Project Report Basic Pathtracer

Zhen Tong 2023-7-9

[Project Requirement]

Ray Generation and Scene Intersection

Generating Camera Rays

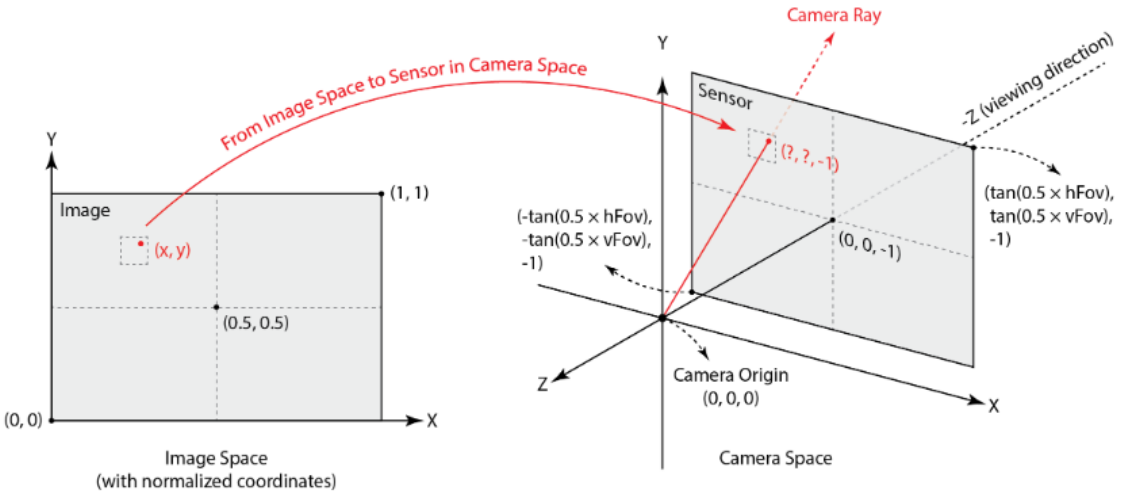


Figure from https://cs184.eecs.berkeley.edu/sp23/docs/proj3-1-part-1

In camera space, bottom left corner is at $(-tan(\frac{hFov}{2}), -tan(\frac{vFov}{2}), -1)$, $(hFov, vFov)$ are field of view angles along X and Y axis. First map the Image space point to the Camera Space point, and normalize the ray direction vector.

code of `Camera::generate_ray(...)` in *src/pathtracer/camera.cpp*.

```
Ray Camera::generate_ray(double x, double y) const {
    float u = 2 * tan(hFov * PI / 180 / 2) * (x - 0.5);
    float v = 2 * tan(vFov * PI / 180 / 2) * (y - 0.5);

    Vector3D camera_ray_direction(u, v, -1);
    Vector3D world_ray_direction = c2w * camera_ray_direction;
    world_ray_direction.normalize();
    Ray r = Ray(pos, world_ray_direction);
    r.max_t = fClip;
    r.min_t = nClip;
    return r;
}
```

Generating Pixel Samples

The integrate radiance of a pixel can be approximated from a average of N_{sample} sample. Therefore iterate the pixels in the pixel space, random generate sample point $\{(x_i, y_i)\}^N$around the pixel, find their corresponding ray, and average their sum of radiance.

$$\int\int r(x,y)dxdy \approx \sum_{i=1}^{N_{sample}} r(x_i, y_i),$$

code of `PathTracer::raytrace_pixel(...)` in *src/pathtracer/pathtracer.cpp*.

```
void PathTracer::raytrace_pixel(size_t x, size_t y) {

  int num_samples = ns_aa;          // total samples to evaluate
  Vector2D origin = Vector2D(x, y); // bottom left corner of the pixel
  Vector3D radiance(0, 0, 0);
  for (int i = 0; i < num_samples; i++) {
      Vector2D random_sample = gridSampler->get_sample();
      Vector2D pixel_sample = origin + random_sample;
      Ray r = camera->generate_ray(pixel_sample.x / sampleBuffer.w, pixel_sample.y /
sampleBuffer.h);
      r.depth = max_ray_depth;
      radiance += est_radiance_global_illumination(r);
  }
  radiance /= num_samples;

  sampleBuffer.update_pixel(radiance, x, y);
  sampleCountBuffer[x + y * sampleBuffer.w] = num_samples;
}
```

Ray-Triangle Intersection

By Moller Trumbore Algorithm, we can compute the intersection point. The formula means a point on the ray is hit on the triangle.

$$\vec{O} + t\vec{D} = (1 - b_1 - b_2)\vec{P_0} + b_1\vec{P_1} + b_2\vec{P_2}$$

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{S_1 \cdot E_1} \begin{bmatrix} S_2E_2 \\ S_1S \\ S_2D \end{bmatrix}$$

$$E_1 = P_1 - P_0$$
$$E_2 = P_2 - P_0$$
$$S = O - P_0$$
$$S_1 = D \times E_2$$
$$S_2 = D \times E_1$$

O is the starting origin of the ray, i.e. the camera position, and D is the ray direction. P_0, P_1, P_2 is the vertices of triangle surface.

Ray-Sphere Intersection

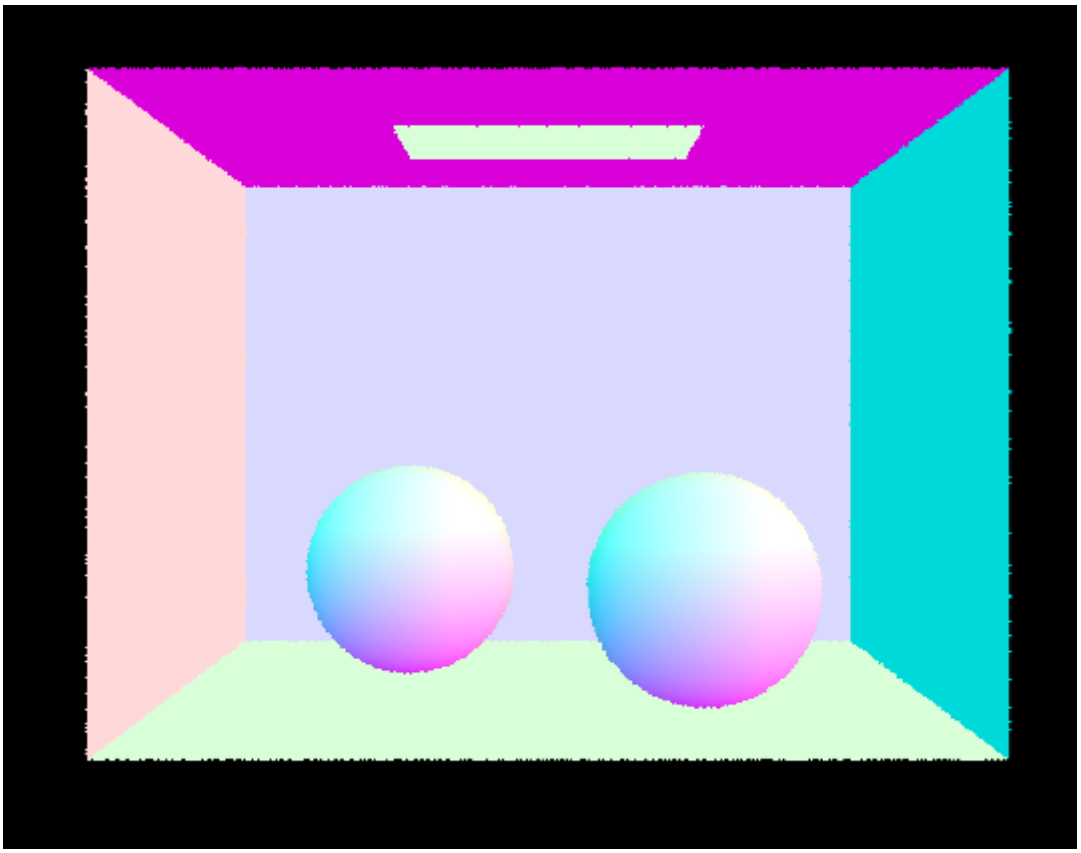
Given the center c and the radius R of the sphere, we can calculate the intersection point by solving:

$$(O - tD - c)^2 = R^2$$

```
bool Sphere::intersect(const Ray &r, Intersection *i) const {
  // TODO (Part 1.4):
  // Implement ray - sphere intersection.
  // Note again that you might want to use the the Sphere::test helper here.
  // When an intersection takes place, the Intersection data should be updated
  // correspondingly.
    if (has_intersection(r)) {
        double a, b, c, t, t1, t2, discriminant;
        a = dot(r.d, r.d);
        b = dot(2 * (r.o - o), r.d);
        c = dot((r.o - o), (r.o - o)) - r2;
        discriminant = b * b - 4 * a * c;
        if (discriminant > 0) {
            t1 = (-b + sqrt(discriminant)) / (2 * a);
            t2 = (-b - sqrt(discriminant)) / (2 * a);
            t = t1 < t2 ? t1 : t2;
        }
        else {
            t = -b / (2 * a);
        }
        //compute surface normal;
        Vector3D intersection_point = r.o + t * r.d;
        Vector3D normal = intersection_point - o;
        normal.normalize();
        //fill in intersection data structure.
        i->t = t;
        i->primitive = this;
        i->bsdf = get_bsdf();
        i->n = normal;
        return true;
    }
    else {
        return false;
    }
}
```

Run and Show Image

```
./pathtracer -r 800 600 -f CBspheres.png ../../../dae/sky/CBspheres_lambertian.dae
```



Bounding Volume Hierarchy

Constructing the BVH

Computing the ray intersection of every primitives is time-consuming, instead, we can use the binary search tree to reduce the searching time from $O(n)$ to $O(\log n)$. First we need to construct the bounding volume tree that each node is a set of primitives. The criterion of split a set of primitives into 2 subset is decided by comparing each primitive's centroid to the mean centroid of the set along the a specific split axis (x, y, z axis). The exact axis chosen is according to the minimal heuristic rule:

$$H_i = N_l V_l + N_r V_r$$
$$i \in \{x, y, z\} \text{ is the axis split along}$$

where N is the number of primitives of the left(l) or right(r) node, and V is the volume of the sub bounding box. Therefore, we can construct the bounding volume binary tree recursively, until the primitive number in one node is less than a threshold.

Intersecting the Bounding Box

After constructing the bounding boxes, we can use the intersection rule to test whether the ray hit the box:

$$\vec{o} + t\vec{d} = p$$

$\vec{o}$ is the origin of the ray, $\vec{d}$ is the direction of the ray, p a point on the plane, which is a side of the cubic. Because we cut the box along x, y, or z axis, we can simplify this formula to:

$$\vec{o}_i + t\vec{d}_i = p_i$$

$$t = \frac{p_i - \vec{o}_i}{\vec{d}_i}$$

$$i \in \{x, y, z\} \text{ is the axis split along}$$

The subscript here means the value of i dimension.

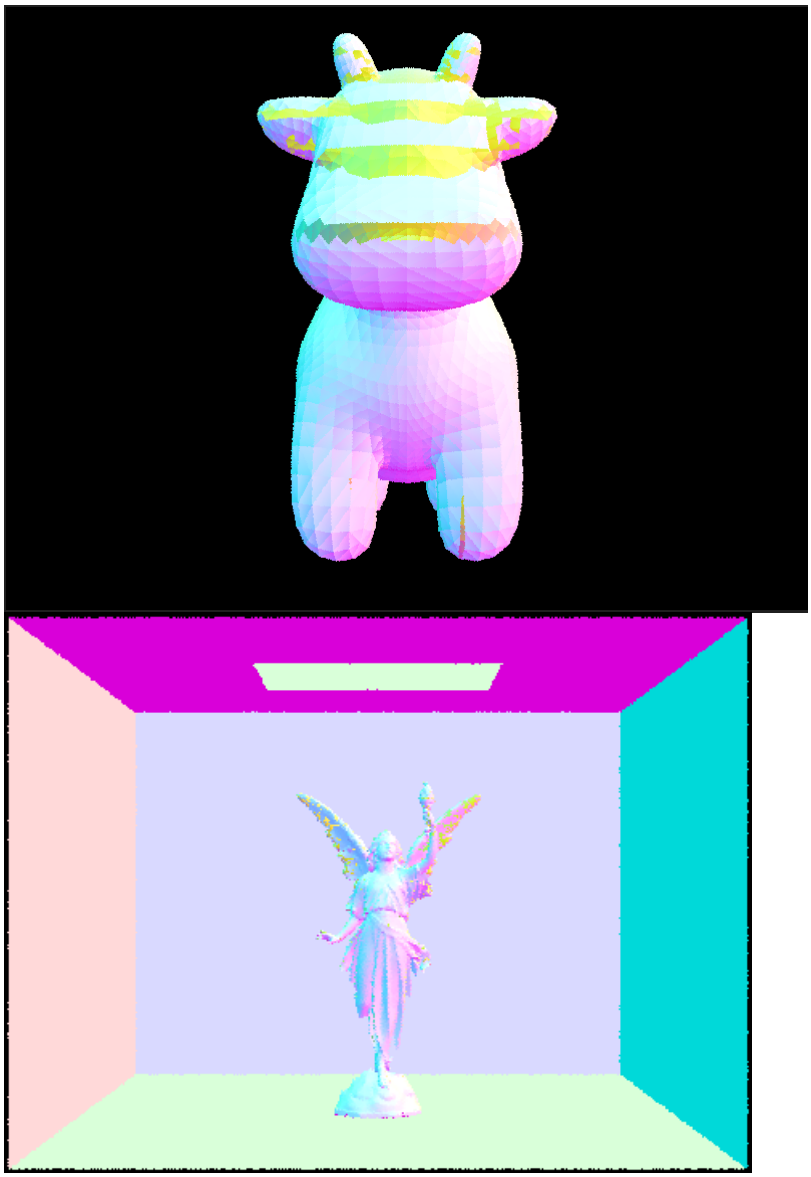
Recursive Traversal BVH

Given the root node, i.e. the universal set of the primitive bounding box, we test either the right node or the left node is hit by the ray first.

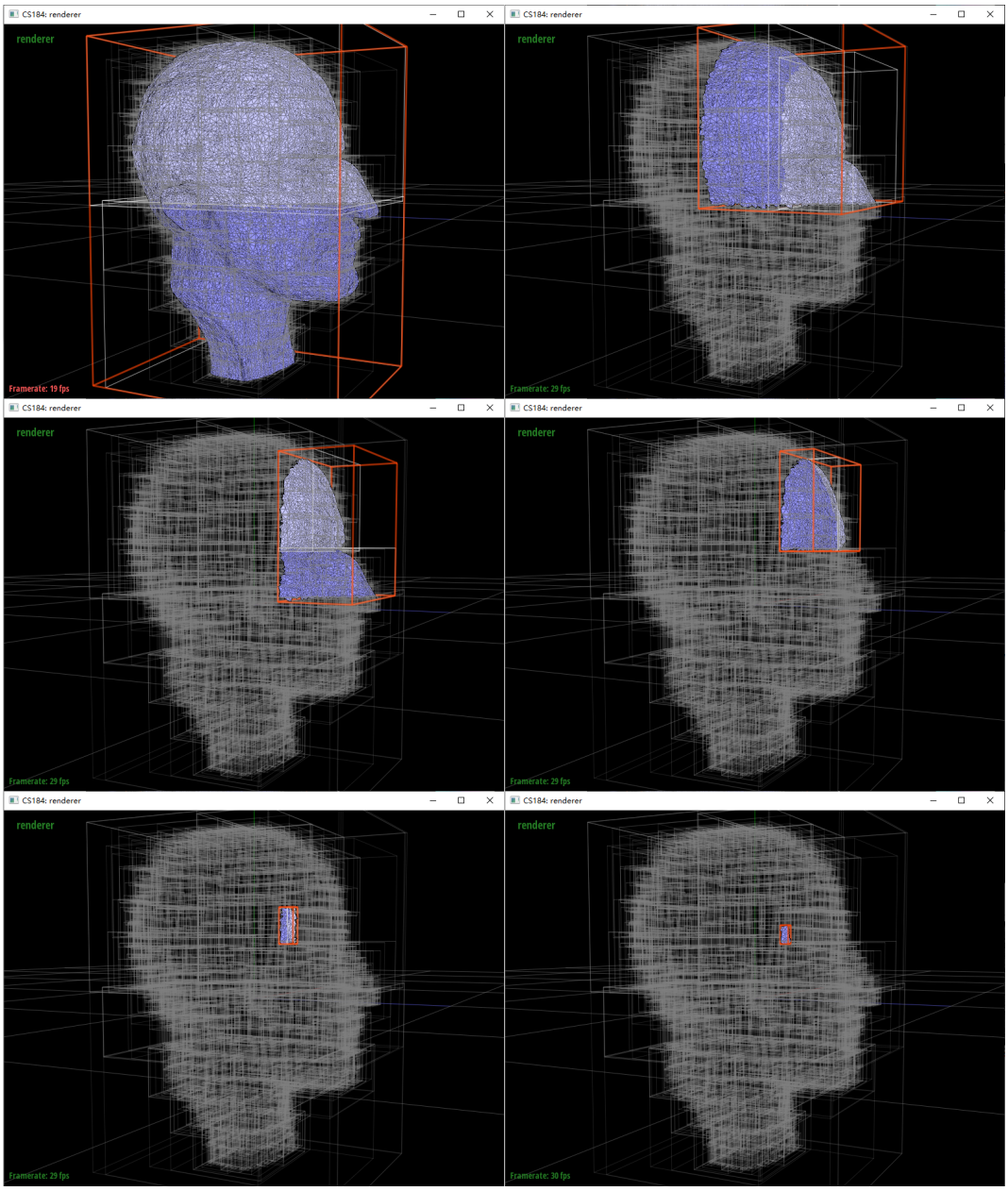
Run and Show Case

With BVH, the ray tracer is faster

```
./pathtracer -t 8 -r 800 600 -f cow.png ../../../dae/meshedit/cow.dae
.\pathtracer.exe -t 8 ..\..\..\dae\meshedit\maxplanck.dae
 .\pathtracer.exe -t 8 ..\..\..\dae\meshedit\beast.dae
```



press `v` and use ↑, →, ← to traversal the tree



Direct Illumination

Diffuse BSDF (bidirectional scattering distribution function)

The reflectance of a hemisphere is $\frac{I}{2\pi}$, because the solid angle of a hemisphere is half of a entire sphere solid angle(4π). In the diffuse BSDF model, the reflected ray is uniformly random independent with the hitting ray direction d_{in}. Therefore ray-out direction follow sampling from:

$$d_{out} = (cos(\phi)cos(\theta), cos(\phi)sin(\theta), sin(\phi))$$
$$\phi \sim U(0, \pi), \theta \sim U(0, 2\pi)$$

Zero-bounce Illumination

If the ray tracer only focus on the ray directly shot at the camera, the only thing in the picture is the light, because although ray from camera hit something that is not the light, querying its emission will return 0.

Direct Lighting with Uniform Hemisphere Sampling

In order to estimate how much light *arrived* at the intersection point, we need to integrate all the light arriving at the hitting point on surface. In practice, we use Monte Carlo estimator to approximate the integral

$$L_r(p, w_r) = \int_{H^2} f_r(p, w_i \to w_r) L(p, w_i) \cos\theta_i dw_i$$
$$\approx \frac{1}{N} \sum_{j=1}^{N} \frac{f_r(p, w_j \to w_r) L(p, w_j) \cos\theta_j}{P(w_j)}$$

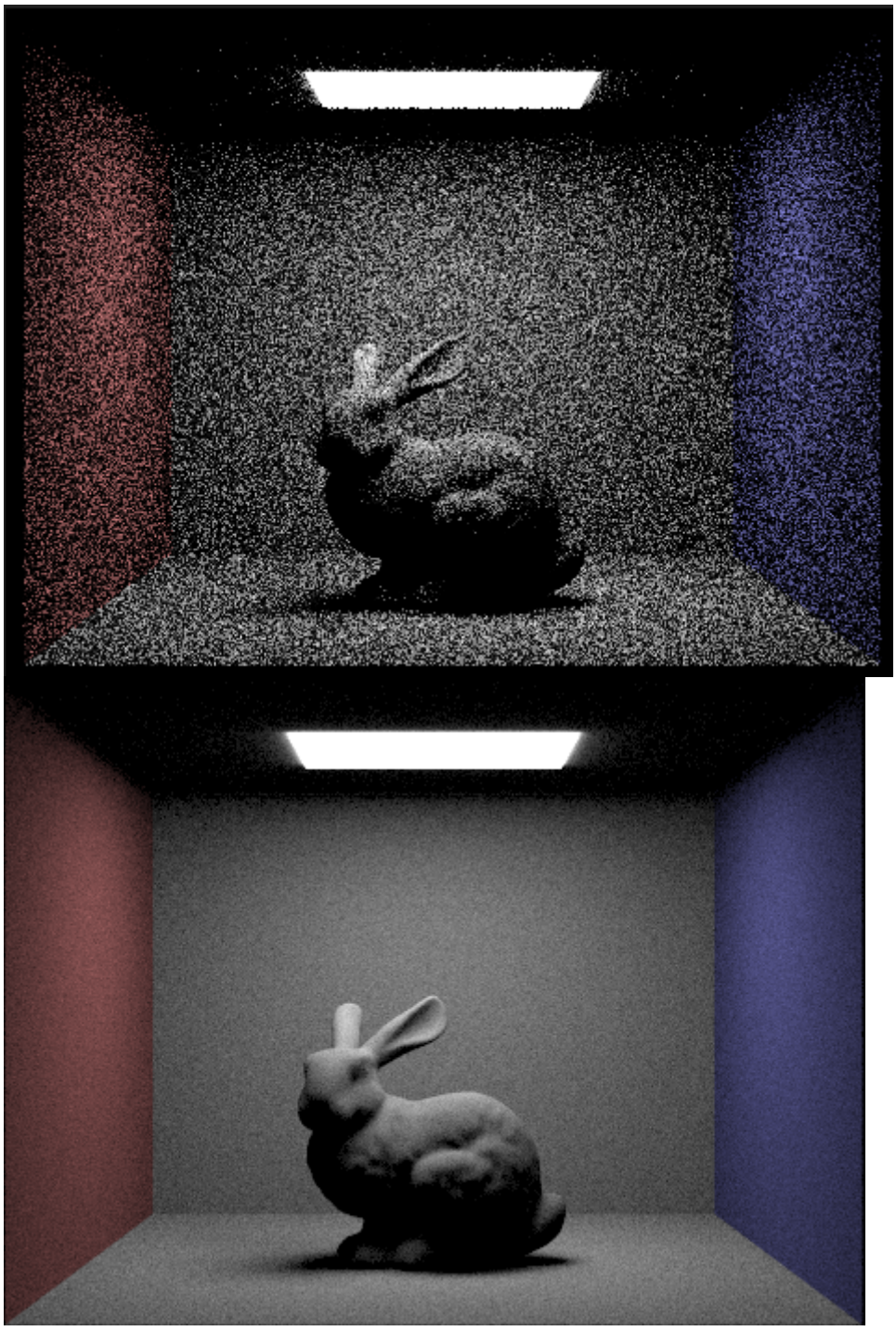
$L_r(p, w_r)$ is the reflection radiance of hitting point p, in the direction to the camera ray (the direction of the solid angle w_r). The integral subscript H^2 means integrate with hemisphere solid angle. f_r is the reflective BSDF of the intersection object surface (initialized as diffuse BSDF). $L(p, w_i)$ is the emission along the solid angle w_i from hitting point p to the light source. $\cos\theta_i$ is defined by Lanbert, to realize actual radiance hitting on per surface area. In the approximation, we sample N number of ray to imitate the integral according to the probability of the sample direction.

Sum up the zero-bounce and one-bounce Illumination, we can render the picture.

```
./pathtracer -t 8 -s 16 -l 8 -H -f CBbunny_H_16_8.png -r 480 360
../../../dae/sky/CBbunny.dae
./pathtracer -t 8 -s 64 -l 32 -H -f bunny_64_32.png -r 480 360
../../../dae/sky/CBbunny.dae
```

The bunny on the right is render with 16 camera rays per pixel, 8 samples per area light

The bunny on the right is render with 64 camera rays per pixel, 32 samples per area light



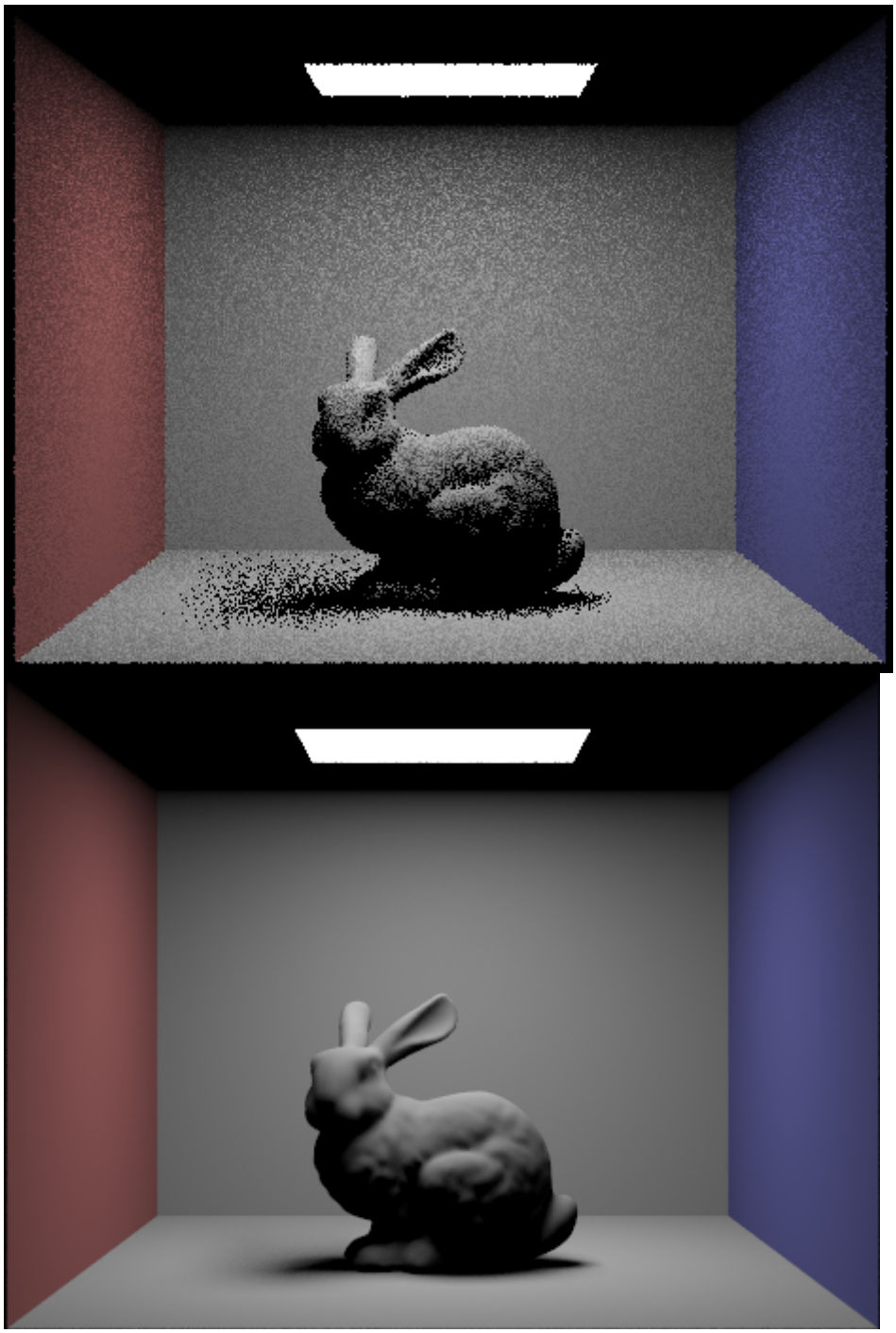
Direct Lighting by Importance Sampling Lights

The uniform hemisphere sampling is noisy, which can be improved by Importance Sampling. The uniform policy is noisy because the most of the sample ray point to non-light area. Therefore, to improve the sample efficiency, we can sample the ray point to the light source, therefore the hitting point area can always find a light as long as there is no blocking.

```
 ./pathtracer -t 8 -s 1 -l 1 -f bunny_1_1.png -r 480 360
../../../dae/sky/CBbunny.dae
 ./pathtracer -t 8 -s 64 -l 32 -f bunny_64_32.png -r 480 360
../../../dae/sky/CBbunny.dae
```

The bunny on the right is render with 1 sample per area light, 1 samples per area light

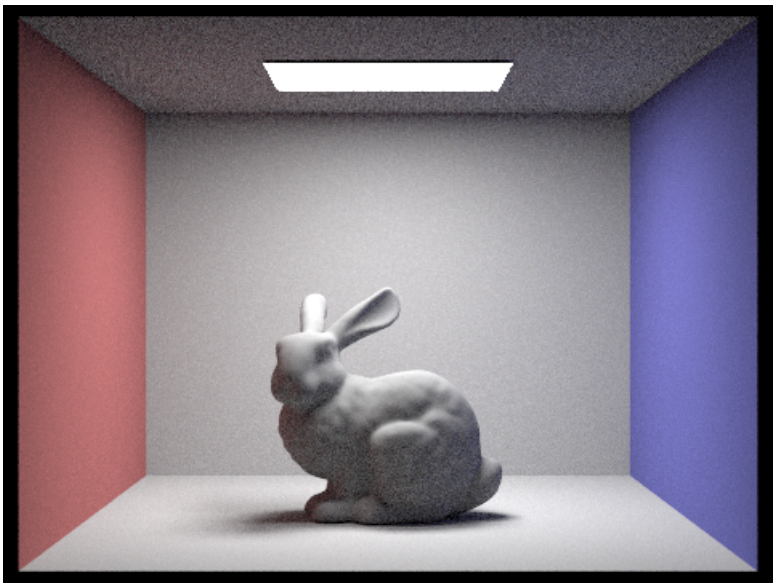
The bunny on the right is render with 64 sample per area light, 32 samples per area light



Global Illumination

In the direct illumination model, the ray only reflect once. To make the picture more real, we can use a recursion to let the ray reflect more than once, and accumulate every reflection. Until the ray depth reach the maximum, we can flip a coin to decide one more reflection for that ray. For additional reflection, we don't store the radiance of that surface, so, we cannot achieve the environment reflection effect. But the simple recursion is enough to "light up" the shade of the object.

Ray depth = 2



Adaptive Sampling

If we sample large amount of ray for single pixel, the computing time can be undesirable. We can use adaptive sampling strategy to improve. For each pixel, we can record the mean and variance during sampling. If the color converge before sampling all the ray, we can stop earlier. To measure convergence, we define

$$I = 1.96\frac{\sigma}{\sqrt{n}}$$

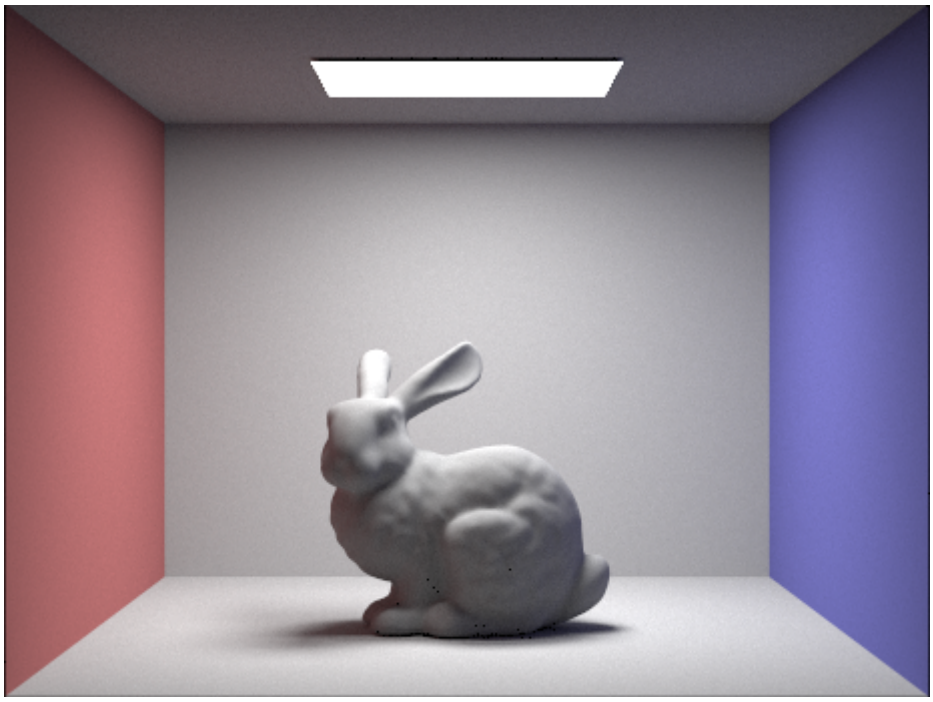
where σ is the variance of the sample set, and n is the number of sample. When the convergence measure is low than a maxTorlarance, we can stop smapling.

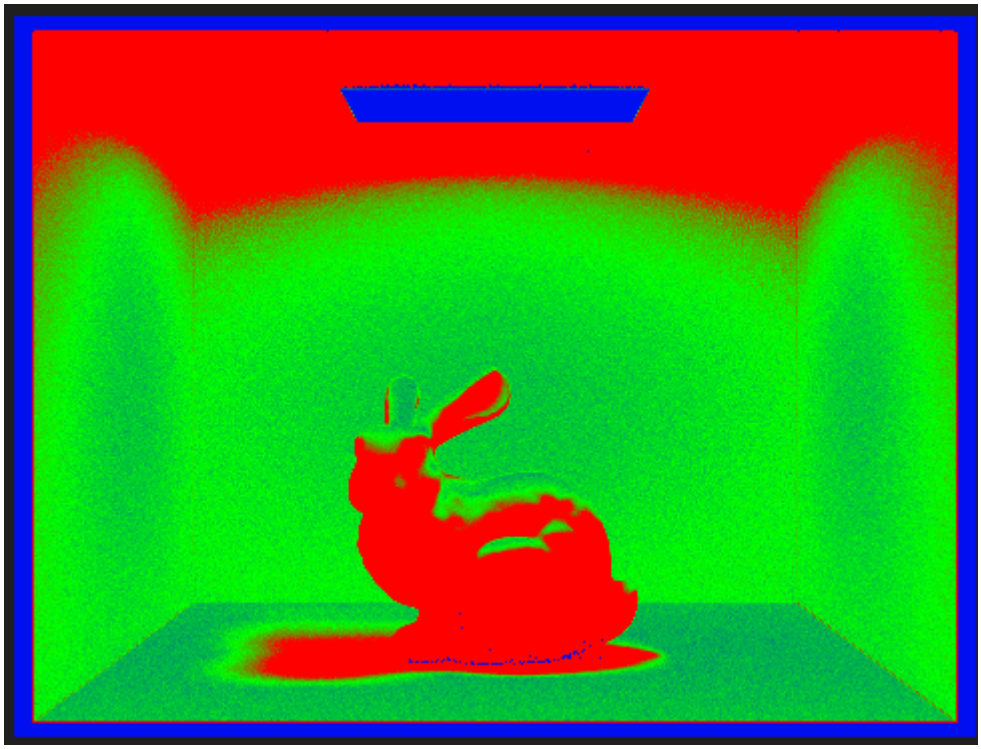
$$I \leq \text{maxTolerance} \cdot \mu$$

μ is the mean of sample set. Specifically, the data is converted to from rgb color vector to illumination.

```
./pathtracer -t 8 -s 2048 -a 64 0.05 -l 1 -m 5 -r 480 360 -f bunny.png
../../../dae/sky/CBbunny.dae
```

After 2048 ray per pixel with sample tolerance = 0.05, we can get a high-quality rendered picture, and the sample frequency image that the shade and up-corner of the wall is hard to converge during sampling.





Reference

UC Berkeley CS 184 project-requirement 3-1 https://cs184.eecs.berkeley.edu/sp23/docs/proj3-1