Writeup

Overview

In this project, I build a renderer that uses ray tracing to render images simulating physical lighting in the reality. I accelerate the renderer largely by implementing Bounding Volume Hierarchy and simulate direct illuminance (including uniform hemisphere sampling and importance sampling) and indirect illuminance (also known as global illuminance) based on the bounces of radiance. I learn a lot how the ray tracing works and how lighting working to render a good and real image.

Part1

The part1 of the assignment starts with the implementation of ray tracing in each pixel. There are ns\_aa random rays going through the image space, I get one random sample in each pixel the ray goes through by calling gridSampler->get\_sample() and get each ray by calling camera->generate\_ray(). I sum up all the radiance of these rays and average it, then update the color of the samplebuffer.

Generating ray is to convert the position of a ray from camera space into world space. Because the coordinates in camera space is different from coordinates in image space, we need to compute the corresponding position in the camera space based on the coordinates in image space. Then, once there is a ray, I convert the direction vector of it from camera space into world space by multiplying c2w.

The primitive intersection is the basic of ray generation. Rays are calculated by the origin point, time and direction vector. When they hit objects, there are different t values indicating different intersection points. The valid t values are those between min\_t and max\_t that the ray can be seen in the camera. What I want is the nearest the intersection points. Therefore, each time there is an intersection, I update the max\_t to be that t that generates this intersection point.

I use Möller Trumbore Algorithm for the triangle intersection algorithm, determining whether there is an intersection with the triangle.

Firstly, as I mentioned before, Ray is computed as o+td where o is the origin point, t is time, d is direction, and also can be calculated in Barycentric Coordinate with three vertices of the triangle (1 – b1- b2)\*P1 + b1\*P2 + b2\*P3. t, b1 and b2 can be represent as the cross product and dot product of different vectors.

Secondly, compute the normal of the triangle with Barycentric Coordinate (1 – b1- b2)\*n1 + b1\*n2 + b2\*n3 where n1, n2, n3 are the normal of vertices.

Finally, update the max\_t to the new t and also update the n, t, primitive and bsdf of the intersection points.

Part2

In the part2, I implement Bounding Volume Hierarchy (BVH) in order to accelerate the ray tracing. BVH is a binary tree structure where there is a rood node and followed by left and right child node. The internal nodes store bounding boxes, and left and right child nodes, while leaf nodes store bounding box and list of objects.

For the BVH construction algorithm:

Firstly, I initialize a bounding box. For all the objects in the primitives, I get the bounding box of them by calling (\*p)->get\_bbox(), and put them in the initialized bounding box, creating a root node.

Secondly, if the size of the primitives is under max\_leaf\_size, the root node is the leaf node, so set the start and end pointers to the exiting start and end pointers, simply returning the node.

Thirdly, if the size of the primitives is more than max\_leaf\_size, the bounding box will be split into left and right branches based on the splitting point.

Finally, I set node for left and right child and splitting the two children recursively.

The heuristic I choose for picking the splitting point is that:

Firstly, I initialize a bounding box for storing the centroid of each bounding box of the objects. I get the centroids by calling bb.centroid() and put them into this bounding box, all the centroids generate the splitting axis.

Secondly, I calculate the extent of each side with centroid.extent.xyz coordinates. Then I sum up the centroids of all the points in the primitive, average xyz coordinates of the them, which is the splitting point.

The main problem I debug in the part 2 is I always got the bad access exception. In order to solve the problem, I create new left and right child vectors in the BVHNode class so that each node holds its own vector. Then I set the start and end iterator to the vector.

Part3

In the part3, I implement the direct illumination which are sampling light uniformly in a hemisphere and important sampling. Zero-bounce and one-bounce illumination are direct illumination, where zero-bounce illumination only calculate the emission of the light from the light source by calling get\_emission() while one-bounce illumination call the one of these two direct lighting functions based on whether direct\_hemisphere\_sample from hemisphere or not.

For implementing the direct lighting with uniform hemisphere sampling:

Firstly, I loop through all samples of the light sources.

Secondly, for each sample, I get the sample direction by calling hemisphereSampler->get\_sample(), and convert coordinates of it into world space by multiplying o2w.

Thirdly, I offset the hit point hit\_p by EPS\_D to be the origin of a new ray. Then I cast a new ray based on the new origin and world space sample direction.

Fourthly, I check where the new ray intersects another light source. If true, I calculate the reflecting illuminance by calling f(wo, wi) which is the BSDF and light emission of the next light source by calling get\_emission(). I calculate the illuminance by multiplying the reflecting illuminance, light emission and cos(theta) of the incoming light.

Finally, I sum up all the illuminance I calculated in the last step during the loop and normalize it with pdf which is 1/(2 \* PI) and num\_samples.

Unlike uniform hemisphere sampling, importance sampling will sample all the lights directly:

Firstly, I loop through all the lights in the scene. For each light, if it is a delta light, I set num\_samples to be 1 since all samples are the same, otherwise, the num\_samples is set to be ns\_area\_light.

Secondly, I loop through all the samples in each light, get the sampling emitted radiance by calling SceneLight::sample\_L(Vector3D& p, Vector3D\* wi, float\* distToLight, float\* pdf), and convert the incoming direction vector into object frame.

Thirdly, I check whether the light is in front of the object by calling w\_in.z >= 0, if true, I offset the hit point hit\_p by EPS\_D to be the origin of a new ray. Then I cast a new ray based on the new origin and world space sample direction.

Fourthly, I check where the new ray intersects another light source. If false which means no other object between the hit point and the light source, I calculate the reflecting illuminance by calling f(wo, wi) and calculate the illuminance by multiplying the reflecting illuminance, sampling emitted radiance in got in step 2 and cos(theta) of the incoming light.

Finally, I sum up all the illuminance I calculated in the last step during the loop and normalize it with pdf and num\_samples.

After finishing all the parts, I found my rendering is a little bit dark. Therefore, instead of offset the hit point by EPS\_D in the direction, I keep the hit point to be hit\_p in both sampling function, but set the min\_t of the new ray to be EPS\_F in uniform hemisphere sampling, and set the max\_t of the new ray to be distToLight - EPS\_F as well as min\_t to be EPS\_F. Then it works better.

I compare the renderings of CBbunny.dae between uniform hemisphere sampling and importance sampling. They are both rendered at 64 samples and 32 light rays. The rendering time is roughly the same. But the qualities of the image are different. Uniform hemisphere sampling is more noise because the sample we get are uniformly in all direction around the intersection point. We only sample and calculate part of rays that point to a light. However, importance sampling is much smoother because by iterating all the lights in the scene we can get all the samples are from the light source that the emitted radiance is non-zero.

Part4

In the part4, I implement global illumination which is also considered as indirect illumination with more than one bounces illumination.

Firstly, this method uses Russian Roulette to sample unbiasedly. I set L\_out to be one\_bounce\_radiance(), get the BSDF by calling BSDF::sample\_f(Vector3D& w\_out, Vector3D\* w\_in, float\* pdf), and select the continuation probability to be 0.7.

Secondly, I check whether max\_ray\_depth is 0, 1 or over 1. If max\_ray\_depth is 0, return the radiance at zero-bounce; if max\_ray\_depth is 1, return L\_out which radiance at one-bounce; if max\_ray\_depth is over 1, then loop through over one bounces.

Thirdly, by looping through over one bounces, I check whether coin\_flip() is true with the continuation probability. If true, I convert coordinates of the incoming unit vector into world space, cast a new ray based on the hit point, incoming unit vector and depth of the ray, and also set the min\_t of the new ray to be EPS\_F.

Fourthly, I check where the new ray intersects another light source. If true, I calculate the radiance of that bounce of the light by calling at\_least\_one\_bounce\_radiance() recursively with the new ray and next intersection.

Finally, I sum up the normalized radiance by multiplying BSDF, each bounce of radiance and cos(theta) of the incoming light.

Part5

After part4, I have seen that Monte Carlo path tracing is very powerful in generating realistic images but also there always results in a large amount of noise. Therefore, in the part5, I implement adaptive sampling in order to avoid the problem of using a fixed high number of samples per pixel, by concentrating the samples in the more difficult parts of the image.

I update PathTracer::raytrace\_pixel() by creating another two illuminance parameters s1, s2 in order to track every sample's illuminance x to compute mean μ and standard deviation σ.

μ = s1 / n and σ^2 = 1 /(n – 1) \* (s2 – s1^2 / n)

where s1 is the sum of x and s2 is the sum of square of x.

Therefore, I trace the n samples through a pixel, and get mean μ and standard deviation σ, so pixel's convergence is

I = 1.96 \* σ / sqrt(n).

I check if

I <= maxTolerance \* μ,

If true, pixel has converged and stop tracing more rays for this pixel. If not, we continue the tracing-and-detecting loop.