

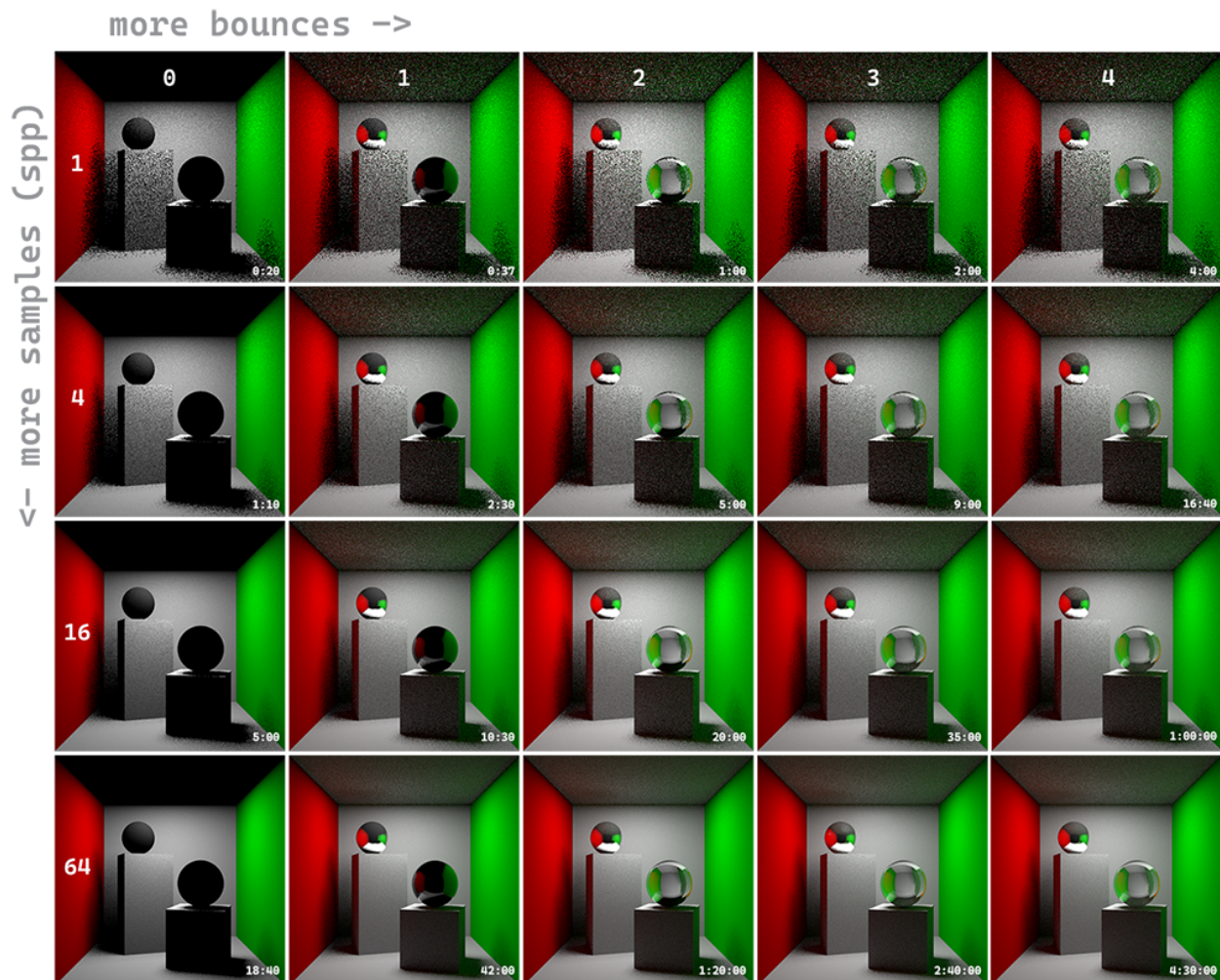
HW5 Understanding Raytracer

CS 148 Fall 2021-2022

Due Date: Monday, October 25th 2021 by 7 pm PT

Follow the instructions carefully. If you encounter any problems in the setup, please do not hesitate to reach out to CAs on Piazza or attend office hours on Nooks.

Be aware of the **Checkpoints** below. Make sure you complete each one since we will do the grading based on them.



Rendered with the code from this homework. Timed on i9-9900X CPU @ 3.50GHz (single-threaded).

This week we will continue building our SimpleRT ray tracer. Last week we explored area lights and BSDF materials in Cycles. Starting from point lights and direct illumination only, we will then implement area lights, sampling, and global illumination in this homework.



Start early! Like HW3, this assignment is more coding intensive than the others.



Rendering can take a long time! The final image will take an hour to render.

To better demonstrate the effect of global illumination, we will use the classic Cornell box. Download the blender scene with an updated UI script `simpleRT_UIpanels` :

http://web.stanford.edu/class/cs148/assignments/hw5_understanding_raytracer.blend

Then add your HW3 code to the current Blender file by selecting **File ▶ Append**, browse to your HW3 .blend file ▶ **Text ▶ `simpleRT_plugin`**. (Of course, you can also create a new text and copy-paste your code). The code will not work directly with HW5 scene, see below for the fix.

Make the following changes to your code from HW3:

1. Update the code to use the same lighting system as Blender Cycles.

Change the line for `light_color` from:

```
light_color = np.array(light.data.simpleRT_light.color * light.data.simpleRT_light.energy)
```

to:

```
light_color = np.array(light.data.color * light.data.energy / (4 * np.pi))
```

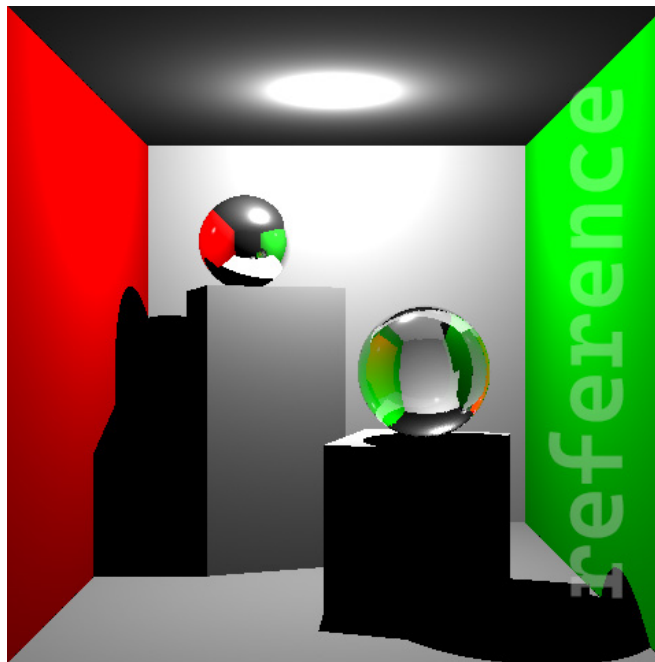
Your code should run at this point after you run `simpleRT_UIpanels`. The image may look incorrect — we'll fix that in the next step.

2. Update the condition that determines if a location is in shadow or not. In the HW3 code, we did not account for the situation where shadow rays hit objects behind the light. This means that the current location will be shadowed when there are objects behind the light, which should not happen. We need to fix it by having a stricter condition to determine shading.

Introduce `light_hit_loc` to receive the hit location of the shadow ray:

```
# cast the shadow ray from hit location to the light
has_light_hit, light_hit_loc, _, _, _, _ = ray_cast(scene, new_orig, light_dir)
```

Then add another condition to the if-statement that checks `has_light_hit` to tell if the object is in shadow. (Hint: the distance that the shadow ray travels from the hit location should be less than the distance to the light.)



- ☐ **Checkpoint 0: Update shadow ray condition.** (For self-check only, no need to save the image)



Your submission images should be rendered at 100%, e.g. 480 px × 480 px, and depth set to 3. You can turn down the resolution during debugging.

I. Area Light



In homework 3, we lit our scene with point lights, which give us sharp shadows. In the real world, most lights are area lights that create soft shadows. The larger the area, the softer the shadow. Here we implement a disk area light.

The idea of implementing area lights is simple: instead of keeping the light location fixed (as with point lights), we will make it a randomized location over the light-emitting area.

When iterating over all the lights in the scene, we first check if it's an area light using the condition `light.data.type == "AREA"`:

```
# iterate through all the lights in the scene
for light in lights:
    # get light color
    light_color = np.array(...)

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

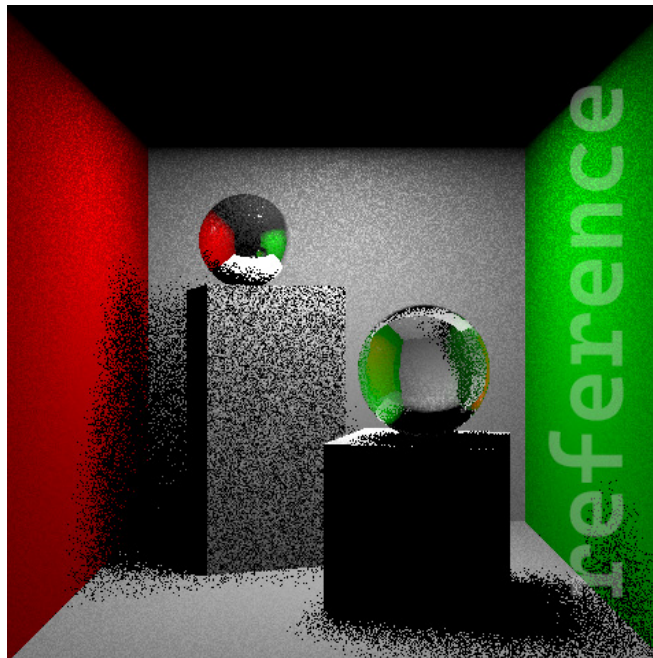
If it is indeed an area light, then we do the following to update the light color and light location:

1. Calculate the light normal in world space. In local space, the light is emitting downwards, so `light_normal = Vector((0, 0, -1))`. We then rotate it in-place with `light_normal.rotate(light.rotation_euler)`.
2. Update the light color. If we are at an angle to the light, then we are effectively receiving less light. 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, and be mindful of vector directions! If the dot product is negative, then we can set `light_color` to zeros since the area light does not emit from the backside.
3. Calculate the local emitting location. 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). To uniformly sample the disk, we will need to take the square root of the uniformly sampled r . In cartesian coordinates, that is $[x, y] = [\sqrt{r} \cos(\theta), \sqrt{r} \sin(\theta)]$, with r uniformly

sampled from 0 to 1, and θ from 0 to 2π . Now we obtained uniform samples on the unit disk, we multiply the coordinate with the radius of the disk `light.data.size/2` to get the final sample coordinates. Use the cartesian coordinate to form the local emitting location `emit_loc_local = Vector((x,y,0))`. You can use `np.random.rand()` to generate random samples from a uniform distribution over $[0, 1)$. Learn more about the math [here](#).

4. Transform the emitting location from local to global by left multiplying `emit_loc_local` by `light.matrix_world`. This gets the global emitting location. In python, matrix-vector multiplication can be done with the `@` operator. Use this vector to calculate `light_vec` below.

If you used `light.location` directly in any code below, you should change every instance to use the new transformed emitting location.



☐ **Checkpoint 1: Save the image with area light.**

II. Sampling



As you may have noticed, area lights will introduce noise in the shadows. We can get rid of this by averaging over multiple results. We can also perturb the direction we shoot rays originating in the film plane to get a better estimate of the pixel value over the pixel area. In this section, we will create our sampling process with a low-discrepancy sequence.

We will modify the `RT_render_scene` function by rendering the image multiple times, then averaging the results together to get our final image. We then perturb ray directions slightly to accelerate convergence.

First, we add the sampling loop:

1. Passing the sample count to `RT_render_scene()`.

In `render(self, depsgraph)`, add an instance variable to read the sample count:

```
def render(self, depsgraph):
    ...
    self.size_x = int(scene.render.resolution_x * scale)
    self.size_y = int(scene.render.resolution_y * scale)

    # ADD CODE HERE
    self.samples = depsgraph.scene.simpleRT.samples
```

In `render_scene()`, pass the sample count variable to `RT_render_scene()` by adding an extra parameter to the `RT_render_scene()` function, where we will do the main sampling process.

2. Update the code for log messages to show the correct progress.

You can come up with your own way of doing this. This part is mainly for self-verification and doesn't count toward assignment points — don't spend too much time on this.

The logging code is in the loop of `render_scene()`:

```
# start ray tracing
for y in RT_render_scene(scene, width, height, depth, buf):
```



```

...
# print render time info in the console
print(
    f"rendering... Time {timedelta(seconds=elapsed)}"
    + f"| Remaining {timedelta(seconds=remain)}",
    end="\n\r",
)
# update Blender progress bar
self.update_progress(y / height)

```

The `y` variable that gets yielded by `RT_render_scene()` is what is being examined here. You are free to modify how the progress is calculated, the yield value of `y`, and any related code to fit whatever update scheme you decide to write.

We recommend printing out which sample count your render is currently on, along with the remaining time. You can also display the stats in the Render Result window using `self.update_stats("", some_status_str)`.

3. In `RT_render_scene()`, create a new buffer `sumbuf = np.zeros((height, width, 3))` before the main for loop. This will be the buffer for accumulating the samples. We will loop over the sample count and divide the buffer by the current sample count to get the final image, i.e. averaging.

Add an additional for loop that iterates over your sample count variable so that you are iterating over the image pixels for each sample count (three nested for loops in the order: sample count, height, width). Then, for each sample count, modify the code so that the pixel color returned by the call to `RT_trace_ray()` is added to the sample buffer, `sumbuf`, instead of stored in the main buffer, `buf`.

From here, we can get the final image for this current sample count with

```

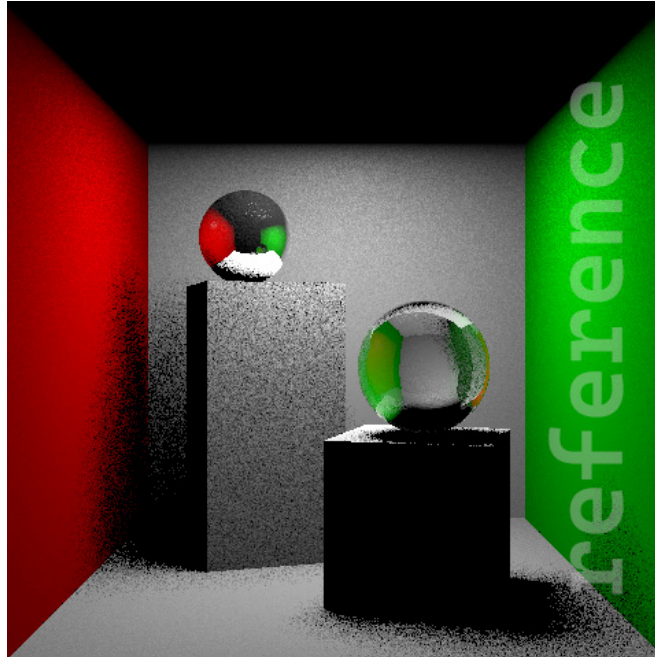
sumbuf[y,x]/sample_count

```

and transfer over this result to the main buffer: `buf[y, x, 0:3] = sumbuf[y,x]/sample_count`.

(Be careful when modifying the loops so that the main buffer still gets returned by the `RT_render_scene()` function appropriately).

Set the number of samples to 4 in Properties Editor ► Render Properties and render your result. This will take ~4 min.



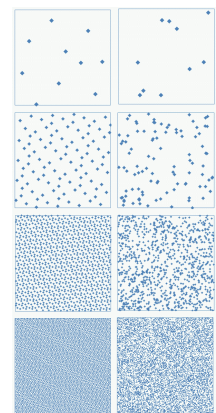
- **Checkpoint 2.1: Save the image with simple sampling. Discuss how this is improved from 1.**

After adding the sampling loop, we will perturb the ray directions. We can of course just randomize the offset applied to a ray, but using a low-discrepancy sequence will make the convergence faster, i.e. the noise will go away faster.

There are various ways of constructing low-discrepancy sequences. Blender Cycles uses the Sobol sequence by default. We are going to use a simpler one called the Van der Corput sequence ([Wikipedia](#)) for this homework.

Here we provide the function to generate a Van der Corput sequence between -0.5 and 0.5:

```
def corput(n, base):
    q, denom = 0, 1
    while n:
        denom *= base
        n, remainder = divmod(n, base)
        q += remainder / denom
    return q - 0.5
```



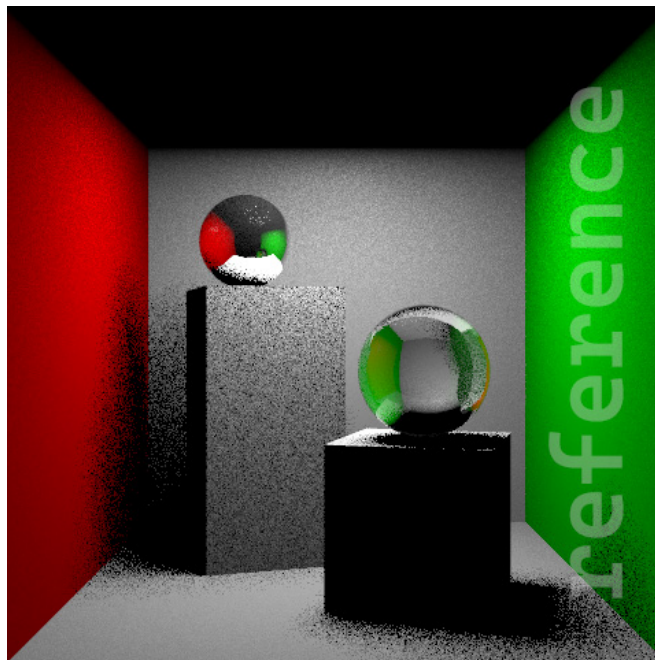
Coverage of the unit square.
Low-discrepancy (left) vs random (right).
[Wikipedia.](#)

With this function, we can pre-compute offsets `corput_x` and `corput_y` for each sample count using:

```
dx = 1 / width
dy = aspect_ratio / height
corput_x = [corput(i, 2) * dx for i in range(samples)]
corput_y = [corput(i, 3) * dy for i in range(samples)]
```

Then, for each sample count `s` in `RT_render_scene()`, we modify the ray direction to include the corresponding Corput offsets with:

```
ray_dir = Vector((screen_x + corput_x[s], screen_y + corput_y[s], -focal_length))
```



- ☐ **Checkpoint 2.2: Save the image with low-discrepancy sampling. Discuss how this is improved from 2.1.** (The difference may not be visible with a low sample count. You can talk about theoretical improvements without referring to your rendered images.)

III. Indirect Diffuse — Color Bleeding



The image above has dead black shadows, and the colors from the red and green walls do not "bleed" onto the cubes. This is because we only have direct diffuse/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 other objects in the scene. To mimic real-world lighting, we should add more bounces for the rays to achieve global illumination. Here we implement recursive bounces for diffuse rays to achieve global illumination for diffuse. Note that we are not solving the rendering equation here. This is only an approximation of the full BRDF since we are not doing recursion for the specular component.

We will add this part of the code to the end of `RT_trace_ray()`, right before `return color`.

Similar to reflection and transmission rays, we only shoot recursive rays when the recursive depth `depth>0`. The direction of the ray should be uniformly sampled from the hemisphere oriented at the normal of the shaded location `hit_norm`. The resulting color will be multiplied by the diffuse color and added to the final color.

1. First, we need to establish a local coordinate system with `hit_norm` as the Z-axis, i.e. finding a pair of X-axis and Y-axis so that the Z-axis becomes `z=hit_norm`:

For X-axis, make it a unit vector orthogonal to `z`, starting with an initial guess of `(0,0,1)`, or `(0,1,0)` if the former is too close to `z` (Hint: two vectors are "too close" if they are almost parallel, i.e. the angle between them is nearly 0 or 180). We subtract the z-direction component `x.dot(z) * z` from `x` to make it perpendicular to `z`, then normalize `x` (this step is essentially Gram Schmit).

For Y-axis, we obtain it by taking the cross product of `z` and `x`.

2. Then, we sample a direction from the hemisphere oriented at $[0, 0, 1]$:

We can write a random point on the hemisphere as $[\sin(\theta) \cos(\phi), \sin(\theta) \sin(\phi), \cos(\theta)]$, with $\theta \in [0, \pi/2)$, $\phi \in [0, 2\pi)$. To sample uniformly on the hemisphere surface, $\cos(\theta)$ will be a uniform variable between 0 and 1.

We compute the random point by creating two random variables `r1` and `r2` between 0 and 1, and let `r1` be $\cos(\theta)$ and `r2*2*pi` be ϕ . The coordinates of this point on the

hemisphere can then be taken as our **local** sampled direction. Learn more about the math [here](#).

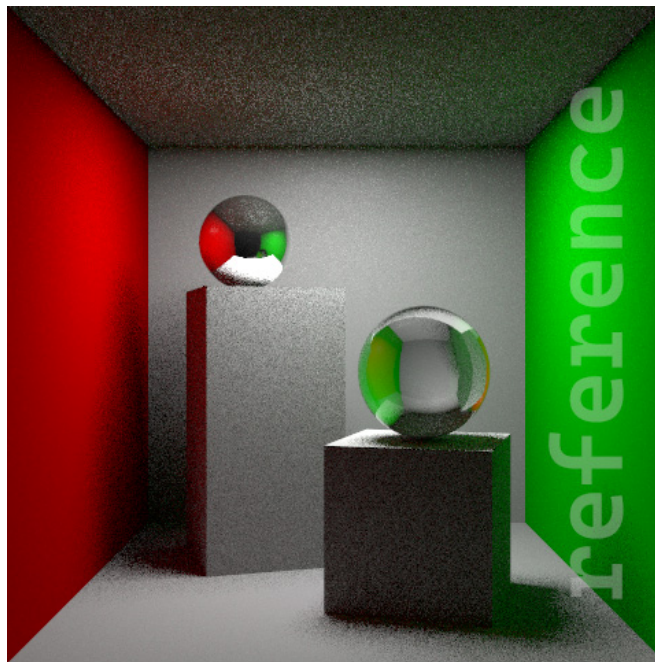
3. Next, we transform the local sampled direction back to **global** coordinates:

Form the transformation matrix with the x, y, z-axes you got from step 1 with the `Matrix()` function from `mathutils` (remember to import it!). Left multiply it to the vector you got from step 2 to get the global ray direction.

4. Finally, we calculate the color contribution.

Cast a ray in the direction from Step 3 above to get the raw intensity. The indirect diffuse color will be that returned, raw intensity multiplied by `r1` to account for face orientation, and then by `diffuse_color` to account for absorption. Add the indirect diffuse color to `color`, and you're done!

Render your final image with at least 16 samples and depth set to 3. It should take ~40 min for 16 samples. If you want a higher quality image, then you can set the samples to a very high count and let it run overnight, and then press `ESC` when you want to stop the render and save that result.



- ☐ **Checkpoint 3: Save the image with indirect diffuse. Discuss how this is improved from 2.2, and why it is darker around the ceiling line**

(You don't need to follow this step) In case you are wondering: with this code, we can re-render the scene from HW3 to get the following image with nicer shadow and global illumination. Of course, the point lights need to be changed to area lights.



IV. Grading (5 pts total)

This assignment will be graded on the following requirements

Complete all the checkpoints (4 pts)

1. (1 pt) **Save the image with area light (1).**
2. (1 pt) **Save the image with simple sampling (2.1). Discuss how this is improved from 1.**
3. (1 pt) **Save the image with low-discrepancy sampling (2.2). Discuss how this is improved from 2.1.**
4. (1 pt) **Save the image with indirect diffuse (3). Discuss how this is improved from 2.2, and why it is darker around the ceiling line.**

Quiz Question (1 pt)

The TAs will choose one of the following questions during the grading session. Please be prepared to give a one-minute answer with your partner.

1. Why is light tracing inefficient?
2. Explain the process of bidirectional ray tracing. What two types of tracing is it a combination of?
3. In Iterative Radiosity, why might “Sorting and Shooting” converge to an acceptable solution faster than randomly choosing the order in which surfaces distribute light?
4. What is albedo? How is it computed when the BRDF is diffuse?
5. What's the difference between direct illumination and global illumination?
6. Can you explain how a light map is used and why it can be helpful in rendering scenes? What are some advantages and disadvantages?
7. How does Monte Carlo Integration compare with Newton-Cotes quadrature? Why do we not use Monte Carlo for low dimension problems? (Hint: think in terms of dimensionality and accuracy)
8. How can Photon Mapping render different colors?