Precomputed Probe Based Global Illumination With Visibility

Ziqi Luo, B.Eng.

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Augmented and Virtual Reality)

Supervisor: Michael Manzke

August 2022

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Ziqi Luo

August 15, 2022

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Ziqi Luo

August 15, 2022

Precomputed Probe Based Global Illumination With Visibility

Ziqi Luo, Master of Science in Computer Science

University of Dublin, Trinity College, 2022

Supervisor: Michael Manzke

Real-time global illumination has become a very important research topic, since it can provide better overall visual quality for many modern video games. Using probes to precompute the lighting information is one of the most popular way. However, original irradiance probes fail to capture the visibility information of the surrounding scene which lead to some light leaking artifact. NVIDIA has proposed a probe based method to solve this problem and this project mainly aims to have a further investigation on it.

Acknowledgments

My sincere thank you to my supervisor, Michael Manzke, for his valuable guidance. I would also like to thank my family for their support.

Ziqi Luo

University of Dublin, Trinity College
August 2022

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Global Illumination is one of the most important research area in computer graphics. Many researchers have already come up with a lot of solutions for rendering a scene with global illumination effect. Among all these methods, probe based global illumination is widely adopted by many games and game engines. This dissertation mainly aims to research on the probe based global illumination methods.

1.1 Motivation

Irradiance probes were very popular solution in game engines to bake global illumination and apply light maps to the game objects in the scene. However, this method has several drawbacks:

- Light leaking artifact caused by interpolating two probes if one of the probe is occluded from the light source.
- It can only capture the first bounce of the light

The first problem can be alleviated by manually placing the light probes and it can be a very expensive operation. The second problem will let some of the objects in a complex scene still have a completely dark surface even if it is supposed to be lit. In order to find a solution for the above problems, many researches are done already.

1.2 Objective

The objective is to have an investigation based on the NVIDIA's existing probe-based global illumination method and see if it can properly solve the problems I have just mentioned, compared to the original irradiance probes, and check out if there is any

performance issue after solving the problems. Further more, find out if there can be any improvement based on this method. The experiment is expected to be done in Unity3D with built-in render pipeline.

1.3 Structure of The Dissertation

Chapter 1 Introduction: This chapter points out the problems of the irradiance probe, and give the motivation and the objective of the dissertation.
Chapter 2 Related Work: This chapter will describe a list of work that is related to the dissertation. This will include some previous work that is the fundamental knowledge of the probe based global illumination, some techniques that can be used to optimize both the performance of the probes and the visual effect, and some state-of-the-art technologies that is currently being used by industries.
Chapter3 Design: This chapter describes the design of the experiment that can be used to see if the problem is resolved.
Chapter4 Theory & Implementation: This chapter focuses on explaining the techniques that I used and the implementation details of the original irradiance probes and NVIDIA's precomputed light field probes.
Chapter5 Evalutaion: This chapter lists the experiment hardware background and describe the experiment details based on the implementation. The result will contain the rendering frames and the performance for both original irradiance probe method and precomputed light field probe method.
Chapter6 Conclusions & Future Work: This final Chapter gives the conclusion of the experiment and list several drawbacks which might be able to improve in future work.

Chapter 2

Related Work

In this chapter, fundamental knowledge of the probe based global illumination, techniques that can be used for optimization, and some state-of-the-art is described.

2.1 Background & Previous Work

2.1.1 Image-based Lighting

This is exactly the technique where the idea of probe-based global illumination is coming from. There is already a lot of documentations and talks about image-based lighting and its variants. For example, Martin (2010), Ritschel et al. (2009), Sébastien and Zanuttini (2012) and so on. The goal of image-based lighting is to store the spherical surrounding environment lighting into a cube map, so that we can sample the lighting result in the later rendering process. This is why it is also called environment map. These maps were considered to be infinitely far away from the camera view and can directly sampled with given direction. However, the sample result can be view point biased if it is sampled based on some surface parameters, for instance, the reflection computed from surface normal. Further more, environment maps is not aware of the visibility information.

2.1.2 Irradiance Gradients

Original irradiance probe have some problem with the view point, because the samples taken from the surface is not at the probes' position. This will lead to some parallax errors for the rendering result. One of the way for the volume based probe is that by moving the surface to the center of the probe in probe space and compute the right sample location. However, we cannot guarantee that every time the scene also have a cube shape boundaries. Irradiance Gradients Ward and Heckbert (1992) is a method that

can be used to alleviate the problem. See Figure 2.1.

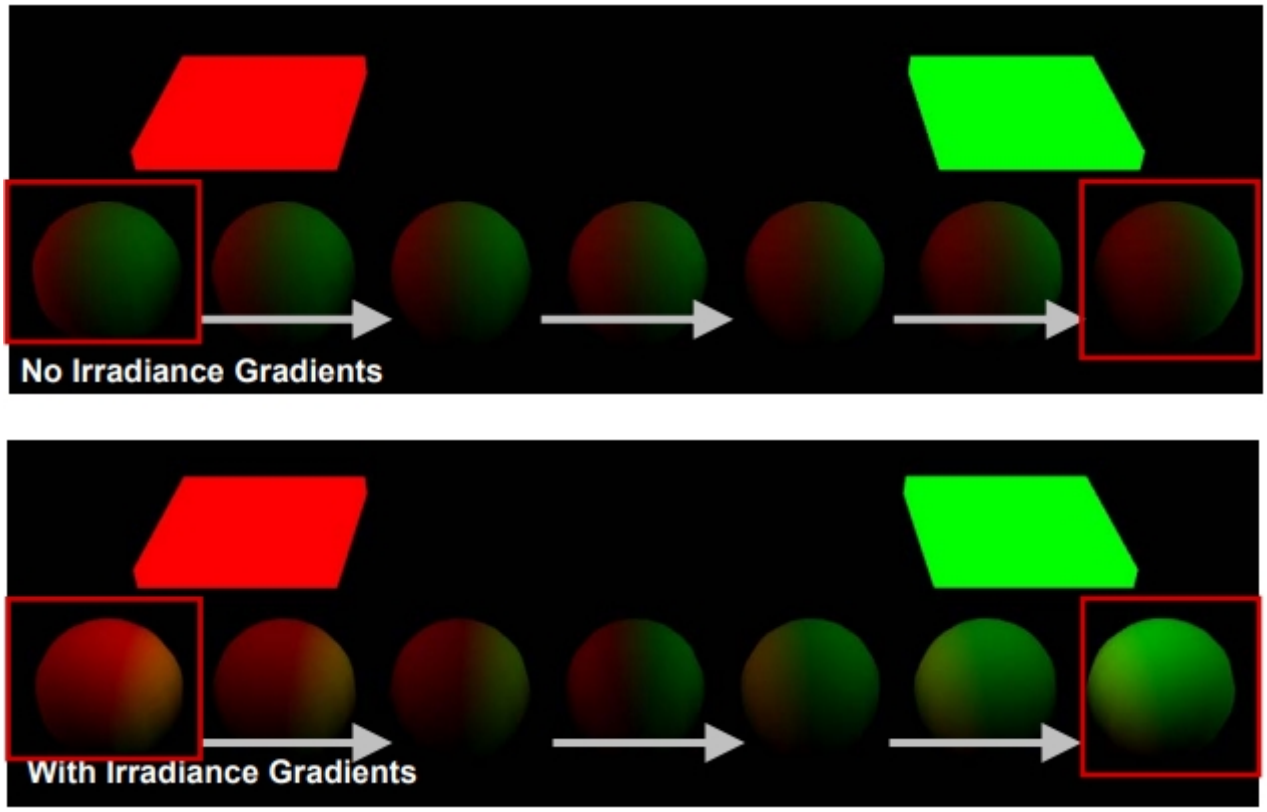

Figure 2.1: The irradiance varies with different positions. The figure is from Tatarchuk (2005)

2.1.3 Octahedral Mapping

Cigolle et al. (2014) did an investigation on unit vector encoding methods, and I used octahedral mapping(Figure 2.2) instead of cube mapping or spherical mapping for it has less distortion and lower channel use. It use only 2D vector to encode a 3D vector, which makes it possible to use only one square texture to store the spherical information. It can be down sampled based on a higher resolution cube map and this is exactly what I did in the implementation. The only problem is that the when sampling the boundaries of the octahedral encoded map with bilinear filtering, it cannot pick the correct texel on the other side of the texture. It can be solved by using concentric octahedral map Clarberg (2008), or simply by padding the texture with one texel width to match the correct one. What I did is the later one, detailed implementation can be found in the later description. Octahedral mapping is very important for light probes since when sampling the probes, there will be a number of probes to sample. If the texture size is too great may lead to

some performance issue. With octahedral mapping to down sample the original six-face cube map texture, this problem can be alleviated.

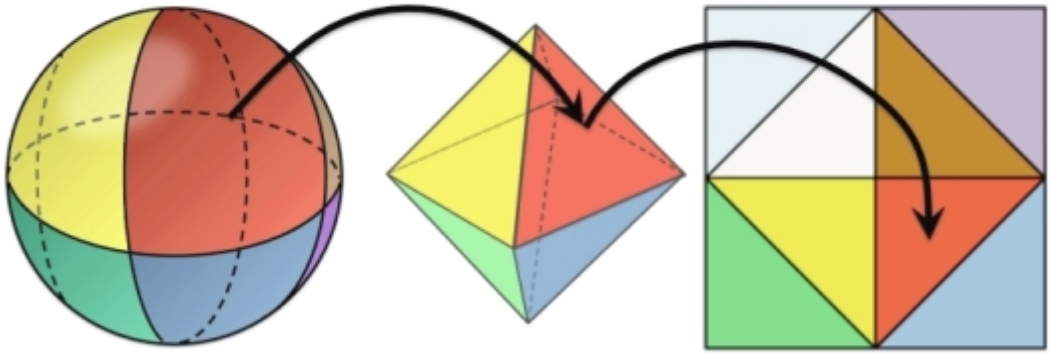

Figure 2.2: The oct maps faces is projected to a plane and unfolds into a unit square. The figure is from Cigolle et al. (2014)

2.2 State of the Art

2.2.1 Precomputed Light Field Probe

NVIDIA introduced a method which is called precomputed light field probes McGuire et al. (2017) which gets the problems resolved. It has the following features:

- A new probe data structure which encodes the geometric information in the scene, which is similar to a cube map G-Buffer.
- Allow ray tracing using the probes instead of the camera view.
- Prefiltered irradiance map with visibility information which can remove the light leaking artifact.

2.2.2 Dynamic Diffuse Global Illumination

Based on the previous work on precomputed light field probe, Majercik et al. (2019) further extend the idea of precomputed light field probe for rendering dynamic objects. It still have a cube map G-buffer data structure, but the rendering process has a little bit change. When ray tracing the radiance in the scene, in the previous work, the result is simply stored in a frame buffer so that they can iteratively update the radiance to get multi-bounce result. This result has to be precomputed for large amount of computations.

But for dynamic objects, if the objects move, the precomputed result is no longer correct, and the lighting information needs to be updated. In order to let the ray tracing algorithm can be computed in real-time. They decided to compute the multi-bounce result in different frames. Every time the ray tracers and trace rays base on the indirect illumination computed in the previous frame. By accumulating the indirect radiance results, it can be expected to reach infinite bounces. Then the radiance can be updated in real-time, which means it can support dynamic objects. The problem is that, since the illumination information is updated in real-time, the denoising and filtering also have to be done in real-time. This actually limits the resolution of the textures. If the texture size is too great, image processing pass will lead to some performance issue. They finally decided to use 88 resolution for irradiance octahedral maps, which is enough for diffuse, but it has to combine with some other state-of-the-art glossy ray tracing algorithm to get complete rendering effect.

Chapter 3

Design

In order to achieve the objective I mentioned. I will compare the original irradiance probe global illumination with the precomputed light field probe global illumination results.

3.1 Overview of the Approach

I will first implement both original irradiance probe global illumination and precomputed light field probe global illumination in Unity3D with Unity built-in render pipeline and the support of DirectX 12. They will have similar properties like the cube map resolution, and the number of probes that is placed in the scene. The probe will be placed in a cube grid volume uniformly so that it is more likely to get light leaking artifact for original irradiance probe method. The objects for the scene will be completely diffuse for easier implementation and will be shape in boxes for easier sampling. Implementation details can be found in the next chapter.

Chapter 4

Theory And Implementation

4.1 Common Theory

This section describes the knowledge and techniques that are applied by the both probe based methods.

4.1.1 The reflectance Equation

The reflectance equation is known as:

$$L_o(p, \omega_o) = \int_{\Omega+} L_i(p, \omega_i) f_r(p, \omega_i, \omega_o) \cos\theta_i V(p, \omega_i) d\omega_i \tag{4.1}$$

which L_i is the incoming radiance, f_r is the BRDF(Bidirectional Reflectance Distribution Function), and V is the radiance visibility of the shading point. For very special cases, for example, when the light source is directional light or point light, the visibility part is actually having the same function as the one of the shadow map.

4.1.2 Bidirectional Reflectance Distribution Function

For the BRDF, I used Cook-Torrance BRDF which contains 2 different terms,which are Lambert diffuse part and a glossy part:

$$f_r = k_d f_{lambert} + k_s f_{cook-torrance} \tag{4.2}$$

Where k_d is the ratio of the energy gets refracted and k_s is the ratio gets reflected.
The Lambertian diffuse BRDF is $\frac{c}{\Pi}$ which means the incoming light is uniformly distributed and the integration of the light is the sum of the incoming light.
The Cook-Torrance BRDF is $\frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$, Where D is the normal distribution function(NDF),

F is the Fresnel term, and G is the geometry function. I used Trowbridge-Reitz GGX for NDF term, Schlick-GGX for geometry function, and Fresnel-Schlick approximation for the Fresnel term. The integration of the light is the weighted sum of the incoming light where the light direction that has small angles between the specular reflection direction gets higher weight.

4.1.3 Split Sum Approximation

One of the problem for solving the glossy part of the integration is that it has too many parameters. Unreal engine introduced an idea called Split Sum Approximation Karis and Games (2013) to split the equation into light part and BRDF part. It is based on a very classic approximation which is:

$$\int_{\Omega} f(x)g(x)dx \approx \frac{\int_{\Omega_G} f(x)dx}{\int_{\Omega_G} dx} \cdot \int_{\Omega} g(x)dx \tag{4.3}$$

This approximation is accurate when the integration support area of is small or the is smooth(which means the value does not vary to much in integration support area). For glossy BRDF, it has small support area, and for diffuse BRDF, it has smooth value changes(Figure 4.1).

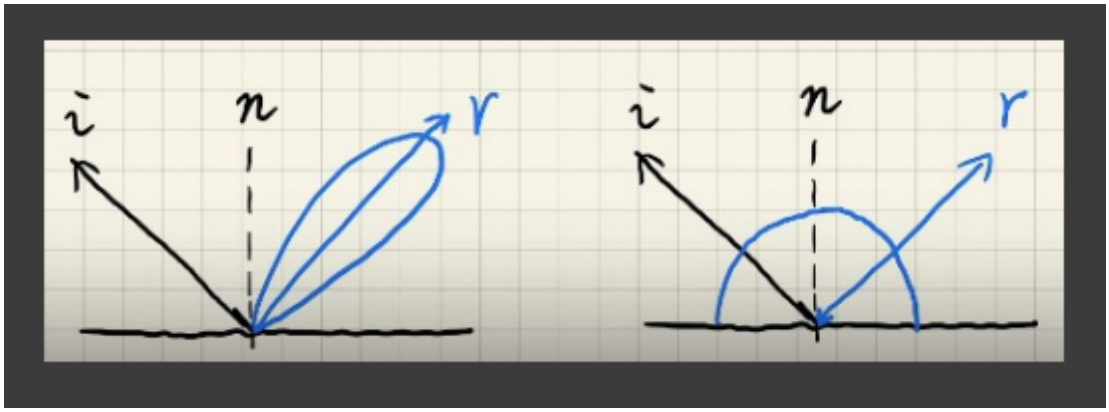

Figure 4.1: Integration support of glossy(left) and diffuse(right) BRDF

So I can split the integration into two different part and the rendering equation becomes like this:

$$L_o(p,\omega_o) \approx \frac{\int_{\Omega_{fr}} L_i(p,\omega_i)d\omega_i}{\int_{\Omega_{fr}} d\omega_i} \cdot \int_{\Omega+} f_r(p,\omega_i,\omega_o)\cos\theta_i d\omega_i \tag{4.4}$$

From the above, we can see that what we need to collect by the light probes is:

- An irradiance map which is the sum of the incoming light

- An filtered radiance map which is the weighted sum of the incoming light

For different types of probes, I used a little bit different methods to compute the irradiance and filtered radiance.

4.1.4 BRDF Look Up Table

From what I previously mentioned, we know that the BRDF integration have too many parameters that it is not efficient to generate a high dimensional texture to precompute the BRDF. In order to split the parameter, I have to move the Fresnel term out of the BRDF by following method:

$$\int_{\Omega+} f_r(p, \omega_i, \omega_o) \cos\theta_i d\omega_i = \int_{\Omega+} \frac{f_r(p, \omega_i, \omega_o)}{F(\omega_o, h)} F(\omega_o, h) \cos\theta_i d\omega_i \tag{4.5}$$

With Fresnel-Schlick approximation we can get:

$$\int_{\Omega+} f_r(p, \omega_i, \omega_o) \cos\theta_i d\omega_i \approx \int_{\Omega+} \frac{f_r(p, \omega_i, \omega_o)}{F(\omega_o, h)} (F_0 + (1 - F_0)(1 - \cos\Phi)^5) \cos\theta_i d\omega_i \tag{4.6}$$

Finally we can split the Fresnel function F over two parts:

$$F_0 \int_{\Omega+} \frac{f_r(p, \omega_i, \omega_o)}{F(\omega_o, h)} (1 - (1 - \cos\Phi)^5) \cos\theta_i d\omega_i + \int_{\Omega+} \frac{f_r(p, \omega_i, \omega_o)}{F(\omega_o, h)} ((1 - \cos\Phi)^5) \cos\theta_i d\omega_i \tag{4.7}$$

We can see that the integration becomes a scale and a bias based on F_0, and both scale and bias can be precomputed with certain roughness and $\cos\theta_i$. This means that I can get BRDF value by checking the BRDF look up table(BRDF LUT) during the rendering process(Figure 4.2).

4.1.5 Low-Discrepancy Sequence & Importance Sampling

Quasi-Monte Carlo Integration requires generating uniformly distributed random samples. It can be done with a low-discrepancy sequence. A commonly seen low-discrepancy sequence is the two dimensional Hammersley Sequence which is based on the Van Der Corput Sequence which mirrors a decimal binary representation around its decimal point and this just lead to the fact that when generating samples, the sequence will always try to fill the samples into places that has the greatest spacing between other samples, and this is the key for generating uniformly distributed samples. Its definition is very simple:

$$X_i := (\frac{i}{N}, \Phi_2(i)) \tag{4.8}$$

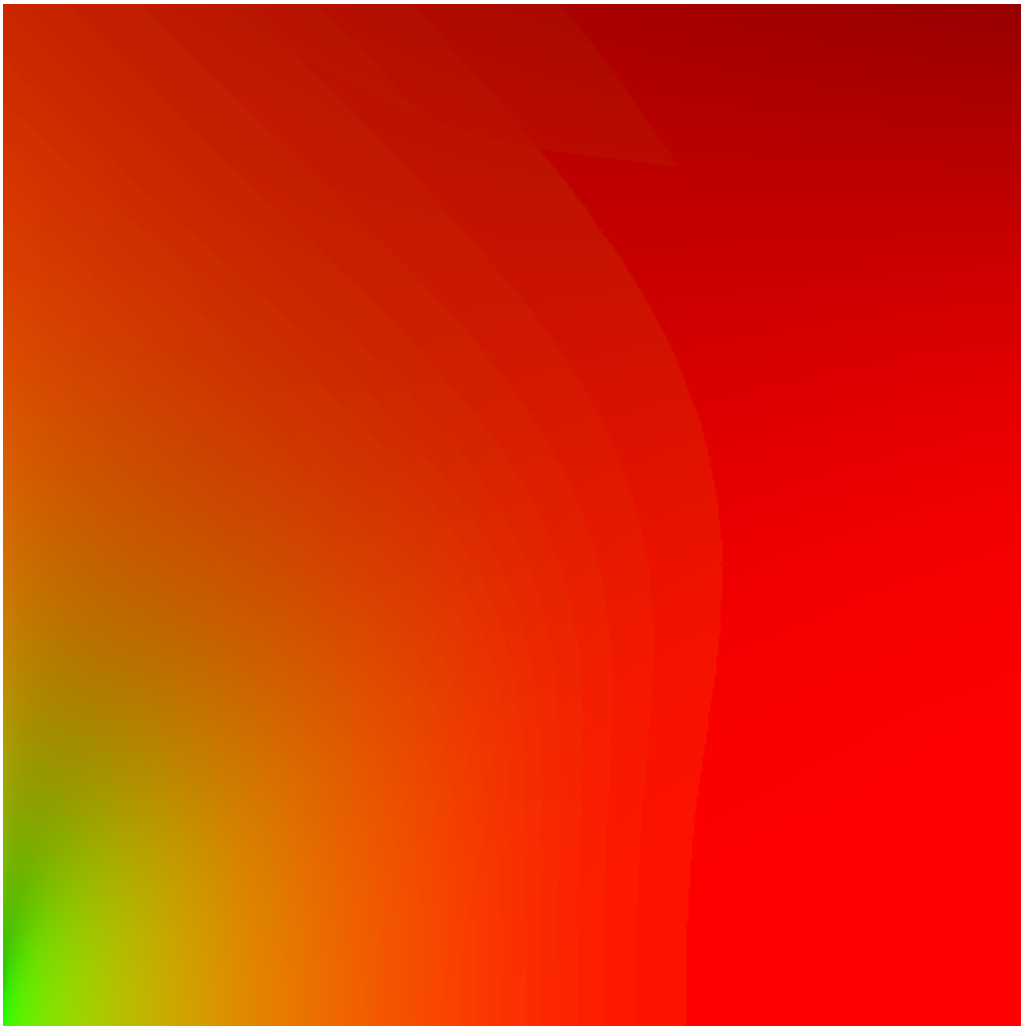

Figure 4.2: BRDF LUT (horizontal parameter is $\cos\theta_i$, vertical parameter is roughness, R channel is scale value, G channel is bias value)

N is the total number of the samples, i is the sample index, $\Phi_2(i)$ is the radical inverse function that returns the number obtained by reflecting the digits of the binary representation of i around the decimal point (See Table 4.1).

Index	**Binary**	**Binary Inverse**	**Binary Value**
1	1	0.1	0.5
2	10	0.01	0.25
3	11	0.11	0.75
4	100	0.001	0.125
...	...	...	...

Table 4.1: Hammersley Sequence

Monte Carlo Integration can greatly reduce the number of samples needed for approx-

imating the integration by taking uniformly distributed random samples. In order to get fast convergence rate, it is better to reduce samples by taking samples which are very close to the reflection direction for glossy reflections, however, the result is biased for trade off. Detailed mathematical topic will not be delve further here. It can be find in Colbert and Krivanek (2007). The code for implementing low-discrepancy sequence and importance sampling can be found below (Listing 4.1).

```
float RadicalInverseSeq(uint bits)
{
    uint inversedBits = 0u;
    uint bitNumValue = 1u;
    for (; bits > 0; bits /= 2u)
    {
        inversedBits = inversedBits * 2u + bits % 2u;
        bitNumValue = bitNumValue * 2u;
    }
    return inversedBits / float(bitNumValue);
}

float2 HammersleySeq(uint i, uint N)
{
    return float2(float(i) / float(N), RadicalInverseSeq(i));
}

float3 GetRandomImportanceSampling(in uint2 sampler,
    in float3 dir, in float rough)
{
    float r = rough * rough;
    //Bias sample vectors to the specular lobes
    float phi = 2.0 * M_PI * sampler.x;
    float cosTheta = sqrt((1.0 - sampler.y) /
        (1.0 + (r * r - 1.0) * sampler.y));
    float sinTheta = sqrt(1.0 - cosTheta * cosTheta);

    //transform from sphereical coord to cartesian coord
    float3 sampleDirT;
    sampleDirT.x = cos(phi) * sinTheta;
    sampleDirT.y = sin(phi) * sinTheta;
```

```
    sampleDirT.z = cosTheta;

    //transform from tangent to world
    float3 up = abs(dir.z) < 0.999 ?
        float3(0.0, 0.0, 1.0) :
        float3(1.0, 0.0, 0.0);
    float3 tangent = normalize(cross(up, dir));
    float3 bitangent = cross(dir, tangent);

    float3 sampleDirW = tangent * sampleDirT.x +
        bitangent * sampleDirT.y +
        dir * sampleDirT.z;
    return normalize(sampleDirW);
}
```

Listing 4.1: Low-discrepancy sequence and importance sampling

4.1.6 Parallax Correction

One of the most important issue with IBL is that the position of the reflection is not right since the probe is sampling the surrounding lighting information from its own position, not from the point of interests that we are rendering, which may lead to some specluar bias artifact (Figure 4.3).
Since my main objective is to solve the visibility problem. I simply created a box scene which is a lot easier to find correct sample direction to avoid the view point biased problem instead of using irradiance gradients.

4.2 Global Illumination With Original Irradiance Probe

This section describes the theory and implementation for the original irradiance probe method. The workflow of the global illumination based on this method is shown in Figure 4.4:

4.2.1 Radiance

In order to compute the irradiance in the space, I need to capture the radiance coming from the surround scene. Radiance can be directly obtained by placing a camera at the center of the probe to render into a cube map.To do that, camera view port needs to be

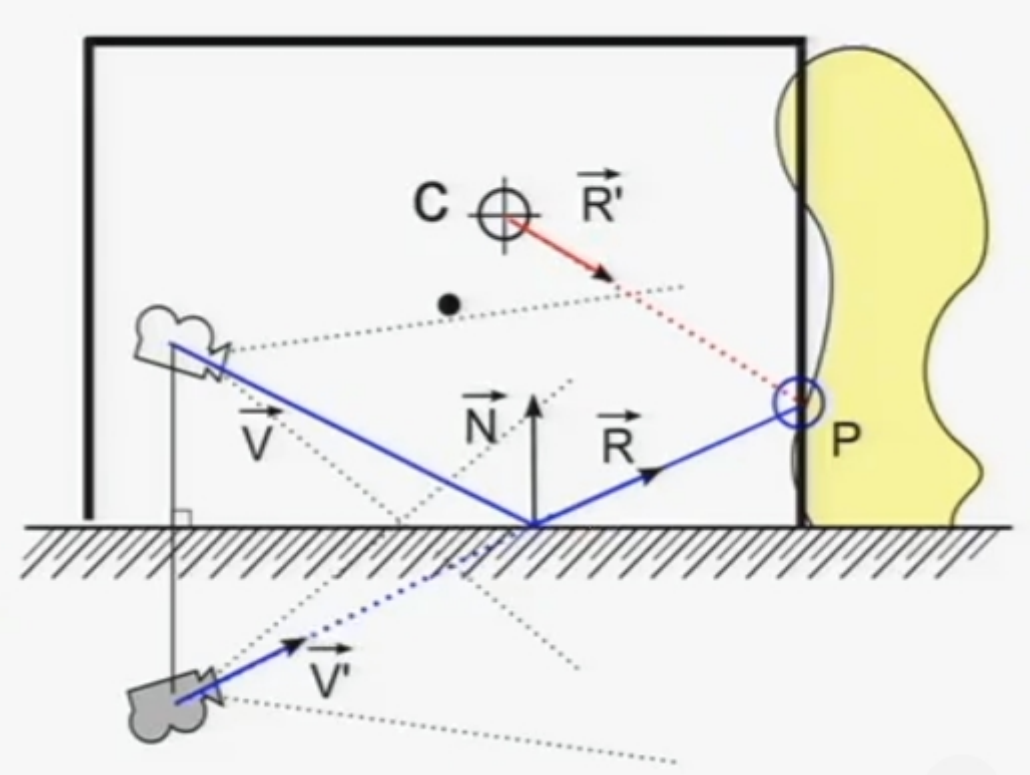

Figure 4.3: The sampling direction is corrected Sébastien and Zanuttini (2012)

set to a square shape and render in to 6 different faces of the cube map 6 times with matching direction. It will capture the radiance coming from the emissive objects, sky box (if any), and directly lit surface by none-area light. This can be done with a specific function in unity with one render pass. The texture size is set to 256^2 for each cube map face with R11G11B10Float format. The far plane is set to the same size as the one of the probes boundaries.

4.2.2 Irradiance

For the original irradiance probe, irradiance is computed directly by uniformly taking samples from the hemisphere of the surface normal direction. I simply set a constant number of samples. The integration can be approximated by the above method. Code for generating irradiance can be found in listing 4.2

```
void GenerateIrradiance(uint3 id : SV_DispatchThreadID)
{
    float3 irradiance = float3(0, 0, 0);
```

```
float3 tangentUp = float3(0.0, 1.0, 0.0);
float3 tangentRight = normalize(cross(tangentUp, bufferDir));
tangentUp = normalize(cross(bufferDir, tangentRight));

float azimuthOffset = (PI * 2) / 500;
float zenithOffset = (PI * 2) / 500;
float weightSum = 0;
for (float phi = 0; phi < 2.0 * PI; phi += azimuthOffset)
{
    for (float theta = 0; theta < 0.5 * PI;
        theta += zenithOffset)
    {
        float3 sampleDirTan;
        sampleDirTan.x = cos(phi) * sin(theta);
        sampleDirTan.y = sin(phi) * sin(theta);
        sampleDirTan.z = cos(theta);
        sampleDirTan = normalize(sampleDirTan);
        float3 sampleDir = sampleDirTan.x * tangentRight
            + sampleDirTan.y * tangentUp
          + sampleDirTan.z * bufferDir;
        float2 samplerDirOct =
            float3ToOct(normalize(sampleDir)) * 0.5 + 0.5;
        float3 L = normalize(2.0 * dot(bufferDir, sampleDir)
            * sampleDir - bufferDir);

        float cosWeight = max(dot(bufferDir, L), 0.0);
        irradiance += _RadianceOct.SampleLevel
            (sampler_linear_clamp, samplerDirOct, 0).rgb
            * cos(theta) * sin(theta);
        weightSum += 1;
    }
}


irradiance *= (PI / float(weightSum));
```

```
    _IrradianceOct[id.xy] = irradiance;
}
```

Listing 4.2: Irradiance Generator

Note that I also did an octahedral encoding for all the cube maps for original irradiance probes since it is a very nice optimization for the memory.

4.2.3 Prefiltered Radiance

From left part of the formula 4.4, we can see that it is integrating the irradiance on the BRDF supported area, and there is a normalization factor on the dominator. This means blurring the image or averaging the image values in certain sized filter box and I only have to sample the pixel value on reflection direction (Figure 4.5). From the GGX distribution, we can know that the filter box size is defined by the roughness of the surface. So based on different steps of roughness values, I have to define different filter box sizes. Luckily, GPU has already got a function that matches this requirement, which is mipmapping. So we can define the mipmap levels based on the roughness value of the object's surface and look up the right level of the filtered irradiance to get the correct lighting. This can be precomputed if the objects in the scene are static, which means they are not moving around.

4.2.4 Shading With Multiple Probes

Because there are multiple probes placed in the scene, I have to decide which probes to sample for each shading point of interest. I can compute weights based on the distance between the shading point and the center of the probes. It has to follow the rule that the closer probes should have higher weight. Here I use trilinear interpolation. However, trilinear interpolation can only applied when the probes are placed in cube volumes pattern, and this means giving up manually placing probes. This is what exactly precomputed light field contribute to. Code for trilinear interpolation can be found below:

```
UNITY_LOOP
for(int k = 0; k < numOfProbe; k++)
{
    float3 probeDir = _ProbePositionW[k] - i.posW;
    float3 trilinearXYZ = probeDir / float3(4.0, 4.0, 4.0);
    trilinearXYZ = clamp(abs(trilinearXYZ),
        float3(0.0, 0.0, 0.0), float3(1.0, 1.0, 1.0));
    trilinearXYZ = float3(1.0, 1.0, 1.0) - trilinearXYZ;
```

```
    float probeTrilinearWeight = trilinearXYZ.x *
        trilinearXYZ.y * trilinearXYZ.z;
}
```

Listing 4.3: Trilinear Interpolation

4.3 Global Illumination With Precomputed Light Field Probe

This section describes the theory and implementation for the precomputed light field probe method. The workflow of the global illumination based on this method is shown in Figure 4.6:

4.3.1 Cube Map G-buffer

Different from the original G-buffer which is used for deferred rendering techniques. This G-buffer stores different data which contains: radiance of direct illumination, surface normal, and radial distance (the distance between the probe center and the shading surface point). Unity3D's built-in function for rendering surrounding scene does not support multiple render target, so it is not advised to use this function to render the G-buffer since this will greatly increase the number of draw calls. It is better to draw each face into 2D textures with multiple render target feature and then copy the 2D texture to each cube map faces by exploiting GPU. For each cube map face, the resolution is set to 2562 for each data I previously mentioned. Radiance cube map format is set to R11G11B10Float. As for normal cube map, I only use 2 channels to store it with R16G16Float format. It can be stored in only 2 channels is because I encode the normalized normal direction with octahedral mapping, which is the same technique I use for encoding cubemaps into octahedral maps. Radial distance uses R16SFloat format.

4.3.2 Octahedral Down sampling

Octahedral maps have less distortion than cube maps and the data is more uniformly distributed. This makes octahedral maps can use smaller resolution to get the same visual quality of the one that cube maps with higher resolution can provide. Here I used the sample approach used in the investigation, which is by down sampling the higher resolution cube maps into lower resolution octahedral maps. However, if I directly resample the texture by averaging the samples taken from the cube maps will make the results look

very washout. I add an additional parameter which is used to adjust the weight of samples. For each cube map sample, I compute a dot production of the direction of the octahedral map sample and the direction of the cube map sample to get a weight and do the following computation:

$$weight = (dir_{oct} \cdot dir_{cubemap})^{sharpness} \tag{4.9}$$

Where sharpness is used to control the weight. Higher the sharpness means the cube map sample direction which is close the octahedral map sample direction will have higher weight.

4.3.3 Iterative Ray Tracing With Probe Data

Here my implementation is a little bit different from the one mentioned in the paper, since when they did their research, NVIDIA did not include acceleration structure for ray tracing in their graphics card. Since I have already got RTX series graphics card, I decided to implement the ray tracing function with DirectX12's DXR API to exploit the GPU.

Ray tracing is usually done recursively with tracing depth value to define how many times the ray bounce off the surface. However, iterative ray tracing only compute the first bounce of the ray by sampling the radiance I previously saved, and store the indirect radiance into a frame buffer and accumulate the radiance. For later iteration, instead of sampling the radiance, it will sample the indirect radiance I have just saved to simulate multi-bounce. Every time when sampling the buffer, I also have to compare the sampling distance with radial distance which I previously saved in cube map G-buffer. If the sampling distance is greater than the radial distance, that means the current hit is not visible to this probe. In this case, the radiance should not be sampled, and it should be computed with other probes nearby.

Reflection ray is dispatched with a random direction based on importance sampling. I used the same method for generating pseudorandom number in GPU which is introduced in Howes and Thomas (2007). Since I only take one sample per pixel during ray tracing progress. This means the final accumulate radiance result can be very noisy. I denoised the radiance with cross-bilateral filters, using G-buffer normals and depths for measuring high frequency details.

4.3.4 Compute Irradaince From Radiance and Filtered Radial Distance

Instead of using Quasi-Monte Carlo Integration to approximate, which is mentioned in the paper, I directly distributed the samples uniformly on the hemisphere since I will be simulating the scene with all diffuse objects. In this case, the result is pretty accurate. Notice that the Chebychev Visibility Test requires the variance of the radial distance. And variance can be computed with this function:

$$\sigma^2 = |E[r]^2 - E[r^2]| \tag{4.10}$$

Where r is the radial distance. For this, I can save filtered radial distance and filtered squared radial distance into one texture with 2 channels. Since the normals in the cube map G-buffer is no longer needed in later computation. I can use the texture, which format is R16G16Float and used by the normals to store the new radial distance information and discard the original single-channel texture to save more memory.

4.3.5 Octahedral Map Padding

One problem with the octahedral map is that when I need to do a bilinear filtering on the texture, the boarders is not taking the correct texel samples (Figure 4.7), and bilinear filtering on radial depth is required for later Chebychev Visibility Test. One texel width padding is required to make sure the bilinear filtering can work properly. Padding method can check out the following figure(Figure 4.8). The source code for padding can be found in listing 4.4.

```
void OctPaddingFloat2(uint3 id : SV_DispatchThreadID)
{
    uint bufferSizeMOne = _BufferSize - 1u;
    uint bufferSizeMTwo = _BufferSize - 2u;
    uint bufferSizeHalf = _BufferSize >> 1;

    if (id.x > bufferSizeMOne || id.y > bufferSizeMOne)
        return;

    if (id.x == 0)
    {
        if (id.y == 0)
            _PaddedOctFloat2[id.xy] =
```

```
_PaddedOctFloat2[uint2(bufferSizeMTwo, bufferSizeMTwo)];
        else if (id.y == bufferSizeMOne)
            _PaddedOctFloat2[id.xy]
= _PaddedOctFloat2[uint2(bufferSizeMTwo, 1u)];
        else if (id.y < bufferSizeHalf)
            _PaddedOctFloat2[id.xy] =
_PaddedOctFloat2[uint2(1u, bufferSizeHalf +
(bufferSizeHalf - id.y - 1u))];
        else
            _PaddedOctFloat2[id.xy] =
_PaddedOctFloat2[uint2(1u, 1u + bufferSizeMTwo - id.y)];
    }
    else if (id.x == bufferSizeMOne)
    {
        if (id.y == 0)
            _PaddedOctFloat2[id.xy] =
_PaddedOctFloat2[uint2(1u, bufferSizeMTwo)];
        else if (id.y == bufferSizeMOne)
            _PaddedOctFloat2[id.xy] =
_PaddedOctFloat2[uint2(1u, 1u)];
        else if (id.y < bufferSizeHalf)
            _PaddedOctFloat2[id.xy] =
_PaddedOctFloat2[uint2(bufferSizeMTwo, bufferSizeHalf +
(bufferSizeHalf - id.y - 1u))];
        else
            _PaddedOctFloat2[id.xy] =
_PaddedOctFloat2[uint2(bufferSizeMTwo,
1u + bufferSizeMTwo - id.y)];
    }
    else if (id.x < bufferSizeHalf)
    {
        if (id.y == 0)
            _PaddedOctFloat2[id.xy] =
_PaddedOctFloat2[uint2(bufferSizeHalf +
(bufferSizeHalf - id.x - 1u), 1u)];
        else if (id.y == bufferSizeMOne)
            _PaddedOctFloat2[id.xy] =
```

```
_PaddedOctFloat2[uint2(bufferSizeHalf +
(bufferSizeHalf - id.x - 1u), bufferSizeMTwo)];
    }
    else
    {
        if (id.y == 0)
            _PaddedOctFloat2[id.xy] =
_PaddedOctFloat2[uint2(1u + bufferSizeMTwo - id.x, 1u)];
        else if (id.y == bufferSizeMOne)
            _PaddedOctFloat2[id.xy] =
_PaddedOctFloat2[uint2(1u + bufferSizeMTwo -
id.x, bufferSizeMTwo)];

    }

}
```

Listing 4.4: Octahedral Map Padding

I cannot come up with better idea to complete the padding operation but using a large number of branches. But, considering that this is the precomputation part, it is still acceptable.

4.3.6 Shading With Multiple Probes

Here the probe is also place in a cube volume shape. I also use trilinear interpolation for this type of probes. One thing that has to be noticed that, for a very large scene using cube volume. There is going to be a very large number of probes. It is not possible to pass all the probe data in to the shader that I used for final rendering result, since it can be a great impact on GPU memory. Usually the scene should be divided using a 3D grid and pass only necessary probes data into the shader, since the probes that are too far away from the object will not have any contribution to the shading result.

4.3.7 Chebychev Visibility Test

Using the weights computed directly from the trilinear filtering process will still lead to light leaking artifact, since the visibility information is not yet applied to the rendering technique. Here I did something very similar to the one with the technique used in Variance Shadow Map Donnelly and Lauritzen (2006). In their case, they used a Chebychev

Inequality which is:

$$P(x > t) \leq \frac{\sigma^2}{\sigma^2 + (t - \mu)^2} \tag{4.11}$$

Where μ is the mean of x, σ is the variance, and t is the compared distance or depth. Here I take the maximum value that the possibility can reach as the a visibility weight for each probe and multiplied by the weight I previously computed by trilinear interpolation to get the final weight. For those probes which are completely occluded from the shading point will have zero weight which means they will never be sampled during the shading progress.

Srouce code for visibility test can be found in listing 4.5.

```
//Visibility
float3 sampleDir = normalize(i.posW - _ProbePositionW[k]);
float2 sampleDirOct = float3ToOct(sampleDir) * 0.5 + 0.5;
sampleDirOct *= (1.0 - _TexelSize);
sampleDirOct += _TexelSize;
float2 meanDistance = UNITY_SAMPLE_TEX2DARRAY(
  _RadialDistanceArray,
  float3(sampleDirOct, k));
float variance = abs((meanDistance.x * meanDistance.x)
  - meanDistance.y);
float varianceWeight = 1.0;
float probeDistance = distance(i.posW, _ProbePositionW[k]);
float biasedDistance = probeDistance - 0.01;
if(meanDistance.x < biasedDistance)
{
    float j = biasedDistance - meanDistance.x;
    varianceWeight = variance / (variance + (j * j));
    varianceWeight *= varianceWeight * varianceWeight;
}
probeTrilinearWeight *= varianceWeight;
```

Listing 4.5: Chebychev Visibility Test

As we can see that, the texture coordinates for sampling the radial distance are scaled and biased. This is because that I previously did a one texel padding for the octahedral maps. The correct sampling position has changed (The texture coordinates for generating the octahedral maps are also changed)

Another important detail is that I also biased the surface plane a little bit to prevent self

shadowing aliasing.

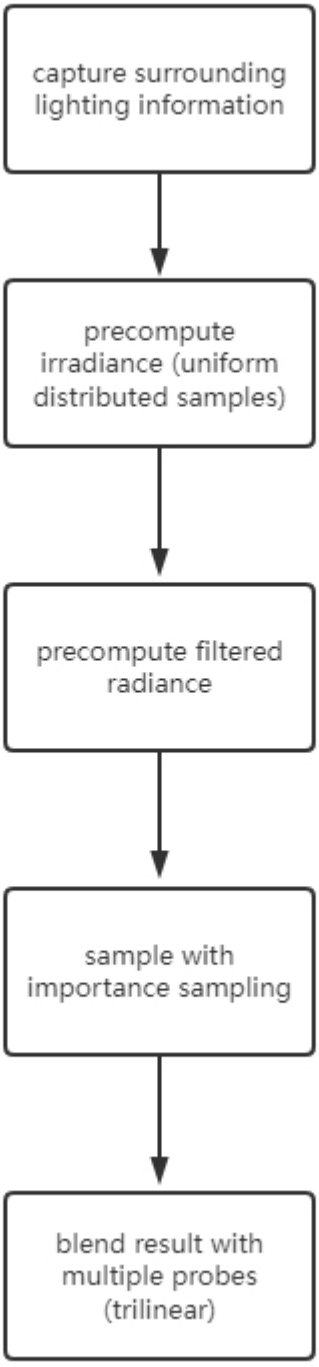

Figure 4.4: The global illumination Workflow of the original irradiance probes

Figure 4.5: Sampling filtered radiance from a glossy reflection vector

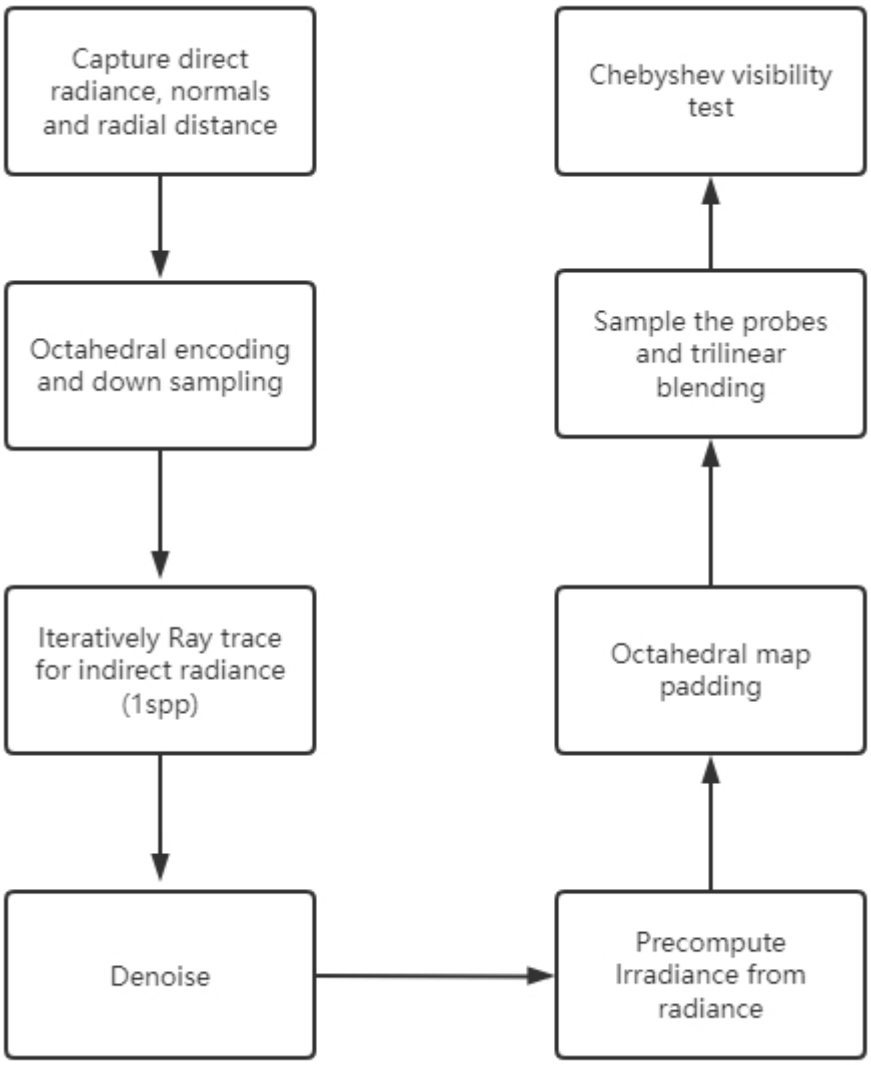

Figure 4.6: The global illumination Workflow of the light filed probes

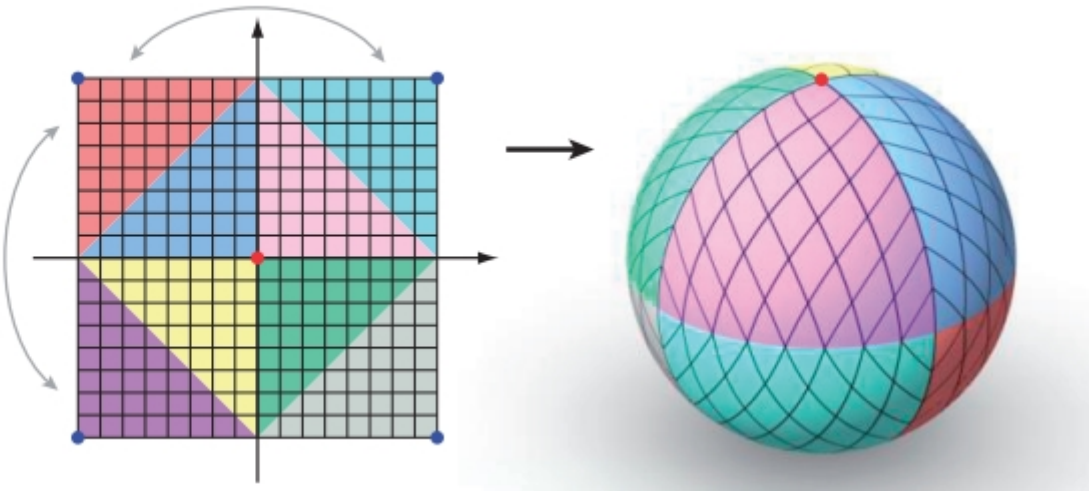

Figure 4.7: Instead of the original selected interpolated texels, the texels that arrows indicate are the correct ones Clarberg (2008)

Figure 4.8: One texel size padding for octahedral maps

Chapter 5

Evaluation

5.1 Experiments

5.1.1 Hardware Support

Ram: 32.0GB
CPU: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
GPU: NVIDIA GeForce RTX 2070

5.1.2 Experiment Property Settings

Original cube map size:
Original irradaince probe: 128×128 (Octahedral)
Light field probe: 128×128 (Octahedral)

Number of probes in the scenes:
16 probes for each scene
2 Different scenes. One with a complete wall that can fully block the light, and another one will incomplete wall that light is allow to leak through one side pass. The second one is for proving that the remaining light energy after several reflection is enough for travelling such a long distance, so the complete dark effect is actually because of the visibility test.

5.2 Results

5.2.1 Rendering Effect

This part is tested with Scene 1.

Original irradiance probe method based global illumination(Figure 5.1):

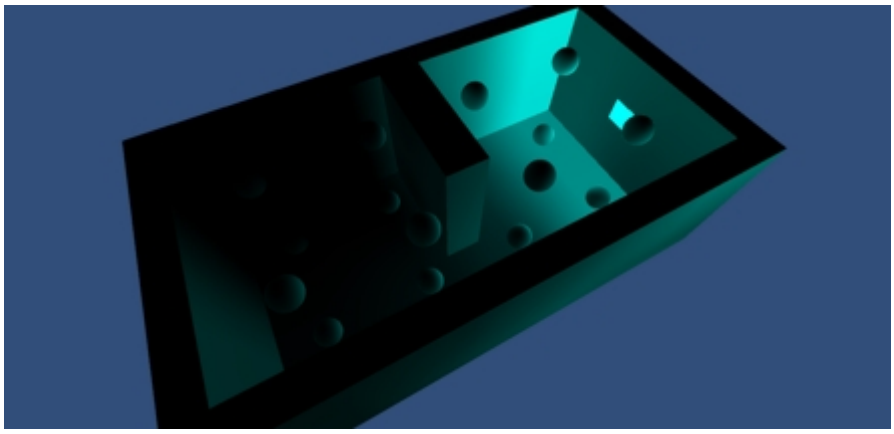

Figure 5.1: Global illumination based on original irradiance probe method (Scene1)

Light field probe method based global illumination(Figure 5.2):

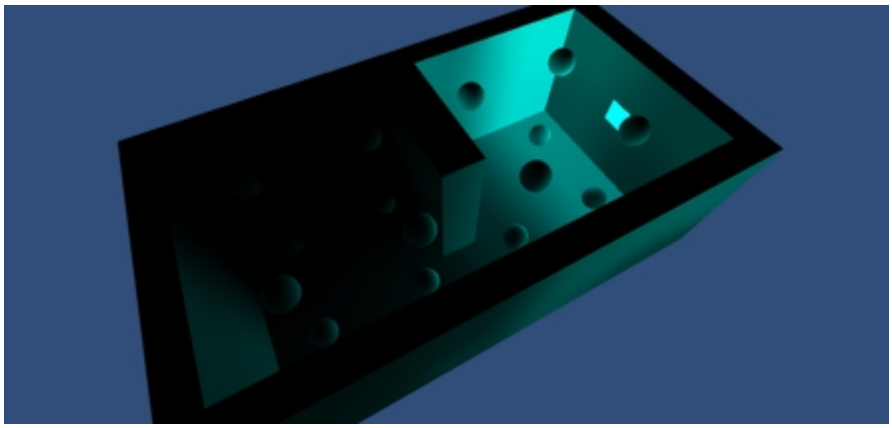

Figure 5.2: Global illumination based on light field probe method (Scene1)

This part is tested with Scene 2.

Original irradiance probe method based global illumination(Figure 5.3):

Light field probe method based global illumination(Figure 5.4):

Note that the surface of the outgoing direction of the tested box scene (Figure 5.5) is lit by the light source. This is caused by the trilinear interpolation, because there is no

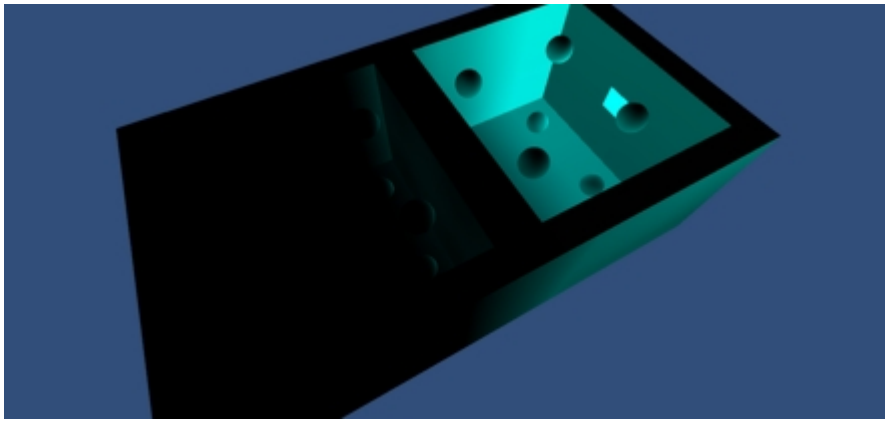

Figure 5.3: Global illumination based on original irradiance probe method (Scene2)

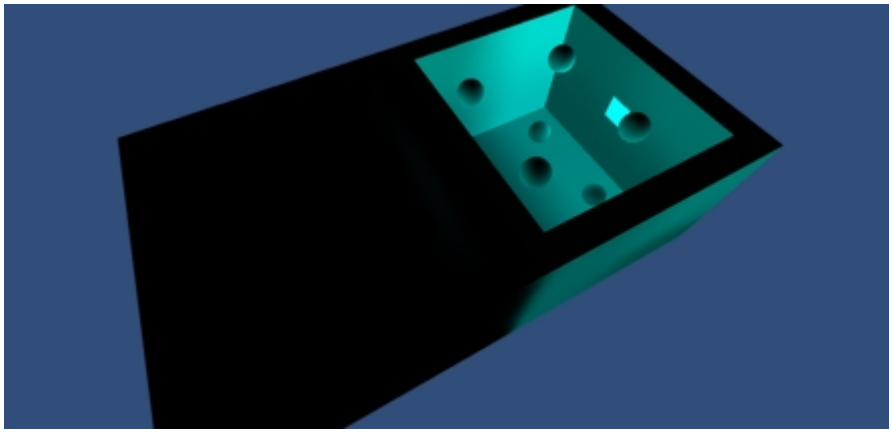

Figure 5.4: Global illumination based on light field probe method (Scene2)

probe placed outside the box. Since usually probes are placed in visible area, this artifact will not be noticed.

5.2.2 Performance

Time consumed during run time:
Original irradiance probe method based global illumination(Figure 5.6):
Light field probe method based global illumination(Figure 5.7): //Memory consumed by each probe:
Original irradiance probe method based global illumination(Figure 5.8):
Light field probe method based global illumination(Figure 5.9):

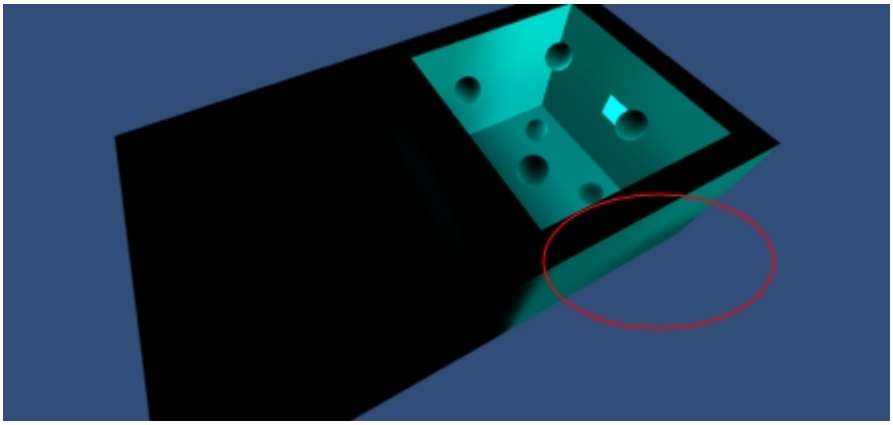

Figure 5.5: The outgoing direction surface is lit by trilinear interpolation

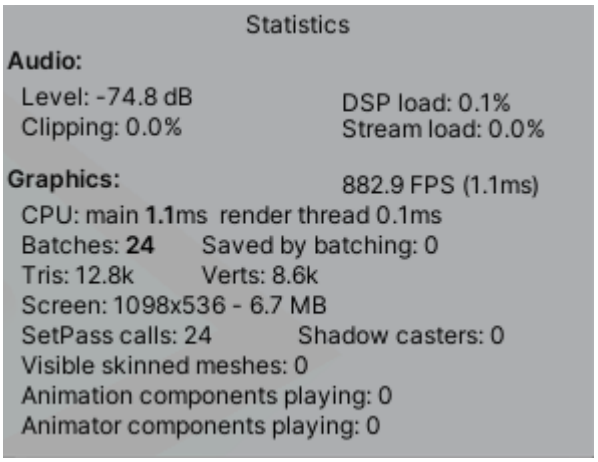

Figure 5.6: Time consumed by global illumination based on original irradiance probe method (run time)

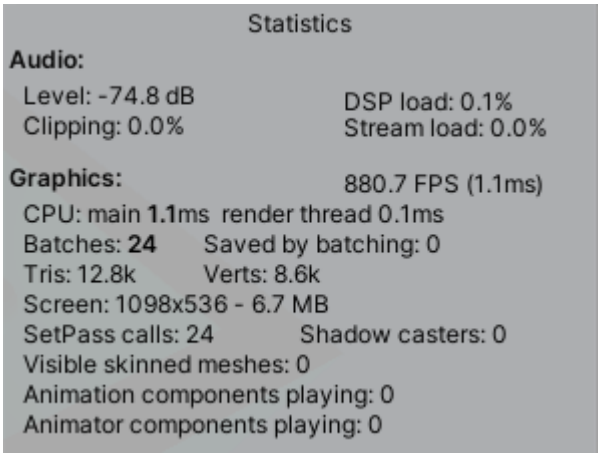

Figure 5.7: Time Consumed by global illumination based on light field probe method (run time)

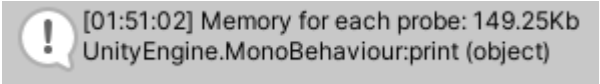

Figure 5.8: Memory consumed by global illumination based on original irradiance probe method (run time by each probe)

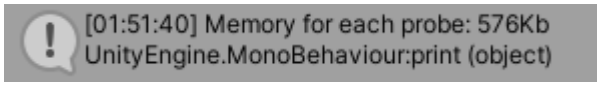

Figure 5.9: Memory Consumed by global illumination based on light field probe method (run time by each probe)

Chapter 6

Conclusions & Future Work

6.1 Conclusions

From the experiment result (Table 6.1), we can know that the precomputed light field probe method can properly handle the light leaking problem, however it have to consume more memory for storing some additional geometric information. Since most of the computation is done before the rendering loop, the time consumed at run time is very similarly short as expected.

Method	**Visibility**	**Time Consumed**	**Memory Consumed(Each Probe)**
Irradiance	False	1.1ms	149.25Kb
Light Field	True	1.1ms	576Kb

Table 6.1: Result compared between two methods

6.2 Future Work

The iterative ray tracing idea has provide a very nice fundamental idea for updating the probe in real-time. In iterative ray tracing technique, I can store the first bounce result in a frame buffer for later bounces to sample and ideally it can reach infinite number of bounces. Suppose that I can actually compute one bounce for each frame instead of finishing the computation just in one frame. Every time I compute the one bounce indirect result and accumulate to a frame buffer, it can be sampled by the next frame. So the computation is spread out to different frame time. The time consumed by ray tracing is greatly decreased. This would allow us to have a dynamic global illumination solution with the probes.

One more problem for updating the probes in real-time is that there is a very large number

of probes in the most scene. Updating each probe is not possible in real-time. However, we can combine LOD(Level of Details) techniques with this by setting up state machines for the probes. If the probes are far away, we do not update the probes since the indirect illumination is usually less noticeable, especially in far distance since the light energy is very low after long travelling distance. This will greatly decrease the number of probes that needs to be updated in each frame.

Bibliography

Cigolle, Z. H., Donow, S., Evangelakos, D., Mara, M., McGuire, M., and Meyer, Q. (2014). A survey of efficient representations for independent unit vectors. *Journal of Computer Graphics Techniques*, 3(2).

Clarberg, P. (2008). Fast equal-area mapping of the (hemi) sphere using simd. *Journal of Graphics Tools*, 13(3):53–68.

Colbert, M. and Krivanek, J. (2007). Gpu-based importance sampling. *GPU Gems*, 3:459–476.

Donnelly, W. and Lauritzen, A. (2006). Variance shadow maps. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 161–165.

Howes, L. and Thomas, D. (2007). Efficient random number generation and application using cuda. *GPU gems*, 3:805–830.

Karis, B. and Games, E. (2013). Real shading in unreal engine 4. *Proc. Physically Based Shading Theory Practice*, 4(3):1.

Majercik, Z., Guertin, J.-P., Nowrouzezahrai, D., and McGuire, M. (2019). Dynamic diffuse global illumination with ray-traced irradiance fields. *Journal of Computer Graphics Techniques Vol*, 8(2).

Martin, S. (2010). A real-time radiosity architecture for video games. *SIGGRAPH course on" Advanced in Real-Time Rendering in 3 D Graphcs and Games", 2010*.

McGuire, M., Mara, M., Nowrouzezahrai, D., and Luebke, D. (2017). Real-time global illumination using precomputed light field probes. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 1–11.

Ritschel, T., Grosch, T., and Seidel, H.-P. (2009). Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 75–82.

Sébastien, L. and Zanuttini, A. (2012). Local image-based lighting with parallax-corrected cubemaps. In *ACM SIGGRAPH 2012 Talks*, pages 1–1.

Tatarchuk, N. (2005). Irradiance volumes for games. In *Game Developer's Conference*, volume 3.

Ward, G. J. and Heckbert, P. S. (1992). Irradiance gradients. Technical report, Lawrence Berkeley Lab., CA (United States); Ecole Polytechnique Federale ....