UCSD CSE 168 Assignment 4:
Indirect lighting, BRDFs, and Multiple Importance Sampling

Figure 1: Images we will render in this homework. 3D scenes courtesy of Marko Dabrovic, Blender Swap creator Wig42, Eric Veach, Wenzel Jakob, and Benedikt Bitterli.

In this homework, we will implement an actual path tracing algorithm. We will finish the RTRYL book this time, and we will render lots of beautiful images (Figure 1)!

Warning: this homework is more difficult than previous ones. Start early!

1 Diffuse interreflection (30 pts)

So far in our code, we seem to be treating the diffuse and specular surfaces separately: diffuse surfaces gather contributions from lights, and specular surfaces trace out rays that collect color from other surfaces (and plastic is a mixture of them). In real world, diffuse surfaces also collect colors from other surfaces. We will implement this in this part.

In the previous homeworks, we do something like this:

```python
def radiance(scene, ray, rng):
    if (ray intersect scene):
        # [emission] add emission of the intersected shape
        # ...
        # [direct_lighting] loop over lights, sample them, and sum over their contributions
        # ...
        # [scattering] recursively tracing the ray by calling radiance
        if (hit a metal or plastic):
            # recursively trace a ray towards the mirror reflection direction
            # ...
        else:
            return scene.background_color
    else:
        return scene.background_color
```

In this part, we will temporarily suspend direct_lighting and the handling of metal and plastic – we will add it back later in part 2 and 3.

We will extend the scattering code to handle diffuse materials in this part. As with the case of area light, conceptually this is simple: instead of always tracing rays towards the mirror reflection direction, we randomly sample rays at different directions and accumulate contributions (Chapter 8 of RTOW describes one approach to do this). But what distribution should we use, and how do we weigh different directions? That’s where we need to do the math.

Remember the area light integral, where we integrate over all points on a light source $S$:

$$
\int_{x \in S} f(x) \, dA(x) = \int_{x \in S} K_d \cdot \max(n_s \cdot l, 0) \cdot \frac{I \cdot \max(-n_x \cdot l, 0)}{d^2} \cdot \text{visibility} \cdot dA(x),
$$

where $K_d$ is the diffuse reflectance, $n_s$ is the normal at the shading point, $l$ is the unit vector pointing from the shading point towards $x$, $n_x$ is the geometric normal at point $x$ on the light source, and $d$ is the distance between the shading point and $x$.

We will derive the equations of diffuse interreflection from Equation (1). Let’s play with an idea: what if everything is light source? We will figure out how to assign the light color $I$ later. Apart from the light
color, all we need to change is the integration domain: instead of integrating a particular light source $S$, we integrate over all surfaces $\mathcal{M}$. Next, since we are tracing rays, we want to deal with directions, instead of points on surfaces. We achieve this using a change of variable (again!) – instead of integrating over position $x$, we integrate over (outgoing) direction $\omega_o$:

$$\int_{x \in \mathcal{M}} \frac{K_d \cdot \max (n_x \cdot l, 0)}{\pi} \cdot \frac{L \cdot \max (-n_x \cdot l, 0)}{d^2} \cdot \text{visibility} \cdot dA(x) = \int_{\omega_o \in \Omega} \frac{K_d \cdot \max (n_x \cdot \omega_o, 0)}{\pi} \cdot \frac{L}{\omega} \cdot d\omega_o. \quad (2)$$

Firstly, we have replaced $S$ with $M$ and $I$ with $L$ to emphasize that things are different now. Also note that $\omega_o = l$. Here, the measure $d\omega_o$ is often called the solid angle – it represents an infinitesimal area of a point on a unit sphere. You will notice that the equation gets a lot simpler after we switch to the solid angle measure. This is because the Jacobian of the change of variable is exactly the reciprocal of $\max(-n_x \cdot \omega_o, 0)$ (see Chapter 9.1 of RTRYL for an explanation). Intuitively, the Jacobian captures the ratio of area on the surface $\mathcal{M}$ and its projected area on a unit sphere (if it is not blocked).

Also note that we use the geometric normal (normal defined by the triangles/spheres) $n_x$ for the cosine at the light, and the shading normal (normal defined by the interpolation of the vertex normals) $n_x$ at the shading point, just like the previous homework. The change of variable explains why we need to do this: the Jacobian needs to be consistent with the scene geometry, since they depend on the ray casting operation, which does not relate on the shading normal.

We still need to decide what is the light color $I/L$. Remember in the mirror case, we recursively trace rays to determine the color. We can do the same thing for diffuse surfaces too!

$$L = \int_{\omega_o \in \Omega} \frac{K_d \cdot \max (n_x \cdot \omega_o, 0)}{\pi} \cdot L \cdot d\omega_o. \quad (3)$$

This is however vacuous, because there is no base case for this recursion. We will instead add the emission whenever we hit a light source$^1$:

$$L = L_e + \int_{\omega_o \in \Omega} \frac{K_d \cdot \max (n_x \cdot \omega_o, 0)}{\pi} \cdot L \cdot d\omega_o. \quad (4)$$

We have arrived at the (in)famous rendering equation (though specialized at diffuse BRDFs).

Now, we need to sample a direction $\omega_o$ for evaluating the integral in Equation (4). For this, read Chapter 5-8 of RTRYL. We will implement the cosine hemisphere sampling described in the book. You do not need to implement the light sampling yet (Chapter 9). You also only need to handle diffuse materials. We will add light sampling and other materials back later!

Go to the function `hw_4_1` in `hw4.cpp` and implement diffuse interreflection using cosine hemisphere sampling. The function also takes an extra command line parameter `-max_depth [max_depth]`, which you will use to limit the maximum recursion depth of your path tracer.

**Debugging tip.** First set `-max_depth 2`, and compare the rendered images with your Homework 3.4 renderer. They should produce more or less the same image (but with different level of noise).

To test your results, type:

```bash
./torrey -hw 4_1 ../scenes/hw1_spheres/scene0_spherical_light.xml
./torrey -hw 4_1 ../scenes/chbox/chbox.xml
./torrey -hw 4_1 ../scenes/party/party_bgonly.xml
./torrey -hw 4_1 ../scenes/sponza/sponza_bgonly.xml -max_depth 6
./torrey -hw 4_1 ../scenes/living-room-Wig42/scene_bgonly.xml
```

See Figure 2 for the references of our rendering. You can notice that the images are much more noisy this time. I did not run all of them to convergence since it’s starting to take a while to render (even so, these images can already take tens of minutes to render!). If you run them overnight (with, say, 10000 samples per pixel), you should see relatively clean images. Since we are relying on randomly trace light path

$^1$The infinite recursion defined in Equation (4) is mathematically well-behaved as long as $K_d < 1$ for all channels. The reason is that after each bounce, some energy will be absorbed by $K_d$, and eventually the ray will carry zero energy.
until we hit a light source, the chance that it hits a light can often be small. The light sampling we will implement later helps with this, but it would not fully solve the problem as we have very large light sources in these scenes already. This is why there are active research projects on trying to make rendering faster – it’s hard! Modern renderers will likely apply path guiding, importance resampling and denoising to speed up the rendering. These could be cool final projects if you are interested.

Another thing to notice is how the global illumination automatically make the images look significantly more realistic! By physically simulating lights, we are able to automagically create realistic images without much efforts. Before people start to simulate global illumination in their renderers, artists have to manually place lights and draw textures to approximate the effect, which is extremely time consuming – now we just let computers spend that time! Even better, compared to recent learning-based generative models, the physically-based rendering approach still let us have full precise control of how the image looks like. We fully understand how the color of each pixel is generated, and we can change the way it is generated if we know how to write a renderer.

[Energy conservation] Quiz (3 pts): Remove the $\frac{1}{\pi}$ in Equation (4) and render the $\text{cbox}$ scene again (you still need to have the $\pi$ in your PDF since PDFs need to integrate to 1). What do you observe? Why?

2 Adding non-diffuse materials (30 pts)

Let’s add back mirror, plastic, and let’s add more.

2.1 Adding back mirror and plastic

Adding back mirror should be easy, we will let you figure that out. Adding back plastic is slightly more involved. Should we follow the diffuse sampling direction or the specular sampling direction? The answer is – we choose them stochastically! Since the diffuse and specular components are weighted by the Fresnel term $F$, we roll a dice and trace the specular ray with probability $F$, and trace the diffuse ray with probability $1 - F$. To be more concrete, mathematically it works like this – we have an expression:

$$ L = Fa + (1 - F)b. \quad (5) $$

We want to use Monte Carlo sampling to evaluate this expression. We evaluate the first term with probability $F$, and the second term with probability $1 - F$. Our Monte Carlo estimator is:

$$ \langle L \rangle = \begin{cases} 
\frac{Fa}{F} & \text{with probability } F \\
\frac{(1-F)b}{1-F} & \text{with probability } 1 - F 
\end{cases}. \quad (6) $$

If we take the expectation of $\langle L \rangle$, we can see that it is an unbiased estimator:

$$ E \left[ \langle L \rangle \right] = F \cdot \frac{Fa}{F} + (1 - F) \cdot \frac{(1-F)b}{1-F}. \quad (7) $$

The same math works even if $\langle a \rangle$ and $\langle b \rangle$ are both Monte Carlo estimators themselves:

$$ \langle L \rangle = \begin{cases} 
\frac{F(a)}{F} & \text{with probability } F \\
\frac{(1-F)(b)}{1-F} & \text{with probability } 1 - F 
\end{cases}. \quad (8) $$
Quiz (3 pts): Let’s consider the following, different estimator:

$$\langle \hat{L} \rangle = \begin{cases} F(a) + (1-F)(b) \frac{F}{F(a) + (1-F)(b)} & \text{with probability } F \\ F(a) + (1-F)(b) & \text{with probability } 1-F \end{cases}$$

(note that (a) and (b) can be Monte Carlo estimators) Is it unbiased? Should we use this one or Equation (6)?

Check your mirror and plastic renderings using the following scenes:

./torrey -hw 4_2 ../scenes/hw1_spheres/scene1_spherical_light.xml
./torrey -hw 4_2 ../scenes/cbox/cbox_area_light_spheres.xml

Figure 3 shows our renderings of them.

![Figure 3: References for mirror and plastic materials for HW 4.2.](image)

2.2 Phong BRDF

After you add back plastic, let’s add a few more materials. It feels weird that our rays either uniformly spread like a diffuse material, or it pinpoint at the mirror reflection direction – there must be something in between! In general, instead of a cosine distribution multiplied by $\frac{K_s}{\pi}$, we can replace it with a bidirectional reflectance distribution function (BRDF) $\rho$:

$$L = L_e + \int_{\omega_o \in \Omega} \rho \cdot L \cdot d\omega_o.$$  \hspace{1cm} (10)

Designing a good BRDF $\rho$ is an active research topic. A simple BRDF, again goes back to the legendary Phong, is to define a cosine falloff function around the mirror reflection direction:

$$\rho_{\text{phong}} \propto \begin{cases} K_s \max(r \cdot \omega_o, 0)^{\alpha} & \text{if } n_s \cdot \omega_o > 0 \\ 0 & \text{otherwise} \end{cases},$$ \hspace{1cm} (11)

where $K_s$ is the reflectance color, $r$ is the mirror reflection direction, and $\alpha$ is usually called the Phong exponent – the larger it is, the sharper the distribution is. Notice that we only define the distribution up to a normalization constant. Choosing a proper normalization is important for BRDFs (why?). When Phong

\footnote{If you love math, Chapter 2.3 of Milos Hasan’s thesis has some cool theoretical discussions on what is the best way to do this.}

\footnote{In many literatures, BRDF does not include the cosine term, but I find it cleaner to combine the both most of the time.}

\footnote{Some variants add a cosine term to the end, but I find that produces darkening appearance at grazing angles.}
derived his reflection model, they did not have to worry about normalization because they were not doing path tracing. To use Phong’s reflection model in our renderer, we need to derive the normalization constant.

For a BRDF $\rho(\omega_i, \omega_o)$ to be energy conserving, it needs to satisfy the following condition:

$$\int_{\omega_o \in \Omega} \rho(\omega_i, \omega_o) \, d\omega_o \leq 1 \text{ for all } \omega_i.$$  \hfill (12)

Throughout this homework, we assume $\omega_i$ to be pointing out from the surface. To ensure the Phong BRDF $\rho_{\text{phong}}$ satisfies this criterion, we need to choose a constant $C$ such that

$$C \cdot K_s \int_{\omega_o \in \Omega} (r \cdot \omega_o)^\alpha \, d\omega_o \leq 1$$  \hfill (13)

for any incoming direction $\omega_i$ (where $r = -\omega_i + 2n_s (n_s \cdot \omega_i)$). We find that the maximum of the integral is achieved when $\omega_i = n_s$. In such case, the reflection direction $r$ is also at the normal direction $n_s$:

$$K_s \int_{\omega_o \in \Omega} (n_s \cdot \omega_o)^\alpha \, d\omega_o.$$  \hfill (14)

The second integral in Equation (14) has a closed-form. To see this, let’s convert it to a spherical coordinate where the $z$-axis is the direction of $n_s$:

$$\int_{\omega_s \in \Omega} (n_s \cdot \omega_o)^\alpha \, d\omega_o = \int_0^{2\pi} \int_0^\pi (\cos \theta)^\alpha \sin \theta \, d\theta \, d\phi = \int_0^{2\pi} -\frac{1}{\alpha + 1} (\cos \theta)^{\alpha+1} \bigg|_0^{\frac{\pi}{2}} \, d\phi = \frac{2\pi}{\alpha + 1}. \hfill (15)$$ (Why is there a $\sin \theta$ in the second expression?)

So, if we set $C = \frac{\alpha + 1}{2\pi}$, as long as $K_s \leq 1$, Phong BRDF always conserve energy:

$$\rho_{\text{phong}} = \begin{cases} K_s \cdot \frac{\alpha + 1}{2\pi} \max (r \cdot \omega_o, 0)^\alpha & \text{if } n \cdot \omega_o > 0 \\ 0 & \text{otherwise.} \end{cases} \hfill (16)$$

Sampling the Phong BRDF isn’t that different from sampling the diffuse BRDF. We first sample the cosine lobe ($\max (r \cdot \omega_o, 0)^\alpha$) assuming the mirror reflection direction $r$ is the $z$ vector. Then we build an orthonormal basis with the $z$ axis pointing towards the mirror reflection direction. We then use the orthonormal basis to transform the local 3D vector to the mirror reflection direction basis.

To importance sample the local cosine lobe, we turn into spherical coordinates and define the PDF we want to sample from:

$$p_{\text{phong}}^{\theta, \phi} (\theta, \phi) \propto (\cos \theta)^\alpha \sin \theta. \hfill (17)$$

From Equation (15), we already know that the normalization factor of this PDF:

$$p_{\text{phong}}^{\theta, \phi} (\theta, \phi) = \frac{\alpha + 1}{2\pi} (\cos \theta)^\alpha \sin \theta$$

$$p_{\text{phong}} (\omega) = \frac{\alpha + 1}{2\pi} (z \cdot \omega)^\alpha \hfill (18)$$

(Why are there two PDFs above? What are the differences between them?)

**Quiz (5 pts):** Apply inverse transform sampling to derive the sampling procedure for the PDF above. Show your derivation process.

After we get a local direction assuming the mirror reflection direction $r$ is the $z$ vector, we can then convert this local direction to the coordinate basis defined by $r$. This conversion is also a change of variable, and we normally have to account for the Jacobian of this coordinate transformation. However, because the coordinate basis is orthonormal, the conversion has an Jacobian of 1.

Now, add the Phong BRDF (Equation (16)) into your `hw_4_2` code. A Phong BRDF of a scene is stored in the following format:

```c
struct ParsedPhong {
    ParsedColor reflectance; // Ks
    Real exponent; // alpha
};
```
**Debugging tip.** A common bug in rendering code is the inconsistency between the sampling procedure and the PDF code (and they are hard to check just by looking at the images!). One way I find useful for testing sampling procedure is to use finite differences to compute the Jacobian determinant of the sampling procedure, and check if our PDF code is consistent with the finite differences. You can look at the BRDF unit tests in my other renderer lajolla to see how this is done. I have caught real bugs using these tests!

We provide yet another Cornell box scene (yes, I am uncreative) and two other scenes for you to test your code:

```
./torrey -hw 4_2 ../scenes/cbox/cbox_area_light_spheres_phong.xml
./torrey -hw 4_2 ../scenes/teapot/teapot_textured_phong.xml
./torrey -hw 4_2 ../scenes/teapot/teapot_textured_phong_grazing_view.xml
```

Our rendering of the scene is in Figure 4.

![Reference for the Phong materials for HW 4.2. For the left most scene, the Phong exponents are 100, 40, 8, 1, from left to right. For the middle and right scenes, the Phong exponent is 50.](image)

Figure 4: Reference for the phong materials for HW 4.2. For the left most scene, the Phong exponents are 100, 40, 8, 1, from left to right. For the middle and right scenes, the Phong exponent is 50.

### 2.3 Blinn-Phong BRDF

![Phong reflection's cosine lobe can lead to discontinuity.](image)

Figure 5: Phong reflection’s cosine lobe can lead to discontinuity.

There are two issues of the Phong BRDF: firstly, the cosine lobe can be extended below the surface (Figure 5), and this can lead to undesired discontinuities.\(^5\) See the last image in Figure 4: the teapot appears less specular at grazing angle, and there is some darkening of the floor because of the clamping. This makes it difficult for Phong BRDF to model reflection at grazing angle, such as sun’s reflection on the sea close to the horizon. Secondly, it is not fully clear how to assign the Fresnel term like we did for mirror: do we apply Fresnel using the normal and the viewing direction \(n_s \cdot \omega_i\)? Or do we apply Fresnel using the normal and the outgoing/light direction \(n_s \cdot \omega_o\)? Both can be good or bad depending on the situations.

The Blinn-Phong BRDF attempts to fix the two issues with one trick.\(^6\) Instead of defining a cosine lobe round the mirror reflection direction, we use the cosine between the normal and the half vector \(h = \frac{\omega_i + \omega_o}{\|\omega_i + \omega_o\|}\):

\[
\rho_{\text{blinn}} \propto \begin{cases} 
F_h (n_s \cdot h)^\alpha & \text{if } n_s \cdot \omega_o > 0 \\
0 & \text{otherwise.}
\end{cases}
\]

\[
F_h = K_s + (1 - K_s)(1 - h \cdot \omega_o)\]

\(^5\)This article illustrates the issue too.

\(^6\)The original paper from Blinn did not include the Fresnel term, but it’s natural to add it.
What’s so special about the half vector \( h \)? First, notice that \( \omega_i \cdot h = \omega_o \cdot h \) (due to the symmetry). Next, when \( h = n_s \), this means that the incoming direction \( \omega_i \) and the outgoing direction \( \omega_o \) satisfies the mirror reflection criteria. Therefore, the similarity between the half vector and the normal is a good metric for measuring how close the two directions are to form a mirror reflection.

Note about two things: firstly, we no longer need to clamp the cosine lobe – as long as viewing direction \( \omega_i \) and lighting direction \( \omega_o \) are on the same side of the surface, \( n_s \cdot h \) is always positive. Secondly, \( F_h \) now takes the half vector as the normal, and \( h \cdot \omega_o = h \cdot \omega_i \), so the evaluating the Fresnel for view direction and the light direction is exactly the same now. Turns out that Blinn-Phong is not only more mathematically well-behaved, it turns out that it also fits significantly better to the highlight shapes of real surfaces. See the article by Ngan et al. for more detail.

The Blinn-Phong BRDF can be normalized using the similar way we did for the Phong BRDF. The closed-form is more complicated, so we just post the end result here and refer you to the article by Yoshiharu Gotanda for the derivation:

\[
\rho_{\text{blinn}} = \begin{cases} \frac{\alpha + 2}{4\pi(2 - 2^{-\alpha})} F_h(n_s \cdot h)^{\alpha} & \text{if } n \cdot \omega_o > 0 \\ 0 & \text{otherwise.} \end{cases}
\] (20)

Sampling the Blinn-Phong BRDF is slightly more involved than Phong, because the cosine lobe is defined using the half vector instead of the outgoing direction directly. Our strategy of sampling Blinn-Phong would be the following:

- Sample a half vector \( h \) with PDF \( p_{\text{blinn}}^h(h) \propto (n_s \cdot h)^{\alpha} \).

- Given an incoming direction \( \omega_i \), get the outgoing direction \( \omega_o \) by applying mirror reflection over the half vector \( h \).

Step 1 is similar to the Phong BRDF sampling, but instead of transforming the coordinates to the \( a \) basis formed by \( r \), we use the basis formed by \( n_s \). Generating an outgoing direction from Step 2 is then straightforward. However, the reflection in Step 2 is yet another change of variable that transform the PDF in the half vector space to the outgoing direction space. This time the Jacobian is not identity. We need to account for the derivatives between \( \omega_o \) and \( h \):

\[
\left| \frac{d\omega_o}{dh} \right| = \frac{1}{\left| \frac{dh}{d\omega_o} \right|}.
\] (21)

Expanding \( h = \frac{\omega_i + \omega_o}{\|\omega_i + \omega_o\|} \), algebraically, we need to compute a \( 3 \times 3 \) Jacobian matrix. The derivation is not too hard but very tedious. Fortunately, there is a very elegant geometric derivative (taken from Microfacet Models for Refraction through Rough Surfaces from Walter et al.): Figure 6 illustrates the proof: the cyan line represents the incoming direction \( \omega_i \), the green line represents the outgoing direction \( \omega_o \), and the half vector before normalization \( \omega_i + \omega_o \) is the yellow line. The normalization of the half vector brings it back to the sphere of \( \omega_i \). The Jacobian is the ratio of the small area \( d\omega_o \) on the \( \omega_o \) sphere, and the small area \( dh \)
on the $\omega_i$ sphere. The ratio of the area is exactly the geometry term: cosine divided by squared distance. There is one more simplification we can do: notice from the figure that $||\omega_i + \omega_o|| = \omega_i \cdot h + \omega_o \cdot h$. Also note that $\omega_i \cdot h = \omega_o \cdot h$. So the Jacobian can be simplify into:

$$\left|\frac{dh}{d\omega_o}\right| = \frac{1}{4\omega_o \cdot h}.$$  \hfill (22)

Multiplying the Jacobian to the PDF, we obtain the final PDF of the Blinn-Phong BRDF sampling procedure:

$$p_{\omega_{\text{blinn}}}^{\omega_o}(\omega_o) = \frac{(\alpha + 1)(\alpha)\alpha}{2\pi \cdot 4(\omega_o \cdot h)}.$$  \hfill (23)

Implement Blinn-Phong BRDF in your hw_4_2 code. A Blinn-Phong BRDF has the same parameters as a Phong BRDF:

```cpp
struct ParsedBlinnPhong {
    ParsedColor reflectance; // Ks
    Real exponent; // alpha
};
```

According to Wikipedia, \texttt{ParsedBlinnPhong::exponent} $\approx 4 \cdot \texttt{ParsedPhong::exponent}$.

Test your method in these scenes:

./torrey -hw 4_2 ../scenes/cbox/cbox_area_light_spheres_blinn.xml
./torrey -hw 4_2 ../scenes/teapot/teapot_textured_blinn.xml
./torrey -hw 4_2 ../scenes/teapot/teapot_textured_blinn_grazing_view.xml

Our rendering of the scene is in Figure 7. Unfortunately, the results are a bit underwhelming. The good news is that for the first two scenes, we maintain similar highlight shape, and for the third scene, we manage to make the teapot looks more specular at the grazing view. However, there is an overall darkening of the color, especially at grazing angle. Our next BRDF will fix the darkening issue.

![Figure 7: Reference for the blinnphong materials for HW 4.2.](image)

### 2.4 (Blinn-Phong) Microfacet BRDF

The use of half vectors in the Blinn-Phong BRDF reveals an important physical insight. Turns out we can build a geometrical/physical explanation of the idea of using half vectors to define BRDFs. The physical explanation is called Microfacet theory.\(^7\) The idea is to treat the surface as a statistical collection of small oriented mirrors (and these mirrors are called the microfacets). Mirrors only reflect lights when the half vector is aligned with its normal direction. So if we are reflecting light at half vector $h$, we must have hit a small mirror with micronormal $m = h$. The half vector cosine lobe $(\alpha = (m \cdot h))^\alpha$ can be seen as defining a distribution of normals, by counting how many mirrors have the normal $h$. Each individual mirror also has

\(^7\)Microfacet theory was developed in optical physics in 1960s by Beckmann anad Spizzichino to study electromagnetic wave scatter on surfaces. Torrance and Sparrow then used similar theory to design microfacet BRDFs. Blinn introduced the idea to graphics. Cook and Torrance later wrote a more comprehensive introduction.
its own Fresnel reflection, which is why we use half vector instead of shading normal for the Fresnel. Finally, the microfacets can block each other, causing shadowing effects. Mathematically, a microfacet BRDF is defined as:

\[ \rho_{\text{microfacet}} = \begin{cases} \frac{F_h \cdot D \cdot G}{4(n_s \cdot \omega_o)} & \text{if } n_s \cdot \omega_o > 0 \\ 0 & \text{otherwise} \end{cases} \tag{24} \]

where \( F_h \) is the Fresnel term (we will use the Schlick Fresnel as in Equation (19)) but evaluated at the half vector, \( D \) is the Normal Distribution Function (NDF) that describes the distribution of the normals of the microfacets, and \( G \) is the geometric shadowing masking term, which accounts for the portion of unblocked microfacets. The \( n_s \cdot \omega_i \) models the total projection area of microfacets towards incoming (if you look at a surface from the grazing angle, you will see more area). The 4 in the denominator is a normalization constant for energy conservation.

Microfacet BRDFs are the most popular BRDFs in computer graphics now, because of how well they fit to our current BRDF datasets (see, again, the article by Ngan et al. for more detail).

For \( D \), we will use a Blinn-Phong inspired NDF:

\[ D \propto (n_s \cdot h)^\alpha. \tag{25} \]

Notice how Equation (24) is basically the Blinn-Phong BRDF but taking into account the microfacet projection area and occlusion. The division of the projection area of microfacets is what will fix our darkening issue in Blinn-Phong! \( G \), on the other hand, is crucial for physical consistency. \( G \) can usually be derived from the geometric configuration of the microfacets. The most popular configuration is developed by Smith: they assume that the microfacets’ orientations are uncorrelated to each other. With this assumption, the geometry term is a function solely on the two viewing directions:

\[ G(\omega_i, \omega_o) = G_1(\omega_i)G_1(\omega_o) \]

\[ G_1(\omega) = \begin{cases} 0 & \text{if } \omega \cdot h \leq 0 \\ \hat{G}_1(\omega) & \text{otherwise.} \end{cases} \tag{26} \]

We can derive \( \hat{G}_1 \) using the conservation of projection area of the microfacet:

\[ \omega \cdot n_s = \int_{\Omega_h} \hat{G}_1(\omega)D(h)\omega \cdot h dh \]

\[ \hat{G}_1(\omega) = \frac{\omega \cdot n_s}{\int_{\Omega_h} D(h)\omega \cdot h dh}. \tag{27} \]

(See Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs for a comprehensive derivation).

Unfortunately, the integral in Equation (27) does not have a known closed-form for the Blinn-Phong NDF. We will use the rational polynomial fit from Walter et al. instead:

\[ a = \frac{\sqrt{0.5\alpha + 1}}{\tan \theta} \]

\[ \hat{G}_1(\omega) = \begin{cases} \frac{3.535a + 2.181a^2}{1 + 2.276a + 2.977a^2} & \text{if } a < 1.6 \\ 1 & \text{otherwise.} \end{cases} \tag{28} \]

Here, \( \tan \theta \) can be computed using trigonometry identities and \( \cos \theta = \omega \cdot n_s \).

Notice that our NDF (Equation (25)) is only defined up to a constant. Normalization of NDF is slightly different from normalizing the Phong BRDF. We, again, need to take the projection area of the microfacets into account:

\[ \int_{\omega_h} D(h)n_s \cdot h dh = 1. \tag{29} \]

See this blog post from Nathan Reed for a nice explanation. Fortunately, the integral above has a closed-form this time. Let \( D(h) = C(n_s \cdot h)^\alpha \), the integral is exactly the same as Equation (15), so we get \( C = \frac{4 + 1.2}{2\pi} \) again.
We can reuse the sampling procedure for sampling Blinn-Phong for sampling the Blinn-Phong microfacet BRDF. It does not importance sample \( G \), but typically the BRDF is dominated by the \( D \) term, so it is usually fine.\(^8\)

Now, implement the Blinn-Phong microfacet BRDF in your `hw_4_2` code. Our microfacet BRDF has the same parameters as a Phong BRDF:

```c
struct ParsedBlinnPhongMicrofacet {
    ParsedColor reflectance; // Ks
    Real exponent; // alpha
};
```

If we render the same scenes again (Figure 8):

```bash
./torrey -hw 4_2 ../scenes/cbox/cbox_area_light_spheres_blinn_microfacet.xml
./torrey -hw 4_2 ../scenes/teapot/teapot_textured_blinn_microfacet.xml
./torrey -hw 4_2 ../scenes/teapot/teapot_textured_blinn_microfacet_grazing_view.xml
```

We will notice that the darkening issue is gone, and we reproduce the desired grazing angle behavior in the last scene!

![Reference for the blinn_microfacet materials for HW 4.2.](image)

### 3 Multiple importance sampling (30 pts)

It’s time to add explicit light sampling (i.e. next event estimation) from the previous homework back. Now at each shading point, we have two sampling strategies: one importance samples the BRDF, and the other one importance sample the light source. We will adopt Multiple Importance Sampling (MIS) for combining the two strategies. In particular, we will adopt the one-sample variant of MIS: instead of deterministically shooting rays for both lights and BRDFs and weighing them, we randomly choose one and combine the distribution. The deterministic version of MIS is usually more efficient (because of the stratification effect), but the one-sample variant fits much better to our current code and is easier to implement.

Read Chapter 10-12 of RTRYL and combine light sampling with BRDF sampling by blending their sampling distribution in your code. You should combine light sampling with the sampling procedures we implemented for diffuse, plastic, phong, blinnphong, and blinn_microfacet. For mirror, light sampling won’t have any effect. Implement the code in `hw_4_3`. Your renderer should converge to the same images before, but hopefully faster in some cases (especially in the case where the light source is mostly not blocked).

**Code structure.** I find it useful to refactor the path tracing code in Homework 4.2 to look like the following (again, you don’t have to do the same as me):

```python
def sample_brdf(...) :
    if pure_specular:
        # sample mirror reflection direction
```

\(^8\)A paper from Heitz and d’Eon discusses how to importance sample \( D \cdot G \).
return wo, is_pure_specular, sampling_weight
else:
    # sample BRDF
    return wo, not_pure_specular, null

def eval_brdf(...):
    # compute both the BRDF value (with cosine) and the sampling PDF
    # ...
    return value, pdf

def radiance(scene, ray, rng, depth):
    if depth <= 0:
        return (0, 0, 0)
    if intersect:
        L = 0
        if hit light source:
            L += emission
        wo, is_pure_specular, sampling_weight = sample_brdf(...)
        next_ray = ...
        if is_pure_specular:
            if sampling_weight > 0:
                L += sampling_weight * radiance(scene, next_ray, rng, depth - 1)
            else:
                value, pdf = eval_brdf(...)
                if value > 0 and pdf > 0:
                    L += (value / pdf) * radiance(scene, next_ray, rng, depth - 1)
        return L
    else:
        return scene.background
I will let you figure out how to add light sampling to this!

Spherical sampling PDF computation. If you reuse your sphere area sampling code from previous homework (instead of the cone sampling strategy in the RTRYL book), since a ray can intersect a sphere twice, you need to sum up the PDF of both intersections when computing the mixture PDF. Alternatively, you can just implement the sphere sampling strategy in Chapter 12.3 of the RTRYL book. It’s more efficient anyway. The references here are rendered using the sphere area sampling, which is less efficient.

We can use some previous scenes to test our new MIS code.

./torrey -hw 4_3 ../scenes/hw1_spheres/scene0_spherical_light.xml
./torrey -hw 4_3 ../scenes/cbox/cbox.xml
./torrey -hw 4_3 ../scenes/hw1_spheres/scene1_spherical_light.xml
./torrey -hw 4_3 ../scenes/cbox/cbox_area_light_spheres_phong.xml

Since the previous scenes are designed to have larger light sources, the variance reduction may not be too significant. See Figure 9 for our renderings.

Figure 9: References for Homework 4.3.
We can also render some new scenes!

```
./torrey -hw 4_3 ../scenes/veach_mi/mi.xml
./torrey -hw 4_3 ../scenes/dining-room/scene.xml
```

Our renderings are shown in Figure 10. Try render these two scenes using your Homework 4.2 code!

![Figure 10: More references for Homework 4.3.](image)

Now reflects a bit how far we have come! Remember the first image we rendered? Now we can render a scene with complex geometry, textures, materials, and lighting.

![Figure 11: How it started/How its going.](image)

Similar to the business card ray tracer, it’s actually possible to implement a very compact path tracer. `smallpt` is a great reference that implements a basic path tracer in 99 lines:

```c
#include <math.h> // smallpt, a Path Tracer by Kevin Beason, 2008
#include <stdlib.h> // Make : g++ -O3 -fopenmp smallpt.cpp -o smallpt
#include <stdio.h> // Remove "-fopenmp" for g++ version < 4.2
struct Vec { // Usage: time ./smallpt 5000 &k xv image.ppm
  double x, y, z; // position, also color (r,g,b)
  Vec(double x_=0, double y_=0, double z_=0){ x=x_; y=y_; z=z_; }
  Vec operator+(const Vec &b) const { return Vec(x+b.x,y+b.y,z+b.z); }
  Vec operator-(const Vec &b) const { return Vec(x-b.x,y-b.y,z-b.z); }
  Vec operator*(double b) const { return Vec(x*b,y*b,z*b); }
  Vec operator*(const Vec &b) const { return Vec(x*b.x,y*b.y,z*b.z); }
  Vec mult(const Vec &b) const { return Vec(x*b.x,y*b.y,z*b.z); }
  Vec norm(){ return *this = *this * (1/sqrt(x*x+y*y+z*z)); }
  double dot(const Vec &b) const { return x*b.x+y*b.y+z*b.z; } // cross:
    Vec operator%(Vec&b){return Vec(y*b.z-z*b.y,z*b.x-x*b.z,x*b.y-y*b.x);}
};
```

12
struct Ray { Vec o, d; Ray(Vec o_, Vec d_) : o(o_), d(d_) {};
enum Refl_t { DIFF, SPEC, REFR }; // material types, used in radiance()
struct Sphere {
  double rad; // radius
  Vec p, e, c; // position, emission, color
  Refl_t refl; // reflection type (DIFFuse, SPECular, REFRactive)
  Sphere(double rad_, Vec p_, Vec e_, Vec c_, Refl_t refl_): rad(rad_), p(p_), e(e_), c(c_), refl(refl_) {}
  double intersect(const Ray &r) const { // returns distance, 0 if nohit
    Vec op = p-r.o; // Solve t^2*d.d + 2*t*(o-p).d + (o-p).(o-p)-R^2 = 0
    double t, eps=1e-4, b=op.dot(r.d), det=b*b-op.dot(op)+rad*rad;
    if (det<0) return 0; else det=sqrt(det);
    return (t=b-det)>eps ? t : ((t=b+det)>eps ? t : 0);
  }
};
Sphere spheres[] = {//Scene: radius, position, emission, color, material
  Sphere(1e5, Vec(1e5+1,40.8,81.6), Vec(.75,.25,.25), DIFF),//Left
  Sphere(1e5, Vec(-1e5+99,40.8,81.6), Vec(.75,.25,.75), DIFF),//Rght
  Sphere(1e5, Vec(50,40.8,1e5), Vec(1,1,1)*.999, SPEC),//Mirr
  Sphere(16.5,Vec(50,681.6-.27,81.6), Vec(12,12,12), Vec(), DIFF) //Lite
};
inline double clamp(double x){ return x<0 ? 0 : x>1 ? 1 : x; }
inline int toInt(double x){ return int(pow(clamp(x),1/2.2)*255+.5); }
inline bool intersect(const Ray &r, double &t, int &id){
  double n=sizeof(spheres)/sizeof(Sphere), d, inf=t=1e20;
  for(int i=int(n);i--;) if((d=spheres[i].intersect(r))&&d<t){t=d;id=i;}
  return t<inf;
}
Vec radiance(const Ray &r, int depth, unsigned short *Xi){
  double t; // distance to intersection
  int id=0; // id of intersected object
  if (!intersect(r, t, id)) return Vec(); // if miss, return black
  const Sphere &obj = spheres[id]; // the hit object
  if (!intersect(r, t, id)) return Vec(); // if miss, return black
  Vec x=r.o+r.d*t, n=(x-obj.p).norm(), nl=n.dot(r.d)<0?n:n*-1, f=obj.c;
  double p = f.x>f.y && f.x>f.z ? f.x : f.y>f.z ? f.y : f.z; // max refl
  if (++depth>5) if (erand48(Xi)<p) f=f*(1/p); else return obj.e; //R.R.
  if (obj.refl == DIFF){ // Ideal DIFFUSE reflection
    double r1=2*M_PI*erand48(Xi), r2=erand48(Xi), r2s=sqrt(r2);
    Vec w=n, u=((fabs(w.x)>0.1?Vec(0,1):Vec(1))%w).norm(), v=w*u;
    Vec d = (w*cos(r1)*r2s + v*sin(r1)*r2s + w*sqrt(1-r2)).norm();
    return obj.e + f.mult(radiance(Ray(x,d),depth,Xi));
  } else if (obj.refl == SPEC){ // Ideal SPECULAR reflection
    return obj.e + f.mult(radiance(Ray(x,d-n*2*n.dot(d)),depth,Xi));
  }
  return obj.e + f.mult(radiance(Ray(x,d-n*2*n.dot(d)),depth,Xi)) +
  Ray reflRay(x, r.d-n*2*n.dot(d));
  double nc=1, nt=1.5, nnt=into?nc/nt:nt/nc, ddn=r.d.dot(nl), cos2t;
  if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0) // Total internal reflection
    return obj.e + f.mult(radiance(reflRay,depth,Xi));
  Vec ddir = (r.d-n*nt-n*(1-nnt*nt)*(1-ndd*ddn)).norm();
  double a=nt-nc, b=nt+nc, Ra=a*a/(b*b), c = 1-(into?-ddf:tdir.dot(n));
  double Re=Ro*(1-R0) + c*c*c*c*Tr=1-Re, Pf=25+.5*Re/Tr, TP=Tr/(1-P);
  return obj.e + f.mult(depth>2 ? (erand48(Xi)<P ?
    radiance(reflRay,depth,Xi)*RP:radiance(Ray(x,tdir),depth,Xi)*TP)
    + radiance(reflRay,depth,Xi)*Re+radiance(Ray(x,tdir),depth,Xi)*Tr);
```
int main(int argc, char *argv[]){
    int w=1024, h=768, samps = argc==2 ? atoi(argv[1])/4 : 1; // # samples
    Ray cam(Vec(50,52,295.6), Vec(0,-0.042612,-1).norm()); // cam pos, dir
    Vec cx=Vec(w*.5135/h), cy=(cx%cam.d).norm()*5135, r, *c=new Vec[w*h];
    #pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP
    for (int y=0; y<h; y++){ // Loop over image rows
        fprintf(stderr,\"\rRendering (%d spp) %5.2f%%\", samps*4,100.*y/(h-1));
        for (unsigned short x=0, Xi[3]={0,0,y*y*y}; x<w; x++) // Loop cols
            for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++) // 2x2 subpixel rows
                for (int sx=0; sx<2; sx++, r=Vec()){ // 2x2 subpixel cols
                    for (int s=0; s<samps; s++)
                        double r1=2*erand48(Xi), dx=r1<1 ? sqrt(r1)-1: 1-sqrt(2-r1);
                        double r2=2*erand48(Xi), dy=r2<1 ? sqrt(r2)-1: 1-sqrt(2-r2);
                        Vec d = cx*(  ((sx+.5 + dx)/2 + x)/w - .5) +
                                cy*(  ((sy+.5 + dy)/2 + y)/h - .5) + cam.d;
                    r = r + radiance(Ray(cam.o+d*140,d.norm()),0,Xi)*(1./samps);
                } // Camera rays are pushed ^^^^^ forward to start in interior
                c[i] = c[i] + Vec(clamp(r.x),clamp(r.y),clamp(r.z))*.25;
        }
   (FILE *f = fopen("image.ppm", "w"); // Write image to PPM file.
    fprintf(f, "P3\n%d %d\n%d\n", w, h, 255);
    for (int i=0; i<w*h; i++)
        fprintf(f,"%d %d %d ", toInt(c[i].x), toInt(c[i].y), toInt(c[i].z));
}
```

`minpt` is another cool reference that implements a more sophisticated path tracer in 300 lines (it even supports stuff we didn’t implement!).

### 4 Design your own scene (10 pts)

Now, render some pretty pictures with your renderer!

### 5 Bonus: Russian roulette (10 pts)

Our renderer currently assumes a maximum path depth, but it doesn’t need to. The idea of Russian roulette, invented in the nuclear engineering community and brought to rendering by James Arvo and David Kirk, is that we can probabilistically terminate a path, while maintaining an unbiased estimation of the infinite sum. Read Chapter 13.7 and Chapter 14.5.4 of PBRT for how to implement Russian roulette in a path tracer and add it to your code.

### 6 Bonus: GGX/Trowbridge-Reitz microfacet distribution (10 pts)

There are many normal distribution functions other than the Blinn-Phong NDF. The most popular NDF is developed by Trowbridge and Reitz, often called GGX in graphics (named by Walter et al., it means “Ground Glass Unknown”). Read about it in PBRT and implement the GGX/Trowbridge-Reitz microfacet distribution in your code. You might also be interested in Eric Heitz’s article of importance sampling $D \cdot G_1$ for GGX.

### 7 Bonus: Rough dielectric BSDF (20 pts)

Just like there is a fuzzy generalization of mirrors, there is a fuzzy generalization of glasses too. It’s also easy to model them using microfacet theory. Read Walter et al.’s article about a refractive microfacet model
and implement it in your code.

8  Bonus: deterministic MIS (20 pts)

An alternative, and more standard way to combine light sampling and BRDF sampling, is to always shoot rays towards light and BRDFs at each bounce. This can be potentially more efficient due to two reasons: first, the light sampling now only requires an occlusion test, which is faster than a nearest hit intersection test. Second, the deterministic nature can drastically reduce variance, while allowing us to use more aggressive MIS heuristic e.g., power heuristic. Read PBRT to see how deterministic MIS is incorporated in a path tracer and implement it in your code.