Homework 2: Ray Tracing

By: Junwen Yu and Yuqi Meng Instructor: Prof. Jeong Joon Park

EECS 498-014 Graphics and Generative Models

2024 Fall

v1.0.2

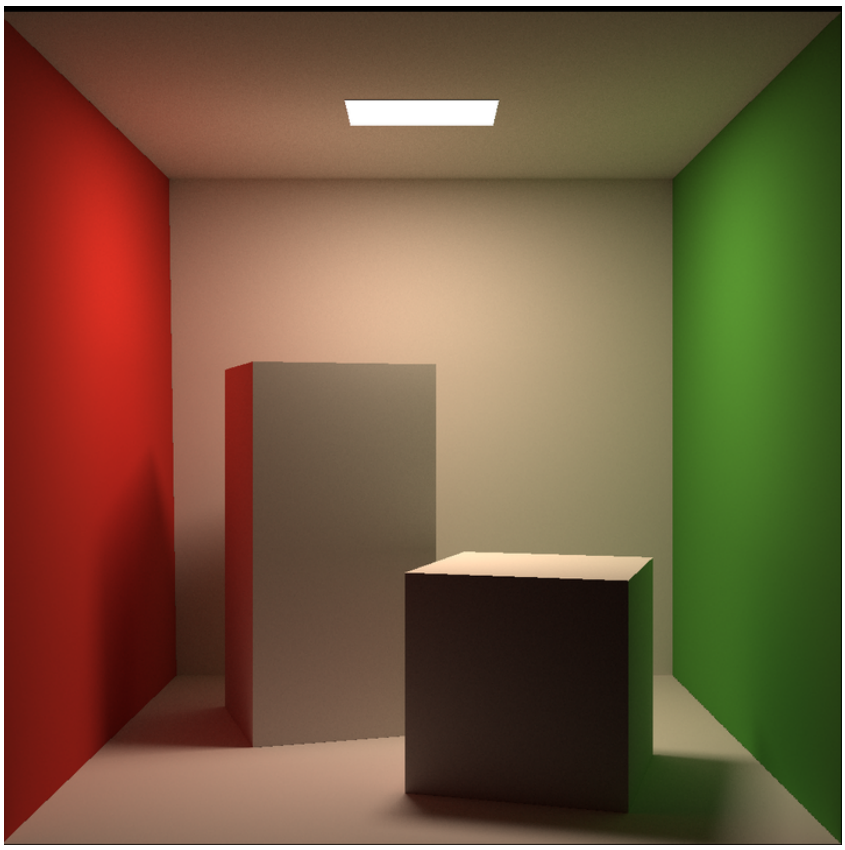
Contents

1 Objectives

This homework is a 2-week project on rendering the cornell box scene by implementing the path tracing algorithm. The learning objectives of this homework are to:

- Gain theoretical familiarity with the rendering equation
- Obtain hands-on experience for solving the physics problems by mathematical formulas

After completing the tasks, you will be able to render an image like this:



2 Instructions

2.1 Downloading the Code

You can download the starter code from the homework repo.

2.2 C++ Environment

You can choose to complete this homework either on the remote CAEN environment, or on your local environment. We will use CAEN as the standard environment for compilation and execution, but you are encouraged to set up the project in your local environment for better coding experience.

2.2.1 CAEN Environment

We strongly encourage you to use the SSH extension in VSCode to connect to CAEN.

1. Download VSCode if you haven't already
2. Install the Remote - SSH extension in VSCode. Here is the official guide for managing extensions.
3. Before connecting to the CAEN Linux, make sure that you are using the campus network, or has connected to the network through UMVPN.
4. Connect to CAEN: first, open the Command Palette by the key combination `Ctrl+Shift+P` (or `Command+Shift+P` on Mac);

   then, search for `Remote-SSH: Connect to Host` and run the command;

   finally, enter `uniquename@oncampus-course.engin.umich.edu` and follow the prompts that appear to login. More details regarding the final step can be found here.

2.2.2 Local Environment

You may also compile and run the code in your local environment with a compiler that supports C++17 (CAEN uses GCC 8.5.0 by default).

2.3 Building the Project

2.3.1 Execute Without Debugging

The simplest way to build the project is to compile and link all the `.cpp` files. To do this, first navigate to the project folder (the one that contains the `.cpp` files) in your terminal, and then type

```
g++ -std=c++17 -O3 -o rayTracing *.cpp
```

and execute. It will generate an executable called `rayTracing` in the current folder, which can be executed through `./rayTracing` in the terminal.

2.3.2 Debugging your code

You may want to debug your code if a crash occurs, or to track intermediate variables. To do this, we recommend you to make use of the CMakeLists.txt we provide. Here is an example based on VSCode:

1. In VSCode, install the CMake Tools extension.
2. Navigate to the source folder (the one that contains the `CMakeLists.txt`). The CMake Tools should automatically detect the CMakeLists.txt and prompt you to configure the project. If it doesn't, restart VSCode.
3. Use the CMake Tools to configure, build, and run the program. Note that CMake doesn't build and execute the program in the current folder - instead, it creates a `build` folder and runs from there. Therefore, you need to adjust the relative paths of the files accordingly, which is described in more details in the following sections.
4. Use Debug mode to run your program when debugging, and Release for performance.

3 Overview

In this homework, you'll work through 4 tasks to implement the main body of the path tracing algorithm, and accelerate the rendering process with importance sampling.

3.1 Config

You can find `Config.h` in the source folder. It exposes the following options:

- SPP: sample per pixel
- SEED: the seed used to generate random numbers
- RESOLUTION: resolution of the image
- MAX_DEPTH: the maximum number of bouncing for each ray
- RR: parameter for Russian Roulette. We pull the trigger (discard the result) if we get a random number higher than this. Should be between (0, 1].
- OBJ_PATH: path of the .obj file with respect to the executable. If you builds and executes your program in place, then the path will be like `./models/cornellBox/CornellBox-Original.obj`, where `./` stands for the current folder. However, other build tools like CMake Tools may run your executable at `build/RayTracing`, or even `build/Release/RayTracing`. In these cases, the path should be set to `"../models"`, or `"../../models"`, respectively, where `"../"` represents the parent folder.

- MTL_SEARCH_DIR: the relative path to the folder that contains the .mtl file.
- OUTPUT_PATH: the relative path to the image that you want to write your output to. The image will be created if it doesn't yet exist. Note that the existing content will be overwritten.

Generally, you'll need to adjust the three paths based on your configuration. Then, you may use a lower `SPP` when testing correctness, and set it back to generate better results. You may also need to modify `MAX_DEPTH` later in the homework.

4 Homework Overview

In this Homework, we are going to implement Monte Carlo path tracing step by step.

We use the standard cornell box as our test scene. The model is specified by `CornellBox-Original.obj`. In our world coordinates, the x axis is towards the right, y is towards the up, and z is pointing outwards from the screen. The entire scene spans roughly from $(-1, 0, -1)$ to $(1, 2, 1)$.

Intuitively, to generate the rays we are going to trace, we place the image in front of the scene, and trace through each pixel to find out what we can see from it.

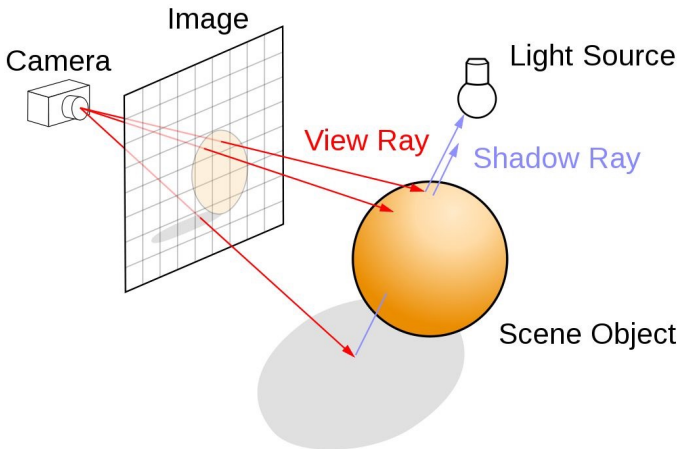
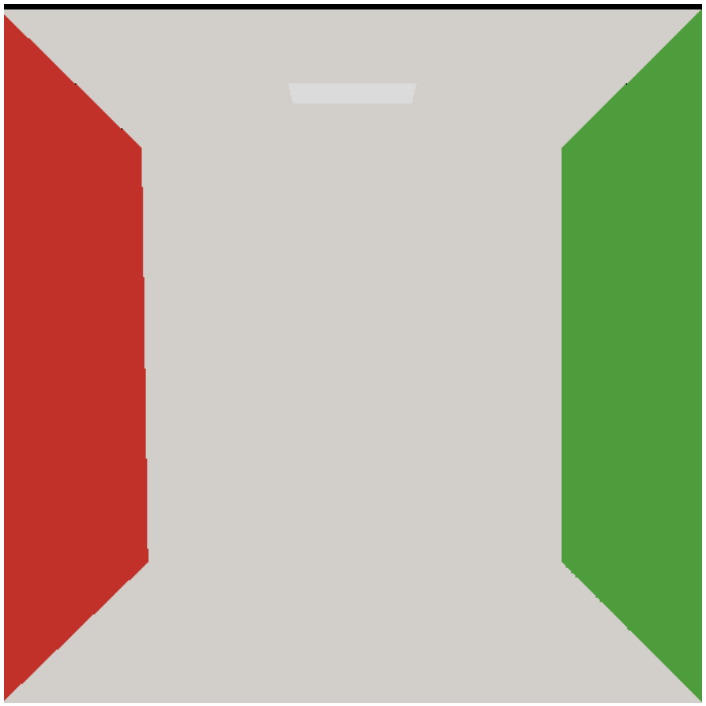


Figure 1: Credit: Nvidia Developer

5 Tracing the Rays

In this part, you will follow the rays cast from the camera, and retrieve the properties of the objects they hit. You will render the scene with the diffuse color of the objects. It aims to help you understand the basic process of tracing a ray, and how to effectively use the helper functions.

You will get an image similar to this:



We cannot distinguish the boxes from the walls as they share the same diffuse color. Therefore, we can only see the colored walls and the light.

5.1 The `trace` function

In this part, you will only need to modify the `trace` function found in `Scene.cpp`.

In the entrance of the program, `main.cpp`, we project each pixel of the image to a 3D location, generate a ray from the camera to it, invoke `trace` on the ray, and convert the result to the final color of each pixel. In a full path tracing process, each ray from the camera will bounce on the surfaces and accumulate the radiance. For now, we simply stop at the first surface each ray encounters, and returns the diffuse color of that surface as the result.

5.2 Helper Functions and Classes

5.2.1 Vec3

`Vec3` is essentially a vector consisting of 3 floats. It can be used to represent an RGB color, a point in the 3D space, a 3D vector representing a direction, etc.

Here are some examples of using `Vec3`:

```
/* Addition */
Vec3 color1, color2;
Vec3 finalColor = color1 + color2;

```

```
/* Subtraction */
Vec3 point1, point2;
Vec3 vec = point2 - point1;

/* Negation */
Vec3 normal;
Vec3 oppositeNormal = -normal;

/* Scaling */
Vec3 vec;
float scale;
Vec3 scaledVec = vec * scale;

/* Normalization */
Vec3 longVec;
longVec.normalize(); // this directly modifies the vector itself

/* Dot Product */
Vec3 v1, v2;
float dotProd = v1.dot(v2);

/* Cross Product */
Vec3 v1, v2;
Vec3 crossProd = v1.cross(v2);

/* Getting Length */
Vec3 longVec;
float length = longVec.getLength();
```

5.2.2 Ray

`trace` has 3 parameters, but for now we only need to use `ray`. ray is an instance from the class `Ray`. It has two member properties: a `pos` representing the starting point of the ray (in this part, it is always the camera's position), and a `dir`, which is a normalized vector representing the direction in which the ray travels.

To construct a ray:

```
Vec3 pos;
Vec3 dir;

// if dir is not yet normalized...
dir.normalize();

Ray ray(pos, dir);
```

```
// Alternatively
Ray ray = {pos, dir};
```

5.2.3 Intersection

`trace` itself is a member function of the class `Scene`, so we can access other functions within this class as well. Here, we are going to use `getIntersection`, which takes a `Ray` object as parameter, and returns an `Intersection` object. An `Intersection` object contains the information describing the intersection of the ray with the scene.

```
struct Intersection {
    bool happened = false; // false if the ray misses the scene
    float time = std::numeric_limits<float>::max(); // the time the ray travels before hitting a
    surface
    const Object* object = nullptr; // the object that the surface belongs to
    Vec3 pos; // the position of the intersection in the world coordinate
    const Mesh* mesh = nullptr; // the mesh that the intersection belongs to

    /* helper functions*/
    Vec3 getNormal() const;
    Vec3 getDiffuseColor() const;
    Vec3 getEmission() const;

    Vec3 calcBRDF(const Vec3& inDir, const Vec3& outDir) const;
};
```

In your implementation, you should first find the intersection through a helper function we provide:

```
Vec3 Scene::trace(const Ray &ray, int bounces, bool discardEmission) {
    // other code...
    Intersection inter = getIntersection(ray);
}
```

then, you should check `inter.happened` property to see whether the intersection has really happened. If not, you should directly return a zero vector. This can be done through

```
Vec3 Scene::trace(const Ray &ray, int bounces, bool discardEmission) {
    // construct a zero vector
    return Vec3(0.0f, 0.0f, 0.0f);
}
```

As the default constructor of Vec3 creates a zero vector, you can make it simpler:

```
return Vec3();
```

Or even simpler:

```
// C++ deduces the return type and default constructs the object
return {};
```

If the intersection did happen, you should directly return its diffuse color. This property is per-object, and you can access it like this:

```
Vec3 diffuseColor = inter.object->kd;
```

Since this involves a pointer indirection and may be confusing for those who don't have much experience with C/C++, we provide a helper function to get the diffuse color, which essentially does the same thing:

```
inter.getDiffuseColor();
```

Likewise, you can access the emission of the object through either of the two ways:

```
inter.object->ke;
inter.getEmission();
```

We will use `getEmission` in the following parts. In addition, we are not going to explicitly use `getDiffuseColor` again (it is only for this part), as it will be encoded into the BRDF.

5.3 Task

Implement the `trace` function as instructed to get the same image as provided above. Attach the code of the entire `trace` function (including the function signature) to your report. You don't have to submit the image for this part.

If you need to add any helper function, write it in `Scene.cpp`, and include the function in your report as well. Do not put it in any other file, as your will only submit the `Scene.cpp` to Canvas.

6 Direct Illumination

In this part, we are going to render the scene with direct illumination. When the ray hits a surface, instead of returning the diffuse color of it, we are going to shade it by estimating the direct lighting, according to the rendering equation.

6.1 DI from the Rendering Equation

Consider the outgoing radiance at some surface point p along the direction ω_o:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\Omega^+} L_i(p, \omega_i) f_r(p, \omega_i, \omega_o)(n \cdot \omega_i) d\omega_i$$

the L_is can directly come from a light source, which is a direct lighting contribution, or come from radiance reflected by other surfaces, which is called indirect lighting. We will only handle the direct lighting in this part.

6.2 Sampling a Uniform Direction

To estimate the integration, i.e., the total incident direct radiance, we are going to use Monte Carlo sampling. In practice, for every intersection, we shoot a single ray from the point, and see whether it intersects with the light source. If it does, we calculate the lighting contribution of this ray.

The integration is defined over the hemisphere oriented towards the normal of the surface. As we want an unbiased estimation of the integration, we need to uniformly sample a direction within the hemisphere as well. "Uniform" means that the probability density at any solid angle within the hemisphere is a constant, i.e., $p(\omega) = \frac{1}{2\pi}$.

There is a predefined but not fully implemented function for the sampling, the `randomHemisphereDirection` in `Math.cpp`. It uniformly samples a spherical coordinates (azimuth, elevation) in the hemisphere oriented towards the positive z axis, then converts it to the local Cartesian coordinates, and finally converts it to world coordinates. You need to complete this function by randomly generate a pair of spherical coordinates in the positive z hemisphere that could produce a uniformly generated direction, described by the following formula:

$$\text{azimuth} = 2\pi p$$

$$\text{elevation} = \arccos(q)$$

, where p and q are random floating points from 0 to 1. We have provided the helper function to generate them.

```
float firstRnd = Random::randUniformFloat();
float secondRnd = Random::randUniformFloat();
```

π is already defined as a global variable `PI`, so you can use it directly. $\arccos$ is a built-in function, which can be invoked through `acos(/*a floating point number as parameter*/)`.

Understanding the mathematics behind the formula is optional. You can check this chapter from Ray Tracing: The Rest of Your Life for a brief explanation.

You should not use the parameter `normal`, as that is only for converting local coordinates to the world, and we have already set that step up for you.

6.3 Estimating DI

Now, we can estimate the rendering equation like this:

$$L_o(p, \omega_o) \approx L_e(p, \omega_o) + \frac{1}{pdf(\omega_i)} L_i(p, \omega_i) f_r(p, \omega_i, \omega_o)(n \cdot \omega_i)$$

where L_o is the return value of the function, and p is the intersection of the camera ray with the scene. w_i is a uniformly random direction we generated around the normal of the intersection point. We will shoot a ray in this direction, and use the emission of the intersection point (if it intersects) as L_i.

Your task is to modify the `trace` function to implement this process. For the three parameters, you will still only need to use the first, `ray`. The function should return a radiance instead of a diffuse color.

To get the value of the BRDF, f_r, you can call the helper function[1] like this:

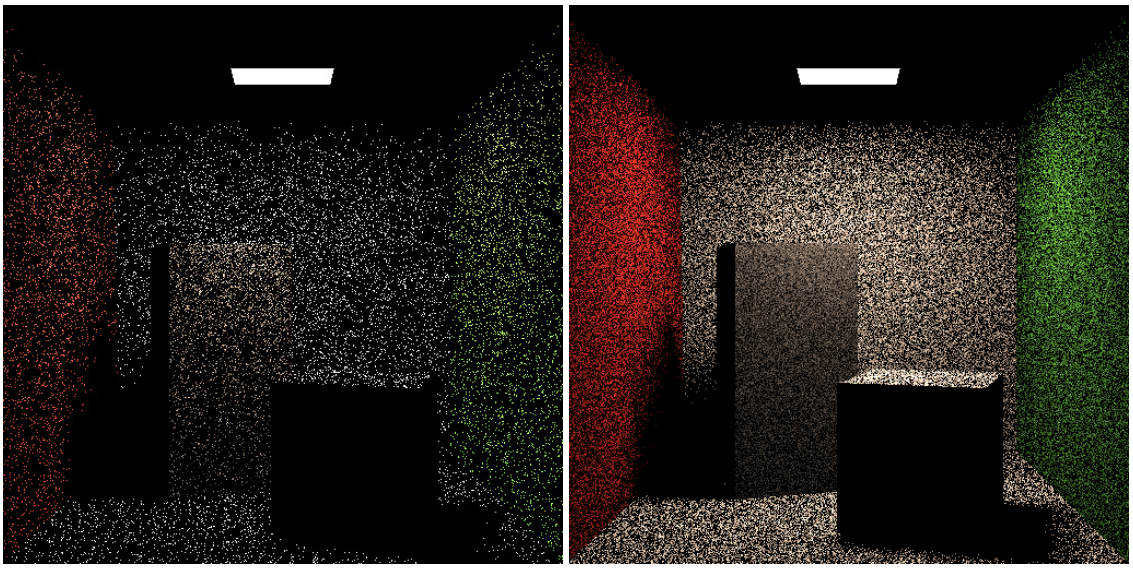
```
Intersection inter = getIntersection(cameraRay);
Vec3 brdf = inter.calcBRDF(-secondRay.dir, -cameraRay.dir);
```

The function takes two directions, one is that of the ray incident to the surface (ω_i), and the other is that of the outgoing ray (ω_o). We need to add negative signs here, since we are tracing back from the camera to the light source.

To calculate the dot product (the cosine term, $n \cdot \omega_i$):

```
float cosineTerm = secondRay.dir.dot(inter.getNormal());
```

Rendering the scene with 16 and 128 SPP looks like this:



(Optional) **Thought question** : why does the image of 16 SPP look darker than 128 SPP?

Thought questions are optional, and will not affect your grade.

[1] for those who are curious, check this chapter from PBRT for diffuse (Lambertian) model

6.4 Task

- Modify `randomHemisphereDirection` in `Math.cpp`
- Modify the `trace` function
- Render the image with exactly **32 SPP**
- (Optional) Answer the thought question above

Attach the code of the entire `trace` function, together with the rendered image in your report. You don't have to submit the code for calculation in `Math.cpp`.

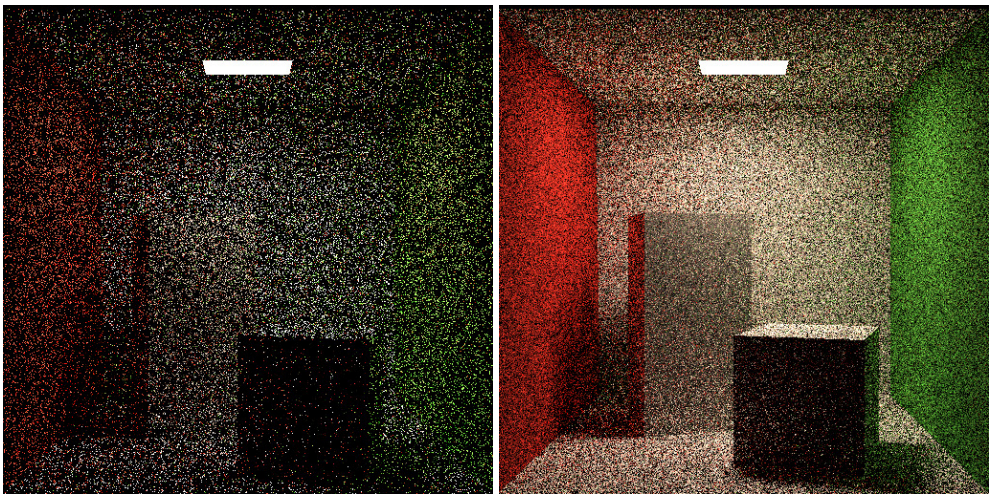
7 Global Illumination

In this part, we will let the ray bounce multiple times until it misses the scene, or reaches the maximum depth we set in `Config.h`. You will need to modify the `trace` function, and use the second parameter `bouncesLeft`. It represents the number of remaining bounces for the ray. When it reaches 0, the function should return a zero vector.

In the previous part, we have generated a ray from the intersection, but we just use it to get direct lighting by finding the next intersection and stopping there. Now, we are going to continue tracing this ray and calculate the radiance it carries. This process is what the `trace` function itself is describing, so we are going to recursively invoke `trace` here.

```
// pseudocode
nextRay = Ray(inter.pos, Random::randomHemisphereDirection(inter.getNormal()))
L_i = trace(nextRay, bouncesLeft - 1, false)
```

Rendering in 16 and 256 SPP gives:



(Optional) **Thought question**: compare the results to direct illumination only, what new features can you identify, and why do we have them? For example, how does the ceiling get illuminated?

7.1 Task

- Modify the `trace` function
- Render the image with exactly **32 SPP**, and `MAX_DEPTH` at least 4
- (Optional) Answer the thought question above

Attach the code of the entire `trace` function, together with the rendered image to your report.

8 Acceleration

More samples per pixel will reduce the noise, but cause the program to run slower. Therefore, we want to make the algorithm faster without changing the SPP.

8.1 Sampling the Light

In the rendering equation, after we find the intersection of the camera ray with the scene (the point p), we estimate L_i by randomly shooting a ray in the hemisphere. If it hits a light source (as what we considered in the direct illumination part), then it contributes a lot to the incident radiance. Otherwise, the ray might have to bounce many times before hitting a light source, or even end up missing the scene, and will not likely contribute much. Therefore, we can separate L_i into direct radiance, and radiance from the rays that bounced one or multiple times:

$$L_i = L_0 + \sum_{k=1}^{\infty} L_k$$

, where k stands for the number of bounces. Each time we estimate L_i, we shoot 2 rays to calculate the two parts separately.

8.1.1 Indirect Radiance

The first ray is to estimate the sum of all indirect radiance. It works the same way as in the previous part (using a recursive call), but it should not take into account the direct radiance. Therefore, you need to set the third parameter, `discardEmission`, to `true` in the recursive call. In your implementation of `trace`, your result should not include the emission of the intersection point if `discardEmission` is `true`.

8.1.2 Direct Radiance

The second ray is to estimate the direct radiance. Since we know the direct radiance always comes from the light, we don't randomly select a direction to find the light source. Instead, we sample a point on the light source and trace a ray towards that point. This way, the domain of the integration is defined on the area of the light source, instead of the solid angle across the hemisphere. This transformation is described by:

$$L_o = \int_{\Omega^+} L_{di}(p, \omega_i) f_r(p, \omega_i, \omega_o)(n \cdot \omega_i) d\omega_i = \int_A L_{di}(p, \omega_i) f_r(p, \omega_i, \omega_o)(n \cdot \omega_i) \frac{(-n' \cdot \omega_i)}{d^2} dA$$

, where n' stands for the surface normal of the light source (which you can get from the `lightSample` described below), and d stands for the distance from the object surface to the light source. The minus sign before n' is because ω_i is defined to be outgoing from the surface and thus incident to the light source, so $(n' \cdot \omega_i)$ alone is negative.

To estimate this integration, we are going to do Monte Carlo sampling as before, and take only one light sample. According to the formula, you will divide the result of the sample by its probability density function to estimate the total contribution of direct radiance. In this case, it should be the inverse of the total area of the light source. This property is a member variable of the `Scene` class and has already been set for you. You can access it like this:

```
float pdfLightSample = 1 / lightArea;
```

However, radiance from the light sample may be blocked by other objects before hitting the current surface and make no contribution. To check this, we set up a shadow ray, which is essentially a ray from the current surface point to the light sample.

Here is how to set up a ray towards the light source:

```
// gets a sample on the light
// This is not an actual "intersection". We just use this class to pass the info, like the
    emission of the light and the position of the sample.
Intersection lightSample = sampleLight();
Vec3 lightDir = lightSample.pos - inter.pos;
float distanceToLight = lightDir.getLength();
// always use a normalized direction to construct a ray!
lightDir.normalize();
Ray rayToLight(inter.pos, lightDir);
```

We will then check the intersection of the shadow ray with the scene. If that is indeed the light sample, we can calculate the direct radiance accordingly. Otherwise, the radiance is blocked. Alternatively, we can simply trace this ray and ask for the direct radiance it carries.

8.2 Importance Sampling

Let's take another look at the rendering equation.

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\Omega^+} L_i(p, \omega_i) f_r(p, \omega_i, \omega_o)(n \cdot \omega_i) d\omega_i$$

Recall that in 6.2 we sample w_i uniformly in the hemisphere. Here, we can do importance sampling based on the $(n \cdot \omega_i)$ term. Intuitively, the incoming radiance from directions closer to the surface normal have larger values on this term, so they are likely to contribute more.

Your task is to complete the `cosWeightedHemisphere` function in `Math.cpp`, such that the probability density function for any direction is proportional to $(n \cdot \omega_i)$. Here is the corresponding formula:

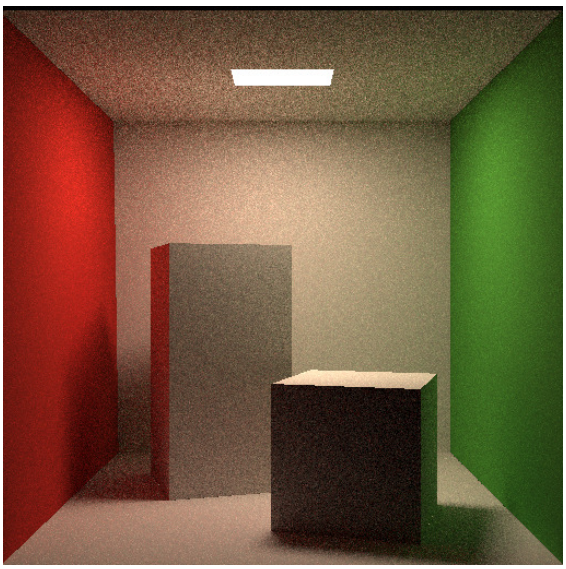
$$\text{azimuth} = 2\pi p$$

$$\text{elevation} = \arccos(\sqrt{q})$$

, where p and p are uniform random numbers as described previously. You can (optionally) check this chapter for more details.

Now, you can use the same formula from 6.3 to implement the `trace` function. Since we are using the cos-weighted importance sampling to generate ω_i, $pdf(\omega_i)$ would be $\frac{(n \cdot \omega_i)}{\pi}$, instead of just $\frac{1}{2\pi}$.

Rendering with only 32 SPP:



8.3 Task

- Modify `cosWeightedHemisphere` in `Math.cpp`
- Modify the `trace` function
- Render the image with at least 32 SPP. Feel free to use a larger SPP in this part to achieve better results.

Attach the code of the entire `trace` function, together with the rendered image to your report. You don't have to include the code in `Math.cpp`.

9 (Optional) Exploration Task

There are many aspects to explore in this homework. However, since the code is not designed to be general-purpose, you may need to modify the existing functions and even create your own. Here are some suggestions, but feel free to develop your own ideas. We also provide the estimated difficulty (from 1 to 5, the higher the more difficult) for your reference.

- **Multi-threading acceleration**. Conceptual Difficulty: 2, Implementation Difficulty: 1.

  You may notice that rendering any pixel of the image is independent of rendering the other, which means that we can render multiple of them in parallel on a multi-core system. Try to modify the main loop in the `main` function to achieve this. For multi-threading functionalities, you may use the `thread` library from stl.

- **Support for spheres**. Conceptual Difficulty: 1, Implementation Difficulty: 3.

  Currently the project only supports triangle meshes. However, it would be interesting to render spheres in the scene as well. To achieve this, you may need to modify the acceleration structure so that it can describe intersection between the ray and the sphere.

- **Physically-based Rendering (PBR)**. Conceptual Difficulty: 4, Implementation Difficulty: 2.

  Currently we only have diffuse surfaces, which looks a bit too flat. Therefore, we can modify the `calcBRDF` function based on a BRDF that more precisely describes the reflection, and supports glossy or reflective surfaces. We suggest you to use the Cook-Torrance BRDF, as it's relatively simple and provides good visual effects. There are many helpful tutorials on this topic, like this from LearnOpenGL.

- **Participating Media**. Conceptual Difficulty: 4, Implementation Difficulty: 3.

  Our path tracing algorithm assumes that the ray doesn't change between surfaces. This is not true if there is any participating media in the scene, like fog or smoke. The media may absorb and scatter the ray, providing completely different visual effects. This lecture from CMU is a good point to start with.

Feel free to discuss with the TAs in advance to see if your idea is feasible. We will grade the exploration task based on both efforts and correctness.

10 Submission and Grading

10.1 Submission

- The entire `trace` function, including the function signature, after completing each of the 4 parts, respectively. If you use any helper functions written by yourself, please include them in your report and code submission as well.
- The rendered images for direct illumination (32 SPP), global illumination (32 SPP), and acceleration (at least 32 SPP).

Gradescope submission:

Submit a single pdf report that includes your code text and images, as described above.

Canvas submission:

Submit your `Scene.cpp` file to Canvas. Put all your helper functions (if any) in this file. We will replace the `Scene.cpp` in the code framework with your version to test your code, so please make sure it compiles.

Exploration Task Submission

The exploration task should be submitted separately from the required part, through specified assignment links on both the Gradescope and Canvas. You'll need to submit a short report to Gradescope that describes your efforts and results, and all your code files (including the framework) in a `.zip` file to Canvas.

10.2 Grading

- Diffuse Scene (10%)
- Direct Illumination (30%)
- Global Illumination (20%)
- Acceleration (40%)
- Exploration (extra credits, 10%)