CSC4140 Assignment VI

Computer Graphics

April 22, 2022

Ray Tracing

This assignment is 10% of the total mark.

Strict Due Date: 11:59PM, April 22^{th}, 2022

Student ID: 118010202

Student Name: Zhixuan LIU

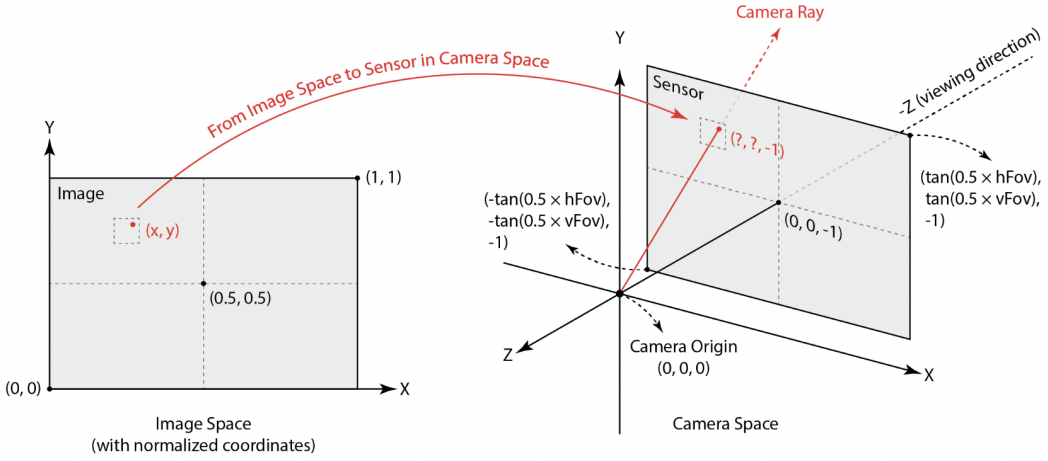
This assignment represents my own work in accordance with University regulations.

Signature: Zhixuan Liu

1 Part 1: Ray Generation and Scene Intersection

1.1 Ray generation and primitive intersection parts

In this section, I will briefly introduce how I generate camera rays. Given a normalized image coordinates as input and outputs a Ray in the world space. So first I first image coordinates to camera space, generate the ray in the camera space, and finally transform it into a ray in the world space, in the function $Camera :: generate\_ray(...)$. In other words, generating the ray is a matter of converting to standardized coordinates between [0,1] and then finding the corresponding position in the sensor plane and convert it to world space using the transform c2w.




```
Ray Camera::generate_ray(double x, double y) const {

  // TODO (Part 1.1):
  // compute position of the input sensor sample coordinate on the
  // canonical sensor plane one unit away from the pinhole.
  // Note: hFov and vFov are in degrees.
  //result = tan ( param * PI / 180.0 );
  //Find the ray in the camera space
  double xCa = (x-0.5) * 2 * tan((hFov/2)*M_PI/180.0);
  double yCa = (y-0.5) * 2 * tan((vFov/2)*M_PI/180.0);
  Vector3D rayCa = Vector3D(xCa, yCa, -1);
  Vector3D dir = c2w * rayCa;
  dir /= dir.norm();
  Ray rayW(pos, dir);
  rayW.min_t = nClip;
  rayW.max_t = fClip;
  return rayW;
}
```

Then I do the Generating Pixel Samples task. This function takes pixel coordinates (x,y)(x,y)

as input and updates the corresponding pixel in sampleBuffer with a Vector3D representing the integral of radiance over this pixel. I made a loop that generates $num\_samples$ camera rays and traces them, then do the update.

```
Vector3D sum_Color = Vector3D(0,0,0);
double s1 = 0.;
double s2 = 0.;
double mju = 0.;
double sigma_square = 0.;
double iTest = 0.;
int count = 0;
int sample_count = 0;
int n=0;
for(int i=0; i<num_samples; i++){
  Vector2D target = origin + gridSampler->get_sample();
  target.x = target.x / sampleBuffer.w;
  target.y = target.y / sampleBuffer.h;

  Ray my_ray = camera->generate_ray(target.x , target.y);
  my_ray.depth = max_ray_depth;

  Vector3D target_Color = est_radiance_global_illumination(my_ray);
  sum_Color += target_Color;
  sample_count++;
}
sampleBuffer.update_pixel(sum_Color/sample_count, x, y);
sampleCountBuffer[x + y * sampleBuffer.w] = sample_count;
```

1.2 Triangle intersection algorithm

Determining intersections between rays and primitive geometric objects required using implicit vector definitions of the primitive geometries themselves. Intersection with a triangle was implemented using the Moller-Trumbore Algorithm. So in the function $Triangle :: has\_intersection(...)$ and $Triangle :: intersect(...)$ functions I mainly convert the following equations into codes to test the intersection. The Moller-Trumbore Algorithm only takes barycentric coordinates and the parameterization of the ray to calculate the intersection with the triangles plane, and it is easy to implement without other resources.

Then I implemented the $Sphere :: has\_intersection(...)$ and $Sphere :: intersect(...)$ functions. I basically followed the idea of this formula and turned it into codes:

Here follows some implementation results after I implement the intersection correctly.

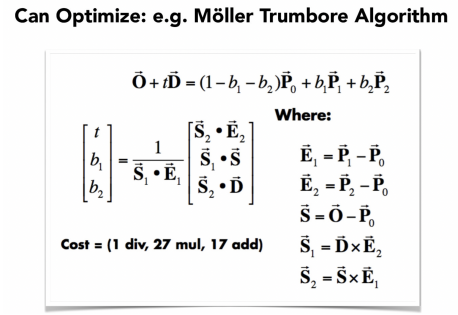


Figure 1: Formula.

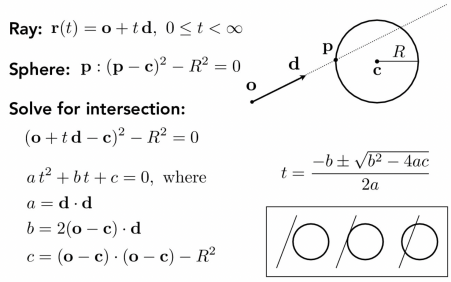
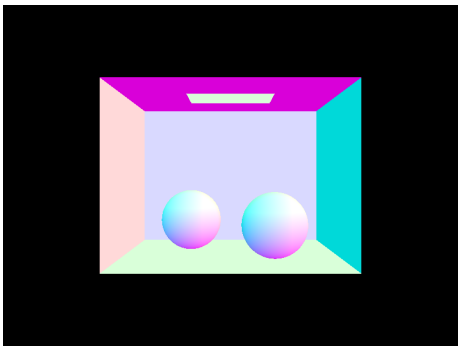
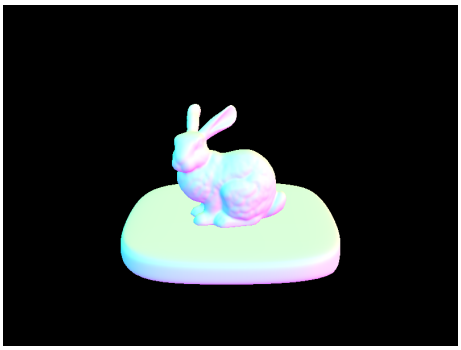
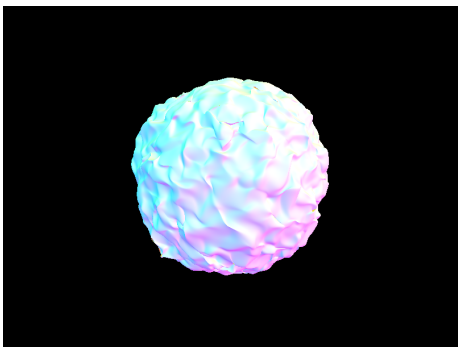
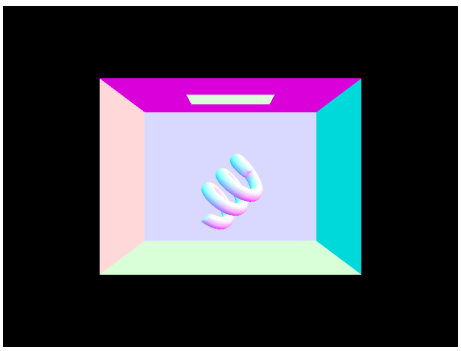


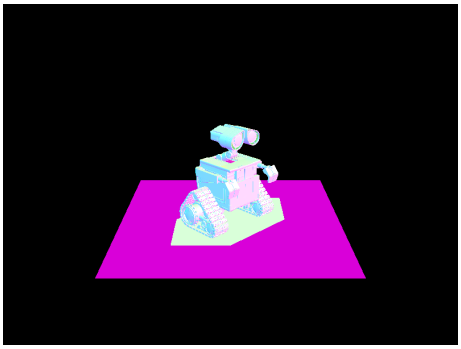
Figure 2: Formula used.

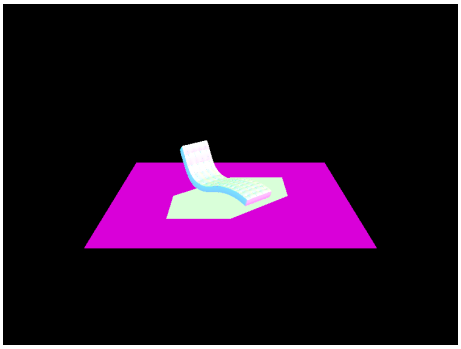












2 Part 2: Bounding Volume Hierarchy

2.1 BVH construction algorithm

In this section, I will talk about how I build my BVH tree structure. BVH tree can drastically improved the ray-tracing efficiency.

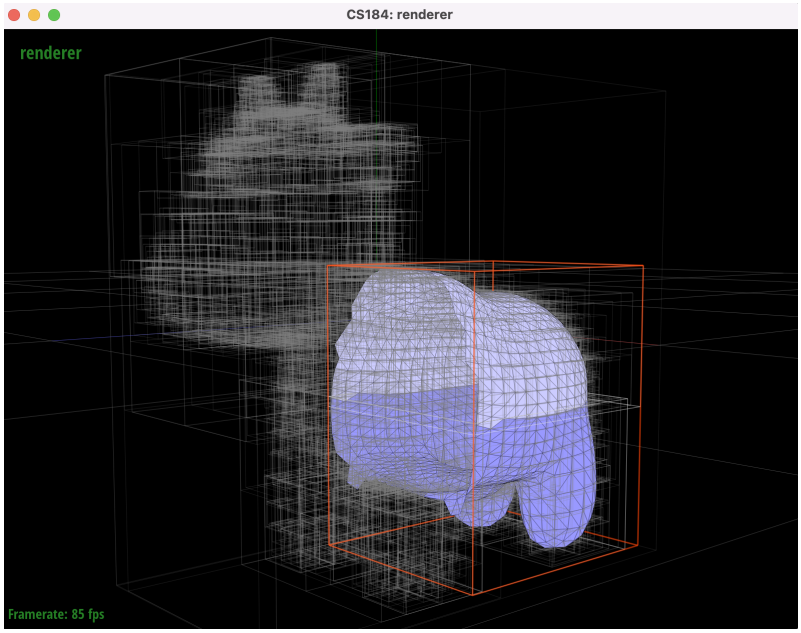
Firstly thing is to decide which axis to split. So here I choose the axis with the largest length.

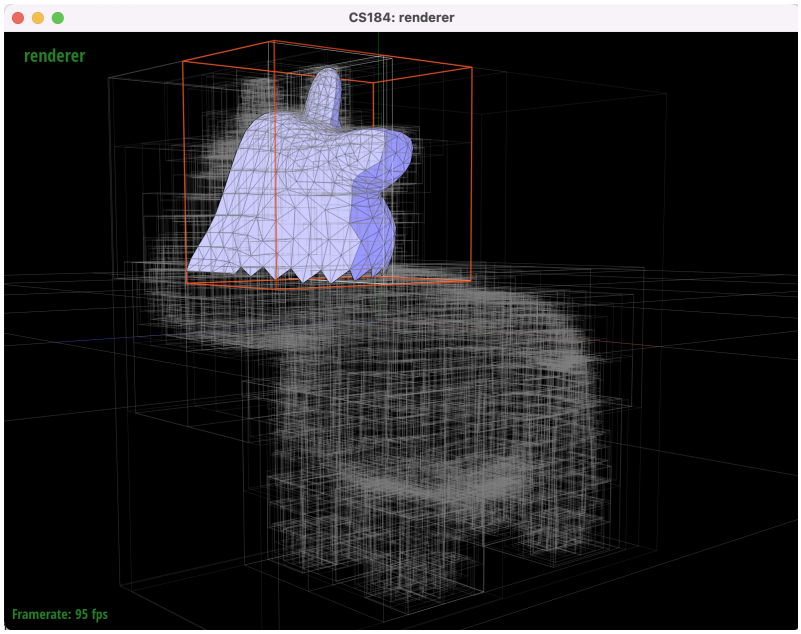
```
1 // 1. Find the larget axis to split
2 int axis = bbox.extent[0] < bbox.extent[1] ? 1 : 0;
3 axis = bbox.extent[axis] < bbox.extent[2] ? 2 : axis;
```

Since get the axis that we want to split, I calculated the average center of all the primitives in that axis using the centriod function. Then loop through all the primitives to decide which primitives go to which subsets.

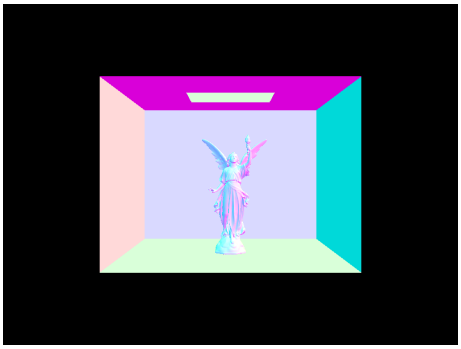
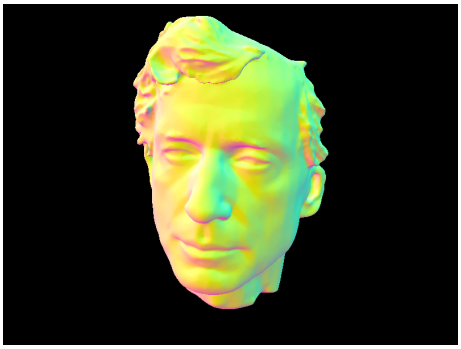
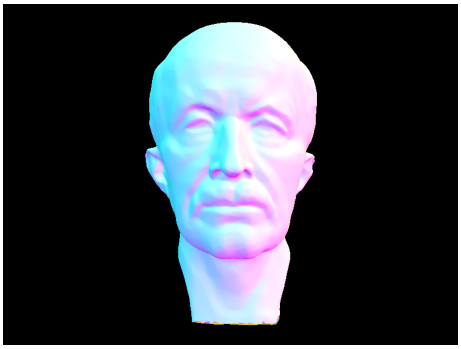
```
1 judge_center += (*p)->get_bbox().centroid();
2 // decide which primitive go to left or right sub tree compared to the calculated center.
3 for(auto p = start; p != end; p++){
4   if((*p)->get_bbox().centroid()[axis] <= axis_center){
5     left_objects.push_back(*p);
6   }else{
7     right_objects.push_back(*p);
8   }
9 }
```

Then using the recursive method to build sub-trees until the number of primitives is lower than the threshold.





2.2 Part 2 Results

2.3 Time consumption comparison

For my Mac M1 pro machine, it took me 0.0364 second to render dae/sky/CBlucy.dae and 0.0495 second to render dae/meshedit/maxplanck.dae. However, if I do not use BVH structure, it will take me 103.5819 seconds to render maxplanck.dae.

I can conclude that the BVH structure is very efficient considering it simply skip the sample of

the ray with no intersection with the block. The computation complexity changed from origiona O(n) to O(logn), which reduce the amount of time a lot when n is large.

3 Part 3: Direct Illumination

3.1 Implementations of the direct lighting function

Direct lighting function consists of three parts, the first is the *zero_bounce_radiance* function, which takes as input a Ray and the Intersection object corresponding with that ray and the scene. The function returns the emission of the object that was intersected.

The next function I implement is *estimate_direct_lighting_hemisphere*. I took advantages of the Monte Carlo estimator, following the steps on the power point.

Monte Carlo estimate:

- **Generate directions ω_j sampled from some distribution $p(\omega)$**
- **Choices for $p(\omega)$**
  - **Uniformly sample hemisphere**
  - **Importance sample BRDF (proportional to BRDF)**
  - **Importance sample lights (sample position on lights)**
- **Compute the estimator**

$$\frac{1}{N}\sum_{j=1}^{N}\frac{f_r(\mathrm{p},\omega_j\rightarrow\omega_r)\,L_i(\mathrm{p},\omega_j)\,\cos\theta_j}{p(\omega_j)}$$

The direct lighting function takes in a ray and an intersection. From this, we extrapolate the hitpoint hit_p and the outgoing direction in the object frame w_out. At this point, we want to calculate the radiance in w_out direction using all of the lights in the scene using monte carlo integration across all lights.

```
for(int target=0; target<num_samples; target++){
    Vector3D wi;
    double pdf;
    Vector3D sample = isect.bsdf->sample_f(w_out, &wi, &pdf);

    Ray next_ray(hit_p, o2w*wi, 1);
    next_ray.min_t = EPS_F;
    Intersection i;
    if(bvh->intersect(next_ray, &i)){
      if(cos_theta(wi)>0){
        L_out += i.bsdf->get_emission() * sample * cos_theta(wi) / pdf;
      }
    }
}
```

```
15 L_out = L_out/num_samples;
16 return L_out;
```

In the estimate_direct_lighting_importance function, I sampled all the lights directly, rather than uniform directions in a hemisphere. For each light in the scene, we want to sample directions between the light source and the hit_p. If we cast a ray in this direction and there is no other object between the hit point and the light source, then we know that this light source does cast light onto the hit point.

```
1  // int count = 0;
2  for (auto L_target = scene->lights.begin(); L_target != scene->lights.end(); L_target++){
3      // count++;
4      Vector3D wi;
5      Vector3D L_sample;
6      double distToLight;
7      double pdf;
8      // start sample, I only sample once per light
9      Vector3D sample = (*L_target)->sample_L(hit_p, &wi, &distToLight, &pdf);
10     Ray next_ray(hit_p, wi, 1);
11     next_ray.min_t = EPS_F;
12     next_ray.max_t = (double)distToLight - EPS_F;
13
14     Intersection ii;
15     if(!bvh->intersect(next_ray, &ii)){
16       if(dot(wi, isect.n)>0){
17         L_sample += sample * isect.bsdf->f(w_out, wi) * dot(wi, isect.n) / pdf;
18       }
19     }
20     if ((*L_target)->is_delta_light()) {
21       break;
22     }
23     L_out += L_sample;
24 }
```

3.2 Part 3 Results

3.3 Deliverable 3

3.4 Comparison

Lighting sampling out performs the uniform hemisphere sampling. Noise in rendering is significantly reduced when importance sampling is used. This demonstrates the effectiveness of the importance sampling technique in producing significantly higher rendering quality for the same

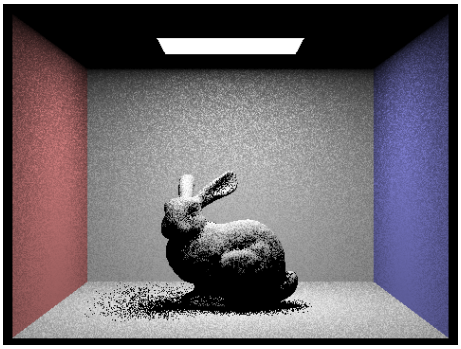


Figure 3: bunny_1_1.

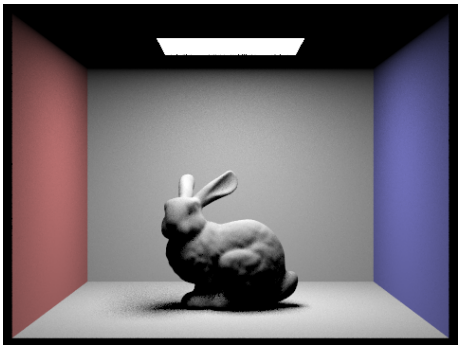


Figure 4: bunny_1_4.

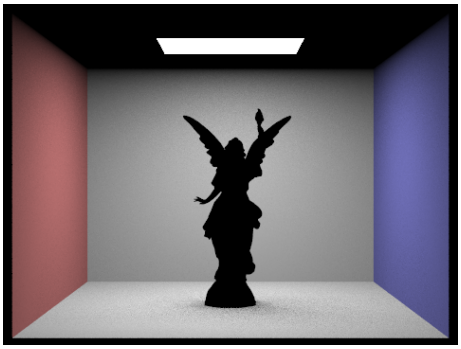


Figure 5: bunny_1_16.

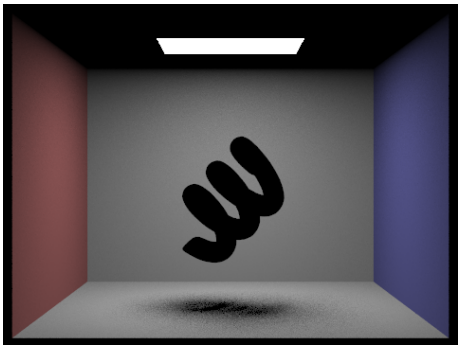


Figure 6: bunny_1_64.

number of samples.

4 Part 4: Global Illumination

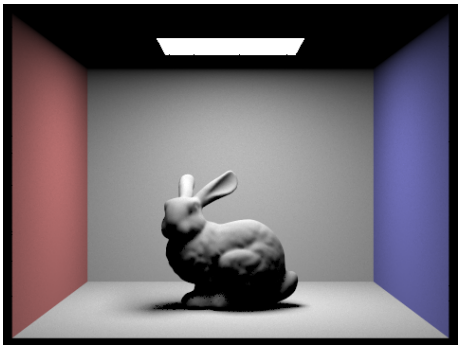
In part 4, I implemented indirect illumination. The indirect lighting function takes in a ray and an intersection. The hitpoint hit_p and the outgoing direction in the object frame w_out were extrapolated. Since the function is recursive and terminates by way of Russian Roulette, there will be a check of stopping condition.

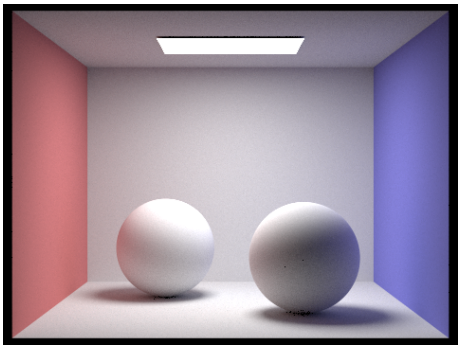
```
double cpdf = 0.7;
  // RANDOM DECISION WITH RECURSION
  Vector3D wi;
  double pdf;

  Vector3D sample = isect.bsdf->sample_f(w_out, &wi, &pdf);
  Ray next_ray(hit_p, o2w * wi, 1);
  next_ray.depth = r.depth - 1;
  next_ray.min_t = EPS_F;

  if (coin_flip(cpdf)){
    Intersection ii;
    if(bvh->intersect(next_ray, &ii)){
      if (next_ray.depth > 0 && cos_theta(wi) > 0) {
          // L_OUT += ONE_BOUNCE_RADIANCE(NEXT_RAY, ISECT);
          Vector3D sample_more = at_least_one_bounce_radiance(next_ray, ii);
          L_out += sample_more * sample * cos_theta(wi) / pdf / cpdf;
      }
    }
  }
```

4.1 Results of part 4





4.2 Direct Only compared with Indirect Only

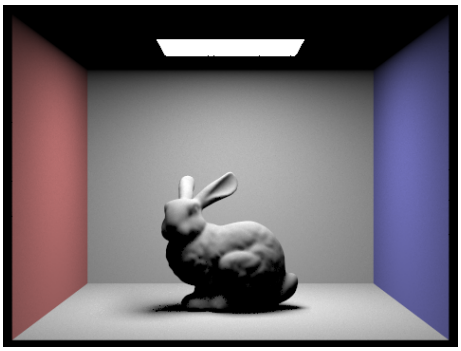


Figure 7: bunny_direct_light_only.

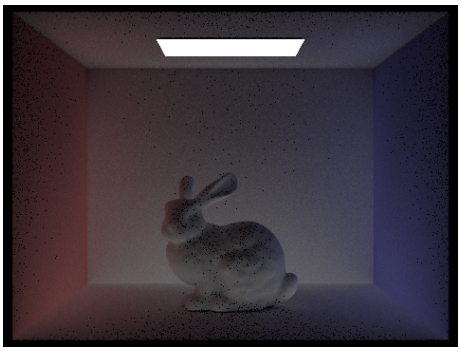


Figure 8: bunny_indirect_light_only.

4.3 Changing with maximum ray depth

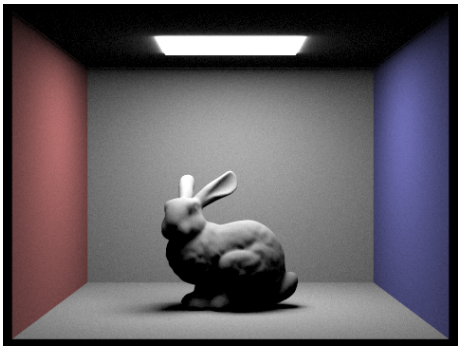


Figure 9: m=1.

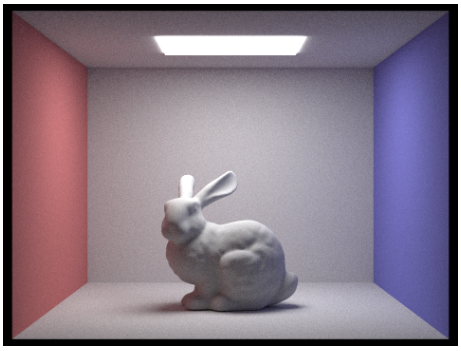


Figure 10: m=2.

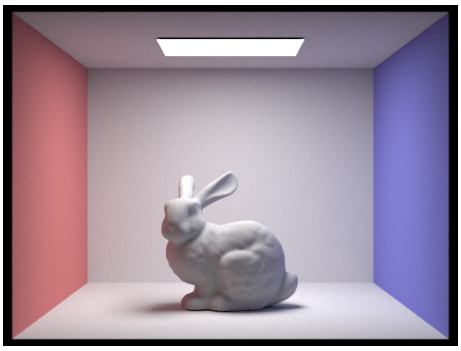


Figure 11: m=3.

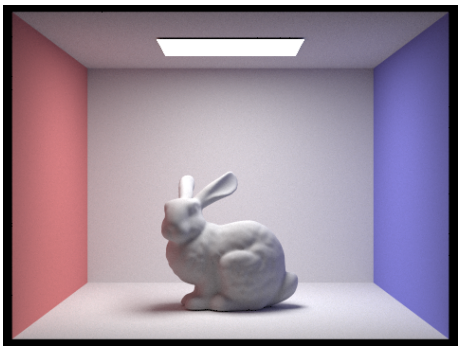


Figure 12: m=100.

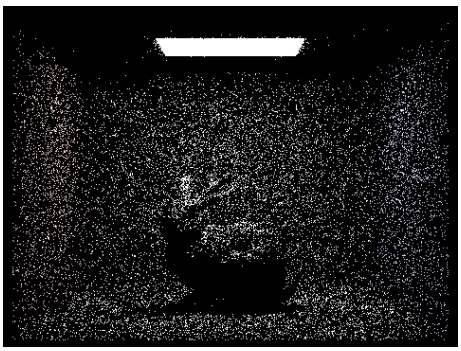


Figure 13: sample rate = 1.

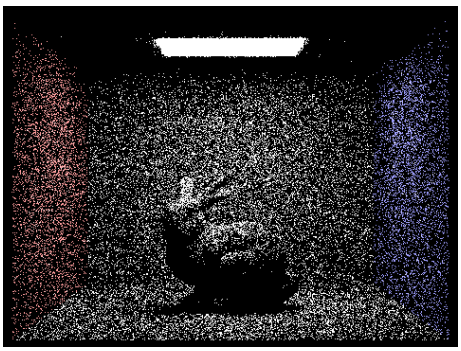


Figure 14: sample rate = 1.

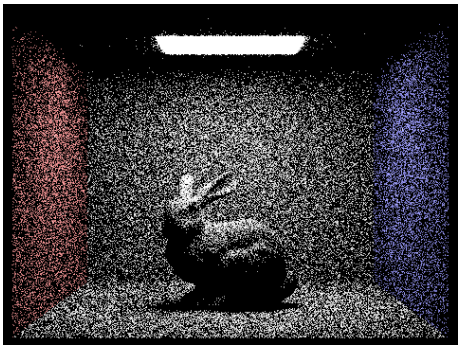


Figure 15: sample rate = 4.

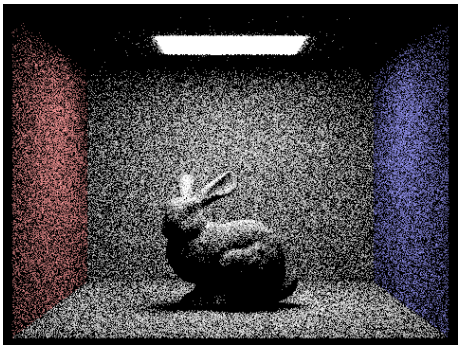


Figure 16: sample rate = 8.

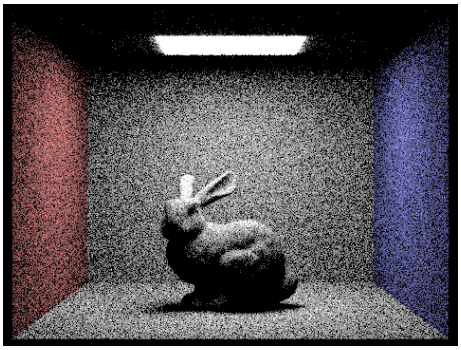


Figure 17: sample rate = 16.

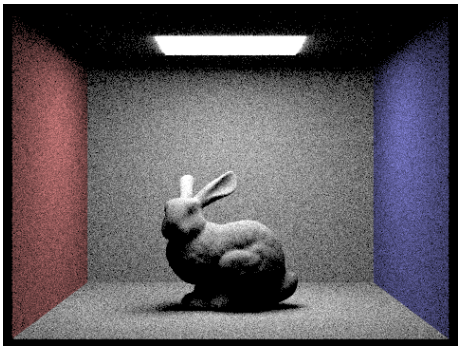


Figure 18: sample rate = 64.

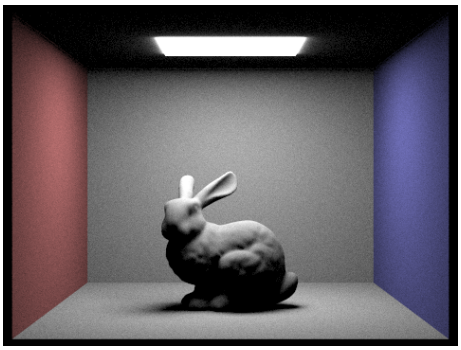


Figure 19: sample rate = 1024.

Figure 20: rate for all the sampling.

4.4 Change with sample rate

5 Part 5: Adaptive Sampling

5.1 Adaptive Sampling Implementation

In this part, I modified raytrace_pixel function to do the adaptive Sampling. Some pixels converge faster with low sampling rates, while other pixels require many more samples to get rid of noise. So we will use this algorithm to determine when to stop sampling.

```
Vector3D target_Color = est_radiance_global_illumination(my_ray);
  sum_Color += target_Color;
  s1 += target_Color.illum();
  s2 += target_Color.illum() * target_Color.illum();
  count++;
  sample_count++;

  // // JUDGE BATCHSIZE
  if(count == samplesPerBatch){
    count = 0;
    mju = s1/double(i+1);
    sigma_square = (1./double(i))*(s2 - (s1*s1)/double(i+1));
    iTest = 1.96 * sqrt(sigma_square)/sqrt(i+1);
    if (iTest <= maxTolerance*mju) {
      // COUT<<"HERE!!!"<<ENDL;
      break;
    }
  }
```

5.2 Results

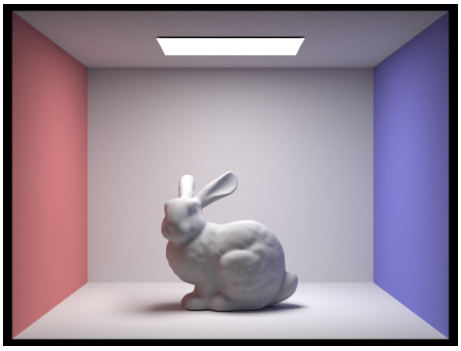


Figure 21: without adaptive sampling.

Figure 22: rate without adaptive sampling.

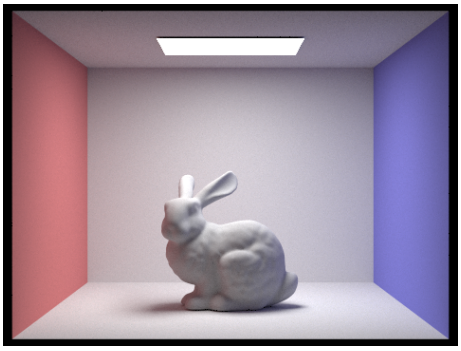


Figure 23: with adaptive sampling.

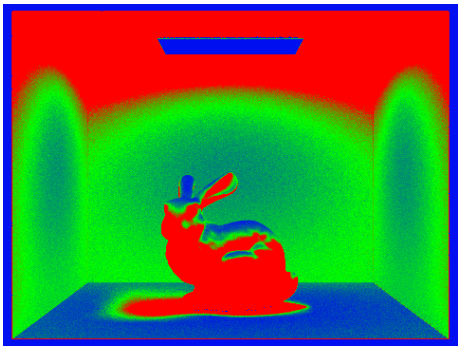


Figure 24: rate with adaptive sampling.