# CS 284A HW3 Report

Zihan Liao, Junming Chen

Webpage: https://cal-cs184-student.github.io/hw-webpages-zh-jm/

GitHub: https://github.com/cal-cs184-student/sp25-hw3-sustech-hw3

## Overview

In homework 3, we successfully implemented a PBR with pathtracing. We first of all set up the ray-scene intersection and accelerate it with BVH. Then we made many effort to implement the global illunimation, this part take us lots of time for debugging. Based on these, adaptive sampling is also applied to further improve the quality and efficiency for global illumination. Eventually we gained lots of fun and a great sense of achievement.

## Part1: Ray Generation and Scene Intersection

The first step to build a path tracer is cast the Rays and intersect with the scene.

For the Ray Generation, in `Camear::generate_ray()` we first creates a ray originating form the camera's position and passing through the sampling points on the image plane. First of all, we need to get the sensor plane coordinates in the range [-1, 1] by `sensorX = tan(hFovRad / 2) * (2 * x - 1)` and `sensorY = tan(vFovRad / 2) * (2 * y - 1)`. For the pinhole camera model we used, the sensor plane is assumed to have a distance 1. Then we define the ray's origin points out to the camera as `Vector3D origin = Vector3D(sensorX, sensorY, -1)` and transform the ray's origin and direction to world coordinates and return it as a `Ray` object.

For the Primitive Intersection, we here only consider the triangle and sphere. For sphere, the `Sphere::intersect` solves the quadratic equation resulting from the ray-sphere intersection. If intersections exist, it will find out the closest intersection point within the ray's `min_t` and `max_t` range, and updates the `Intersection` struct with

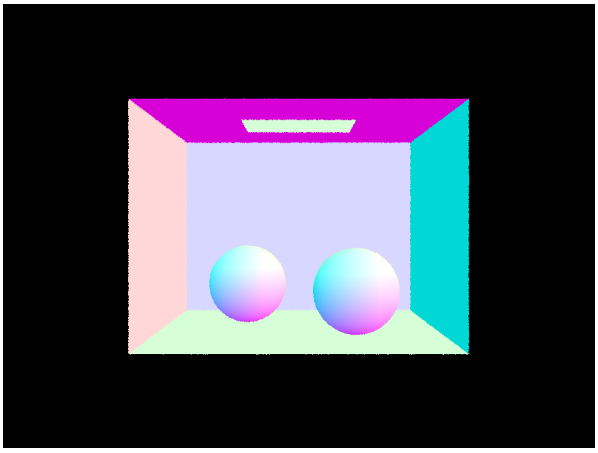
the intersection distance (`t`) and the normal at the intersection point, and the BSDF of the sphere.

For **triangle intersection,** we need to calculate the cross product of the ray direction and the edge vector, to determine the barycentric cooridnates of the intersection point. And we need also to check if the ray is parallel to the triangle. Once get the intersection barycentric corrdinate, we can calculate the intersection distance and check if it's in the valid range of the ray. If it valid for all these conditions, we will update the Ray object's intersection information, including the pointer to the triagnle, distance, normal, and BSDF of the triangle. The triangle intersection code is as below:

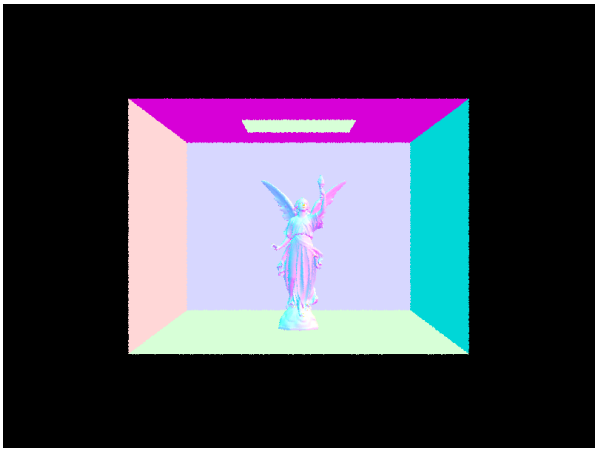
```
bool Triangle::intersect(const Ray& r, Intersection* isect) const {
  Vector3D e1 = p2 - p1;
  Vector3D e2 = p3 - p1;
  Vector3D s1 = cross(r.d, e2);
  double det = dot(e1, s1);

  if (det > -1e-6 && det < 1e-6) {
    return false; // Ray is parallel to the triangle
  }

  double inv_det = 1.0 / det;
  Vector3D s = r.o - p1;
  double b1 = dot(s, s1) * inv_det;
  if (b1 < 0.0 || b1 > 1.0) return false;

  Vector3D s2 = cross(s, e1);
  double b2 = dot(r.d, s2) * inv_det;
  if (b2 < 0.0 || (b1 + b2) > 1.0) return false;

  double t = dot(e2, s2) * inv_det;
  if (t < r.min_t || t > r.max_t) return false;

  r.max_t = min(t, r.max_t);
```

```
  if (isect) {
    isect->t = t;
    isect->primitive = this;
    isect->n = (1 - b1 - b2) * n1 + b1 * n2 + b2 * n3; // Interpolated normal
    isect->bsdf = get_bsdf();
  }

  return true;
}
```

Here is the **result** showing the normal shading:



Sphere



*CBlucy*

# Part2: Bounding Volume Hierarchy

To accelerate the ray objects intersection, we build up BVH in this part using the Surface Area Heuristic. First of all, we need to pre calculate the bounding box that encloses all the primitives, and set it as the root. After that, we need to split the axis follow the max extent rules, that choose the longest axis of the bounding box to split:

```
Vector3D extent = bbox.extent;
int split_axis = (extent.x > extent.y && extent.x > extent.z) ? 0 : (extent.y > extent
```

Then, we will sort the primitives along the splitting axis based on their center points, so that we can do the Surface Area Heuristic.

```
std::sort(start, end, [split_axis](Primitive* a, Primitive* b) {
  return a->get_bbox().centroid()[split_axis] < b->get_bbox().centroid()[split_axis]
});
```

Based on the sorted split position, we will calculate the cost function to optimze the partition. We assume that the triangle and sphere have the same intersection cost. Then we will choose the minumum cost points to be the best split position. We will store the best cost and will use and update it when looping.

```
double best_cost = std::numeric_limits<double>::infinity();
auto best_split = start + num_primitives / 2;

std::vector<BBox> left_bbox(num_primitives), right_bbox(num_primitives);
BBox left_accum, right_accum;
for (size_t i = 0; i < num_primitives; ++i) {
  left_accum.expand((*(start + i))->get_bbox());
  left_bbox[i] = left_accum;
}

for (size_t i = num_primitives; i > 0; --i) {
  right_accum.expand((*(start + i - 1))->get_bbox());
  right_bbox[i - 1] = right_accum;
}

// Evaluate SAH cost at each possible split point
for (size_t i = 1; i < num_primitives; ++i) {
  double left_area = left_bbox[i - 1].surface_area();
  double right_area = right_bbox[i].surface_area();
  double cost = left_area * i + right_area * (num_primitives - i);
```

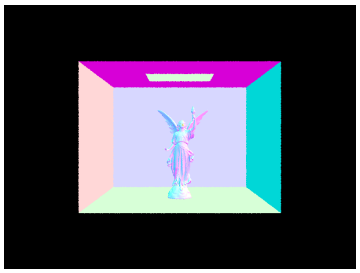
```
  if (cost < best_cost) {
    best_cost = cost;
    best_split = start + i;
  }
}
```

In the end, we need to recursive build the BVH to the left node and right node recursively.

```
node→l = construct_bvh(start, best_split, max_leaf_size);
node→r = construct_bvh(best_split, end, max_leaf_size);
```

**We compare** the rendering times on 3 scenes in different complex level to show our BVH's scalablity. The experiement result is runned on Windows 10, with the thread = 8, and resolution 800 600, return the normal shading. Here we can see BVH get more than 100 times speed up, and as the complexity increase, the speed up will more obvious following the N/log(N). Here is our results and running time for each.

maxplanck



CBlucy

cow

| Running Time (in seconds) | cow | maxplanck | CBlucy |
|---|---|---|---|
| With BVH | 0.2252 | 0.6178 | 1.5329 |
| W/O BVH | 37.1645 | 387.8913 | 1069.4281 |
| Speed Up | 165 times | 628 times | 697 times |

# Part3: Direct Illumination

In this part we implement both the uniform hemisphere sampling based on Diffuse BSDF and lighting importance sampling.

For the **Unifor Hemisphere Sampling,** we need first of generate random directions uniformly across the hemisphere to find the lighting source. Once a ray hit the primitive, we will calculate the outgoing direction in the local coordinate. The PDF for hemisphere sampling is 1/(2π) uniformly.  We will also test the shadow ray, to check if there is any intersection with any objects, and will skip that sample If no intersection found. After all, we will get the emission from the intersected object and calculate BSDF, devide by PDF, accumulate and then get the average estimation for Monte Carlo Integration.

```
Vector3D PathTracer::estimate_direct_lighting_hemisphere(const Ray &r,
                                                         const Intersection &isect) {
  // Create coordinate system at intersection point
  Matrix3x3 o2w;
  make_coord_space(o2w, isect.n);
  Matrix3x3 w2o = o2w.T();

  // Get hit point and outgoing direction in local coordinates
  const Vector3D hit_p = r.o + r.d * isect.t;
  const Vector3D w_out = w2o * (-r.d);

  int num_samples = scene->lights.size() * ns_area_light;
  Vector3D L_out(0.0);

  for (int i = 0; i < num_samples; i++) {
    // Sample a direction uniformly from the hemisphere
    Vector3D w_in = hemisphereSampler->get_sample();
    double pdf = 1.0 / (2.0 * PI); // PDF for uniform hemisphere sampling

    // Convert direction to world space
    Vector3D w_in_world = o2w * w_in;
```

```
    // Cast shadow ray
    Ray shadow_ray(hit_p, w_in_world);
    shadow_ray.min_t = EPS_F;
    Intersection light_isect;

    // Check for intersection
    if (!bvh->intersect(shadow_ray, &light_isect)) continue;

    // Get emission from intersected object
    Vector3D L_i = light_isect.bsdf->get_emission();
    L_out += isect.bsdf->f(w_out, w_in) * L_i * abs_cos_theta(w_in) / pdf;
  }

  return L_out / num_samples;
}
```

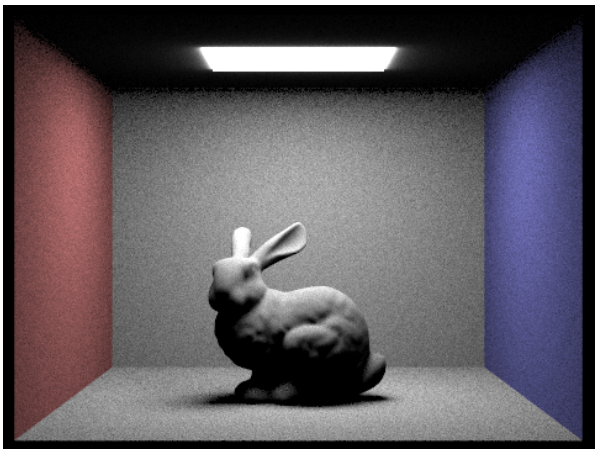
For the **Lighting Importance Sampling,** instead of sampling from random directions, we direct sample based on the distribution of the lighting source. For point lights, we will only sample once, but for area lights, we will sample across it's surface multiple times with `L_i = light->sample_L(hit_p, &wiw, &dist_to_light, &pdf)` . However, the incident light may be below the surface, so we also need to check on the indicident direction. For the Monte Carlo Integration here, we need to estimate on the area lighting's surface.

```
Vector3D PathTracer::estimate_direct_lighting_importance(const Ray &r,
                                                         const Intersection &isect) {
  // Create coordinate system at intersection point
  Matrix3x3 o2w;
  make_coord_space(o2w, isect.n);
  Matrix3x3 w2o = o2w.T();

  // Get hit point and outgoing direction in local coordinates
  const Vector3D hit_p = r.o + r.d * isect.t;
  const Vector3D w_out = w2o * (-r.d);
  Vector3D L_out(0.0);
```

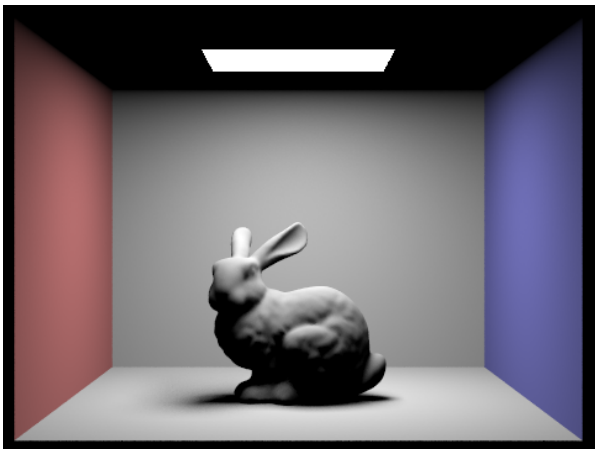
```
  for (const auto& light : scene->lights) {
    int num_samples = (light->is_delta_light()) ? 1 : ns_area_light;

    for (int i = 0; i < num_samples; i++) {
      Vector3D wiw;
      double dist_to_light, pdf;
      // Sample light to get direction, distance and PDF
      Vector3D L_i = light->sample_L(hit_p, &wiw, &dist_to_light, &pdf);

      if (pdf < EPS_F) continue;

      Vector3D w_in = w2o * wiw; // Convert to local space
      if (w_in.z < 0) continue; // Skip directions below the surface

      // Cast shadow ray
      Ray shadow_ray(hit_p, wiw);
      shadow_ray.min_t = EPS_F;
      shadow_ray.max_t = std::max(dist_to_light - EPS_F, shadow_ray.min_t);

      // Check for intersection (shadows)
      if (bvh->has_intersection(shadow_ray)) continue;

      // Compute contribution
      L_out += isect.bsdf->f(w_out, w_in) * L_i * abs_cos_theta(w_in) / pdf;
    }

    if (!light->is_delta_light()) L_out /= num_samples;
  }

  return L_out;
}
```

Here is our comparison on the bunny scene between the Uniform Sampling and Lighting Sampling. **Analysis the results** between the two sampling

implementations, we find that even though the uniform hemisphere sampling is mathmatically correct for the Diffuse BSDF, it's not as efficient as lighting sampling, since most rays cast from it will not hit the light souces. And from the results, we can tell that the uniform sampling alwyas have more noise, which means its estimator have higher variance than the lighting sampling. For the lighting sampling, it converge faster, but might hard for the complex light source geometry, and will also suffers if the scene lighting condition is quiet dense.

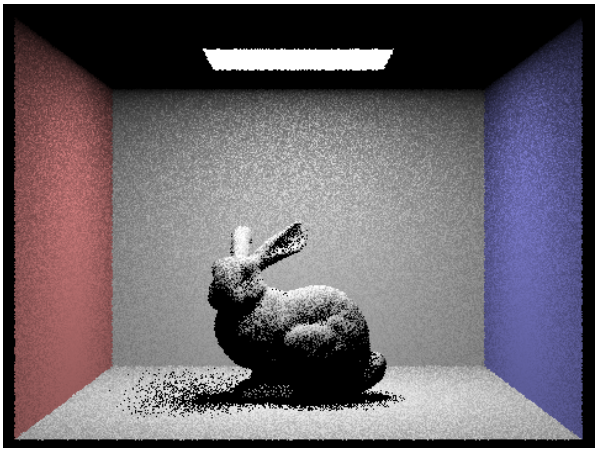


Uniform hemishpere sampling

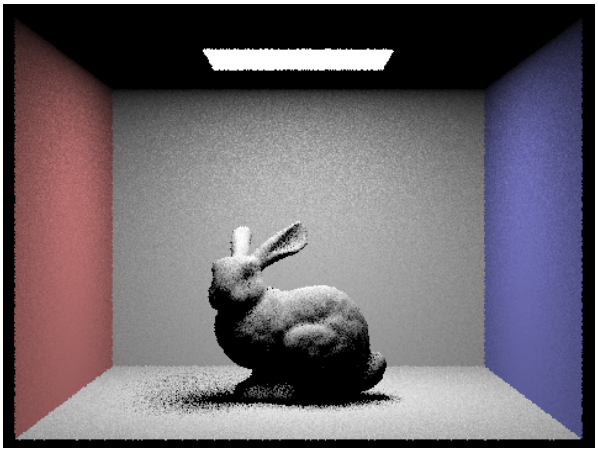


Lighting sampling

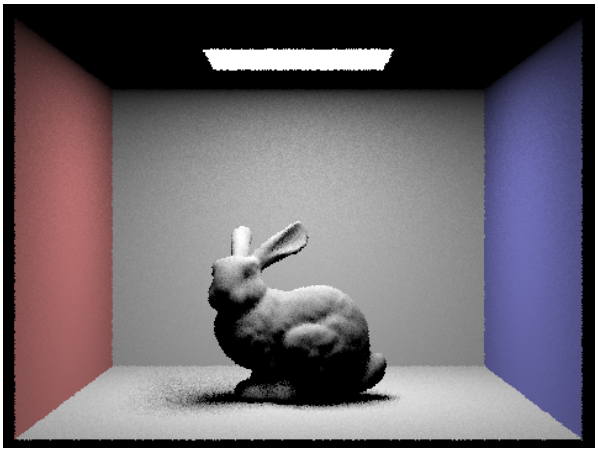
We also compare the nose levels in soft shadows, using the light sampling, with 1, 4, 16, and 64 light rays and only 1 sample per pixel using light sampling. We can see that as the nums of light rays increase, the soft shadow is cleaner, and also centered, showing that the estimation's variance is reduced with more sampling on area light.



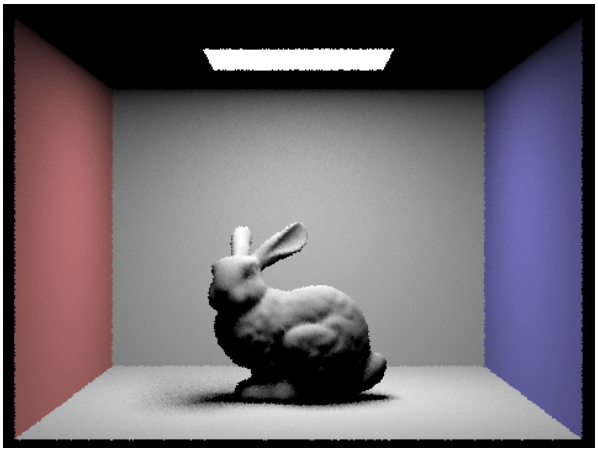
light rays = 1



light rays = 4



light rays = 16



light rays = 64

# Part 4: Global Illumination

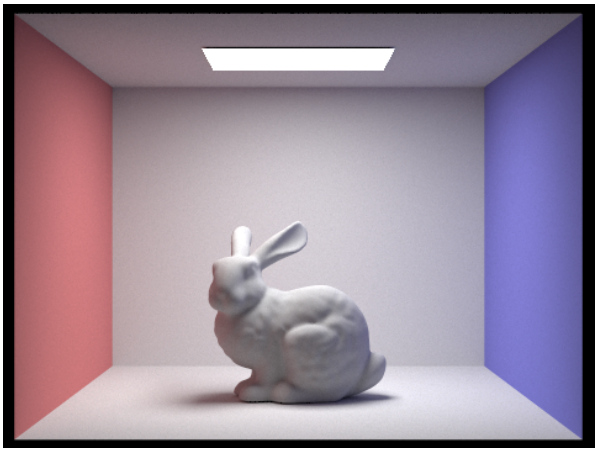
Besides direct lighting from the light source, the light will also bounce infinitely in the scene. In this part we will include the indirect lighting to achieve global illumination. Same as the direction lighting, we will first get the hit points of the ray, and initialize `L_out` to accumulate the radiance. For one ray, we directly use the direct illumination by calling `one_bounce_radiance` . Instead of uniform sampling, we here sample the BSDF to get a incoming direction. After that, we can now create the new ray for the next bounce, that origin from the hit points, and point out in the sampled incoming light direction. To avoid infinite bounces and fast converge, we

use the Russian Roulette with 0.35 probablity for terminate this ray. Based the rendering equation, we will recursively call the `at_least_one_bounce_radiance` on the following new incoming ray.

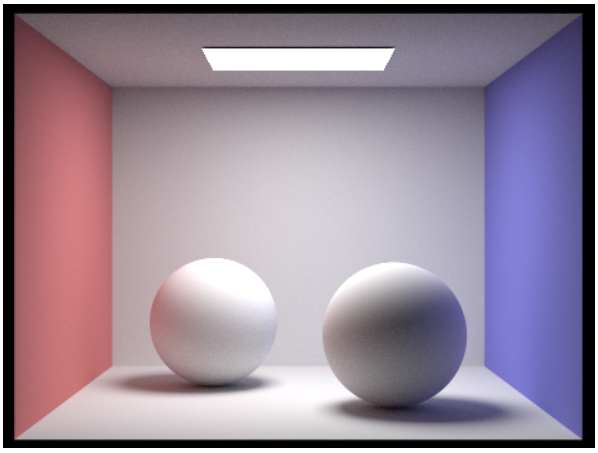
```
Vector3D PathTracer::at_least_one_bounce_radiance(const Ray &r,
                                                  const Intersection &isect) {
  if (max_ray_depth == 0) return Vector3D(0.0);

  Matrix3x3 o2w;
  make_coord_space(o2w, isect.n);
  Matrix3x3 w2o = o2w.T();

  Vector3D hit_p = r.o + r.d * isect.t;
  Vector3D w_out = w2o * (-r.d);

  Vector3D L_out(0, 0, 0);

  Vector3D direct_light = one_bounce_radiance(r, isect);

  if (r.depth >= max_ray_depth - 1) {
    return direct_light; // If only last bounce is needed, return at max depth
  }

  if (isAccumBounces) {
    L_out += direct_light; // Accumulate direct lighting
  }

  Vector3D w_in;
  double pdf;
  Vector3D f = isect.bsdf->sample_f(w_out, &w_in, &pdf);

  if (pdf < EPS_F || f.norm() < EPS_F) return L_out; // Avoid division by zero

  Vector3D w_in_world = o2w * w_in;
  Ray new_ray(hit_p, w_in_world);
    new_ray.min_t = EPS_F;
```

```
  new_ray.depth = r.depth + 1;

  double rr_prob = 0.35;
  if (new_ray.depth >= max_ray_depth || coin_flip(rr_prob)) return L_out;

  Intersection new_isect;
  if (bvh->intersect(new_ray, &new_isect)) {
    Vector3D indirect = at_least_one_bounce_radiance(new_ray, new_isect);
    L_out += (f * indirect * abs_cos_theta(w_in)) / (pdf * (1 - rr_prob)); // Accumula
  }

  return L_out;
}
```

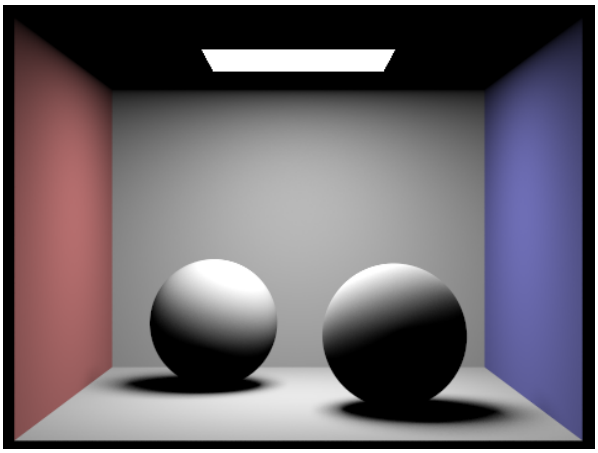
Here is some global illumination **examples with 1024 samples per pixel, lighting samples = 16, max ray depth=5.** We also show the Sphere scene for only direct illumination and only indirection illumination.



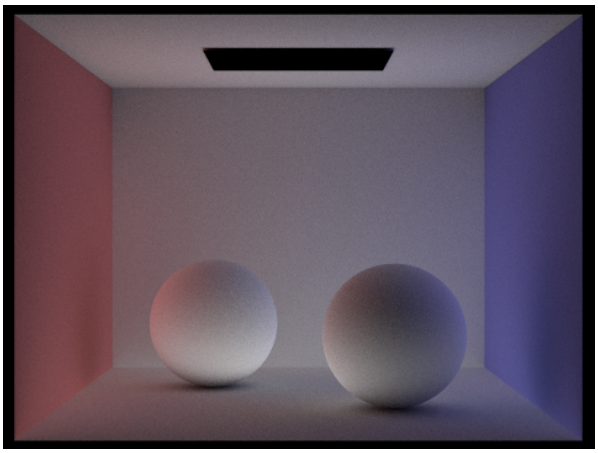
Bunny



Sphere

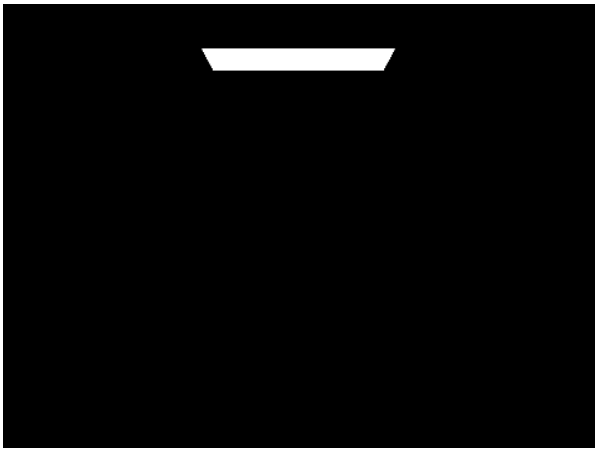


Sphere, Only direct illumination

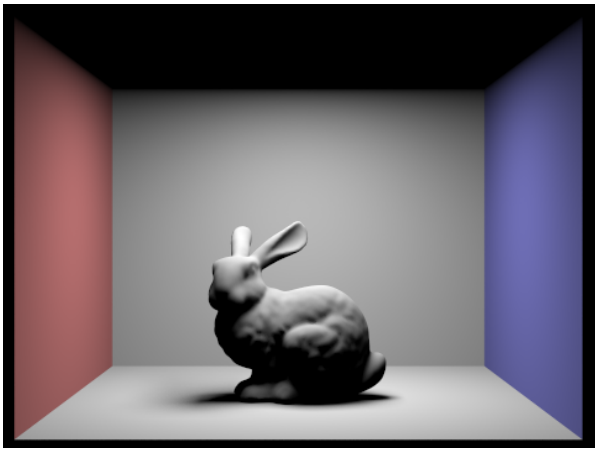


Sphere, Only indirect illumination

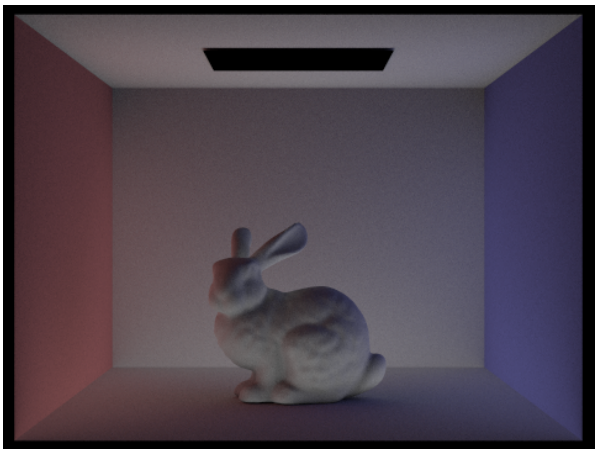
We here output the each ray depth's **kth bounce** of light. For the 2nd and 3rd bouce, compared to rasterization, they mainly contributed to the reflection between the objects. In the following results, we can see that for the 2nd bounce, the bunny get the light from the ground, and also cast the shadow to the left and right walls. And in the 3rd bounce, we can tell that the ground also receive the light reflected by the bunny.



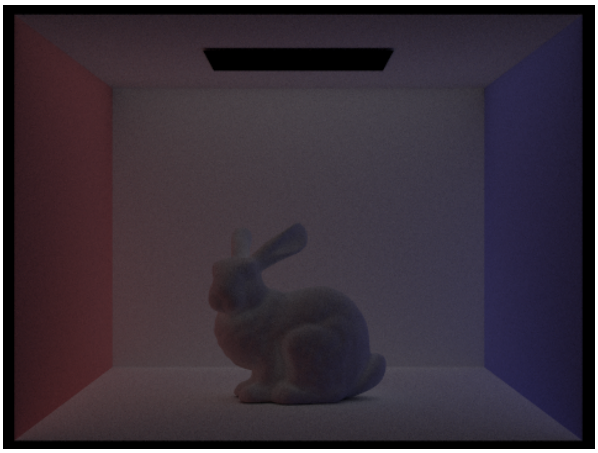
Bunny, 0th bounce



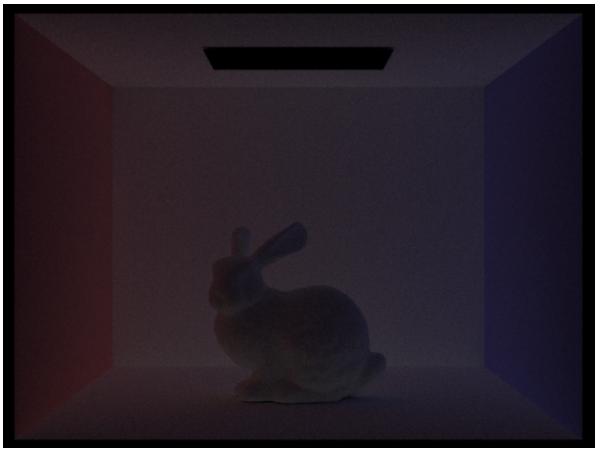
Bunny, 1st bounce



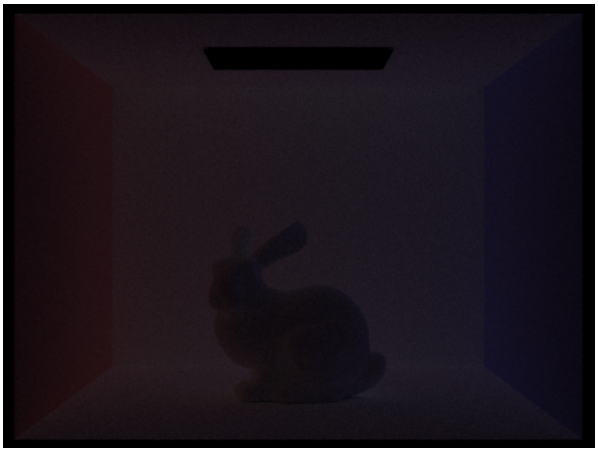
Bunny, 2nd bounce



Bunny, 3rd bounce

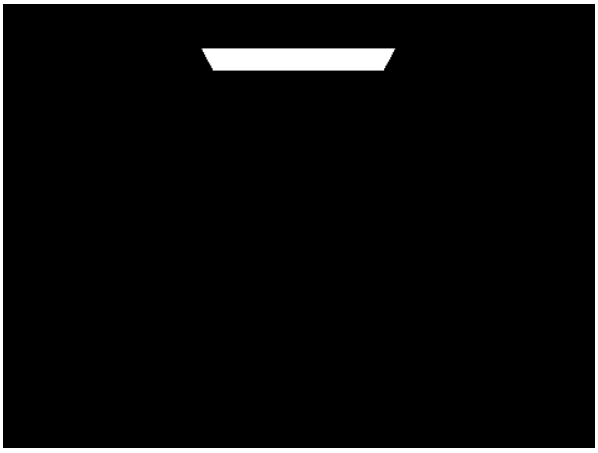


Bunny, 4th bounce

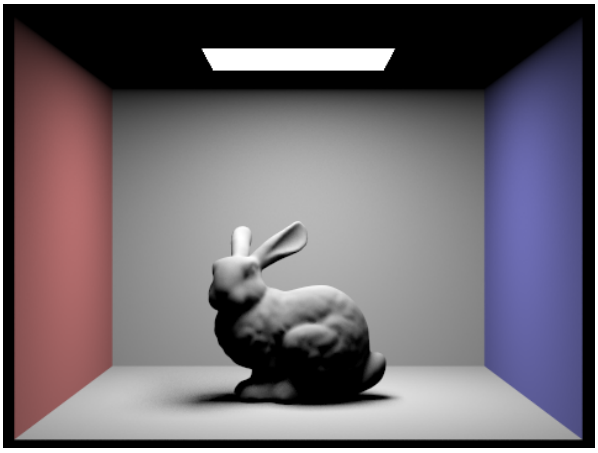


Bunny, 5th bounce

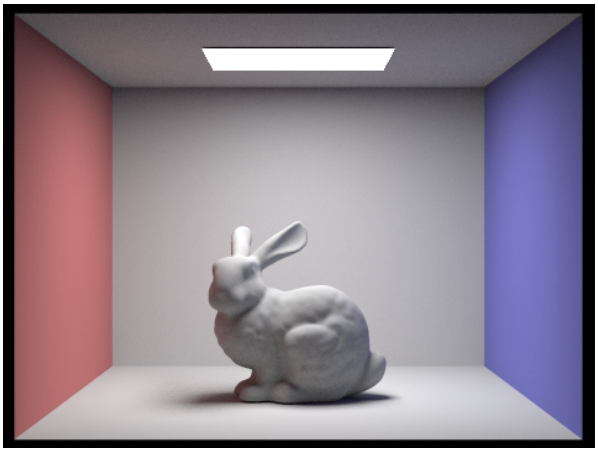
We test the Bunny scene with **Russian Roulette** with different max ray depth showing below, we can tell that after max ray depth=4, the results have already converge, and there is no obvious different even though the depth increase to 1024:



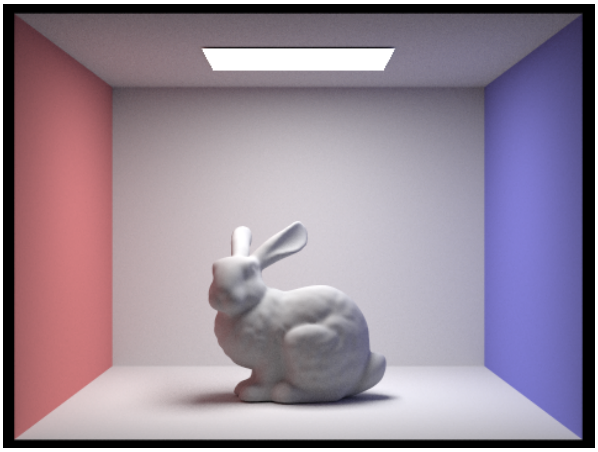
Bunny, Russian Roulette, max ray depth=0



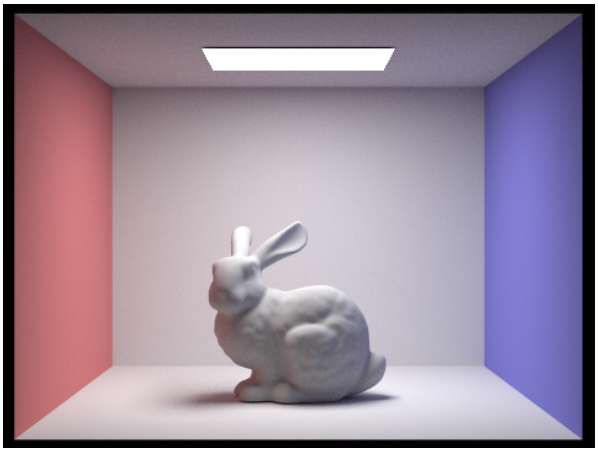
Bunny, Russian Roulette, max ray depth=1



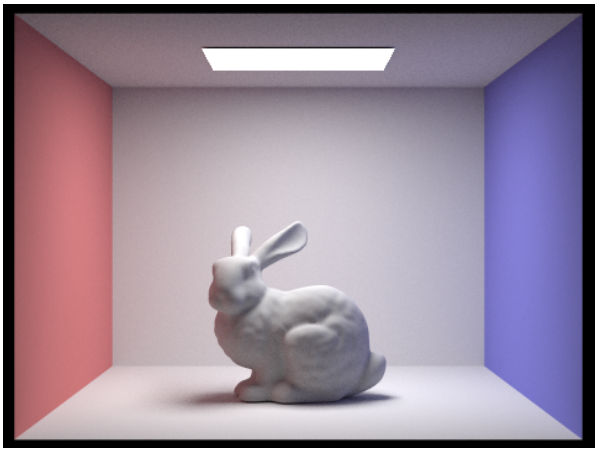
Bunny, Russian Roulette, max ray depth=2



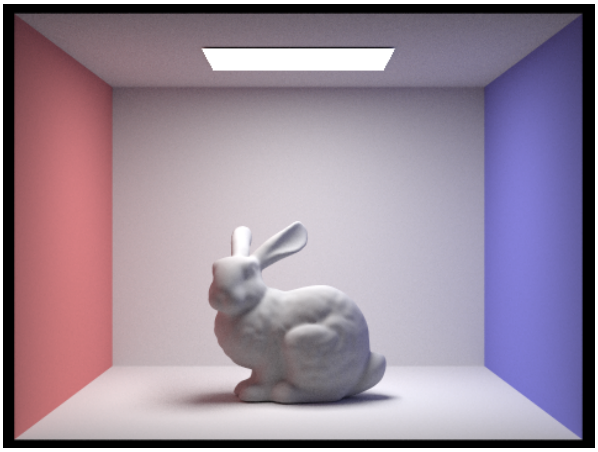
Bunny, Russian Roulette, max ray depth=4



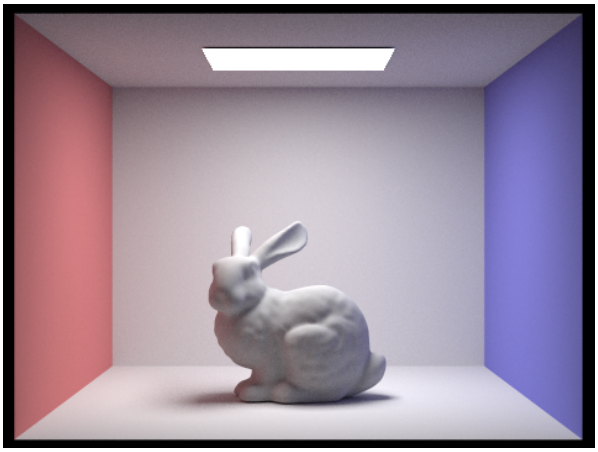
Bunny, Russian Roulette, max ray depth=8



Bunny, Russian Roulette, max ray depth=16

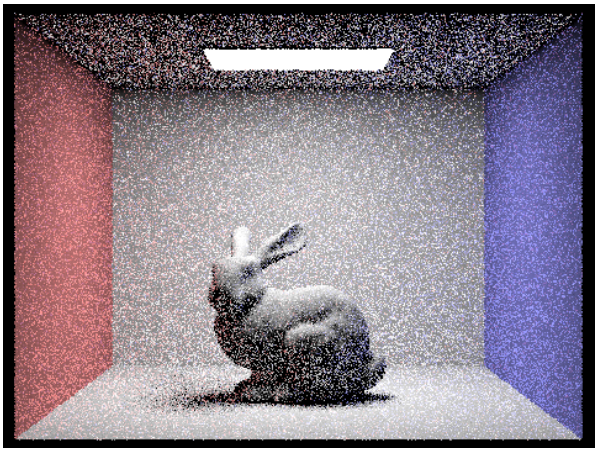


Bunny, Russian Roulette, max ray depth=64

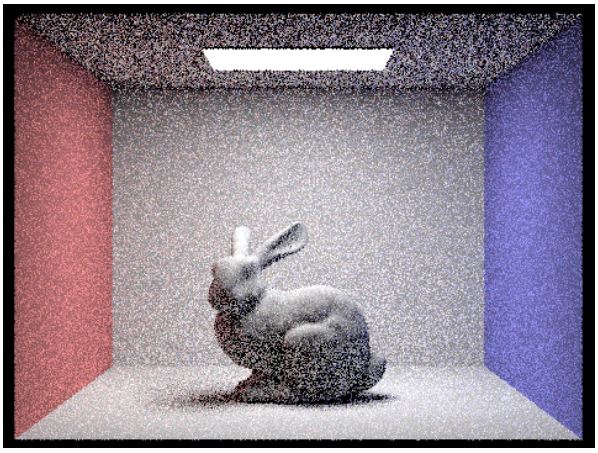


Bunny, Russian Roulette, max ray depth=1024

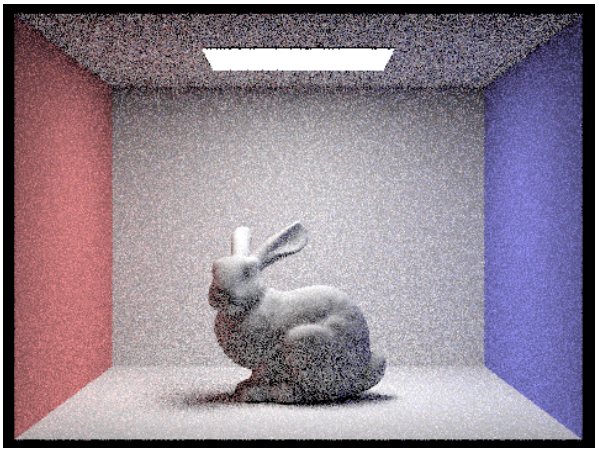
We **also test the different sample-per-pixel rates** with different 4 light rays showing below. We can find that as the sample-per-pixel increase, the noise is reduced.



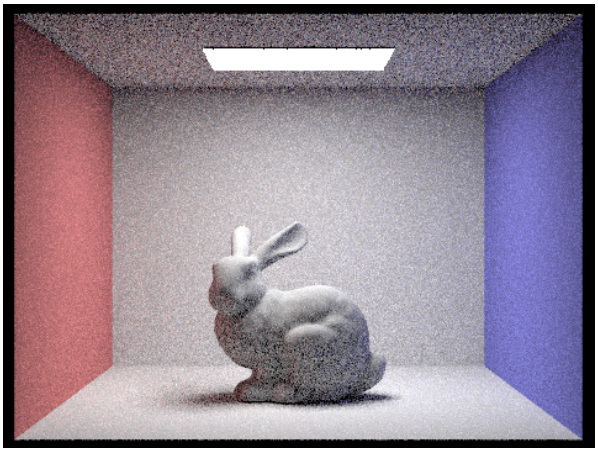
Bunny, light rays=4, sample per pixel = 1



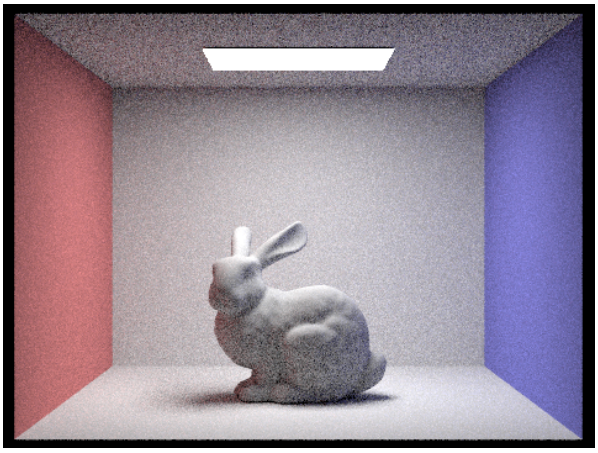
Bunny, light rays=4, sample per pixel = 2



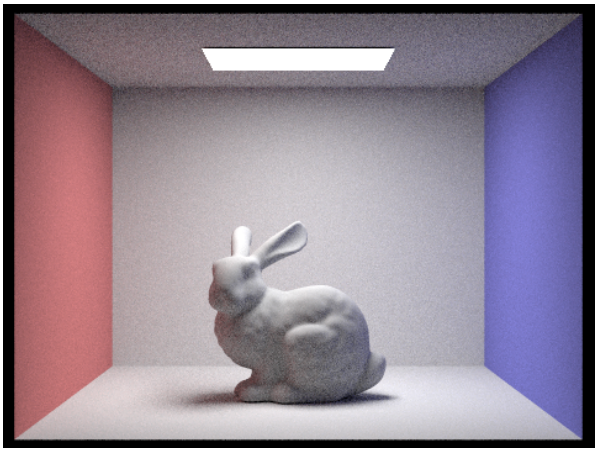
Bunny, light rays=4, sample per pixel = 4



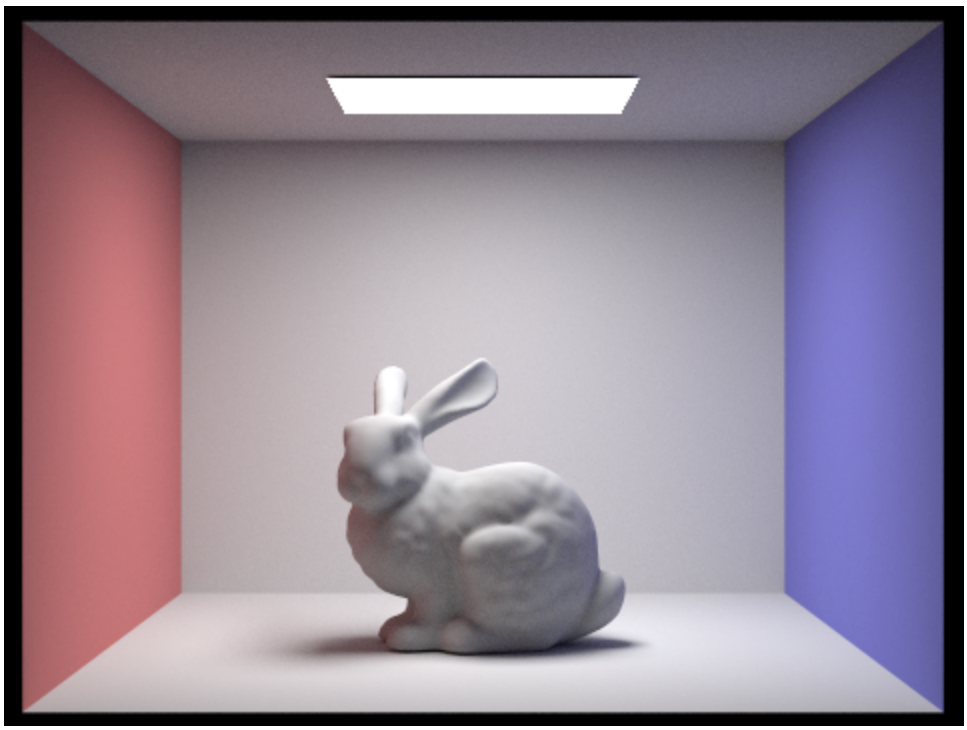
Bunny, light rays=4, sample per pixel = 8



Bunny, light rays=4, sample per pixel = 16



Bunny, light rays=4, sample per pixel = 64



Bunny, light rays=4, sample per pixel = 1024

# Part 5: Adaptive Sampling

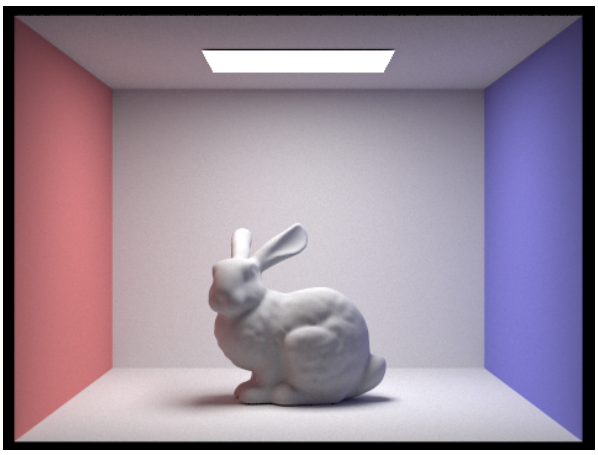
From the previous parts, we can see that the sample rate per pixel is important to get noiseless results. But it will cost too much time, when uniformly sample the pixels with the same sample rate. In this part, we use the dapative sampling to take more samples in noisy areas and fewer samples in converged area.

For each pixel, we first of all need to records it's samples color's distribution. We store the mean and variance of each pixels samples' color to decide whether it's already converged. We will compute the global illumination radiance of the sampled ray, and converts to a scalar luminance and update the mean and variance. We check if the luminance now under the inside the confidence interval ( at 95% confidence level) following the rules `I = 1.96 * stddev / sqrt(num_samples)` . And if it's already converged, we will return the average color and terminate sampling more rays from this pixels. Here is our code:

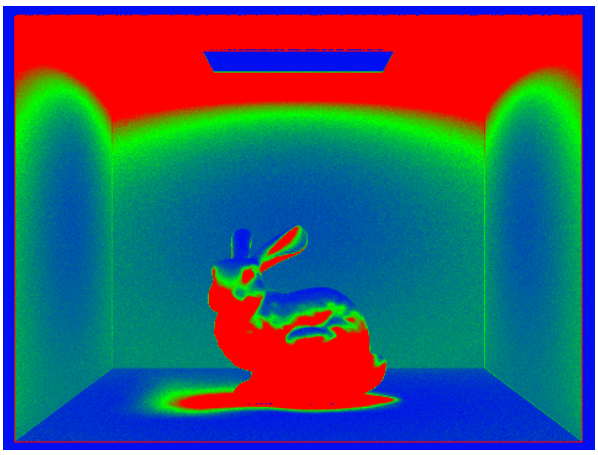
```
void PathTracer::raytrace_pixel(size_t x, size_t y) {
  int num_samples = 0;        // Current number of samples taken
```

```
  Vector2D origin = Vector2D(x, y); // Bottom-left corner of the pixel

  Vector3D sampleColor(0, 0, 0);  // Accumulated radiance
  double s1 = 0, s2 = 0;          // Accumulators for mean and variance

  for (int i = 0; i < ns_aa; i++) {
    Vector2D sample = gridSampler->get_sample();
    Vector2D p = origin + (sample + Vector2D(0.5, 0.5));
    Ray r = camera->generate_ray(p.x / sampleBuffer.w, p.y / sampleBuffer.h);
    Vector3D radiance = est_radiance_global_illumination(r);

    double illum = radiance.illum(); // Convert to scalar luminance
    s1 += illum;
    s2 += illum * illum;

    sampleColor += radiance;
    num_samples++;

    // Perform adaptive sampling check every samplesPerBatch
    if (num_samples % samplesPerBatch == 0) {
      double mean = s1 / num_samples;
      double variance = (s2 - (s1 * s1) / num_samples) / (num_samples - 1);
      double stddev = sqrt(variance);
            double I = 1.96 * stddev / sqrt(num_samples);

      // Check convergence condition
      if (I <= maxTolerance * mean) {
        break; // Stop sampling if converged
      }
    }
  }

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

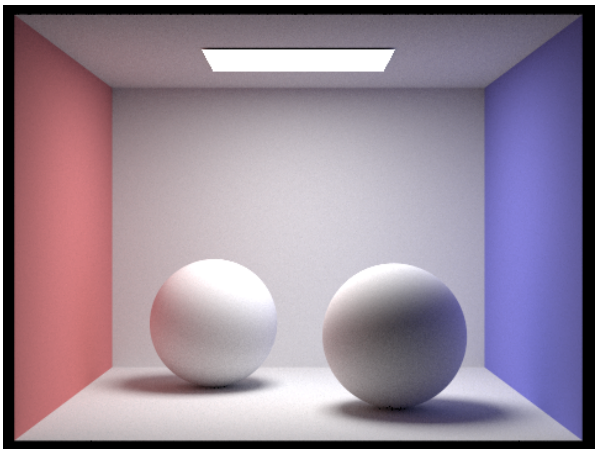
Here is our **results with 2048 samplers per pixel** on two scenes. We can see from the sampling rate map, that the noised (dark) area in original image will have more samples (in red in sampling rate map).



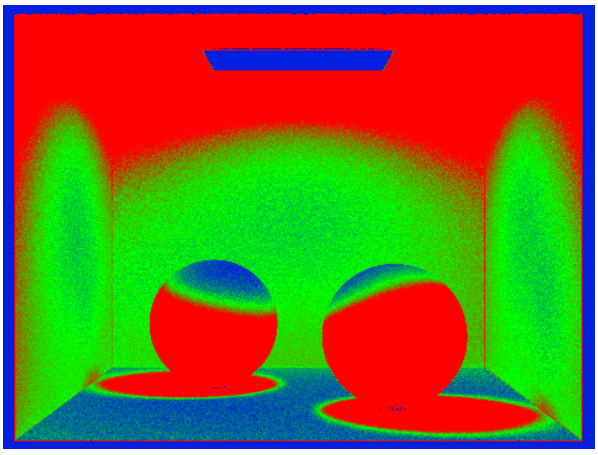
Bunny, samplers per pixel = 2048



Bunny, Adaptive Sampling Rate map



Sphere, samplers per pixel = 2048



Bunny, Adaptive Sampling Rate map