CS148 Homework 5 - Global Illumination

Grading on Monday, Oct 30th

0.1 Assignment Outline

Your task this week is to adapt the simple ray tracer from HW3 from using point lights and direct illumination to using area lights and global illumination for more natural lighting. To demonstrate the effect of global illumination, we will apply the simple ray tracer to the classic Cornell box scene shown below.

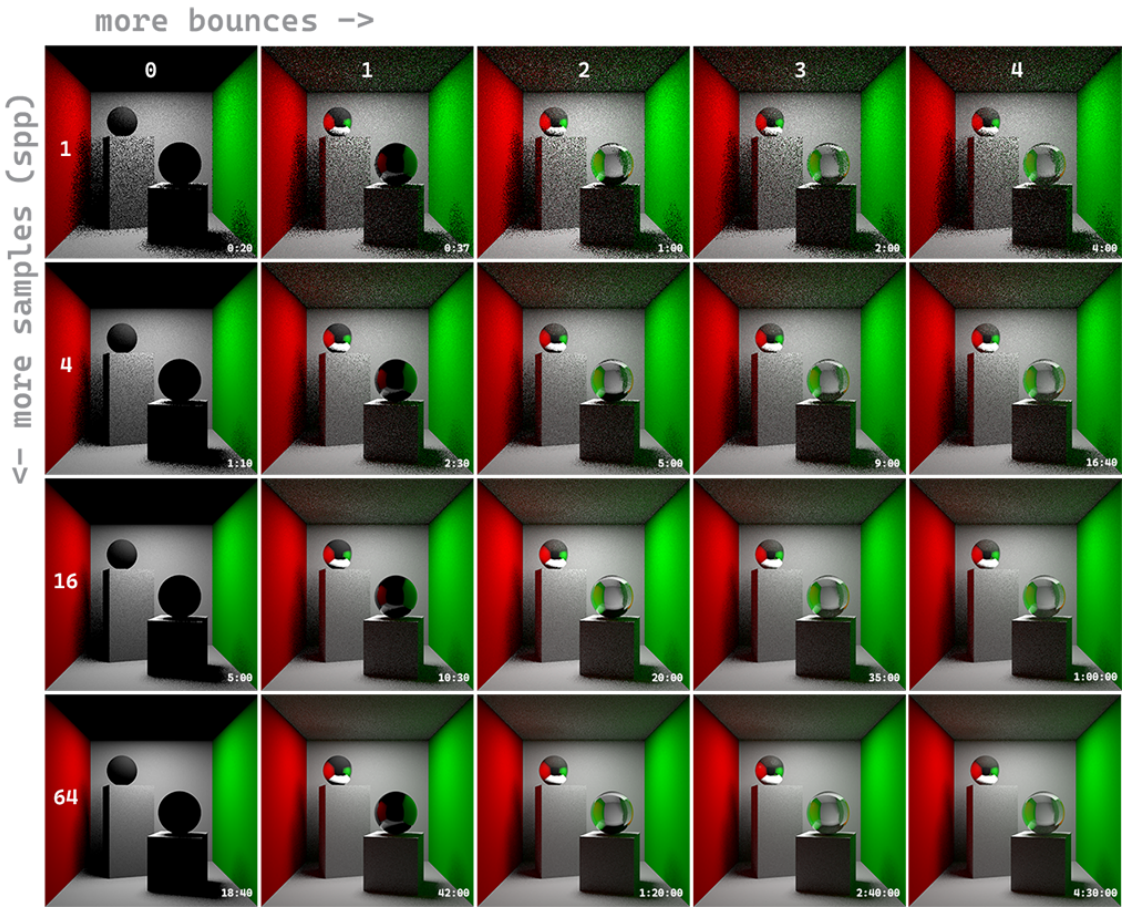
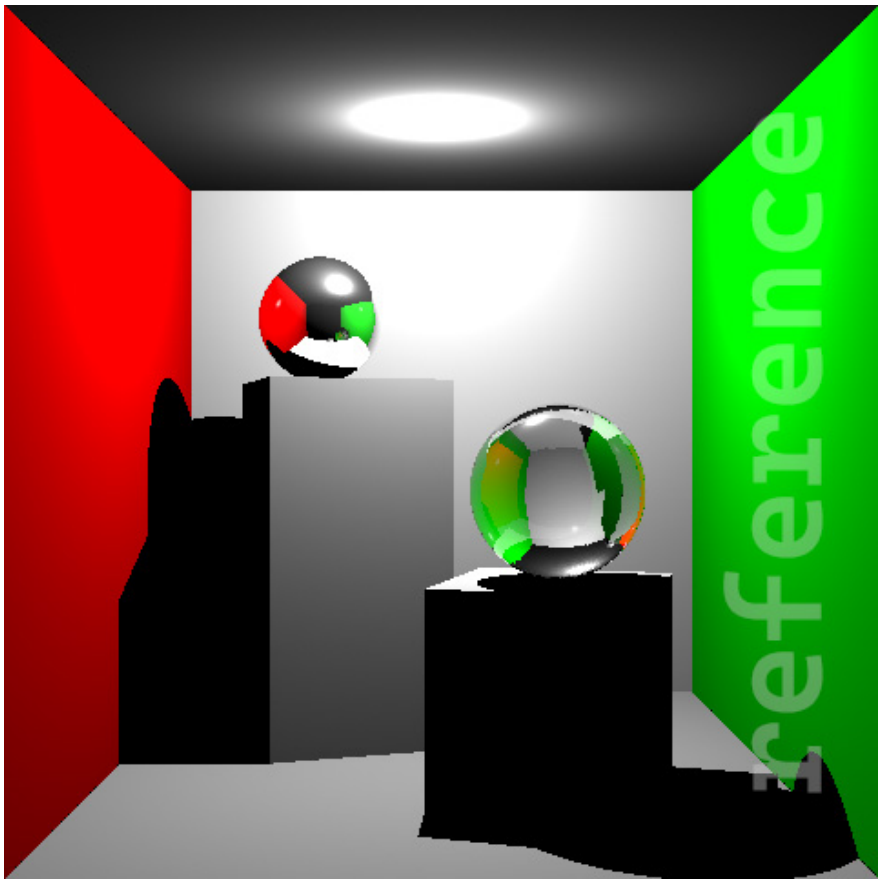


Figure 1: The Cornell box scene rendered using the completed HW code; timed on a i9-9900X CPU @ 3.50GHz (single-threaded)

We've already created the scene for you in Blender and have set up the code infrastructure to access information from the scene using Blender's Python API. **You can find the scene and code all** in this .blend file. Like in HW3, you will need to fill in some missing lines of code that are further elaborated as **Action:** TODO items in this PDF.

Please start early! Unlike previous assignments, this HW5 is not as dependent on the week's lecture material. The area light code can all be done immediately based on last week's lectures. And while the indirect diffuse method for global illumination will be conceptually covered on Thursday, the actual algorithm for it is spelled out for you in this handout. **Furthermore, depending on your computer hardware, the final image can take up to an hour to render! So plan ahead and give yourself enough time!**

To run the code, you will need to run the `simpleRT_UIpanels` script first every time you launch Blender (like in HW3) before running `simpleRT_plugin`. You should try launching Blender from the command line as you did in HW3, as the starter code comes with a function that prints out the estimated wait time to the command line for how long the render will take to finish. Rendering the scene with the starter code should give you the following image:



0.2 Collaboration Policy, Office Hours, and Grading Session

All policies from here on are the same as they were for HW2. See the HW2 document for details.

Quiz Questions (1 pt)

You will be randomly asked one of these questions during the grading session:

- What is light tracing, aka photon tracing? Why is it inefficient?
- What is bidirectional ray tracing? How does it lead to indirect illumination?
- Summarize the tractability issue when doing multiple global illumination bounces. How can we address this issue by looking at the diffuse and specular lighting components separately?
- What is the advantage of Monte Carlo methods for numerical integration? When might it be more appropriate to use e.g. Newton-Cotes instead?
- What is the point of photon maps? How are they involved in gathering radiance for generating effects, such as caustics, indirect lighting, etc, when ray tracing?

1 Assignment Checkpoints

1.1 Area Lights

In HW3, we lit our scene with point lights, which gave us sharp shadows. In the real world, most lights are area lights that create soft shadows. The larger the area, the softer the shadow (as you may have noticed when playing around with the lighting in HW4). For this HW5, we will implement a disk-shaped area light.

1.1.1 TODO: Implementing an Area Light (1.5 pt)

Action: We will go over step-by-step the process of implementing an area light in code.

1. We first need to check if a light is an area light (as opposed to a point light) when iterating over all the lights in the scene for our ray tracing. We can do so using the condition: `light.data.type == "AREA"`.

   Add an if statement within the loop over the lights like so:

   ```
   for light in lights:
   light_color = ... # don't modify
   light_loc = ... # don't modify

   # ADD CODE FOR AREA LIGHT HERE
   if light.data.type == "AREA":
   ... # your code

   light_vec = ... # don't modify
   ```

   If the light is an area light, then we proceed with the following steps to update the light color variable, `light_color`, as well as the light location variable, `light_loc`.

2. Calculate the normal vector for the area light in world space. Area lights are surface patches and thus have normals that we use to determine their tilt angle with respect to the object(s). From HW4, you know that the tilt angle affects the strength of the light.

   We calculate the light normal by first defining it in the light's local space (basically the light's "object space" if we pretend the light were an object). Let the area light be emitting downwards (i.e. in negative z) in its own local space, thus initialize:
   `light_normal = Vector((0, 0, -1))`
   Then, we need to transform this normal vector for the light into the global world space. Conveniently, the light data structure already has the transform we need stored as a member variable, `rotation_euler`, already computed via Blender. So we just need to call:
   `light_normal.rotate(light.rotation_euler)`.

3. Update the light color based on the tilt angle between the area light and object. To do so, we need to multiply `light_color` by a dot product between the light normal and the direction **FROM** the light **TO** the hit location. Remember to normalize the vectors since we are relying on them for a dot product, and be mindful of the vector directions! If the dot product is negative, then we can set `light_color` to zeros, since it means the object is behind the area light. Unlike point lights, area lights only emit forwards in one direction.

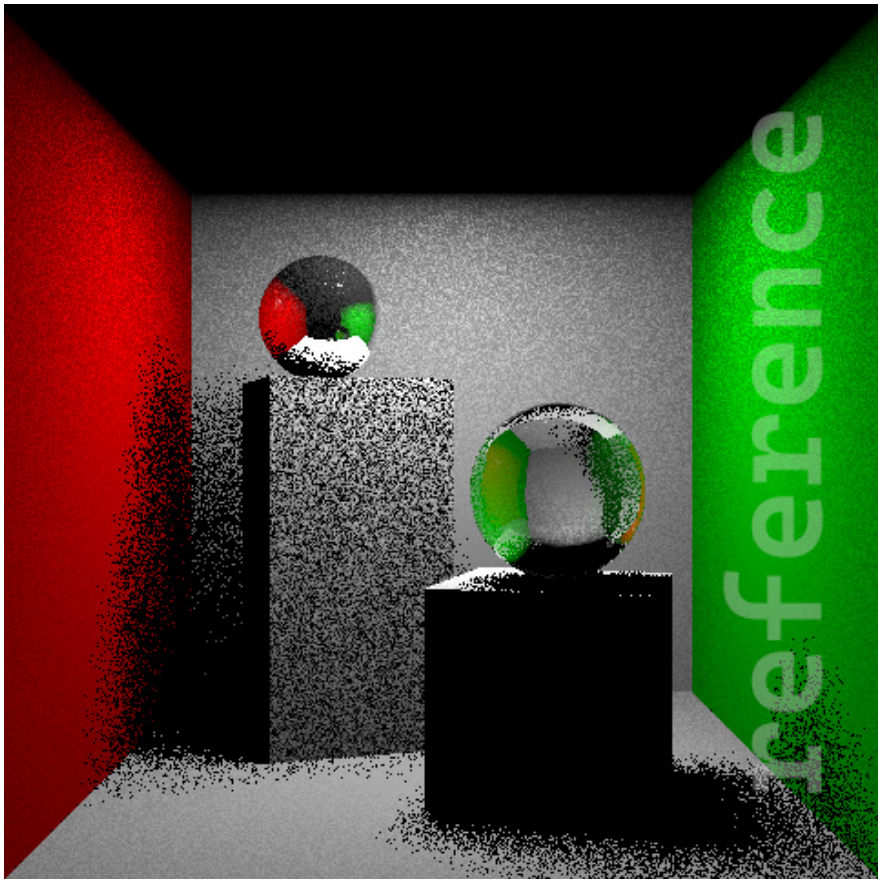
4. Calculate the point on the area light disk from which we will be emitting light. Conceptually, you can think of a (disk-shaped) area light as a collection of point lights that only shine in 1 direction, clustered together into a shape (of a disk). For this step, we will (randomly) pick one of the point lights in this collection to emit light from. We will worry about the other point lights in our collection later. This step will be done in the local space of the light.

   Since the light is in the shape of a disk, we can randomly sample this area by parameterizing the space such that each point has some distance to the disk center r and angle θ (as in polar coordinates). We uniformly sample r from 0 to 1, and θ from 0 to 2π. You can use the Python function `np.random.rand()` to generate random samples from a uniform distribution over $[0, 1)$.

   After we obtain our uniform samples, we can compute the emit location in Cartesian coordinates as $[x, y, z] = [\sqrt{r}\cos(\theta), \sqrt{r}\sin(\theta), 0]$. From here, we scale the coordinates with the radius of the disk, which is stored in the light data structure member variable: `light.data.size/2` to get the final sample coordinates. Save this as a new variable.

5. The previous step computes the light emit location in the light's local space, so we need to transform it into the global world space. Again, the transform that we need is conveniently already stored in the light data structure as a member variable: `light.matrix_world`. This stores the matrix that transforms a point from the light's local space to world space. You simply just need to apply the matrix appropriately to your computed coordinate in the previous step. Set the final result to the `light_loc` variable, thus modifying it appropriately for use later in the code.

After finishing the above steps, your render should look similar to the following (but might not match exactly because of random sampling). **Please save this render at 480x480 100% resolution with a depth of 3 for grading.**



1.1.2 TODO: Sampling the Area Light (0.5 pt)

Action: You may have noticed that implementing the area lights introduced a lot of noise in the shadows. This is because we only emitted light from one point on the area light disk for each area light. To get more accurate lighting with area lights, we need to emit light from multiple points on the disk for each area light, then average over the results. This process is called sampling.

1. First, look for the `render(self, depsgraph)` function definition. Notice how the `self.samples` variable is set to 1. This is telling the code to do only 1 ray trace pass through the whole image. Replace the 1 with the code in comments to its right. This sets the samples variable to whatever number you enter into the `samples` field of the `Render Properties` tab in the Properties Editor.

   The sample number is basically how many ray trace passes we will do. If we only do 1 ray trace pass, then that means we only use 1 random point on each area light disk for lighting and call it a day. If we were to do 2 samples, then we would ray trace 2 times, thus using 2 random points on each area light disk instead. For 4 samples, we would ray trace 4 times, using 4 random points on each area light disk; and so on.

2. Now take a look at the `RT_render_scene` function. Notice the triple loop that first loops over the number of samples, then the height of the image, then the width of the image. Within the loop, notice the call to the ray tracing function that you implemented in HW3: `RT_trace_ray`. Does it make sense to you now how the sample number is the "number of ray trace passes" that we do?

Currently however, the code is not equipped to handle multiple ray trace passes. Look for the line:

`buf[y, x, 0:3] = ...`

This variable is short for "buffer", which refers to the data structure that we use to store our image (which is simply an array of 2D arrays of pixels, one 2D array for each color channel: r, g, b, and alpha transparency). Recall that this is all within a loop over the height of the image, then a loop over the width of the image. This double loop is looping over each pixel in the image. Notice then how we are storing the r, g, and b color computation from our ray tracing function for each pixel directly to this buffer variable via the above line of code. This effectively has each iteration of the outermost sample loop overwrite the last by writing directly to the buffer, thus only the last sample iteration will matter.

We want to instead average our results across all the samples. To do so, you need to create a temporary buffer variable, i.e. call it the sample buffer or `sbuf` for short. Declare this variable outside the triple loop as a 3D array with the same height, width, and 3 color channels as the buffer variable (ignore the alpha transparency channel here):

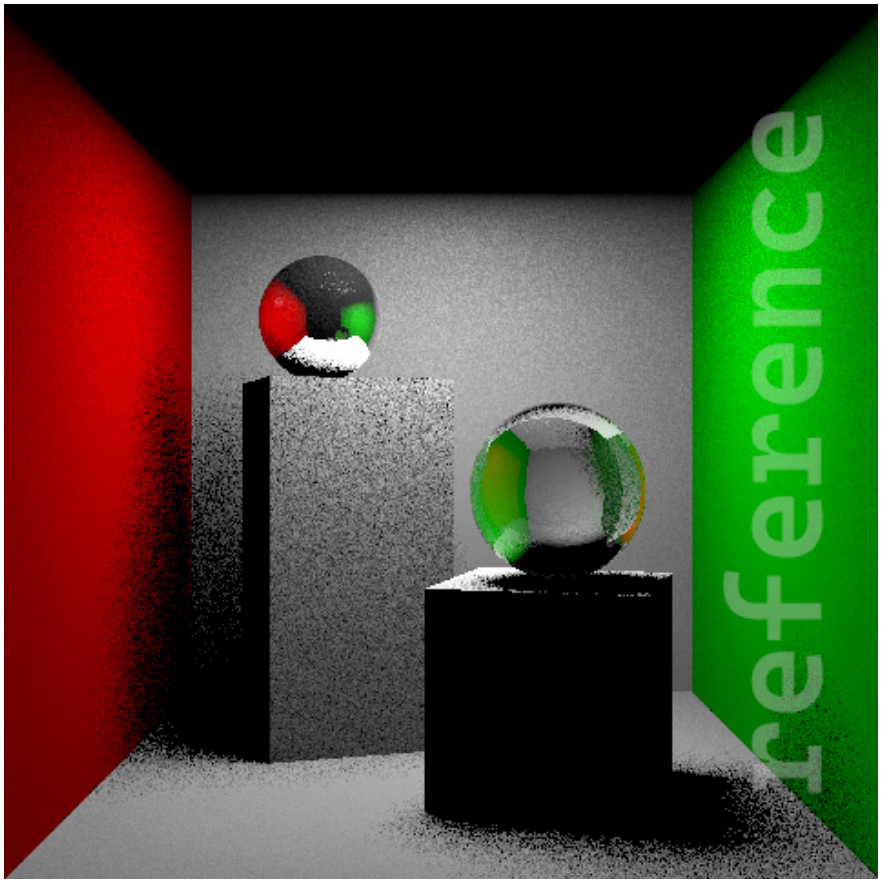
`sbuf = np.zeros((height, width, 3))`

From here, you want to **ADD** (as in accumulate) the results of the ray trace function to the sample buffer instead of the actual buffer within the loop.

Still within the loop, We can then get the appropriate image for the current sample count by dividing the data stored in the sample buffer by the current sample count `s + 1` (Python self-check: why the +1?). This gets us our desired average across all current samples in the loop so far. Finally, you can set the result of your computation to the original buffer:

`buf[y, x, 0:3] = # your result`

After finishing the above steps, your render should look similar to the following (but might not match exactly because of random sampling). **Please save this render at 480x480 100% resolution with 4 samples and a depth of 3 for grading.**



1.2 Ray Sampling

We can extend the idea of sampling to perturb the directions in which we shoot rays originating from the film plane. Doing so lets us get a better estimate of the pixel value over the pixel area to accelerate convergence, i.e. make the noise in the image go away faster. Thus, within each iteration of our sample loop, we will not only sample different points along the disk area of our area lights, but also sample different starting ray locations in the pixel area for each pixel (as opposed to simply originating all rays from the center of each pixel).

Sampling is more formally covered in Lectures 10 and 11. For this HW, we will simply give the code for one particular sampling algorithm and explain the process of using it.

1.2.1 TODO: Low-discrepancy Sampling (0.5 pt)

Action: One particular way to speed up convergence is with low-discrepancy sampling. To do this, we need to construct what is called a low-discrepancy sequence. Blender Cycles uses the Sobol sequence by default. For the purposes of this HW, we are going to use a simpler one called the Van der Corput sequence. We have provided you the code for generating this sequence in the `corput` function (it's simply a Python translation of what is on the Wikipedia page).

To use this sequence for low-discrepancy sampling, you need to do the following in the `RT_render_scene` function:

1. First, compute the dimensions of each pixel dx and dy. dx is 1 over the width of the image. dy can be computed by dividing the aspect ratio by the height.

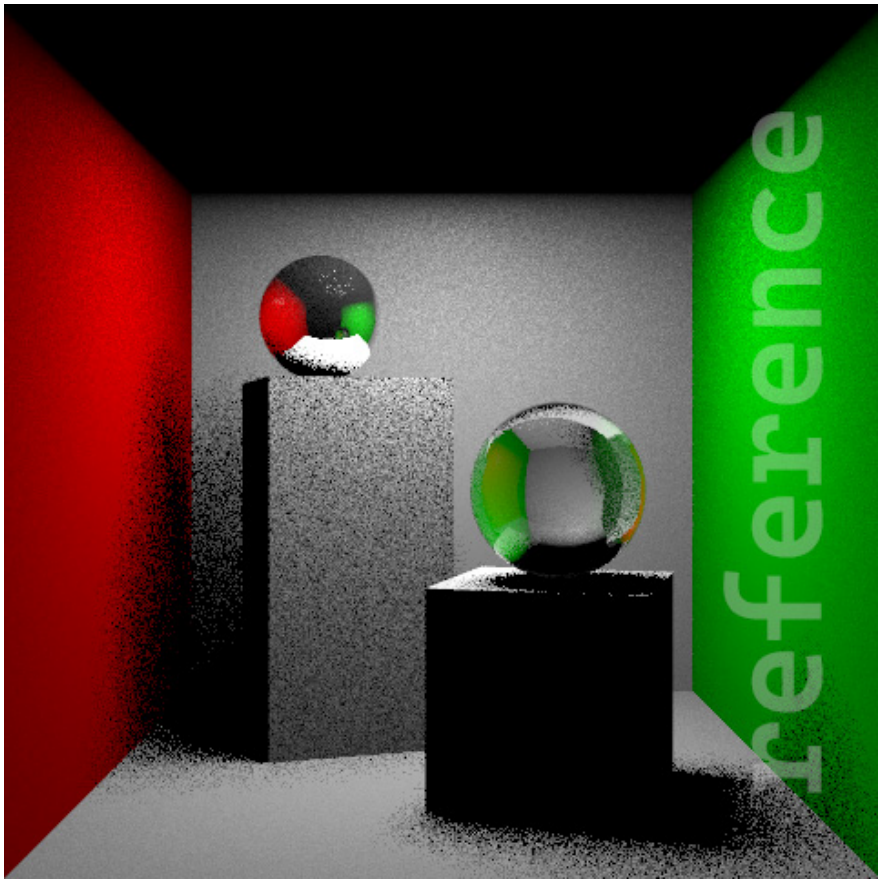
2. Then, compute an x and a y pixel offset for each sample using the Corput sequence:
   `corput_x = [corput(i, 2) * dx for i in range(samples)]`
   `corput_y = [corput(i, 3) * dy for i in range(samples)]`

   Essentially, each sample gets an associated Corput x and Corput y offset. We can **precompute** these offsets ahead of time **outside the triple loop**, and then index them within the loop as `corput_x[s]` and `corput_y[s]` to get the x and y offsets for the current sample iteration.

3. Now, within the loop, find where `ray_dir` is declared. Modify `ray_dir` so that its x and y components are not simply the center of the pixel (i.e. not simply `screen_x` and `screen_y`), but rather the center of the pixel **PLUS** the Corput offset for the current sample iteration.

Visually, it can be hard to tell the effect of low-discrepancy sampling without a large sample count. We've provided a reference image below for an expected result using the same sample count of 4 as before. If you have a fast computer, then you can try upping the sample count by powers of 2 with and without low-discrepancy sampling until you see the difference.

Be ready to show the code that you wrote for this part of the HW during grading.



1.3 Global Illumination (Color Bleeding)

The images so far have all had "dead" black shadows, i.e. the colors from the red and green walls do not "bleed" onto the cubes. This is because we only have direct diffuse and specular, i.e. the

diffuse and specular rays stop once they hit an object. In reality, the object receives light from not only light sources, but also from other objects in the scene. You have heard of this phenomenon as color bleeding from lecture.

To mimic real-world lighting, we need to add more bounces for the rays to achieve global illumination. We will do so for the diffuse rays in this HW by giving them recursive bounces to implement what we call indirect diffuse lighting. Essentially, we will add extra steps to the computation of the diffuse component in the Blinn-Phong BRDF. We will ignore global illumination for the specular component to keep the assignment within reasonable length.

1.3.1 TODO: Implementing Indirect Diffuse (1.5 pt)

Action: You can write this code right before the ambient computation in `RT_trace_ray`.

1. First, check if `depth > 0`. Similar to reflection and transmission rays, we only shoot recursive rays when the recursive depth is greater than 0. Only proceed with the next steps if this true.

2. For the purposes of computing the recursive ray direction, we need to first establish a local coordinate system with the normal vector at the intersection point as the Z-axis. That is, we need to find a pair of x and y axes so that the z-axis becomes `hit_norm`.

   For the x-axis, we will make it a unit vector orthogonal to the normal. Start by initializing this vector to an initial guess of (0,0,1). Now, if (0,0,1) is too close to the normal, then change it to (0,1,0) instead. Two vectors are "too close" if they are almost parallel, i.e. the magnitude of their dot product is close to 1. (Self-check: do you know why? Ask in office hours if you're not sure!)

   We then compute the normal-direction component of x, which can be computed as $x \cdot n * n$ where x and n are the x and normal vectors respectively. Essentially, this is just a dot product of the x and normal vectors, then a component-wise multiplication with the normal vector. Subtract the result from the x vector to make it orthogonal (aka perpendicular) to the normal, then normalize x. Some of you may recognize this process as the Gram Schmit orthogonalization technique from linear algebra.

   The y vector can be obtained via the cross product of the x vector and the normal. Python has a `.cross()` function.

3. Also for the purposes of computing the recursive ray direction, we need to sample a hemisphere oriented at (0, 0, 1). Imagine the top half of a sphere centered at the origin of the xyz-axis, where up is positive z. We want to randomly pick a vector direction along this hemisphere.

   Mathematically, we can express a point on the hemisphere as $[\sin(\theta)\cos(\phi), \sin(\theta)\sin(\phi), \cos(\theta)]$, with $\theta \in [0, \pi/2), \phi \in [0, 2\pi)$. We uniformly sample on the hemisphere surface by making $\cos(\theta)$ a uniform variable between 0 and 1.

   Computationally, we do this by first creating two random variables r_1 and r_2 between 0 and 1, and let r_1 be $\cos(\theta)$ and $r_2 * 2 * \pi$ be ϕ. From here, you can simply plug these values appropriately into the above formula for a point on the hemisphere (remember that $\sin^2(\theta) + \cos^2(\theta) = 1$). Since the hemisphere is centered at (0,0,0), this point is also a ray direction. (Self-Check: do you know why? Ask in office hours if you're not sure!)

4. The ray direction that you computed from the previous step is within some abstract local space of a hemisphere, and thus needs to be transformed into world space to actually be used. To

do this, we use the coordinate system computed in Step 2 to determine a matrix transform. This transform will take the ray from this abstract hemisphere and place it alongside the normal of our object in world space.

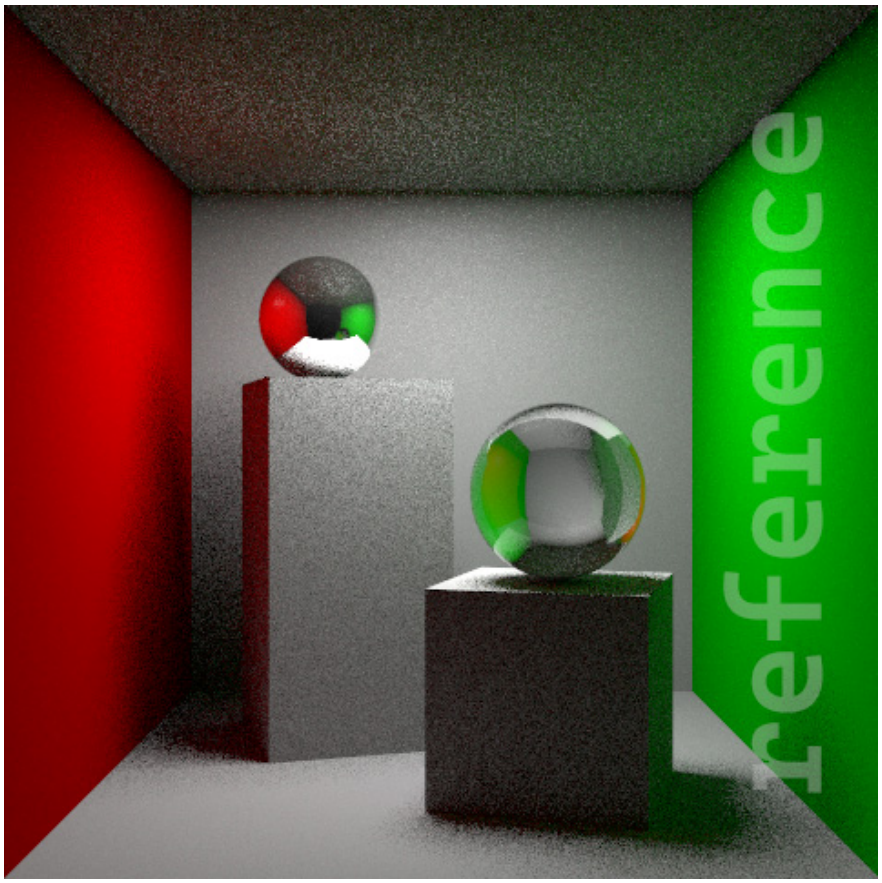
Let `x, y, n` be the coordinate system you computed in Step 2. Then to form the necessary matrix transform in Python, you can do:

```
mat_transform = Matrix()
mat_transform[0][0:3] = x
mat_transform[1][0:3] = y
mat_transform[2][0:3] = n
mat_transform.transpose()
```

Use this matrix to transform your ray result from Step 3.

5. Finally, recursively trace the ray from Step 4 as you did for the reflection and transmission rays from HW3, remembering to take into account self-occlusion. Store the result as the indirect diffuse color. This color needs to be scaled by r_1 to account for face orientation as well as `diffuse_color` to account for absorption. Add the final result of these products to `color`.

After finishing the above steps, your render should look similar to the following. When you think you have your code correct, **render your final image at 480x480 100% resolution with 16 samples and a depth of 3. This may take up to an hour or more. Please save this render for grading.**



In case you are curious, this is how the scene from HW3 looks if rendered with the completed HW5 code. Notice the softer shadows and color bleeding. You do not need to generate this image yourself.