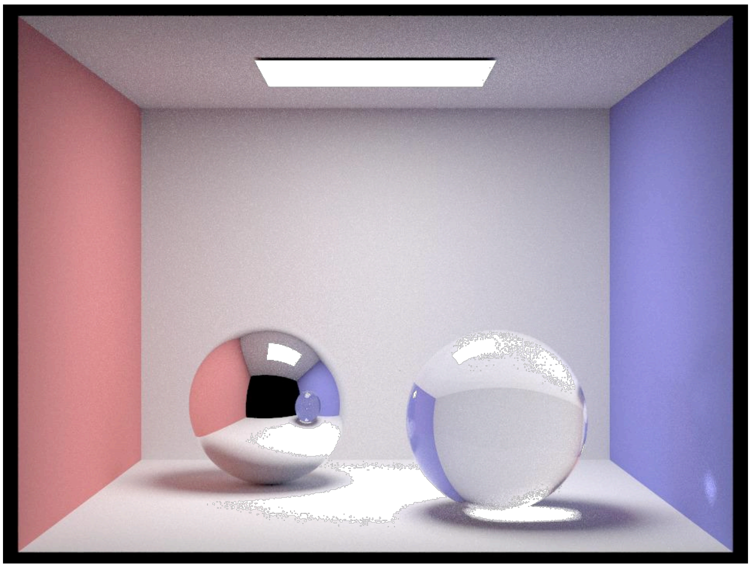
```
./pathtracer -t 8 -s 64 -l 16 -m 5 -r 480 360 -f spheres.png ../dae/sky/CBspheres_lambertian.dae
```



UPDATE: your image will probably look more like



Task 3: Reflection & Refraction



This image took about 20 minutes to render. It has 256 samples/pixel, 4 samples/light, and max ray depth equal to 7. After completing this part, make sure you can render

- ***CBdragon.dae*** (mirror only)
- ***CBlucy.dae*** (glass only)
- ***CBspheres.dae*** (mirror and glass)

Reflection

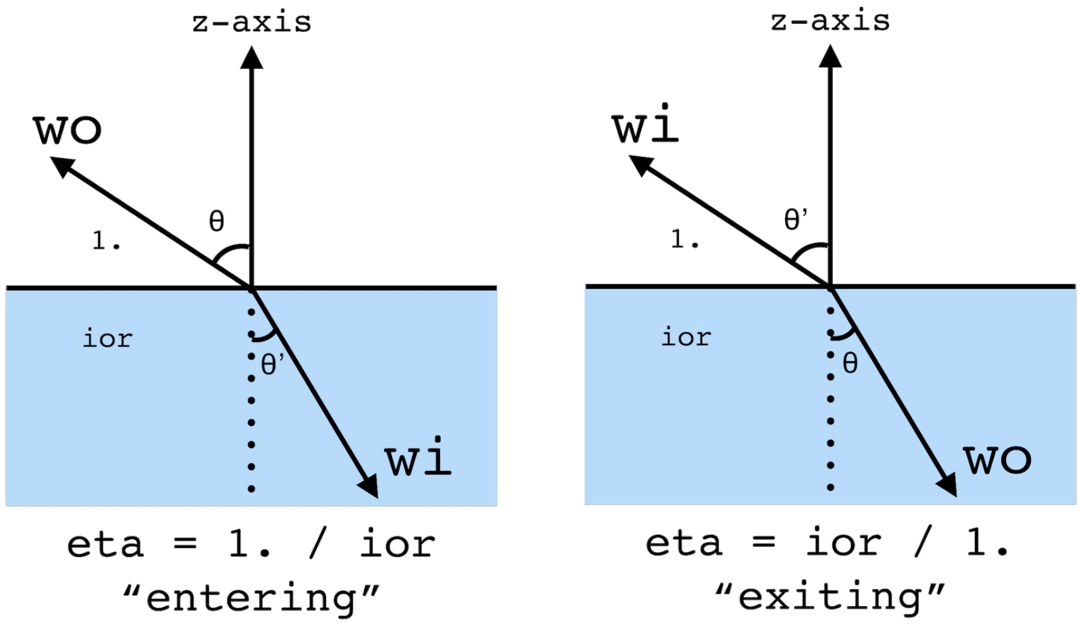
Implement the `BSDF::reflect()` function.

We recommend taking advantage of the object coordinate space that BSDF calculations occur in. This means that the origin is the intersection point and the **z** axis lies along the normal vector. In this situation, reflection should be a trivial one line transformation of the **x**, **y**, and **z** coordinates.

Implement `MirrorBSDF::f()` and `MirrorBSDF::sample_f()`. This should be straightforward if you rely on your `BSDF::reflect()` function. Remember that delta BSDFs like this one are a special case in `PathTracer`'s lighting estimation routines, so you should set `pdf` equal to 1 in `sample_f()`. ***Note*** that you must actually return `reflectance / abs_cos_theta(*wi)` because we need to ***cancel out*** the cosine that the `estimate_indirect_illumination` function will multiply by. Perfect mirrors only change the ray direction, they don't cause any Lambertian falloff.

`MirrorBSDF::f()` returns an empty `Spectrum()` since we assume that a `wi` direction that was not created using `sample_f()` has zero probability of being equal to the reflection of `wo`. (This is a convoluted way of ensuring that we never double count our radiance with delta BSDFs, since we have the special `is_delta_bsdf()` field that we use in `trace_ray`

Refraction



To see pretty glass objects in your scenes, you'll first need to implement the helper function `BSDF::refract()` that takes a `wo` direction and returns the `wi` direction that results from refraction.

As in the reflection section, we can take advantage of the fact that our BSDF calculations always take place in a canonical "object coordinate frame" where the z axis points along the surface normal. This allows us to take advantage of the spherical coordinate Snell's equations on **this slide**. In other words, our `wo` vector starts out with coordinates:

$\omega_o . x = \sin\theta \cos\phi, \quad \omega_o . y = \sin\theta \sin\phi, \quad \omega_o . z = \pm\cos\theta.$

Note: we put a $\pm$ sign on the z coordinate because when `wo` starts out ***inside*** the object with index of refraction greater than 0, its z coordinate will be negative. The surface normal always points out into air. When z is positive, we say that we are entering the non-air material, else we are exiting. This is depicted in the figure.

For the transmitted ray, $\phi' = \phi + \pi$, so $\cos\phi' = -\cos\phi$ and $\sin\phi' = -\sin\phi$. As seen in the figure, define η to be the ratio of old index of refraction to new index of refraction. Then Snell's law states that $\sin\theta' = \eta \sin\theta$. This implies that $\cos\theta' = \sqrt{1 - \sin^2\theta'} = \sqrt{1 - \eta^2 \sin^2\theta} = \sqrt{1 - \eta^2(1 - \cos^2\theta)}$.

In the case where $1 - \eta^2(1 - \cos^2\theta) < 0$, we have total internal reflection--in this case you return false and the `wi` field is unused.

Be careful when implementing this in code, since you are only provided with a single `ior` value in the `refract` function: when entering, this is the new index of refraction of the material ω_i is pointing to, else it is the old index of refraction of the material that ω_o itself is inside. In other words, $\eta = 1/$`ior` when entering and $\eta =$`ior` when exiting.

In spherical coordinates, our equations for ϕ' and θ' tell us that

$\omega_i . x = -\eta\omega_o . x, \quad \omega_i . y = -\eta\omega_o . y,, \quad \omega_i . z = \mp\sqrt{1 - \eta^2(1 - \omega_o . z^2)},$

where we are indicating that $\omega_i . z$ has the opposite sign of $\omega_o . z$.