Automatic Placement of Radiance Probes

Yue WANG,
School of Computer Science,
McGill University

December 19, 2018

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Master of Science

under the supervision of
Dr. Derek Nowrouzezahrai,
Dr. Paul G. Kry,
Soufiane Khiat

MCGILL UNIVERSITY

Abstract

School of Computer Science

Master of Science

Automatic Placement of Radiance Probes

by Yue WANG

Radiance probes are one of the most widely used techniques in real-time rendering to achieve realistic shading effects. This algorithm caches the radiance and geometry information of 3D scenes into probes, which are used to interpolate or trace rays at run-time. Probes are commonly placed in 3D regular grids to enable trilinear interpolation and efficient indexing. However, uniform density of probes usually leads to various problems such as oversampling and undersampling.

We introduce algorithms that automatically generate non-uniform radiance probe positions from 3D scenes, and provide an efficient structure for organizing non-uniform probes for run time tracing algorithms using probe textures.

We generate probe candidates from skeleton information of 3D scenes, which can be built either from medial axes or signed distance functions. This structure, alongside an optional optimization step using gradient descent, allows us to achieve full spatial coverage and reduce grazing angles between probes and surfaces.

At run-time, we adapt a standard probe ray tracer to non-uniform probe placement by caching probe and visibility information into a sparse voxel octree. To further accelerate our ray tracer, we augment the distance probes with data needed to apply hi-Z ray tracing to hierarchically trace rays in distance probes.

Our benchmarks show that our algorithm achieves better performance with fewer probes when rendering at fixed visual quality.

Traduction en français

Positionnement automatique des sondes Radiance

par Yue WANG

Les sondes de radiance sont l'une des techniques de rendu en temps réel les plus utilisées pour obtenir un rendu réaliste. Cet algorithme permet de stocker les informations de radiance et de géométrie d'une scène 3D dans des sondes, qui sont utilisées pour interpoler ou lancer des rayons en temps réel. Les sondes sont généralement placées dans une grille régulière 3D pour permettre une interpolation trilinéaire et une indexation efficace. Cependant, la densité uniforme des sondes pose généralement divers problèmes, tels que le sur-échantillonnage et le sous-échantillonnage.

Nous introduisons des algorithmes pour générer automatiquement les positions non uniformes des sondes de radiance à partir d'une scène 3D. Nous fournissons une structure de données efficace pour organiser des sondes non uniformes pour des algorithmes de lancer de rayons en temps réel utilisant des textures de sondes.

A partir d'informations du squelette de la scène 3D, nous générons des sondes candidates pouvant être construites à partir d'axes médians ou de fonctions de distance signée. Cette structure, associée à une étape optionnelle d'optimisation utilisant la descente de gradient, permet d'obtenir une couverture spatiale complète et de réduire les angles rasants entre les sondes et les surfaces.

A l'exécution, nous adaptons un traceur de rayons de sondes standard à un placement de non uniforme sondes en mettant en cache les informations de sondes et de visibilité dans un octree de voxels clairsemé. Afin d'accélérer notre lancer de rayons, nous ajoutons des données dont nous avons besoin dans le but d'appliquer le lancer dans un tampon-Z hiérarchique selon les distances avec les sondes.

Nos mesures montrent que notre algorithme réalise de meilleures performances avec moins de sondes lors du rendu pour une qualité visuelle fixe.

Acknowledgements

I would like to thank my co-supervisor Professor Derek Nowrouzezahrai for taking me as his student into the rendering field. I am grateful for the consistent encouragement, patience and support he has shown ever since the first rendering class I attended, when I did not have much knowledge about rendering, and found myself stuck. I would also like to thank my co-supervisor Professor Paul G. Kry for taking me as his student to be part of his lab, and I'm grateful for his support and patience, and hands-on teaching in the lab and discussion. Last but not least, I am grateful for the opportunity to join Ubisoft La Forge, which is the best internship I have ever had.

My special thanks to Soufiane Khiat, who gave me constant and in-time support and encouragement during both my internships at Ubisoft as my manager, even when he was sick or on vacation. I will always appreciate those late night debugging sessions, hand-drawing algorithm illustrations, career suggestions, and introduction to great people in the graphics field; I will always be grateful for his constant encouragement when I lost faith or hit roadblocks. His passion, knowledge, attitude and experience towards rendering and other fields were a strong inspiration for me to keep moving forward. I would like to thank my former-manager Olivier Pomarez for the support, guidance, and valuable suggestions during my internship, which helped me gain confidence at work and get involved as a member of La Forge family. My thanks to Ulrich Haar and Stephen McAuley for constant support and valuable insights and feedback from industry, and thanks to other colleagues from Ubisoft: Jendrik Illner, Daniel Bairamian, Kim Goossens, John Huelin and Gauthier Viau for helping me debug and design algorithms.

Last but not least: my thanks to Michal Drobot and Cyril Crassin for their image reference permissions, and Hugo Lamarre for providing the model of iconic temple Ubisoft© from Assassin's Creed® Odyssey. My friends Keven Villeneuve for the French abstract translation and review, Aurora Huang and Xiaozhong Chen for their support in both my thesis and my life; I am very grateful to my parents for their support in my years of exploring what exactly I would like to do with my life, and I'm glad that I found it.

Declaration of Authorship

I, Yue WANG, declare that this thesis titled, "Automatic Placement of Radiance Probes" and the work presented in it are my own. I confirm being the sole contributor for all chapters from Chapters 1 to 6.

Contents

Abstract iii

Traduction en français iv

Acknowledgements v

Declaration of Authorship vii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Goals . . . 2
  - 1.2.1 Automatic Probe Placement . . . 2
  - 1.2.2 Adapting Ray-tracing to Non-Uniform Probe Placements . . . 3
- 1.3 Overview . . . 3

2 Related Work 5
- 2.1 Light Probe Algorithms . . . 5
- 2.2 Non-Uniform Placement and Probe Organization . . . 6
  - 2.2.1 Radiance Probes . . . 6
  - 2.2.2 Irradiance Probes . . . 7
    - Placement on demand . . . 7
    - Spawn, Merge and Remove . . . 7
- 2.3 Conclusion of Literature Review . . . 8

3 Automatic Probe Placement 9
- 3.1 Straight Skeletons and Medial Axes . . . 9
  - 3.1.1 Skeleton from Signed Distance Function . . . 10
  - 3.1.2 Ball Shrinking Medial Axis . . . 12
- 3.2 Position Optimization . . . 13
  - 3.2.1 Optimization Metrics . . . 13
    - Visibility Atlas . . . 13
    - View Direction . . . 15
    - Distance to Surface . . . 15

3.2.2 Gradient Descent . . . 16
3.2.3 Gibbs Sampling . . . 17
3.2.4 Importance Sampling . . . 19

4 Shading with Non-uniform Probes **21**
4.1 Voronoi-based Selection . . . 21
4.1.1 First Probe Texture with Linked List . . . 21
4.1.2 Baking Neighbor Probe Indices into Probes . . . 24
4.1.3 Run-time: Texture Lookup . . . 25
4.2 Baking Probes to Sparse Voxel Tree . . . 26
4.2.1 Baking Probes Based on Visibility . . . 26
4.2.2 Implementation Details . . . 28
Baking Probe Octree: Probe Sorting and Multi-sampling . . . 29
run-time: Traverse Down SVO . . . 30
4.3 Hi-Z Ray Tracer with Probe Textures . . . 30
4.3.1 Hi-Z Ray Tracer . . . 31
4.3.2 Implementation Details . . . 32
Baking Hi-Z Probes . . . 32
Run-time . . . 32
4.3.3 Results and Discussion . . . 32

5 Results and Discussion **35**
5.1 Quality . . . 35
5.1.1 Same Quality, Different Probe Placements . . . 36
Cornell Box . . . 36
Iconic Temple . . . 36
Crytek Sponza . . . 36
5.1.2 Same Number of Probes . . . 39
Iconic Temple . . . 39
Crytek Sponza . . . 43
5.2 Performance . . . 43
Iconic Temple . . . 45
Crytek Sponza . . . 46
5.3 Memory Footprint . . . 46

6 Conclusion and Future Work **49**
6.1 Conclusion . . . 49
6.2 Future Work . . . 49
6.2.1 Probe Placement and Optimization . . . 49

Analytic Visibility 49
Visibility Atlas 50
6.2.2 Run-time Rendering Algorithm 50
Run-time Probe Selection 50
Cone Tracing 50

A Appendix **53**
A.1 Probe Placement Settings 53
A.1.1 Iconic Temple 53
A.1.2 Crytek Sponza 56
A.2 Ground truth 58
A.2.1 Cornell Box 58
A.2.2 Iconic Temple 59
A.2.3 Crytek Sponza 60
A.3 Comparison With Dense Grids 61
A.3.1 Iconic Temple 61
A.3.2 Crytek Sponza 62

List of Figures

1.1 Comparison of uniform and non-uniform probe placement in the iconic temple Ubisoft©. Given the same number of probes, non-uniform placement can better cover important regions in space, while the regular grid has four probes within objects on both sides of the stairs. Model courtesy of Ubisoft© by Hugo Lamarre. 3

3.1 An illustration of a straight skeleton (in blue on the image on the right) in 3D. Image courtesy of Barequet et al. [2008]. 9
3.2 Medial axis (a) and straight skeleton (b). Image courtesy of Cheng et al. [2014] 10
3.3 Skeleton from an SDF of the Cornell box. 11
3.4 Medial axis shrinking ball algorithm. 12
3.5 Medial axis of the Cornell box. 13
3.6 Different probe positions generated from medial axis. Probe positions shown in the top left image is generated from many small clusters. Probe positions shown in the bottom right image is from large clusters. . 14
3.7 After we mapped the visible surfaces into a texture atlas, all probes will have an evaluation of the number of visible surfaces at the same scale. . 15
3.8 Global visibility atlas. Probe 2 is the only probe that can "see" the surfaces in red square. 16
3.9 A demonstration of Gibbs sampling. 18
3.10 Stochastic direction sampling. Left to right: we construct a stochastic probe update dierction by sampling points $x_{\xi,i}$ ($i \in [1,4]$) proportional to a linear model of expected fitness along the x-adjacent support probe grid, then repeating for y-points $y_{\xi,j}$ ($j \in [1,2]$) and z-point $z_{\xi,1}$ 18

4.1 Step 1 of baking neighbor probes: closing Voronoi cells. 22
4.2 Visualization of Voronoi cells after step 2 of baking neighbor probes: shrinking Voronoi cells towards their centers. 22
4.3 An illustration of per-pixel linked list. 23

4.4 One frame of the first probe texture that encodes probe indices. Bright areas indicate large probe indices, dark areas indicate small probe indices. . . . 24
4.5 Delaunay graph and projection onto a unit sphere. . . . 24
4.6 An illustration of neighbor probe indices baked into probes of the Cornell box. . . . 25
4.7 Visualization of baked probe indices into sparse voxel octree for the Cornell box. . . . 26
4.8 Top view illustration of influence regions of probes in the iconic temple Ubisoft©. The top left pillar of the temple is covered by the orange probe (in the middle) and the white probe (top left). . . . 27
4.9 Octree building steps, image courtesy of Crassin and Green [2012]. . . . 28
4.10 Visibility test of probe 1; red as visible, black as occluded. . . . 29
4.11 Hi-Z ray tracing process, image courtesy of Drobot [2018]. . . . 31
4.12 A ray traced in a probe texture of the Cornell box, using the original two-resolution ray marching and hi-Z ray marching. . . . 34

5.1 Comparison of non-uniform and regular grid in final rendering results (first row) and filtered glossy components (second row) of the Cornell box. . . . 37
5.2 Comparison of the same visual quality using non-uniform and regular grids placement in final rendering results (first row and second row) and filtered glossy components (third row) of iconic temple Ubisoft©. Note that when the MSE is similar, our non-uniform placement uses fewer probes to achieve the same quality. . . . 38
5.3 Top view illustration of one pillar gets covered by 3 probes in the iconic temple Ubisoft©. The upper left pillar is covered by probe in the center, and the probe coded in white upper left. . . . 39
5.4 Comparison of Crytek Sponza with the same visual rendering quality using non-uniform and regular grid placement, in filtered glossy components (second row) and error visualization (third row). Note that when the mean squared error is similar, our non-uniform placement uses fewer probes to achieve the same quality. . . . 40
5.5 Comparison of the iconic temple Ubisoft© with the same number of probes using non-uniform and regular grid placement, in filtered glossy component (the third row). . . . 41

5.6 Comparison of visual quality between the same number of probes using non-uniform and in regular grid placement, in filtered glossy component (second row) and error visualization (third row). Note that our non-uniform placement achieves lower mean squared error and fewer visual artifacts. 42
5.7 Comparison of coverage over the statue from top view of Crytek Sponza. 43
5.8 Comparison of visual quality of the iconic temple Ubisoft© between using non-uniform and regular grid placements, in filtered glossy component (second row). Note that to reach the same performance as a dense grid, we use unnecessarily dense sampling for non-uniform placement to achieve similar quality. 45
5.9 Comparison of visual quality between using non-uniform and in a regular grid placement with the same performance, in filtered glossy component (second row) and error visualization (third row). Note that our non-uniform placement generates higher quality results and fewer artifacts with similar performance. 47

6.1 Correct mipmap radiance probe texture. Two rays (in yellow arrows) traced in the probe texture of Crytek Sponza, and the hit texels lie on the edges of the octahedron. To correctly mipmap the texels on boundaries of an octahedral texture, we need to consider the pixels that are reflective symmetric with respect to the midline of the boundary, as shown in red and blue. 51

A.1 Medial axis of the iconic temple Ubisoft©. 53
A.2 Visualization of non-uniform 63 probes in the iconic temple Ubisoft© . 54
A.3 Visualization of non-uniform 48 probes in the iconic temple Ubisoft©. . 54
A.4 Visualization of non-uniform 43 probes in the iconic temple Ubisoft©. . 55
A.5 Visualization of non-uniform 36 probes in the iconic temple Ubisoft©. . 55
A.6 Medial axis of Crytek Sponza. 56
A.7 Visualization of non-uniform 87 probes in Sponza 56
A.8 Visualization of non-uniform 74 probes in Sponza. 57
A.9 Visualization of non-uniform 64 probes in Sponza. 57
A.10 Visualization of non-uniform 54 probes in Sponza. 58
A.11 Global illumination with one-bounce indirect light. 58
A.12 One-bounce indirect glossy component. 59
A.13 Global illumination with one-bounce indirect light. 59
A.14 One-bounce indirect glossy component. 60
A.15 Global illumination with one-bounce indirect light. 60

A.16 One-bounce indirect glossy component. 61
A.17 Visualization of the probe positions, non-uniform and dense grid in the iconic temple. 61
A.18 Comparison of rendering results using non-uniform placed probes and a dense grid in the iconic temple. 61
A.19 Comparison of the filtered glossy component using non-uniform placed probes and dense grid in the iconic temple. 62
A.20 Comparison of per-frame glossy component using non-uniform placed probes and dense grid in the iconic temple 62
A.21 Visualization of the probe positions, non-uniform and dense grid in Sponza. 62
A.22 Comparison of rendering result using non-uniform placed probes and dense grid in Sponza. 63
A.23 Comparison of the indirect component using non-uniform placed probes and dense grid in Sponza. 63
A.24 Comparison of filtered glossy component using non-uniform placed probes and dense grid in Sponza. 63
A.25 Comparison of per-frame glossy component using non-uniform placed probes and dense grid in Sponza. 64

List of Tables

5.1 Performance in Milliseconds of Different Placement. 44
5.2 Memory usage (in MB) with same quality. 46

Chapter 1

Introduction

Rendering realistic images is a longstanding problem in computer graphics. One of the key components for computing realistic images is the simulation of multiple bounces of light, also known as global illumination. With the rapid development of video game and virtual reality techniques, creating realistic images at interactive rates has become an important scenario for many applications. However, computing accurate indirect light can be expensive in terms of computation time and memory consumption.

In this thesis, we focus on one global illumination algorithm – light field probes, which encodes precomputed geometry and radiance information of a 3D scene into a compact representation (light field probe). At run-time, lighting is computed using an optimized ray marcher in texture space and Monte Carlo techniques. This thesis focuses on radiance probe placement, and exploiting precomputed information to improve run-time performance.

1.1 Motivation

Since light probes were first introduced into the video game industry by Tatarchuk [2005], the most commonly used method to place probes is regular grids. While regular grids are straightforward and convenient to compute or interpolate at run-time, there are two main issues with regular grid placement, which are also our main motivations for working on this problem.

One problem with regular grids is that they cannot account for the fact that areas with complex geometries or lighting changes require higher density sampling, while areas with simple geometries can be fully covered by fewer probes. Therefore, most algorithms using regular grids face a dilemma of dense grids versus sparse grids: dense grids can capture all the details but usually suffer from information redundancy, and it is hard to keep the memory footprint within budget; while sparse grids usually result in missing information, which leads to artifacts at rendering time. Most algorithms end up settling for a trade-off, which results in undersampling at places with high

variance in lighting or geometry, and oversampling in places with low variance in lighting and geometry.

From the first motivation, one might argue that sparse regular grids can also fully capture the information of the radiance and geometry, if placed carefully. This leads to our second motivation: video games make use of large, open scenes with houses and buildings of different shapes. It is very challenging to ensure that every probe is in a reasonably good place using a regular grid in large scenes with complex geometry. Since the grid is defined by its grid cell size and grid dimensions, it is impossible to adjust one particular probe's position.

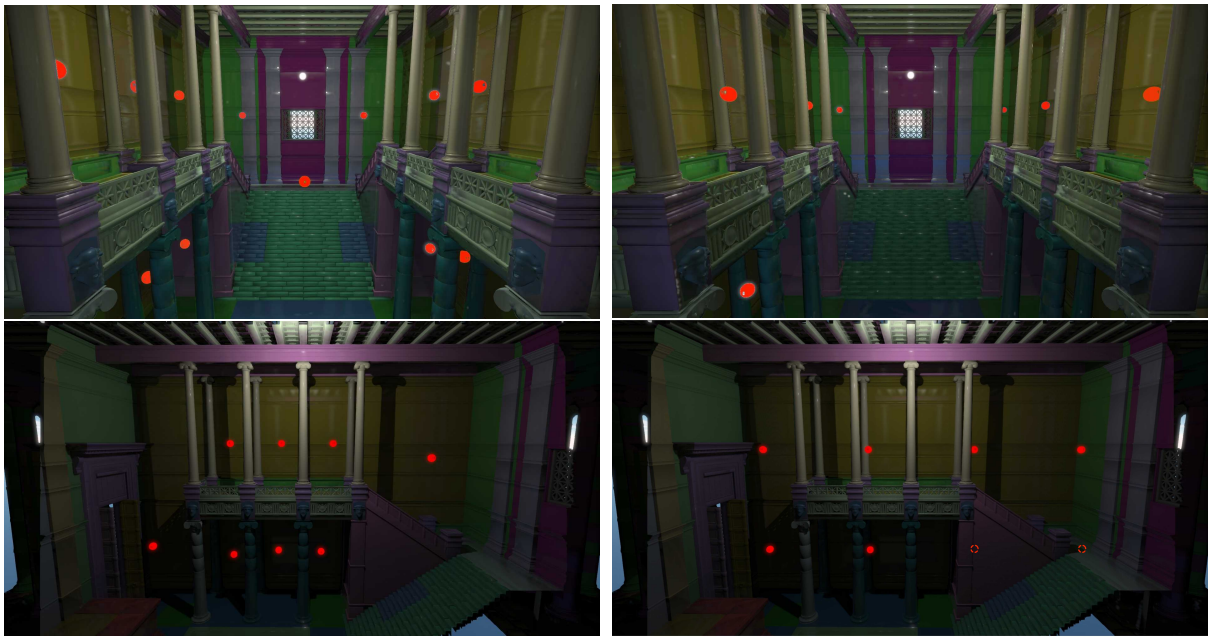
Another motivation concerns the ray tracing algorithm. Besides the full coverage of probes to the geometry, we also want to make sure every surface has at least one probe that with a good view towards it – each surface should subtend a large projected area about (at least) one probe. Poorly placed probes lead to artifacts during ray tracing due to a lack of geometry information. If we would like to hit a certain point on a wall, but only one probe "sees" this wall and worst at a grazing angle, it is likely that the ray tracer will miss the surface. To resolve this undersampling, local cubemaps are usually placed manually by artists. To do so an artist needs to not only determine probe positions, but also manually set their influence volumes and projection planes as introduced by Sébastien and Zanuttini [2012] in order to have the local cubemaps function correctly. However, as the world map in video games grows, it becomes prohibitively time consuming and tedious to do this manually.

1.2 Goals

Our goal is to explore strategies to automatically generate probe positions for any given scene, such that this set of probe positions is optimal in terms of projected scene visibility, memory footprint, and final rendering quality. With a suitable non-uniform probe placement strategy, we would like to adapt the run-time rendering algorithm to rely on fewer probes while still generating high-quality results.

1.2.1 Automatic Probe Placement

For any given scene, we need to define metrics such as visibility to evaluate the quality of a current proposal of probe placement. With these metrics, we can treat probe placement as a metric optimization problem.



(A) Non-uniformly placed 16 probes. (B) Regular grid $2 \times 2 \times 4 = 16$ probes.

FIGURE 1.1: Comparison of uniform and non-uniform probe placement in the iconic temple Ubisoft©. Given the same number of probes, non-uniform placement can better cover important regions in space, while the regular grid has four probes within objects on both sides of the stairs. Model courtesy of Ubisoft© by Hugo Lamarre.

1.2.2 Adapting Ray-tracing to Non-Uniform Probe Placements

Since we are placing probes non-uniformly, the indexing convenience of regular grids is no longer available, such as trilinear interpolation and easy probe position queries. We need a new algorithm that handles non-uniformly placed probes: for any given surface, we need to efficiently find the nearest probe, and next best candidate probes if a ray cannot find a hit point using the current probe.

Due to a scene's arbitrary complexity and scale, our preliminary results show that it is difficult to efficiently find a best candidate probe for an arbitrary surface without any acceleration structure. Therefore we explore different representations of spatial data structures on GPU, such that we can efficiently and accurately query correct probes for any given ray at run-time.

1.3 Overview

Our work is divided into two parts: optimal probe placement, and run-time algorithm improvements. We will introduce related work of the two problems in Chapter 2, and

then detail our solutions in Chapters 3 and 4. In Chapter 3, we will detail our position optimization metrics, and strategies for automatic optimal placement. In Chapter 4, we will focus on how to adapt a non-uniform structure into current rendering algorithms, and introduce techniques to accelerate ray tracing. In Chapter 5, we will present our results, including comparisons of performance and rendering quality with the original algorithm of using regular grids, and a discussion of advantages and disadvantages of our implementation in the context of prior art. In the last chapter, we conclude and discuss open challenges.

Chapter 2

Related Work

In this chapter, we first discuss algorithms that use light probes to compute global illumination, then give an overview of strategies that generate non-uniform probes and how they organize the probes for run time use.

2.1 Light Probe Algorithms

With the development of the capability and computation power of GPU, light probes have become a standard method for approximating global illumination in real time in common game engines like Unreal [Burger, 2013] and Unity[Mortensen, 2014].

Greger et al. [1998] introduce irradiance volumes that cache irradiance information from 3D scenes, and reconstruct the distribution of irradiance for run-time interpolation. However, self-occlusion and dynamic objects cause inaccurate results, and the 2D array representation of probes is expensive. More efficient representations such as spherical harmonics (SH) [Ramamoorthi and Hanrahan, 2001] and precomputed radiance transfer (PRT) [Sloan et al., 2002] achieve better quality low-frequency lighting with similar storage and performance requirements. Jendersie et al. [2016] construct probes using a hierarchy of surfels sampled from direct light. At each frame, surfels are first lit by direct light, then a hierarchy of surfels are built to propagate the indirect light to senders. The light probes are computed form the surfels and precomputed light transport factor between surfels and light probes. Then indirect light are computed by interpolating between light probes. However, it takes extra memory to store the form factors and link factors for surfels and probes.

Silvennoinen and Lehtinen [2017] factorize the transport matrix into global and local transport matrices, and sample non-uniform sparse radiance probes to reconstruct indirect light.

Our work is mainly based on the work of McGuire et al. [2017], which combines radiance and irradiance probes with visibility information. This algorithm is based on a ray tracer that uses the depth information baked into probe textures. At run time, lighting could be either evaluated by tracing rays using Monte Carlo techniques,

or interpolated between irradiance probes with weighted visibility to have smooth results without light leaking. However, sparse sampling of probes usually leads to artifacts due to lack of accurate visibility information for either ray tracing in probes or trilinear interpolation with weights. Our work generates non-uniform probe positions that covers all important geometry in the scene, which is more ray-tracer friendly and generates more accurate results in surfaces with glossy components.

2.2 Non-Uniform Placement and Probe Organization

Many methods have explored how to automatically place probes in a more adaptive way. Due to the fact that different algorithms for using probes are used at run time, the strategies for placement are also different. We will introduce the non-uniform placement algorithms for radiance probes and irradiance probes separately.

2.2.1 Radiance Probes

Non-uniform placement of radiance probes usually involves visibility and light variations. Chajdas et al. [2011] place environment maps based on the variation of intensity, hue or direction of incoming light in 3D space. Probes are sampled along a dense grid, and are pruned later based on similarities and irradiance gradient field. This algorithm provides sparse and irradiance-aware results. However, it does not ensure full coverage, and the irradiance gradient field tends to drive all probes towards light sources. To maximize the coverage of radiance probes, Silvennoinen and Timonen [2015] evaluates the visible surface area and the distance to surfaces from each probe position in 3D regular grids, then selects the best K probes based on the two metrics. The probes are then baked into an octree for run-time interpolation. However, it is difficult to define the visible surface area for indoor scenes. We extend this idea in our algorithm and use a visibility atlas to measure visible surfaces of each probe. Silvennoinen and Lehtinen [2017] use a greedy approach to find probe locations. They first voxelize the scene and flood-fill the empty interior of the scene's geometry, then place a probe in each empty voxel with non-empty neighbors. This set of probes is further merged into a sparser set, based on visibility and influence radii. The probe generated are usually close to the geometry, which is undesirable for ray tracing in our case. Probes that are too close to the geometry would cause grazing angles from probe to surfaces, leading to artifacts at run-time ray tracing.

2.2.2 Irradiance Probes

The general ideas of sampling irradiance probes can be categorized into two different types: One is to traverse the whole scene and place probes if certain conditions are satisfied. Another starts from (dense or sparse) regular grids, spawn, remove, or merge probes from grid points based on certain metrics, which usually requires multiple pass or iterations.

Placement on demand

Greger et al. [1998] use a bi-level regular grid where irradiance probes are first placed in a coarse grid to acquire initial samples, that are then subdivided to a finer level if a voxel contains any geometry. However, this method suffers from the same visibility issue as with a regular grid. Probes placed within objects will be black, which leads to darker results if these probes are used for interpolation at run-time. The visibility issue is resolved by casting vertical rays every several meters (relative to the scene scale) along the ground plane, and probes are only placed above the first hit positions [Gilabert and Stefanov, 2012] [Stefanov, 2016]. To organize the probes, a hierarchical 3D grid is created to store the indices of closest probes. However, this method does not ensure full coverage, and casting rays in 2D grids still suffers from oversampling and undersampling. Our algorithm resolves visibility problems and achieves adaptive sampling by only placing probes on the skeleton of the interior of scenes. Cupisz [2012] uses Delaunay tetrahedralization to organize and interpolate non-uniform irradiance probes. We tested this method in our implementation, directly using barycentric coordinates in tetrahedra to locate nearest probes is expensive in our case. Therefore we extend this method to use Voronoi cells and baked neighbor information into probes.

Spawn, Merge and Remove

This type of algorithm usually exploits geometry and radiance information to guide the probe operations such as spawn, merge and remove. Tatarchuk [2005] introduces adaptive sampling of irradiance volumes at GDC 2005, which starts from a dense grid sampling, then prunes redundant samples while introducing new samples based on the gradient of irradiance. Bowald [2016] uses similar ideas but introduces a bit mask for each probe to indicate whether this probe is occluded by surfaces to resolve visibility issues.

2.3 Conclusion of Literature Review

The probe placement strategies are mostly designed to meet the requirements of run-time algorithms, and the run-time algorithms are usually customized to fit the structure of probes. Due to different run time usages, the information baked in probes is different, therefore the placement strategies are different.

The main difference between the previous algorithms and our work is that we have not only radiance in probes, but also normals and radial distances, which gives us more insights about the geometry, and enables the algorithm to trace rays in probes at run-time. For this reason, we require full visibility of geometry, but also from good views (i.e., no grazing angles for any surface from a probe's point of view).

In the following chapters, we will introduce how we solve this problem.

Chapter 3

Automatic Probe Placement

In this chapter, we introduce how we place probes non-uniformly. Our algorithm has two steps. In the first step, we generate the skeleton from a given scene, and generate candidate positions from the skeleton. In the second step, we design metrics and further optimize the set of positions based on these metrics. We will introduce the two steps in the following sections.

3.1 Straight Skeletons and Medial Axes

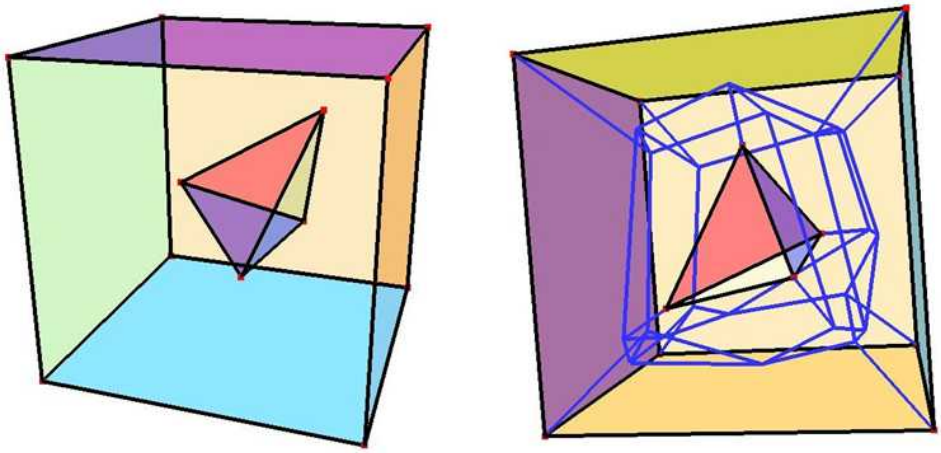


FIGURE 3.1: An illustration of a straight skeleton (in blue on the image on the right) in 3D. Image courtesy of Barequet et al. [2008].

A straight skeleton is a topological skeleton, which can be generated by a progressive shrinking process of all edges (or faces for 3D) at the same speed, until zero volume is reached. The elements of a straight skeleton can be points and lines for 2D, and can be points, lines, and planes for 3D, as shown in Figure 3.1. There are many applications of a straight skeleton. For example, the straight skeleton is used to guide vertex placement in graph drawing algorithms [Bagheri and Razzazi, 2012] , and to model roofs in architectures [Kelly et al., 2017].

One useful property of the skeleton that we use here is that each element of the straight skeleton defines a unique region that has the same visibility of the surfaces. Therefore the 3D straight skeleton is a good reference for probe placement in terms of visibility. Each element of a straight skeleton can be considered as one or more probe positions, depending on probe configurations such as resolution and far plane. By placing at least one probe on each element of straight skeleton, we achieve full coverage with minimal number of probes.

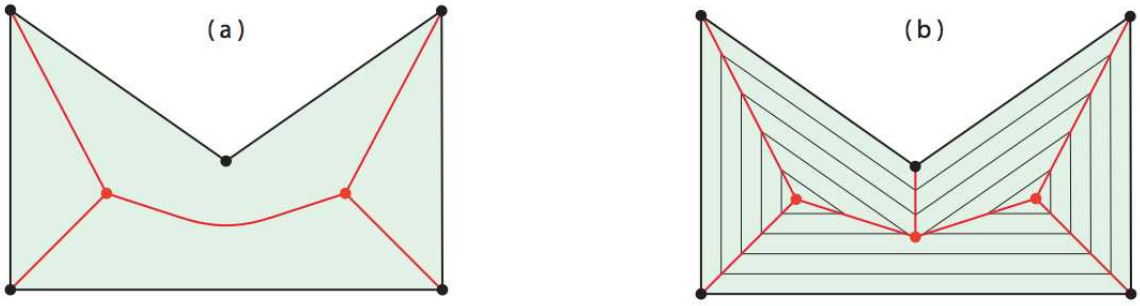


FIGURE 3.2: Medial axis (a) and straight skeleton (b). Image courtesy of Cheng et al. [2014]

However, straight skeletons in 3D are very complex to compute in comparison to other kinds of skeletons. Fortunately there are many other well known skeleton algorithms that also meet our requirements, medial axis is one of them. We experimented with two approximate algorithms for skeletons: signed distance function and medial axes. While they are not as accurate as the straight skeleton, they can still provide a good estimation of the internal framework of 3D space and fit our needs for spatial description of geometry, as shown in Figure 3.2.

3.1.1 Skeleton from Signed Distance Function

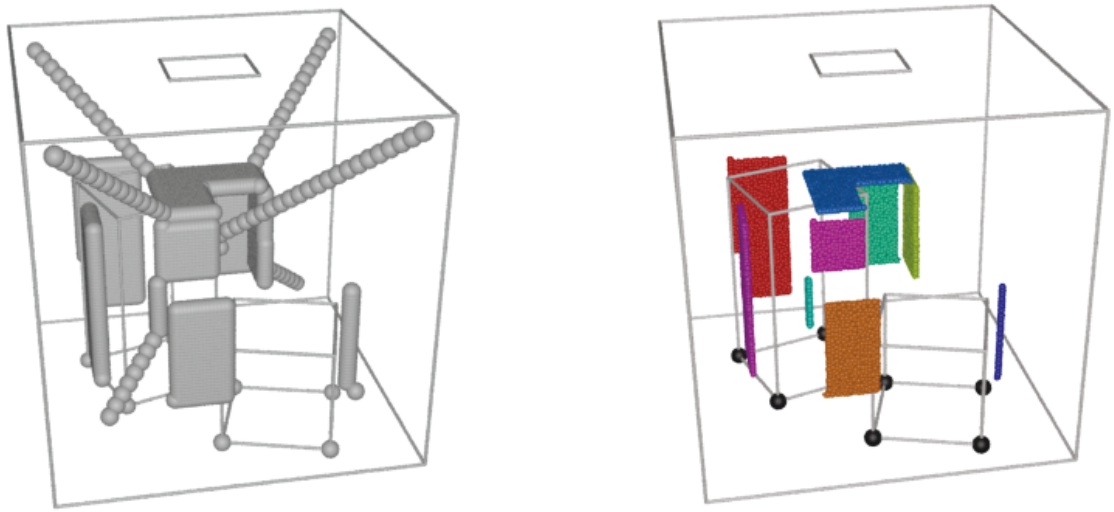
The signed distance function (SDF) is a function that describes the shape of a geometry, which takes as input a coordinate, and returns the shortest distance from this coordinate to the nearest point of this geometry. This distance is defined to be a positive value if the input coordinate is outside the volume defined by the geometry, and negative otherwise.

One way to generate the signed distance function of a given mesh is by first voxelizing the mesh, then assigning to each voxel a distance value based on its position and its neighbors, repeat this assignment process progressively inward or outward from the geometry until all voxels are assigned with distance values.

Algorithm 1 generates candidate positions from an SDF. After the signed distance function is built, we take only the interior of the original mesh, i.e., with negative

Algorithm 1 Probe Positions from SDF

```
skeleton ← SkeletonFromRemoveVoxels(threshold);   ▷ shown in Figure 3.3a
clusters ← ClusterSkeletonVoxels(skeleton);       ▷ shown in Figure 3.3b
for each cluster in clusters do
    newPosition ← averagePosition(cluster);
    candiatePositions.append(newPosition);
end for
```



(A) Remove voxels with SDF values smaller than a threshold.

(B) Cluster on elements of the skeleton.

FIGURE 3.3: Skeleton from an SDF of the Cornell box.

signed distance function values, and remove those voxels that are close to the geometry, i.e., voxels with SDF value lower than a threshold. What is left are voxels with distance greater than the threshold from surrounding geometry, which becomes the skeleton for probe placement. Figure 3.3a shows the skeleton from the SDF of the Cornell box, here we visualize each voxel using a sphere. We remove the clusters with voxels less than a threshold (such as diagonal elements in as shown in Figure 3.3a), then run a cluster on the left voxels, as shown in Figure 3.3b. Based on the resolution of voxelization and the threshold for trimming voxels with lower distance values, cluster results differ from sizes and positions. We take the average position of each cluster as a probe candidate.

We prototype our signed distance function in Houdini using OpenVDB. Due to the definition of interior, a signed distance function does not work with double sided meshes. For example, the temple mesh in Figure 1.1 has geometry for a wall that uses two surfaces with normals pointing outwards on both sides. The interior is defined to be the volume of the wall, which is not what we want. For this kind of mesh, SDF will consider the space inside the geometry to be interior. Therefore we take the bounding

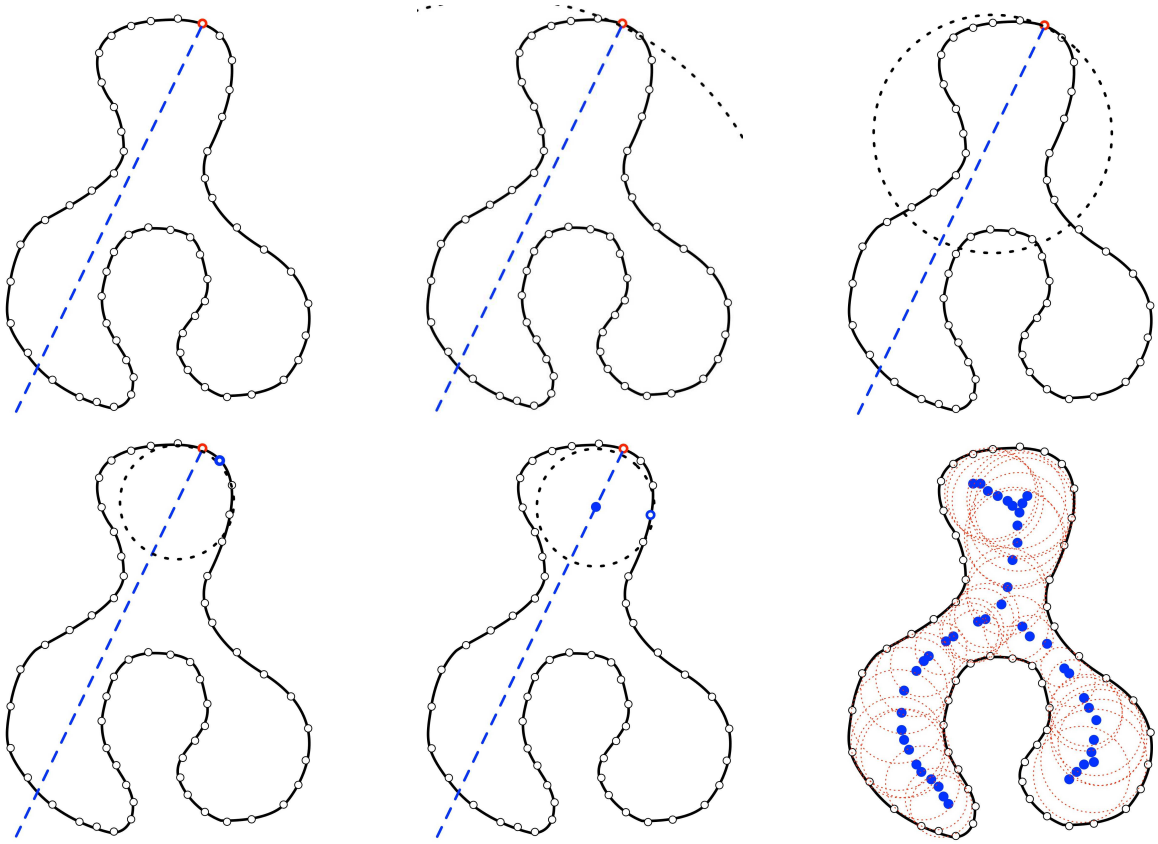


FIGURE 3.4: Medial axis shrinking ball algorithm.

box of the mesh, construct the SDF, and perform a signed distance difference with the SDF of the geometry. In this way we get the interior of the room if the mesh is double sided.

3.1.2 Ball Shrinking Medial Axis

The ball shrinking algorithm for medial axis computation was introduced by Ma et al. [2012]. As shown in Figure 3.4, given a point cloud, this algorithm operates on each point p: a ball $B(r_0)$ centered at $-r\vec{n}_p + p$ with radius r is built. This ball is tangent to p, and r is a large user defined number. The skeleton point is constructed by shrinking the radius r by intersecting the closest points until the ball is maximally inscribed. The center of the ball is taken as a skeleton point.

Because the algorithm takes a point cloud as input, we start with the vertices of the mesh as input to generate the medial axis of the mesh. As the point cloud should describe the surface, an extra tessellation step must be done before running the ball shrinking algorithm. After the skeleton is obtained, as shown in Figure 3.5, we form clusters based on normals and curvature similarities of the point cloud. With different parameters we can change the density of the resulting probe positions (shown in

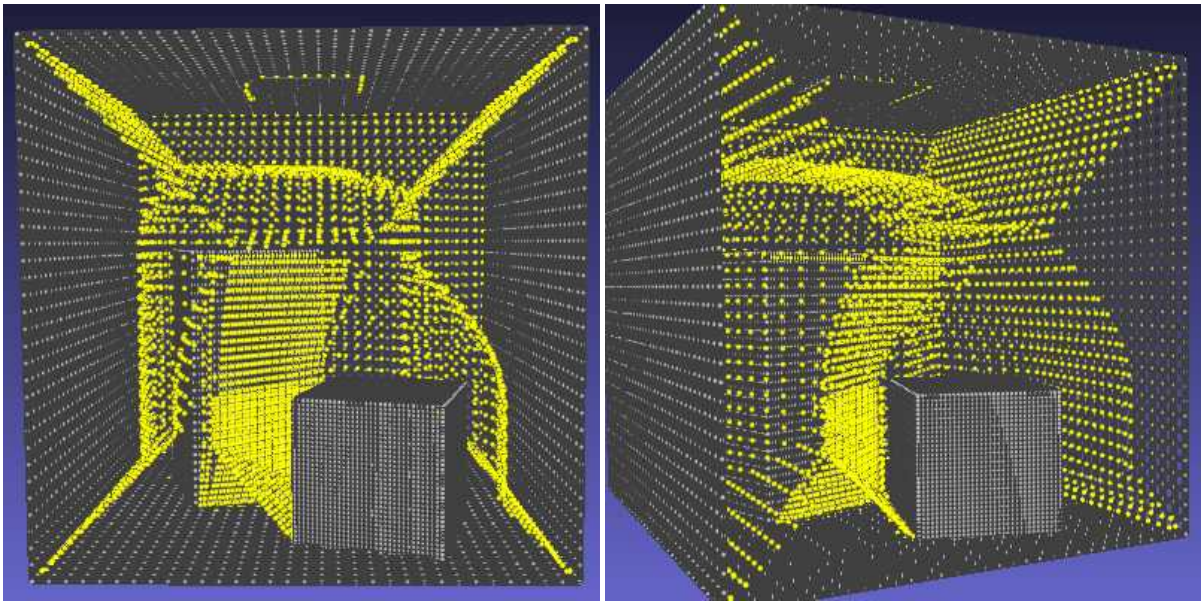


FIGURE 3.5: Medial axis of the Cornell box.

Figure 3.6).

3.2 Position Optimization

After placing probes in the skeleton, the set of probes is usually good enough for runtime use. However, we perform some post-processing to further refine the probe positions.

3.2.1 Optimization Metrics

We define several optimization metrics to further improve the quality of the probe settings that we obtained from Section 3.1.

Visibility Atlas

We define a global objective function to measure how many surfaces are covered for a given set of probe positions. To do this we map the visible surfaces of each probe from an octahedral map to the texture atlas. In this way, we can evaluate the visible area of each probes on the same scale. As shown in Figure 3.7, we first unwrap the geometry of the scene (Cornell box, shown in Figure 3.9a). Then for each probe shown in Figure 3.9b, we map the visible surfaces of the probe into the atlas, the result being shown as Figure 3.7c. Our objective is to have full coverage of the visibility atlas. In order to evaluate the objective function of current probe setting, we accumulate the visibility of each probe into a single global visibility atlas, as shown in Figure 3.8a.

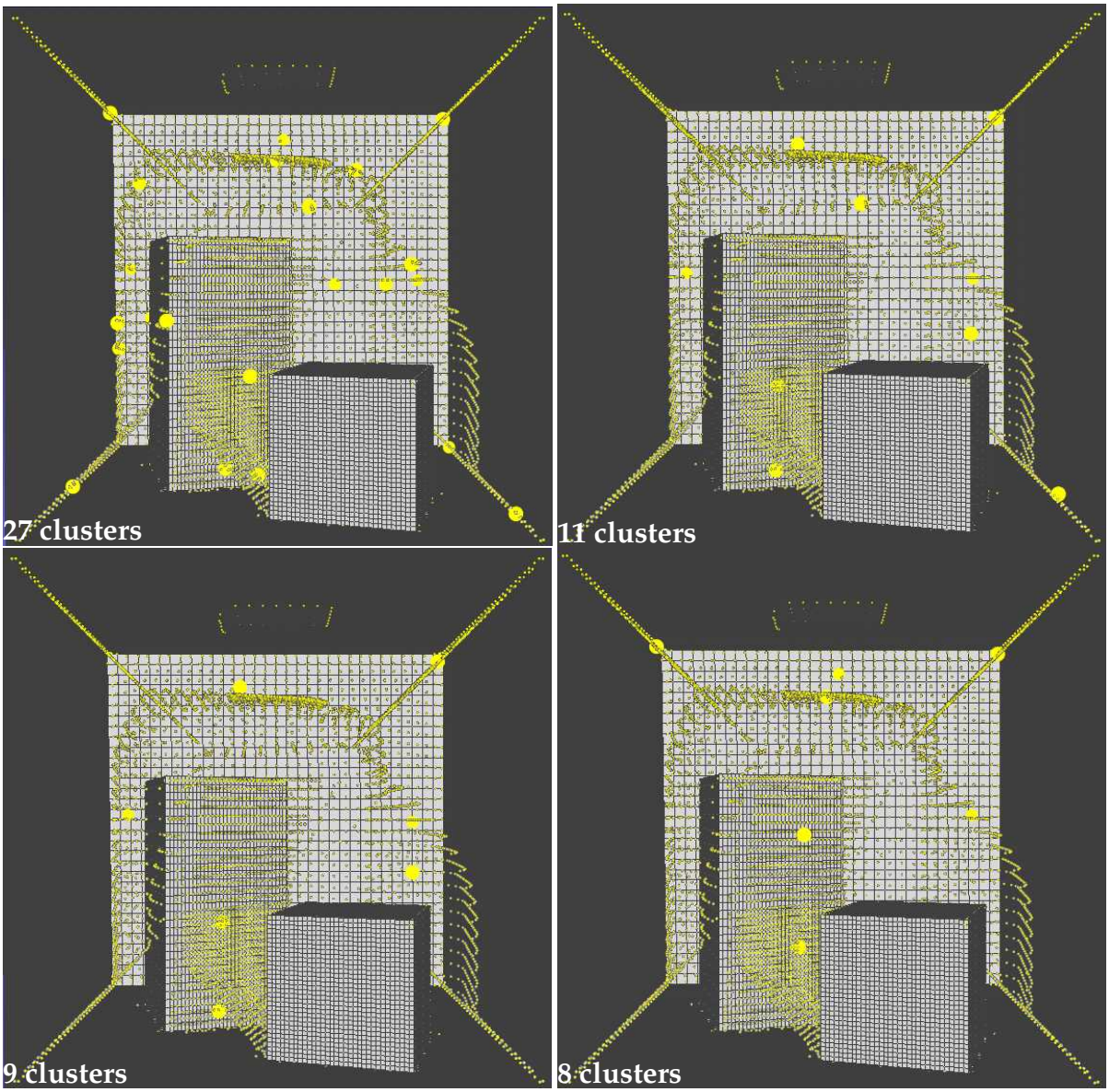
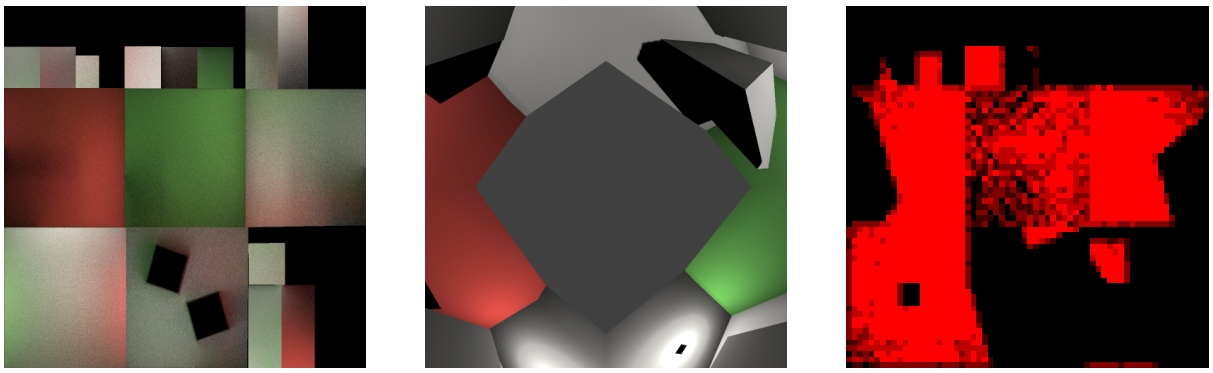


FIGURE 3.6: Different probe positions generated from medial axis. Probe positions shown in the top left image is generated from many small clusters. Probe positions shown in the bottom right image is from large clusters.



(A) Texture altas of the Cornell box.

(B) An example of an octahedral probe.

(C) Mapped visible area into texture atlas.

FIGURE 3.7: After we mapped the visible surfaces into a texture atlas, all probes will have an evaluation of the number of visible surfaces at the same scale.

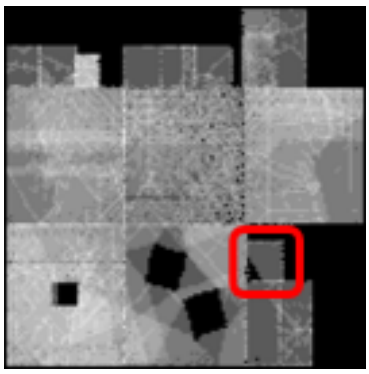
In order to know which direction would contribute most to the global full coverage, we sample eight directions around each probe, and then we compute the global visibility atlas of each direction. To do this, we first subtract the visibility of the probe from global visibility atlas, and then accumulate one of its eight neighbors to it. We sum up the resulting visibility texture to be the contribution of this neighbor (shown in Figure 3.8). Using the contributions of the eight neighbors, we perform gradient descent and move the testing probe in the direction with maximum contribution. This will be elaborated in Sections 3.2.2 and 3.2.3.

View Direction

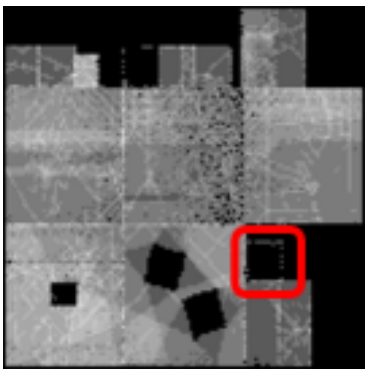
Even if a probe can "see" all the surfaces, at grazing angles, it is very likely to have artifacts if we use this probe for ray tracing. In order to evaluate the quality of how well a probe can see surfaces, we take into account the orientation towards surrounding surfaces by computing the dot product of the view direction and the normal of the surface. Using this metric we would like to drive the probes towards positions that are at less of a grazing angle from the visible surfaces.

Distance to Surface

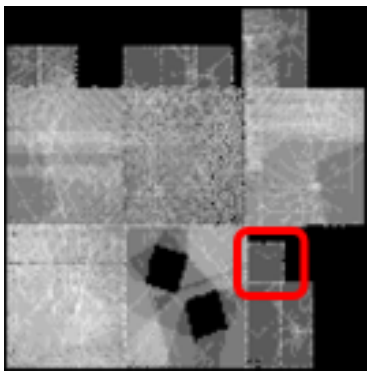
The dot product measures how the probe faces the surrounding surfaces. However, the above two metrics will still produce undesirable positions in some cases. The probes sometimes become stuck in objects because the objective function using the two metrics considers such positions as optimal.



(A) Visibility atlas of all probes accumulated. Surfaces in the red square are not covered by any probe



(B) Visibility atlas of all probes except for probe 2. The surfaces in the red square are not covered without probe 2.



(C) Visibility atlas of all probes after 100 iterations of Gibbs sampling. Full coverage is achieved/

FIGURE 3.8: Global visibility atlas. Probe 2 is the only probe that can "see" the surfaces in red square.

To fix this we added another metric based on distance to surrounding surfaces. We would like to maximize the sum of distances to its visible surface. In this way probes have a greater chance of escaping from occlusion.

3.2.2 Gradient Descent

In order to evaluate one set of probe positions, we define the fitness of a *probe set proposal* $P = \{p_1, p_2, ..., p_n\}$ as

$$R(P) = \sum_{i,j} \pi_{i,j}, \tag{3.1}$$

where$\pi_{i,j}$ is the fitness associated to the $(i,j)^{\text{th}}$ texel in the our global visibility atlas, which comprises a weighted combination of the individual metrics as

$$\pi_{i,j} = \sum_{k=1}^{n} \left(w_1 \, \overline{\Omega}_k + w_2 \, \overline{d}_k \right) \; V_{i,j}^k \,, \tag{3.2}$$

where weights w_1 and w_2 scale the relative importance of the orientation and distance metrics, and we have n probes.

In order to follow the distribution of the metrics, for each probe, we sample eight directions that form a cube with the current tested probe in the center. As shown in Figure 3.9, each probe is surrounded by eight support probes. For each probe, we replace the probe with each neighbor and compute the sum of the visibility atlas as a new reward, then we move the probe position in the direction of the support probe that gives us the largest reward improvement. When updating probe k's location p_k to p'_k, yielding a new set of probe positions $P' = (p_1, \ldots, p'_k, \ldots, p_n)$, we compute a

gradient-based update as

$$p'_k = p_k + \gamma \nabla_k R, \tag{3.3}$$

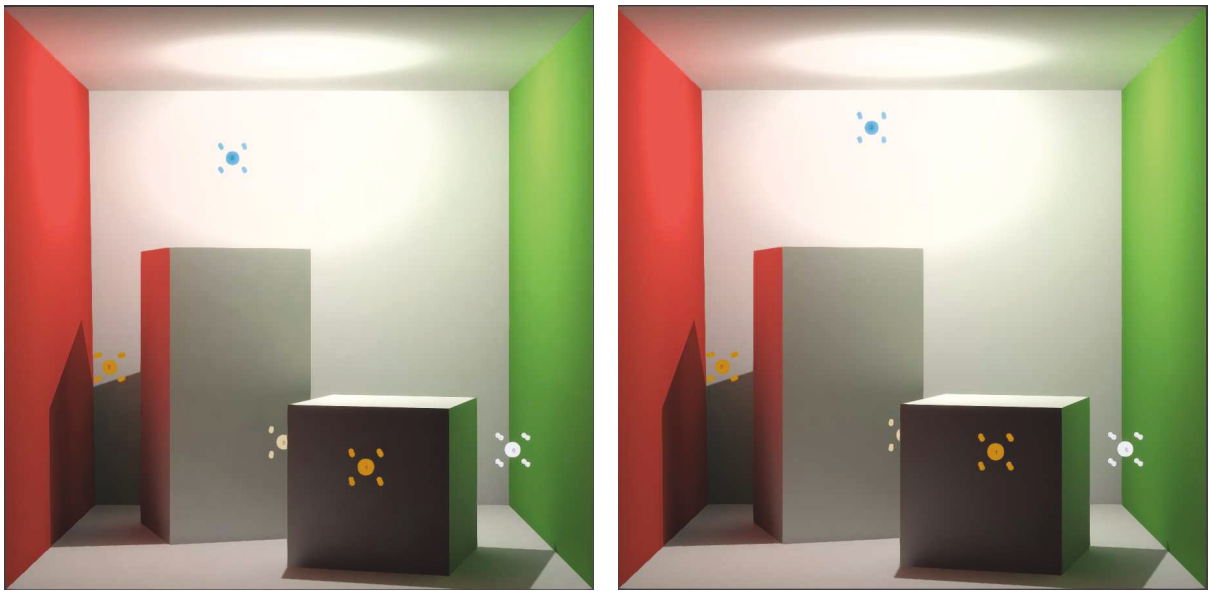
where γ is a step size defined as the distance from from tested probe to a support probe , and $\nabla_k R \approx (\partial / \partial p_k) R$ is an approximated gradient of the reward with respect to the candidate probe's current position p_k.

However, since we use a fixed grid size to sample the eight support probes, we observe that after a certain number of iterations, the probes usually move back and forth between two positions. Because the probes always move to the position of the best support probe, using a fixed learning rate for gradient descent usually leads to divergent behaviors and trapped in local minima. Also, moving all probes at the same time without knowing the moving directions of each other does not ensure that the result is better after iterations. Therefore we apply Gibbs sampling and use importance sampling diagonal directions of a cube of support probes.

3.2.3 Gibbs Sampling

Gibbs sampling [Casella and George, 1992] is a special case of the Metropolis-Hasting sampling [Chib and Greenberg, 1995]. The Metropolis-Hastings algorithm is a Monte Carlo method that can sample from a probability distribution and generate a sequence of samples. For a given large set of variables, Metropolis-Hastings draws the next sample with all variables based on the previous sample, whereas Gibbs sampling considers each variable (or each group of variables) of the current sample in turn. This method is usually used to sample multi-dimensional distributions, where the probability distribution is difficult to obtain and direct sampling is hard to achieve.

In our case, we consider the positions of the probes $P = (p_1, p_2, ..., p_n)$ that we obtained from Section 3.1. We want one sample that meets our optimization metrics. These metrics have different distributions which are difficult to compute, therefore cannot be directly sampled. For each variable p_i (a particular probe position), we sample its nearby positions and advance in the gradient descent direction. A new reward $R(P')$ can be computed by evaluating the optimization metrics on the current set of probes $P' = (p_1, p_2, ..., p'_i, ..., p_n)$ and summing up the visibility atlas. Then we determine to accept or reject the new sample by generating a random number ξ on $[0, 1]$, and accept the sample if $\xi < R(P')$. We repeat this process until current set of probes satisfies our stop criteria.



(A) Original probe setting generated from signed distance function. One corner of the short box is not covered by any probe.

(B) After 100 iterations of Gibbs optimization, probe 2 (in blue) moved up to achieve full coverage.

FIGURE 3.9: A demonstration of Gibbs sampling.

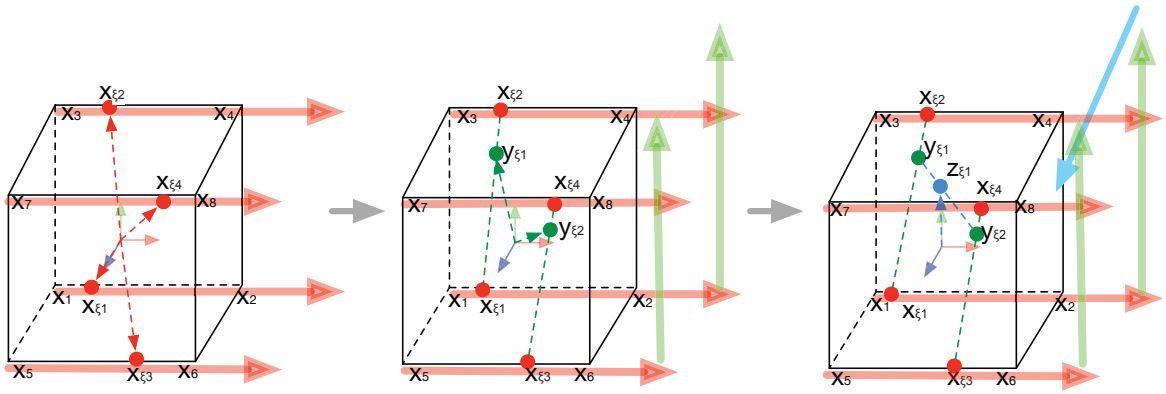


FIGURE 3.10: Stochastic direction sampling. Left to right: we construct a stochastic probe update dierction by sampling points $x_{\xi,i}$ ($i \in [1,4]$) proportional to a linear model of expected fitness along the x-adjacent support probe grid, then repeating for y-points $y_{\xi,j}$ ($j \in [1,2]$) and z-point $z_{\xi,1}$.

3.2.4 Importance Sampling

Instead of computing the position update vector $\nabla_k R$ using the fixed support probe grid locations p_k^s, we perform a stochastic direction sampling reduce coherent structure in the exploration. Our stochastic direction sampling proceeds incrementally along each canonical axis and assumes that the distribution of R along any axis follows a (locally) linear model.

Starting with the x-axis, we pair the *four* x-adjacent support probes on the corners of the support *cube*. We fit four linear models $f(x) = ax + b$ to the four fitness pairs $[R(P'_s), R(P'_{s+1})]$ (for $s \in \{1,3,5,7\}$) obtained by substituting p_k in P with p_k^s in P'_s. After solving for a and b, we can drawn a sample for the x-component of $\nabla_k R$ proportional to each of the $f(x)$, using the inversion method: the CDF of f is $F(x) = (a/2)x^2 + bx$ and its inverse is $F^{-1} = (-b \pm \sqrt{b^2 - 2ax})/a$. Thus, evaluating $F^{-1}(\xi_i)$ at four i.i.d. canonical random numbers ξ_i yield a sample point along each of the four x-adjacent support probe edges (left of Figure 3.10).

We repeat this process along the y-axis next, now considering the *two* y-adjacent support probes on the corners of the (x-presampled) support *square*, to sample *two* y points. Finally, we repeat along the z-axis, considering the *only remaining* z-adjacent support probes on the endpoints of the (xy-presampled) support *line*.

Using the sample obtained from importance sampling, we replace the current tested probe with this new sample and compute a new reward R_{new} by summing up the visibility atlas. We then take the ratio of $R_{new} \big/ R_{old}$ as the threshold for Gibbs sampling to update or reject the new sample.

Compared to the strategy of moving the probe in the direction of the best support probe, importance sampling helps the solution to escape from local minima and to achieve global optimum – full coverage of the scene while ensuring each surface is covered at a relatively good angle from at least one probe. One result of Gibbs with importance sampling is shown in Figure 3.9. After 100 iterations, probe 2 moved upward so that the top of the short box is covered.

Algorithm 2 Probe Position Optimization

```
while !FullCoverage do
    for each testingProbe in probes do                ▷ Gibbs sampling, Section 3.2.3
        Generate sample:
        newSample ← ImportanceSamplingProbe(testingProbe);      ▷ Section 3.2.4
        Calculate reward:
        R_old ← SumUp(OriginalVisibilityAtlas);
        R_new ← SumUp(VisibilityAtlasWithNewSample);
        Accept or reject:
        A ← R_new / R_old;                  ▷ Compute the acceptance probability
        u ← Random(0, 1);                   ▷ Generate a uniform random number
        if u ≤ A then
            UpdateProbeSettings(newSample);
        else
            RejectNewSample;
        end if
    end for
end while
```

Chapter 4

Shading with Non-uniform Probes

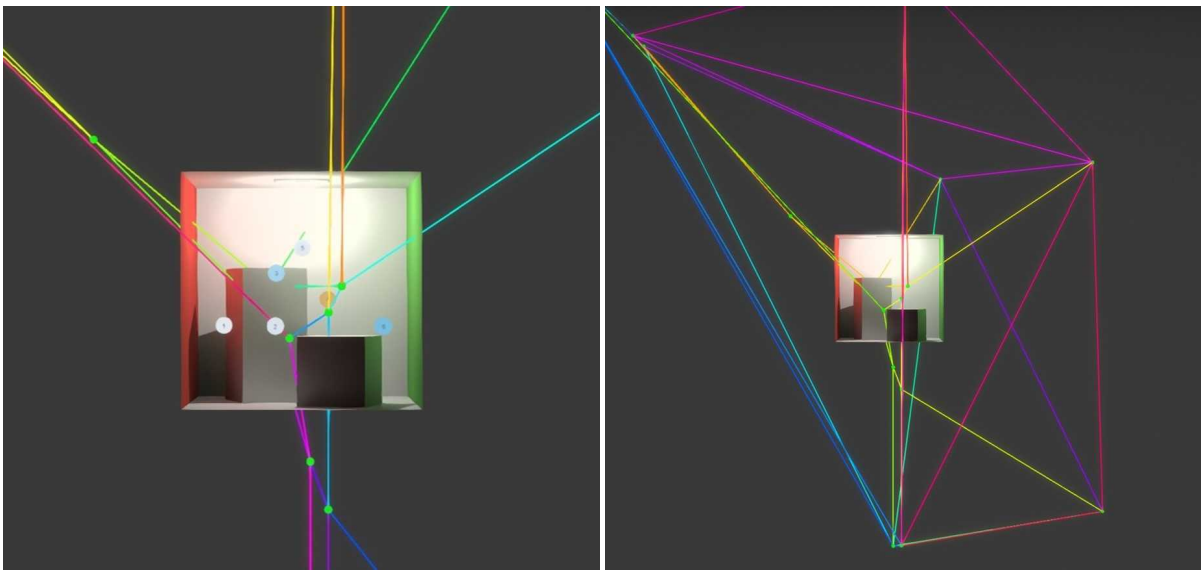
We introduce our algorithms to organize non-uniform probes for run-time rendering, and our probe selection strategies during ray tracing in probes.

4.1 Voronoi-based Selection

Our first algorithm for organizing and selecting non-uniform probes is based on Delaunay tetrahedralization. Delaunay tetrahedralization is a 3D extension of Delaunay triangulation. Delaunay triangulation aims to maximize the minimum internal angle for each face of each triangle, therefore tends to avoid sliver triangles. We build the Delaunay graph from the probe positions, as Cupisz [2012] did for irradiance probes. We also construct the dual graph of Delaunay, i.e., a Voronoi graph, that divides the space into Voronoi cells, with one probe in the center of each cell. A Delaunay graph and its Voronoi graph connect probes and the underlying 3D space. We use this structure to build our probe look up and selection algorithm at run-time.

4.1.1 First Probe Texture with Linked List

Using a deferred renderer, we only need to compute the first probe for the surfaces that are visible to the camera. We do this by computing a screen-sized image with each pixel assigned a probe index. For each pixel we compute the index of the probe that is most likely to provide a hit for rays from the surface in that pixel. Following the same heuristic for regular grids, we take the nearest probe as the first probe. With regular grids, the nearest probe for any pixel could be easily computed by dividing the world space position of the pixel by the grid size. With non-uniform placement, the nearest probe can be computed using Voronoi graph. Each Voronoi cell defines the region that has a closer distance to that probe than to any other probe. Therefore for any surface in the scene, it will fall into the Voronoi cell of the nearest probe. Then we assign this pixel with the probe in this cell as the first probe to use.



(A) Original Voronoi constructed. (B) Closed Voronoi cells.

FIGURE 4.1: Step 1 of baking neighbor probes: closing Voronoi cells.

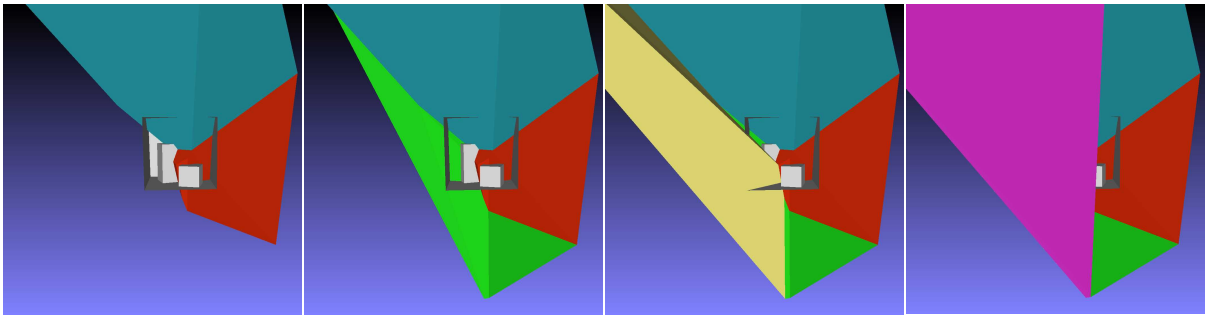
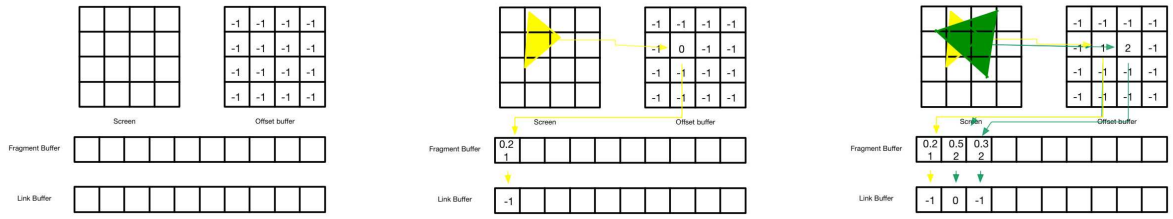


FIGURE 4.2: Visualization of Voronoi cells after step 2 of baking neighbor probes: shrinking Voronoi cells towards their centers.

By definition, Voronoi cells can be open and reach to infinity. Therefore, before we use them at run-time, we need to pre-process the Voronoi cells. First we separate each Voronoi cell as an independent mesh. For rendering purpose, we close the open Voronoi cells with distances that do not intersect with the scene's geometry. Then we triangulate each Voronoi face, as shown in Figure 4.1. For each Voronoi cell, we shrink the cell by moving its vertices to its center (probe position) by an epsilon distance to keep the Voronoi cells separate so that when computing the distance between scene geometry and Voronoi faces, no Voronoi faces will be considered at the same distance to a surface. The result is shown in Figure 4.2. The Voronoi cells are stored in an array with the same order as the probes they linked with. This step can be done at loading time and will be executed only once. The array of Voronoi cells can be used multiple times after construction.

At run-time, we compute the nearest probe for each pixel using Z-buffer and meshes of the Voronoi cells. We build a per-pixel linked list and stack all the Voronoi cells



(A) Initialize the start offset buffer (screen size) to -1. The fragment list and link buffer (large enough to store all fragments) are empty.

(B) For each fragment, update the corresponding offset index, fill the fragment buffer with the current fragment, link buffer with the previous offset index.

(C) Continue processing all triangles, building the linked list by updating the offset index and fragment list, store the previous index to link buffer.

FIGURE 4.3: An illustration of per-pixel linked list.

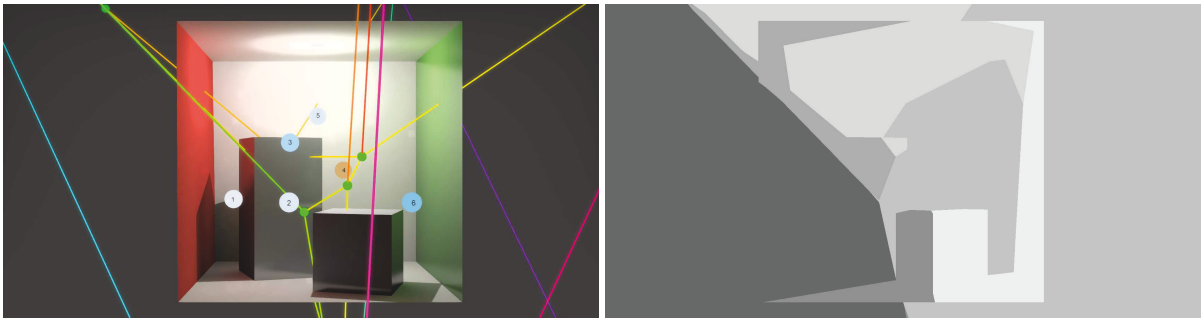
that intersect the current pixel in the list. Then we select the nearest Voronoi cell by traversing the list. This method is introduced by Yang et al. [2010] for computing order-independent transparency, where an A-Buffer is used to store a list of transparent fragments for each pixel, and sort them in place and accumulate them later in the rendering pass.

Because the number of probes can be large, one pixel could be intersected by many Voronoi faces, and the resulting linked lists could be long. We reduce the length of the linked list by culling the Voronoi cells by their centers (i.e., the probe positions). In this way we do not consider probes behind current surfaces.

After the culling pass, we render the mesh of each Voronoi cell (shown in Figure 4.2) using a pixel shader and build the per-pixel linked list. For each fragment, we store the corresponding probe index and the difference of depth between Z-buffer (contains the scene's geometry) into the linked list, as shown in Figure 4.3. In a second pass, we use a compute shader to iterate through the linked list for each pixel to get the probe with minimum difference from the depth buffer. The result of one frame is shown in Figure 4.4. This process is described in Algorithm 3.

Algorithm 3 Compute First Probe Image

```
VisibleVoronoiList← FirstPassCullingByCenter(AllVoronoiCells);
ConstructPerPixelLinkedList(VisibleVoronoiList);
for each pixel in Z − Buffer do
    nearestIndex ← IterateLinkedList(pixelCoord);      ▷ Find nearest Voronoi cell
    ImageStore(pixelCoord, nearestIndex);              ▷ Write to first probe image
end for
```



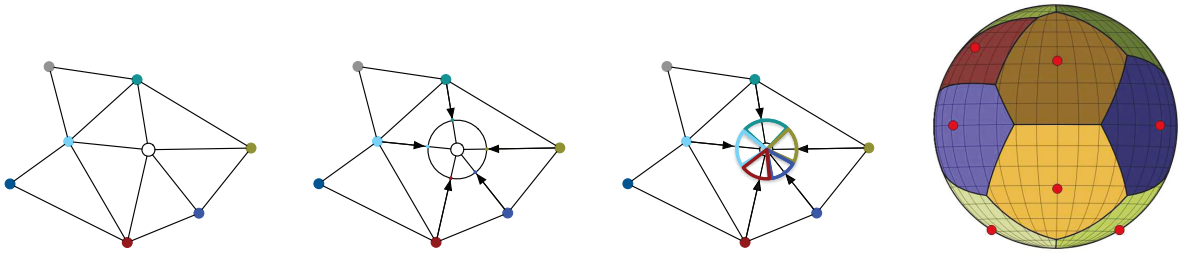
(A) Visualization of Voronoi cells and probes. (B) First probe texture.

FIGURE 4.4: One frame of the first probe texture that encodes probe indices. Bright areas indicate large probe indices, dark areas indicate small probe indices.

4.1.2 Baking Neighbor Probe Indices into Probes

The first probe cannot ensure the ray will find a hit. Therefore we need to select the next best probe when a ray goes behind surfaces.

Consider the ray tracing process in a probe. When the ray goes behind surfaces at a specific texel, a world space position can be constructed from the ray where the current probe does not "see". We want to switch to a probe that knows the surfaces behind, therefore we should search for a probe in the direction from the current probe to the position that the ray goes behind a surface. In fact, each texel of a probe texture indicates a different direction, at which the ray can go behind a surface. Therefore we bake the indices of the neighbor probes that can "see" surfaces behind the surface for each texel by constructing the Voronoi spheres.



(A) Delaunay graph in 2D with tested probe in the center and neighbor probes in different colors.

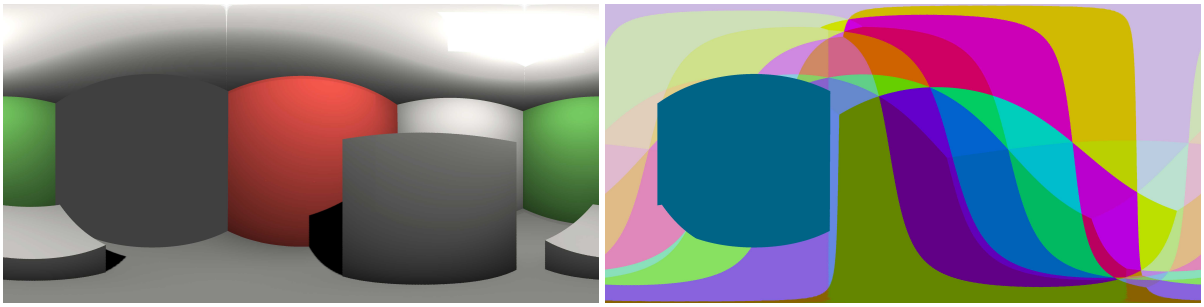
(B) Project neighbor probes onto the unit sphere with its center at the current tested probe.

(C) Construct the Voronoi sphere.

(D) Voronoi sphere in 3D space.

FIGURE 4.5: Delaunay graph and projection onto a unit sphere.

The Delaunay graph connects all probes in tetrahedra. For any probe p, we can get its neighbors from the constructed Delaunay graph. We project the neighbors of p to a unit sphere centered at the probe position of p, and construct the Voronoi sphere, as



(A) Radiance texture of a probe. (B) Neighbor probe texture of a probe.

FIGURE 4.6: An illustration of neighbor probe indices baked into probes of the Cornell box.

shown in Figure 4.5. Each region on the sphere defines an area that has the smallest distance to the neighbor probe. From this graph, we can get information about the next best probe for any given texel. We sample the Voronoi sphere and bake it into the probe texture. An example for a probe is shown in Figure 4.6, we baked 4-neighbor probes for each pixel using RGBA channels.

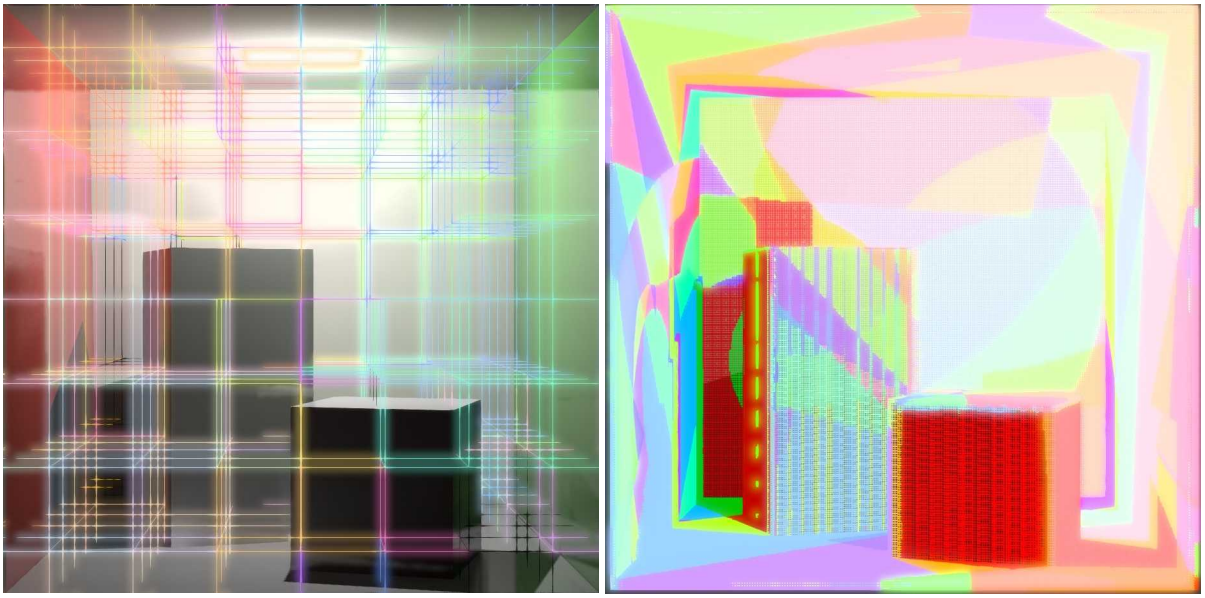
4.1.3 Run-time: Texture Lookup

As mentioned before, at run-time, the first probe texture is constructed per frame by comparing the nearest distance with Z-buffer. With the first probe texture is constructed, the run-time algorithm for ray tracing in probes is described in Algorithm 4.

Algorithm 4 Probe Selection with Linked List and Neighbor Textures

```
function LIGHTFIELDTRACE(ray)                          ▷ trace ray across probes
    result←UNKNOWN;
    currentProbe← FirstProbeTextureLookUp(pixelCoord);          ▷ Section 4.1.1
    (result, ray)← singleProbeHiZTrace(ray, currentProbe);        ▷ Section 4.3
    while result == UNKNOWN do
        currentProbe← NeighborProbeLookUp(currentProbe, ray);   ▷ Section 4.1.2
        (result, ray)← singleProbeHiZTrace(ray, currentProbe);
    end while
end function
```

Although this probe selection strategy is simple and effective, the construction of a linked list per frame is costly. The performance mainly depends on the sorting, which depends on the length of the linked list. When the scene's depth is large, each pixel has a long linked list since most of the Voronoi cells pass the culling test. Therefore the performance varies for different viewports and scene complexities. To avoid run-time construction, we use sparse voxel octree that encodes the probe indices and spatial information in each voxel cell.



(A) Octree level 5. (B) Octree level 9.

FIGURE 4.7: Visualization of baked probe indices into sparse voxel octree for the Cornell box.

4.2 Baking Probes to Sparse Voxel Tree

Voxel representation is widely used in computer graphics. As an acceleration structure, sparse voxel tree can be used to retrieve information in logarithmic time. We encode probe indices into voxels. Therefore for any given position in world space, we can traverse down the tree until the leaf node, and retrieve the best probes that know information about the current position.

4.2.1 Baking Probes Based on Visibility

Sébastien and Zanuttini [2012] propose a method to improve the quality of image-based lighting using local cubemaps. This method defines an inner influence volume (a bounding sphere) where a cubemap fully contributes to the final result, and a boundary volume (a bounding sphere), which the cubemap contributes less but still has influence on the final result. These volumes are associated with different weights, so that for any position the results will be blended between results from different cubemaps. In order to look up the correct cubemap for any position of interest, their method defines an influence volume for each cubemap, and bake them into an octree. Inspired by their work, we bake the probe selection choices for each position into an octree.

For each probe, we define an influence region where this probe can "see" all the geometry, therefore, any ray falling in this region can use this probe to find a hit. We

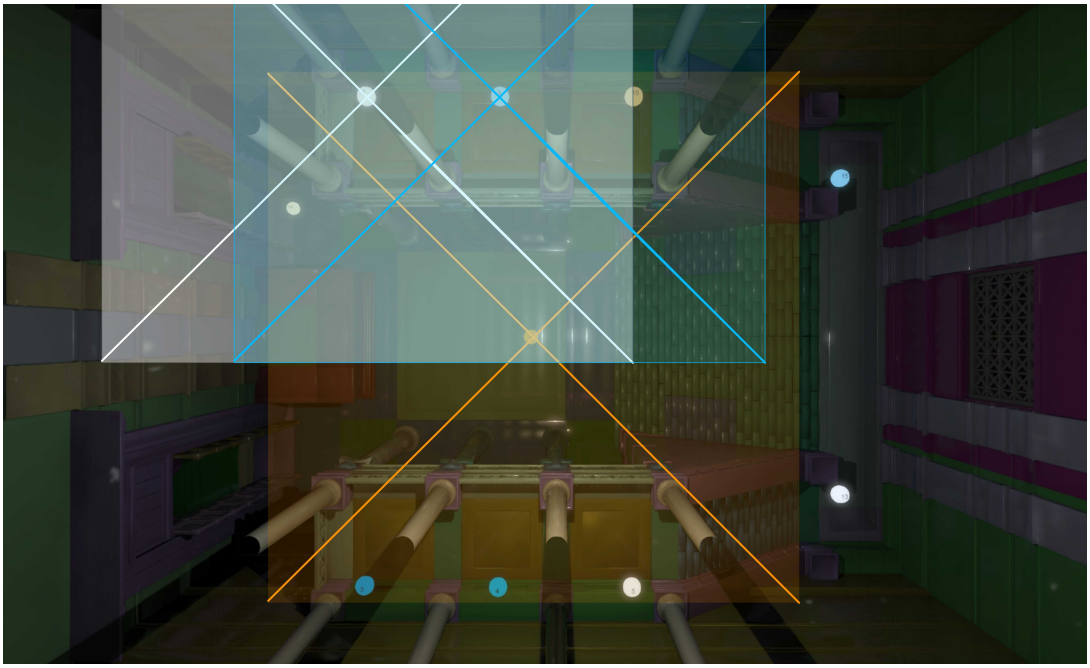


FIGURE 4.8: Top view illustration of influence regions of probes in the iconic temple Ubisoft©. The top left pillar of the temple is covered by the orange probe (in the middle) and the white probe (top left).

also define a boundary volume, where the probe can see some of the geometry, but there might be some occlusions that cause rays to fail to find a hit. Figure 4.8 illustrates how the influence region works. In this figure the influence distance for each face is defined by taking the median of the radial distances captured in this cube face.

There will be overlap between influence volumes. We bake only four probes for each voxel, based on the following metrics:

- **Visibility** Only consider probes that are visible to the current geometry. If a probe is not visible, this probe does not know information about the current geometry, therefore it is not a good candidate to trace rays.
- **Projected area of pixel** If a probe "sees" a surface at a grazing angle, the projected area of the pixel would be large, and the depth information captured in texture would be limited. It is likely to produce artifacts if a ray is marched here using this probe. Therefore this probe will not be a good candidate for this surface.
- **Distance to surfaces** Satisfying the first two metrics cannot ensure the probe is a good candidate. Consider that one probe is visible to a surface, but it has a relatively small projected area of pixel, and very far away from the surface. It is likely that this probe does not know about neighboring surfaces, because long distances increase the possibility of having other occluders between the probe

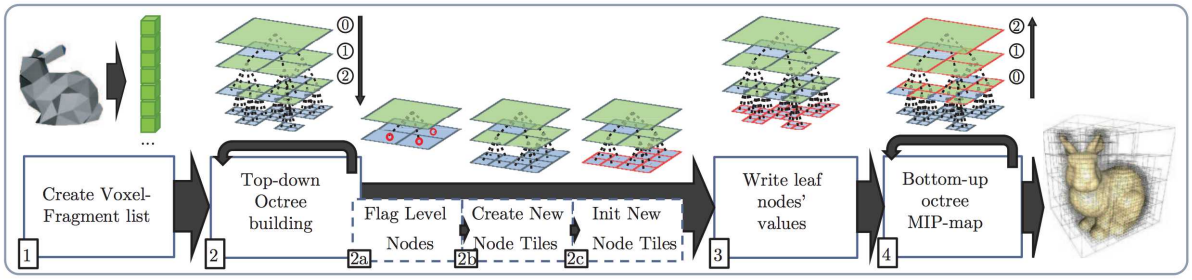


FIGURE 4.9: Octree building steps, image courtesy of Crassin and Green [2012].

and the surface. Therefore we would like to make sure this probe gets less chance to be selected.

4.2.2 Implementation Details

Our sparse voxel octree implementation is based on Crassin and Green [2012], as shown in Figure 4.9. This implementation first voxelizes the scene in the finest resolution, and saves all the fragments into a fragment list. Using this list, we can build the sparse octree structure from top to bottom: Start from the highest level (root node) that contains only one voxel, progressively subdivide a current node to the next level if the voxel is intersected with any fragment in the fragment list, until the finest voxel resolution or target depth is reached. In the second step, the bottom level is filled with fragment information, then the upper levels are filled later by mipmapping their children levels.

Unlike the original implementation, for any empty voxel (i.e., no geometry intersects this voxel), we also need to allocate space and write the probe index into it. This is necessary because

- A ray can go behind surfaces at any position in 3D space. We need to query the octree for the probe that is the best to switch to for the ray at that position.
- To support dynamic geometry. Dynamic objects could be at any arbitrary position. Therefore we need to bake the best probe for any position in 3D space.

For empty voxels, since there is no geometry available, we only bake the probes based on visibility and distances to probes.

Because our placement strategy ensures full coverage, it also ensures that each voxel will be visible to at least one probe. Therefore every voxel will be assigned with at least one probe.

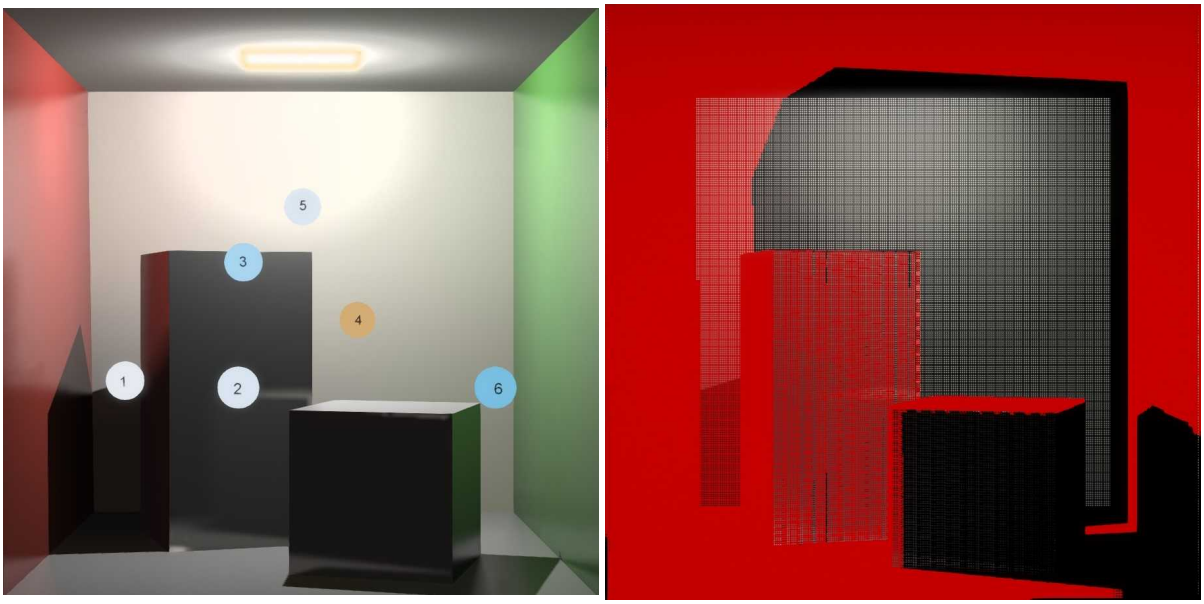


FIGURE 4.10: Visibility test of probe 1; red as visible, black as occluded.

Baking Probe Octree: Probe Sorting and Multi-sampling

Before baking probes into the octree, the tree structure is built using the scene's geometry. This means our probe index octree has the same structure as an octree of the scene. This is important because the scene's geometry also acts as occluder at the same time, which separates the 3D space. Therefore building the octree from the geometry is equivalent to baking the exact boundaries of each probe. Also, since the large portion of the rendering done on static geometry, a finer octree granularity on the geometry can help select accurate probes to use for each surface and ultimately improve the rendering quality.

To fill leaf nodes with probe indices, we use a compute shader per level, one thread for each node in this level. For each node, we first check whether this node has allocated children. We will only bake probes if a current node does not have any children. For each leaf node, we bake at most four probes into this voxel. This is done in three steps.

The first step is to eliminate probes that are not visible from the current voxel. This can be achieved by tracing a ray through the voxel octree from the voxel center to the test probes using the cone tracing algorithm introduced by Crassin et al. [2011]. Figure 4.10 shows an example of the visibility test of voxels in level 9 regarding probe 1. Probe 1 will only be considered as a candidate to those voxels visible to it, which are shown as red. If the voxel is not the top level, which means the voxel represents empty space, we randomly choose multiple samples in this cell to test the probe visibility.

The second step is to compute the metrics. We combine all metrics into one by computing $\frac{\vec{n}\cdot\vec{p}}{d}$, where $\vec{n}$ is the normal of the surface, and $\vec{p}$ is the direction from the

surface to the probe, and d is the distance from surface to probe. For empty voxels, we simply compute the inverse of distance, therefore we can use the same ascending sort algorithm to sort probes for all voxels.

The last step is to compare the current value with sorted probes values. Since we only bake four probes, we do not sort them all, only compare current probe with the best four we have stored, and if the current one is better, we will discard the last one.

We visualize the octree in Figure 4.7. Here we only visualize the voxels that encoded with probe indices, i.e., the leaf voxels in this level. In the finest level (Figure 4.7b), leaf voxels are those that intersect with geometry.

run-time: Traverse Down SVO

The run-time algorithm (shown in Algorithm 5) uses sparse voxel octree to retrieve the best probes for each ray.

Algorithm 5 Probe Selection with SVO

```
function LIGHTFIELDTRACE(ray)                          ▷ trace ray across probes
    result← UNKNOWN;
    while result ==UNKNOWN do
        probeCandidates← TraverseDownSVO(ray.currentPosition);
        (result, ray)← singleProbeHiZTrace(ray, probeCandidates);
    end while
end function
```

Theoretically, for any ray traced from a surface, it would be able to find at least one visible probe. Because we considered visibility from each voxel cell, we will not be trapped in a loop of probes because if a ray goes to a voxel that the current probe does not know, when the ray query the best probe from the voxel, current probe is not in its list because this probe is eliminated due to visibility test failed.

It is possible that the current ray runs out of probes but still can not find a hit. In this case, instead of iterating through all probes, we exit the while loop, since we are sure that the probes that most likely to have a result cannot provide a hit. Other probes have less probability to get results. The performance and memory usage analysis will be provided in Chapter 5.

4.3 Hi-Z Ray Tracer with Probe Textures

The original algorithm by McGuire et al. [2017] traces rays in the distance probes with two different resolutions: For a given ray, the segment is first traced in a low-resolution distance probe, which is downsampled from hi-resolution distance probe. If there is a

conservative hit returned from the low-resolution tracing in the current segment, then this segment will be marched in hi-resolution distance probe to find the accurate hit coordinates.

We further extended this approach into a full hierarchy by adopting hi-Z ray tracer introduced by Uludag [2014].

4.3.1 Hi-Z Ray Tracer

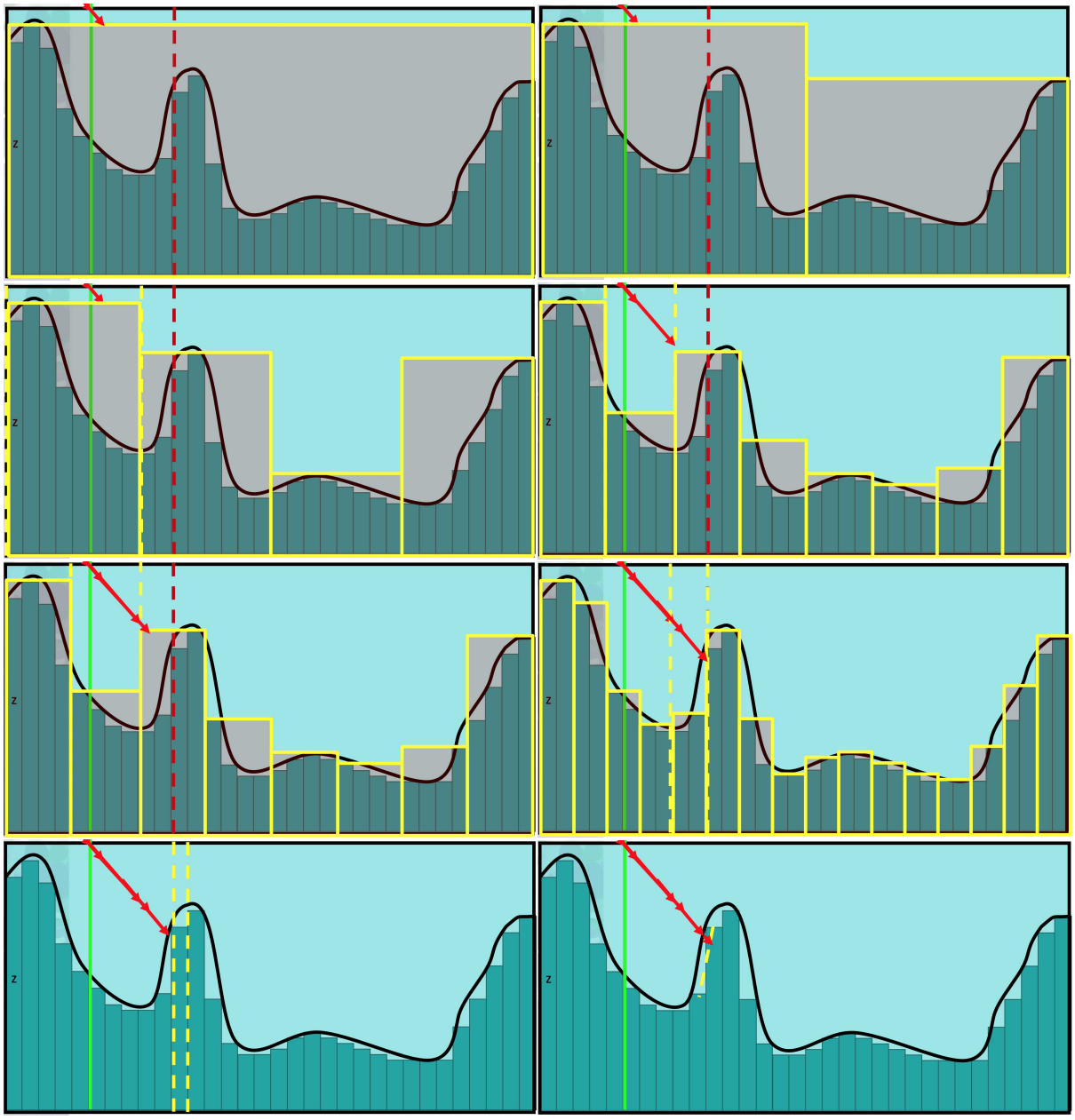


FIGURE 4.11: Hi-Z ray tracing process, image courtesy of Drobot [2018].

The hi-Z (hierarchical Z-buffer) ray tracing algorithm uses a depth buffer with minimum and maximum values stored in a mipmap. Each higher level mipmap contains

the minimum and maximum depth values of the four samples from its lower mipmap. The minimum and maximum depths in a higher level describe the height field in a hierarchical way. At rendering time, as shown in Figure 4.11, for each trace step, the ray compares the depth of the current position with the min-max depth buffer. If the current depth is smaller, which indicates the intersection point is in front of the current position, the ray goes up a level; if the current depth is within minimum depth and maximum depth, indicating there is an intersection in this volume, the ray goes down one level, until mipmap level 0 is reached.

The reason why we can adapt the hi-z ray tracer into a probe is that an octahedral mapping is piecewise linear, and rays are projected into at most four pieces on the octahedral map. Each ray segment is marched through one face, therefore the hi-Z ray tracer can be integrated into each ray segment. However, there are several implementation details that need to be considered.

4.3.2 Implementation Details

Baking Hi-Z Probes

Since our probe texture is a squared map from a cubemap, directly mipmapping the octahedral probe would cause boundary depth artifacts. Therefore we mipmap each cube face, and sample each mip level separately to each squared probe texture, so that we have less distortions and missing information, especially on face boundaries.

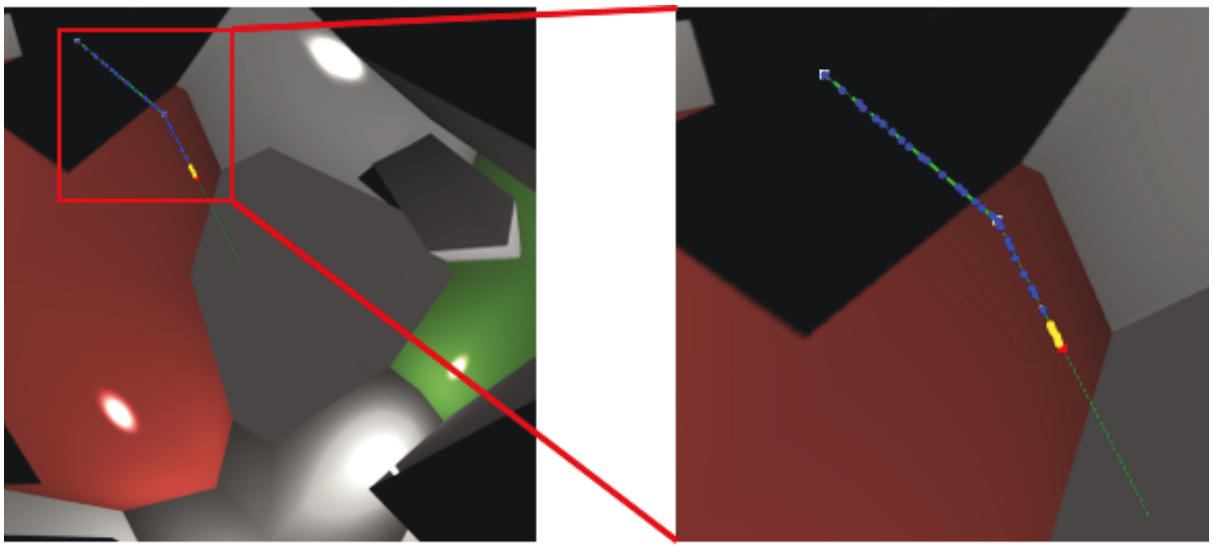
Run-time

Since probe textures store radial distance instead of depth, the hi-Z computation also needs to be adapted accordingly. Therefore we are able to trace rays in world space across probes. Performance and rendering results are provided in Chapter 5.

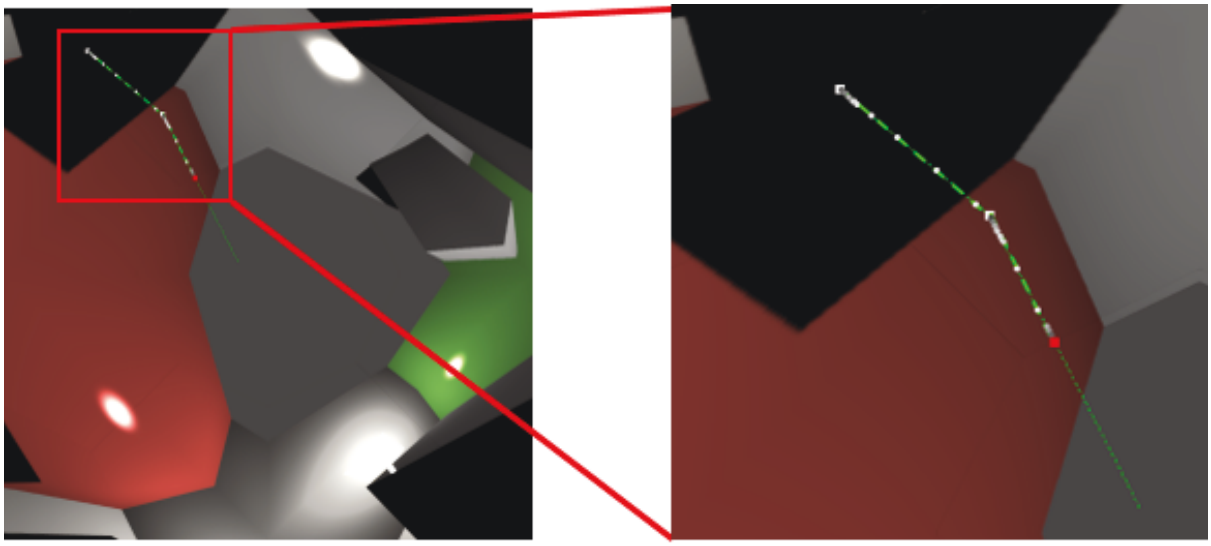
4.3.3 Results and Discussion

With hi-Z ray trace, we use only half the computation time as the original algorithm to reach nearly the same quality (see Chapter 5 for details). This is because the hi-Z uses fewer steps than the original algorithm by switching between different mip levels. The steps of hi-Z and the original algorithm are visualized in Figure 4.12. However, we have some artifacts due to inaccurate sampling in higher level mipmaps. Therefore we limit hi-Z to use only six or seven levels of mipmaps, instead of the whole pyramid. In terms of memory usage, hi-Z relies on the hierarchy of mipmaps to perform fast ray marching, while the original algorithm uses two levels of mipmaps. We use RGB16F 2D texture for minimum and maximum depths and convolved visibility, and

seven probes as shown in Figure 4.10 with resolution 512 by 512. Mips take 64MB, while regular grids use only 2MB for encoding radial distances using R16F with no mipmaps.



(A) Original ray marcher in probes with two different resolutions. In total 42 steps. Yellow dots are hi-resolution ray marching and blue dots are low-resolution ray marching.



(B) Hi-Z ray marching with 20 steps in total, the color indicates which level of mipmap it is sampling. The top mipmap level is limited to 7.

FIGURE 4.12: A ray traced in a probe texture of the Cornell box, using the original two-resolution ray marching and hi-Z ray marching.

Chapter 5

Results and Discussion

In this chapter, we provide our results. We test scenes with different complexities, compare quality, performance, and memory footprint of our placement and regular grids.

We implemented our algorithm in the G3D engine [2017]. Our results are rendered on a PC with an NVidia GeForce GTX 1060 graphic card and a 3.6 GHz Intel Xeon E5-1650 v4 CPU with 64GB of memory. In our implementation, direct illumination is computed with a deferred renderer and shadow mapping, the glossy component of indirect illumination comes from ray tracer in light probes. Since the placement of irradiance probes is a different problem and is not our focus here, we use irradiance probes laid out in regular grids for the diffuse component. Radiance probes are initialized with direct lighting from a deferred renderer, multiple bounces can be generated by updating radiance probe textures to the next bounce. In all the following tests, we use a probe resolution of 512×512.

5.1 Quality

In order to evaluate the rendering quality using non-uniform placement, we provide two comparisons: different probe placement settings with the same rendering quality, and different rendering quality with the same number of probes for regular grid and non-uniform placement. We provide ground-truth images of one bounce indirect glossy component and of final rendering result with one bounce indirect light computed using a path tracer for comparison; we report MSE values. We also render final rendering results and one bounce indirect glossy using a relatively dense grid as a reference of the best results we can get from regular grids. The images are attached in the Appendix.

Since our work mainly targets radiance probes, we show the comparison of final rendering results, and filtered glossy component.

5.1.1 Same Quality, Different Probe Placements

Cornell Box

Figure 5.1 shows probe settings and rendering results. The two boxes inside and walls of the large box are set to be glossy. In the first row of Figure 5.1, we visualized the probe positions. For non-uniform placement, we use seven probes (probe 0 is behind the tall box) generated from signed distance function. To take a closer look at the difference between these two probe settings, we compare filtered glossy components in the second row. Regular grids can do a decent job for most surfaces, except for some artifacts on detailed geometry such as reflection between two boxes. Our non-uniform placement generates a better reflection of the short box on the tall box, because our placement better covers the surfaces between the two boxes.

Iconic Temple

The iconic temple has more complex geometry on both sides than in the center. In this case, non-uniform placement has the advantage of adaptive sampling - we can place more probes at interesting places. The regular grid of 4×2 has eight probes under the stairs.

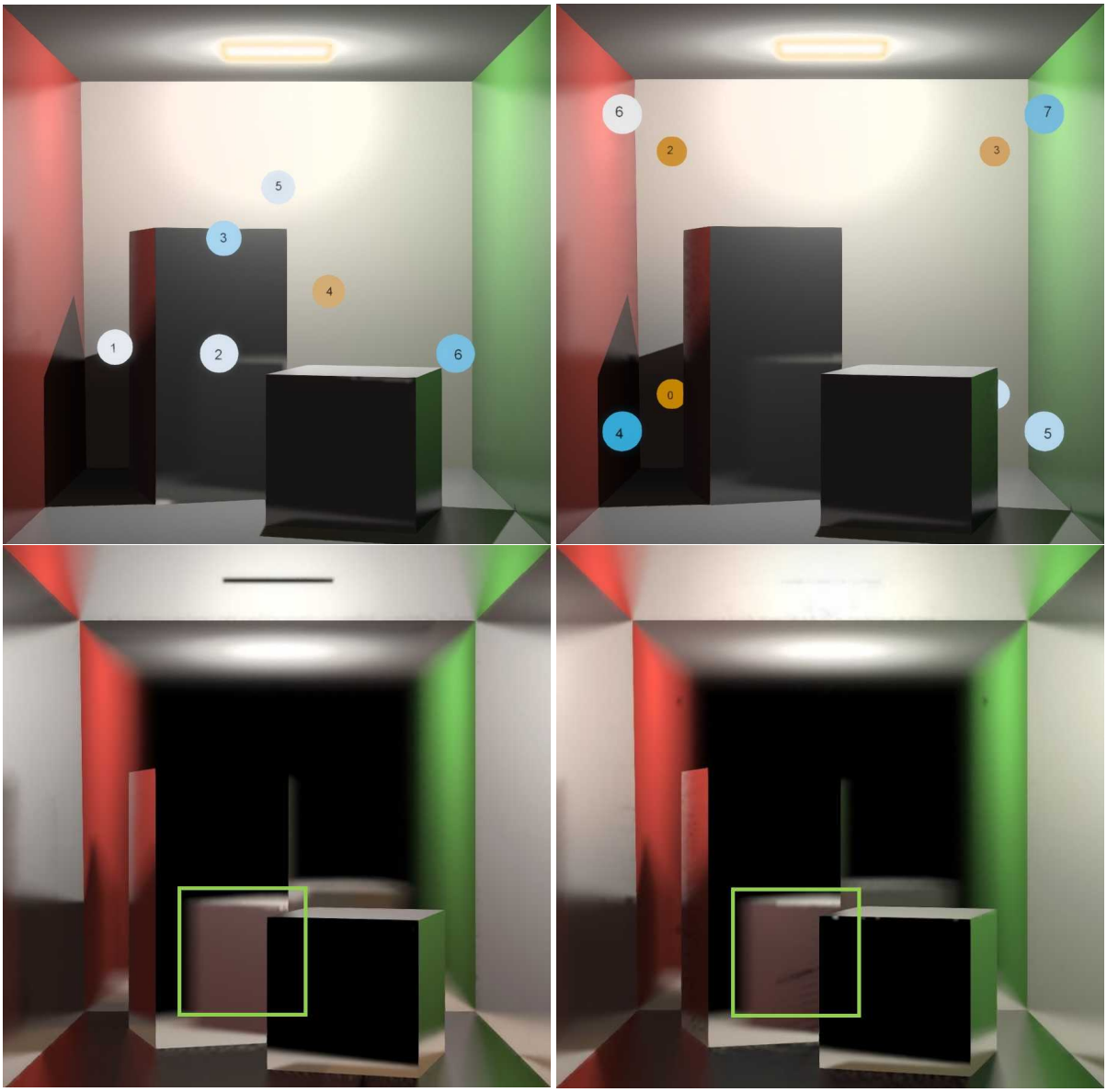
For a pillar made in a cylinder shape, we need at least three probes around it to cover the side. In Figure 5.2 we place one probe between each pair of pillars, and one in the center of the corridor, thus every pillar is covered (shown in Figure 5.3). This set of probe positions is generated from signed distance functions. Comparing the filtered glossy component of a $4 \times 2 \times 4$ regular grid and a non-uniform placement, we have generally fewer hits for the vertical side of the stairs in non-uniform placement, because we only placed one probe to capture the central corridor, but we captured most glossy features such as pillars reflected on walls and walls reflected on the floor using half of the number of probes.

Crytek Sponza

Sponza has a similar structure as the iconic temple, with two floors on both sides that need at least two layers of probes, and one corridor in the center.

For non-uniform placement, we generated different probe settings from medial axis skeleton. The number of probes varies from 54 to 84; here we take the result with 64 probes and 54 probes as examples, the rest of the probe settings are provided in Appendix A.

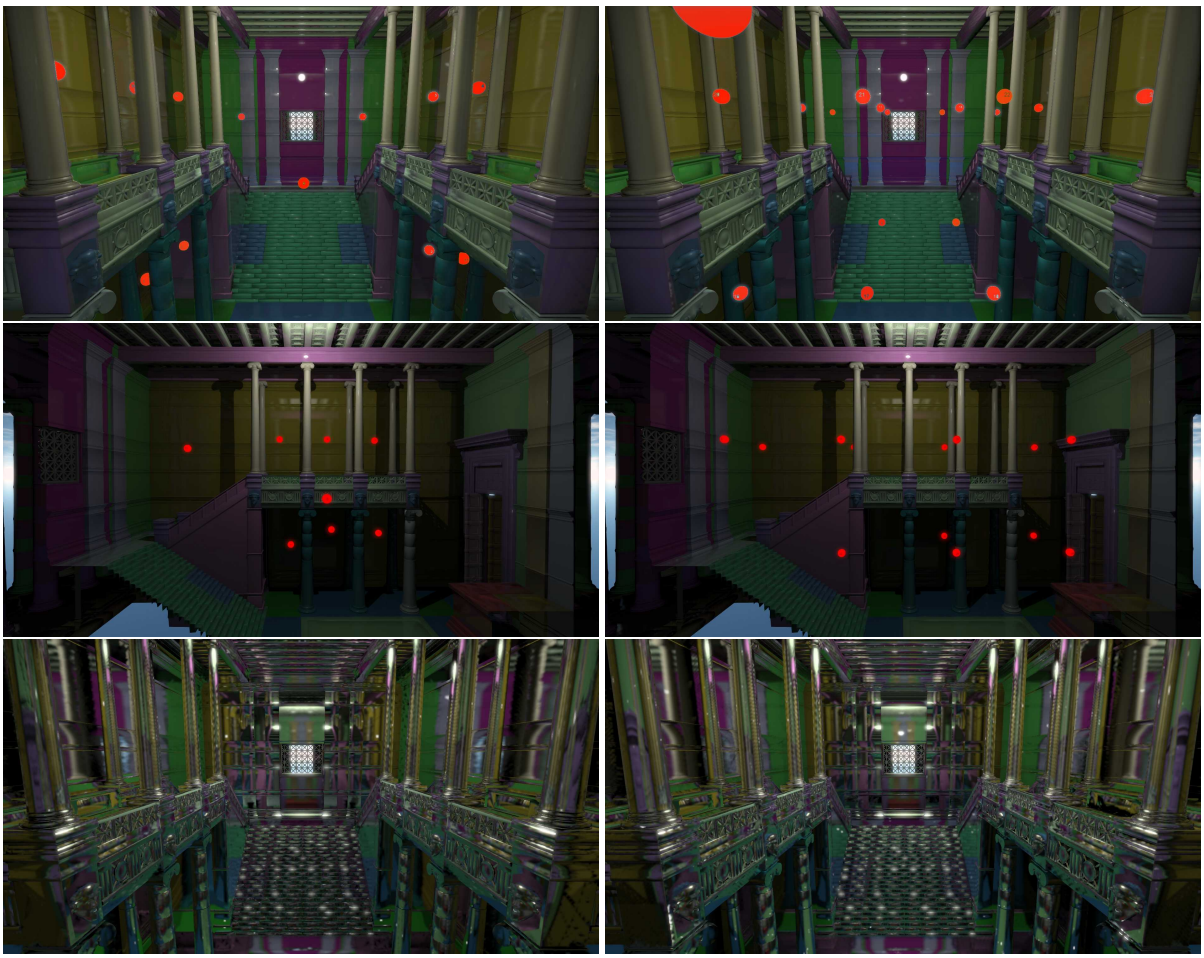
For regular grids, with the statue and area light, the minimum number of vertical layers needed is four. We tested multiple settings with three vertical layers, one in



(A) Non-uniform placement
7 probes,
render time 11.3ms,
MSE 2.7×10^{-2}

(B) Regular grids
$2 \times 2 \times 2 = 8$ probes,
render time 6.1ms,
MSE 2.8×10^{-2}

FIGURE 5.1: Comparison of non-uniform and regular grid in final rendering results (first row) and filtered glossy components (second row) of the Cornell box.



(A) Non-uniform placement
16 probes,
render time 17.3ms,
MSE 2.78×10^{-3}

(B) Regular grids
$4 \times 2 \times 4 = 32$ probes,
render time 27.7ms ,
MSE 2.70×10^{-3}

FIGURE 5.2: Comparison of the same visual quality using non-uniform and regular grids placement in final rendering results (first row and second row) and filtered glossy components (third row) of iconic temple Ubisoft©. Note that when the MSE is similar, our non-uniform placement uses fewer probes to achieve the same quality.

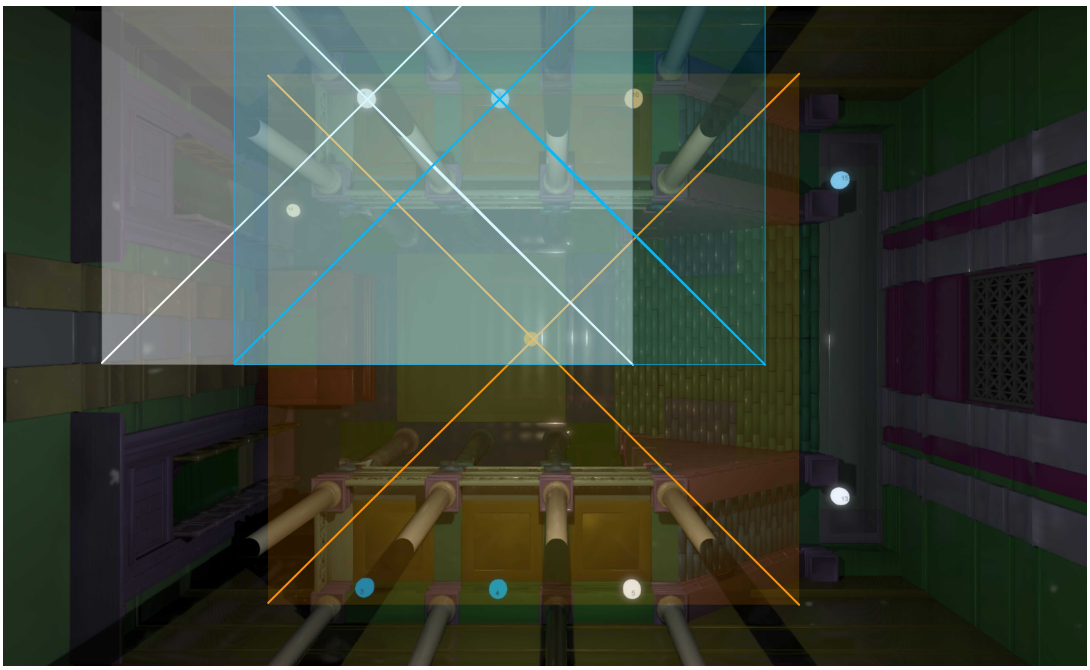


FIGURE 5.3: Top view illustration of one pillar gets covered by 3 probes in the iconic temple Ubisoft©. The upper left pillar is covered by probe in the center, and the probe coded in white upper left.

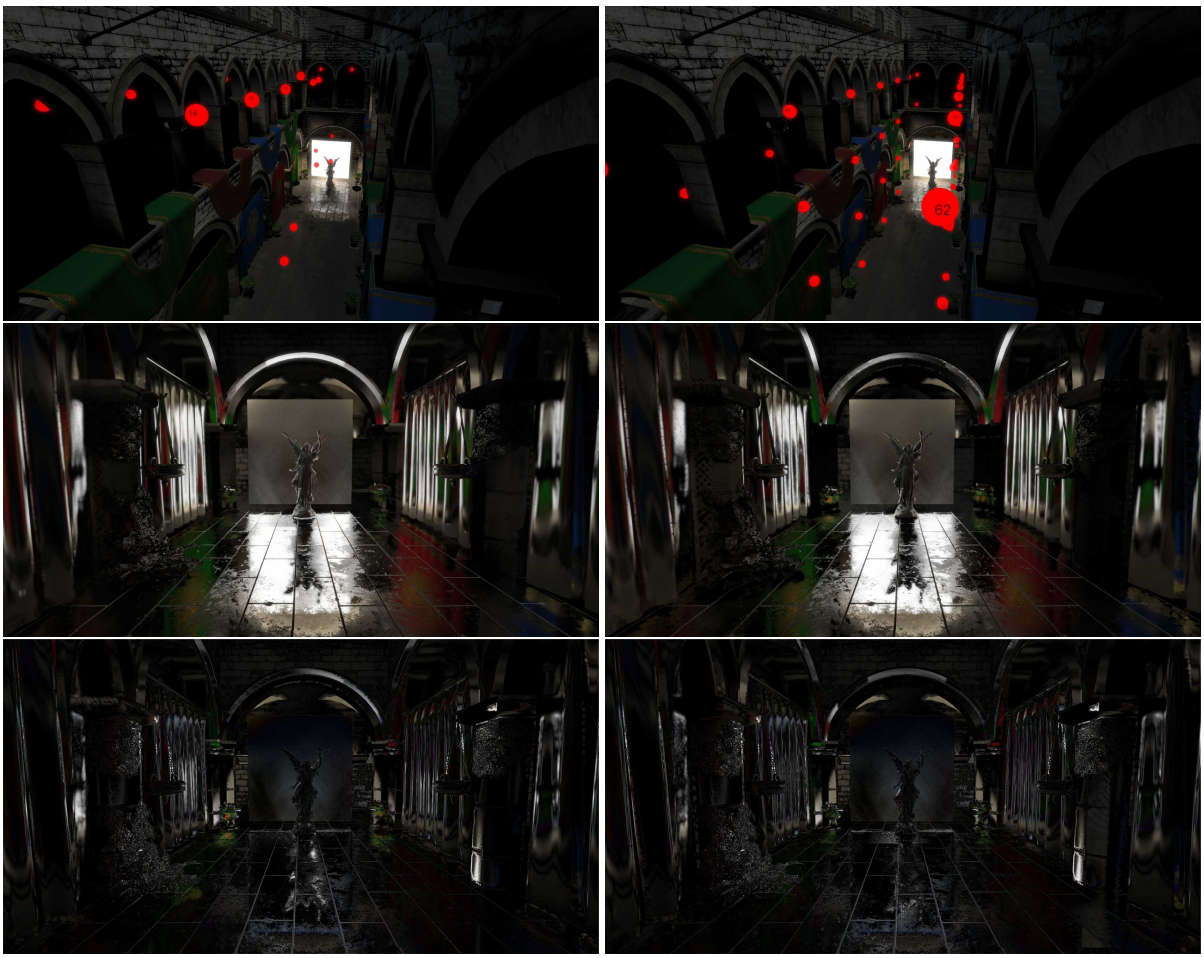
the corridor for capturing the statue and other geometry, one for each side behind curtains. However, none of them could fully capture the occlusion of the statue and the area light. The reason is that the regular grid aligned all probes with the axes, which limited the view directions towards certain geometry. In Sponza, if we would like to achieve full coverage and good rendering results with no obvious artifacts, a minimum resolution of a $8 \times 4 \times 4$ grid is needed, as shown in Figure 5.4.

5.1.2 Same Number of Probes

Iconic Temple

It is tricky to limit the number of probes to the same number as non-uniform while still achieving a good coverage, especially when there are irregular shapes like stairs. In the temple, four probes are placed under the stairs, which are useless for rendering (second row of Figure 5.5). This is unavoidable when using regular grids if we need probes above the stairs.

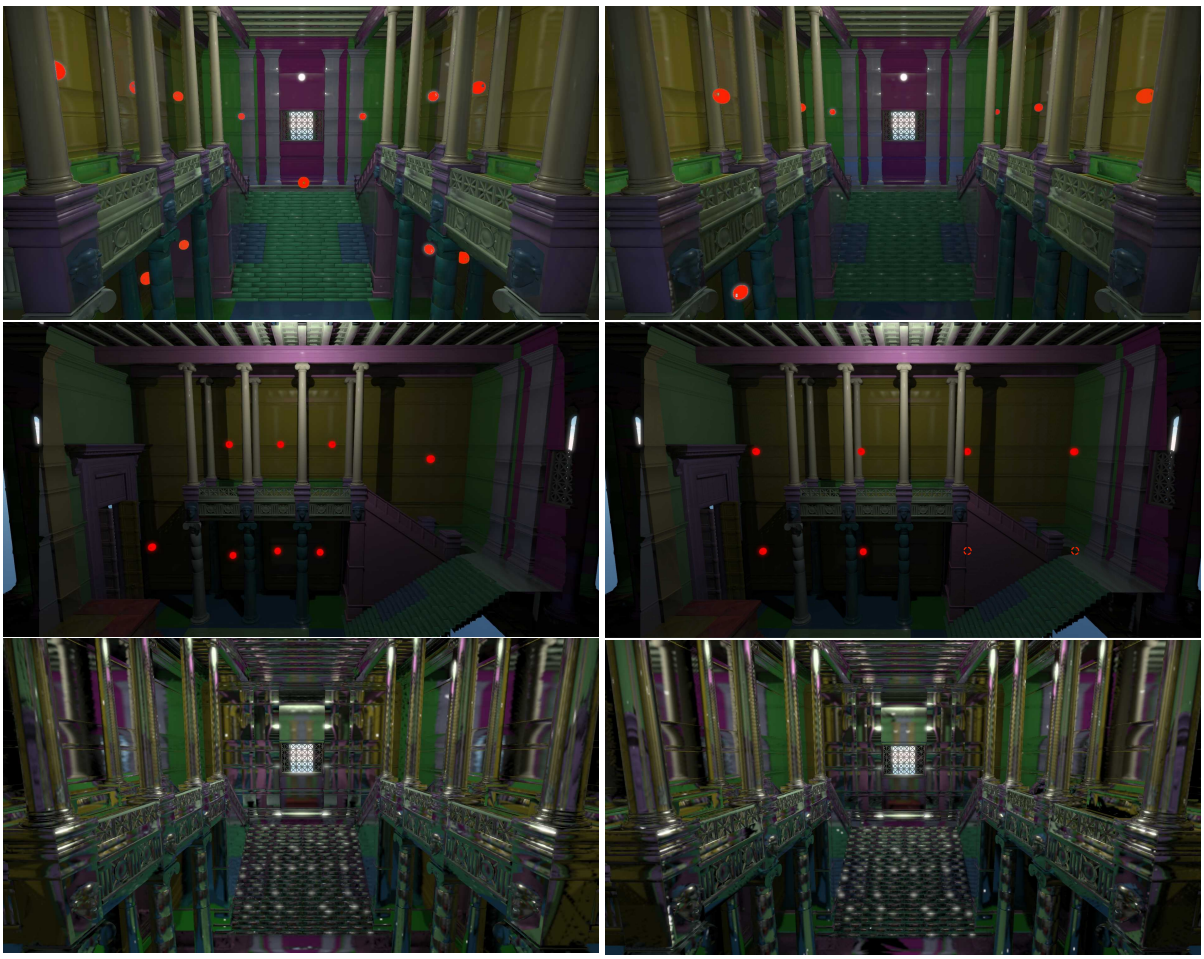
Only placing probes on both sides cannot ensure full coverage of the surfaces in the center corridor. Therefore, there are artifacts due to information missing when we look down on the floor.



(A) Non-uniform placement
64 probes,
render time 23.2ms,
MSE 1.10×10^{-2}

(B) Regular grids
$8 \times 4 \times 4 = 128$ probes,
render time 30.0ms,
MSE 1.05×10^{-2}

FIGURE 5.4: Comparison of Crytek Sponza with the same visual rendering quality using non-uniform and regular grid placement, in filtered glossy components (second row) and error visualization (third row). Note that when the mean squared error is similar, our non-uniform placement uses fewer probes to achieve the same quality.



(A) Non-uniform placement
16 probes,
render time 17.3ms,
MSE 2.78×10^{-3}

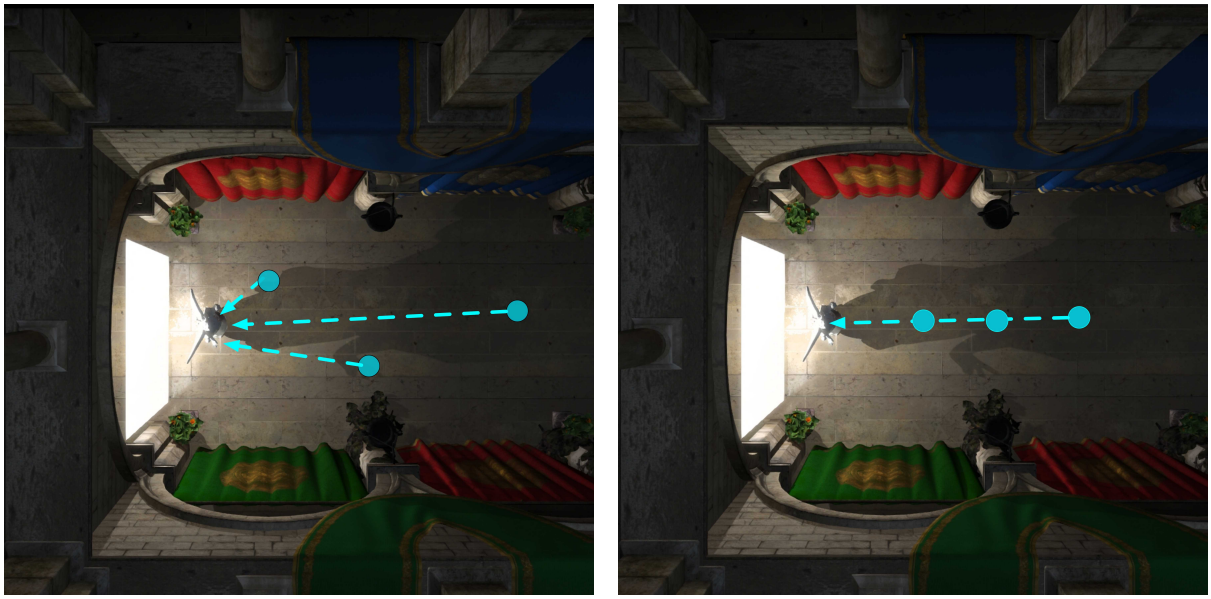
(B) Regular grids
$2 \times 2 \times 4 = 16$ probes,
render time 32.5ms,
MSE 2.93×10^{-3}

FIGURE 5.5: Comparison of the iconic temple Ubisoft© with the same number of probes using non-uniform and regular grid placement, in filtered glossy component (the third row).

(A) Non-uniform placement
54 probes,
render time 19.0ms,
MSE 1.27×10^{-2}

(B) Regular grids
$6 \times 3 \times 3 = 54$ probes,
render time 24.1ms,
MSE 4.14×10^{-2}

FIGURE 5.6: Comparison of visual quality between the same number of probes using non-uniform and in regular grid placement, in filtered glossy component (second row) and error visualization (third row). Note that our non-uniform placement achieves lower mean squared error and fewer visual artifacts.



(A) Non-uniform placed probes can see different parts of the statue.

(B) All uniformly placed probes have the same viewport towards the statue.

FIGURE 5.7: Comparison of coverage over the statue from top view of Crytek Sponza.

Crytek Sponza

In Figure 5.6 we placed 54 probes to cover the scene, which is not possible to generate the same quality image with the same number of probes using regular grids (a $6 \times 3 \times 3$ grid is shown as an example). For full coverage of the scene, at least four vertical layers are needed. Beside occlusions of the statue that is not fully captured, we also observed artifacts due to the thickness and grazing angles with $6 \times 3 \times 3$ grid. Our non-uniform placement alleviates this problem by adding randomness into the probe position adjustment process, so that although there is only one vertical layer of probes in the corridor, we still have a diversity of viewports towards the statue and area light. The illustration of comparison is shown in Figure 5.7.

5.2 Performance

Table 5.1 shows the rendering time of the indirect glossy component. We render each scene at a 1900×1080 resolution. Since the glossy component depends on the number of glossy surfaces, we render all surfaces as glossy.

We have better performance using non-uniform placement, in comparison with the original algorithm. The Hi-Z ray tracer further accelerates the tracing process. Our final algorithm saves roughly 30% to 53% of the rendering time of the original algorithm with the same visual quality.

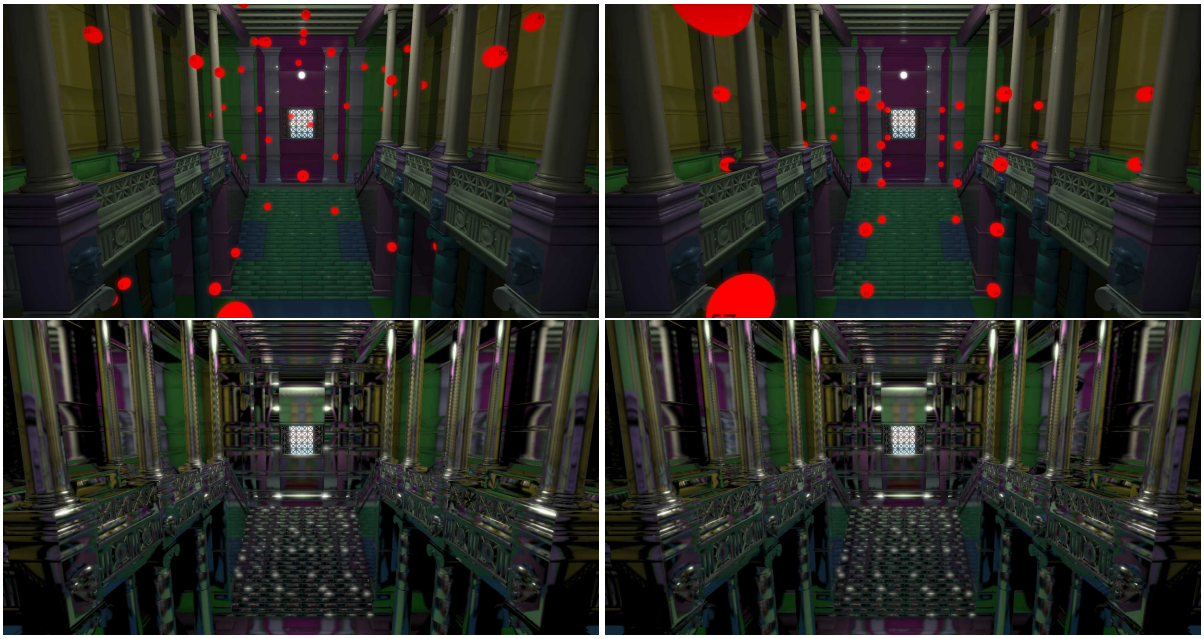
Scene	Probes	Trace Algo	Performance
Cornell Box	$2 \times 2 \times 2$ grid	Original	11.3
	$2 \times 2 \times 2$ grid	Hi-Z	8.2
	7 non-uniform	Original	7.8
	7 non-uniform	Hi-Z	**6.1**
Iconic Temple	$2 \times 2 \times 4$ grid	Original	32.5
	$4 \times 2 \times 4$ grid	Original	27.7
	16 non-uniform	Hi-Z	**17.3**
Crytek Sponza	$8 \times 4 \times 4$ grid	Original	30.0
	64 non-uniform	Original	23.1
	64 non-uniform	Hi-Z	**24.8**
Crytek Sponza	$6 \times 3 \times 3$ grid	Original	24.1
	54 non-uniform	Original	20.6
	54 non-uniform	Hi-Z	**19.0**

TABLE 5.1: Performance in Milliseconds of Different Placement.

Generally speaking, our algorithm could maintain a good performance even if we use dense non-uniform probe settings. Since our run-time probe selection takes visibility into account, we can save time on jumping between useless probes. Due to the same reason, regular grids do not generalize well when visibility complexity increases, such as the decoration between pillars in the iconic temple, because the nearest probe does not necessarily know the surface we are computing, it tends to fail in the nearest probes then re-project the ray to new probes. For example, the nearest probe for the statue could be behind the curtain or the area light. Switching between probes creates extra cost and increases the chance of potential cache misses, which tends to slow down the algorithm.

Besides ray tracing, the most time-consuming component in our runtime algorithm is to traverse down the voxel octree until reaching the leaf to find a probe to use. This can be improved by encoding more efficient acceleration structures, such as bounding volume hierarchy (BVH). However, we observe that even traversing octree is costly. We still win several milliseconds (comparing the performance of Sponza $6 \times 3 \times 3$ grid and 54 non-uniform grid, both using original two-resolution ray marching and hi-Z ray marching) by not wasting time tracing probes that are not visible to a current ray.

For the pre-processing part, generating the skeleton either from the medial axis or the signed distance functions (SDF) takes several seconds to a minute for a given scene, depending on the resolution and complexity of the geometry. Baking sparse voxel tree (SVO) only takes several seconds at loading time. The time for baking probes vary from seconds to minutes depending on the number of probes and resolution.



(A) Non-uniform placement
79 probes,
render time 29.7ms,
MSE 2.51×10^{-3}

(B) Regular grids
$4 \times 4 \times 4 = 16$ probes,
render time 31.3ms,
MSE 2.50×10^{-3}

FIGURE 5.8: Comparison of visual quality of the iconic temple Ubisoft© between using non-uniform and regular grid placements, in filtered glossy component (second row). Note that to reach the same performance as a dense grid, we use unnecessarily dense sampling for non-uniform placement to achieve similar quality.

We compare the visual quality of rendering results using regular grids and non-uniform probe placements for the same performance. One observation is that there are lower limits for computing the glossy component given a fixed resolution and a camera configuration because the number of rays is fixed. Each ray needs to traverse the grid to find a hit or to trace to infinity, the number of steps being usually within a certain range. Here we push performance to the limits using extreme cases: we test the temple in very dense probe settings using both regular grids and non-uniform placements, whereas benchmark Sponza in sparse probe settings.

Iconic Temple

We choose a dense non-uniform probe setting that generated from the medial axis to match performance of regular grids. As shown in Figure 5.8, non-uniform 79 probes still takes less time to render the glossy component. Although our mean squared error is slightly higher, we have fewer visual artifacts compared to a regular grid of $4 \times 4 \times 4$.

Scene	# Probes	Distance Probes	Other Textures	SVO	Total
Cornell Box	non-uniform 7	9.33	12.25	15.41	36.99
	regular grid 8	4.57	14.88	–	19.45
Iconic Temple	non-uniform 16	21.33	28	61.21	110.54
	regular grid 24	13.71	59.52	–	73.23
Crytek Sponza	non-uniform 54	72	94	243.23	409.23
	regular grid 128	146.29	238.08	–	384.37

TABLE 5.2: Memory usage (in MB) with same quality.

Crytek Sponza

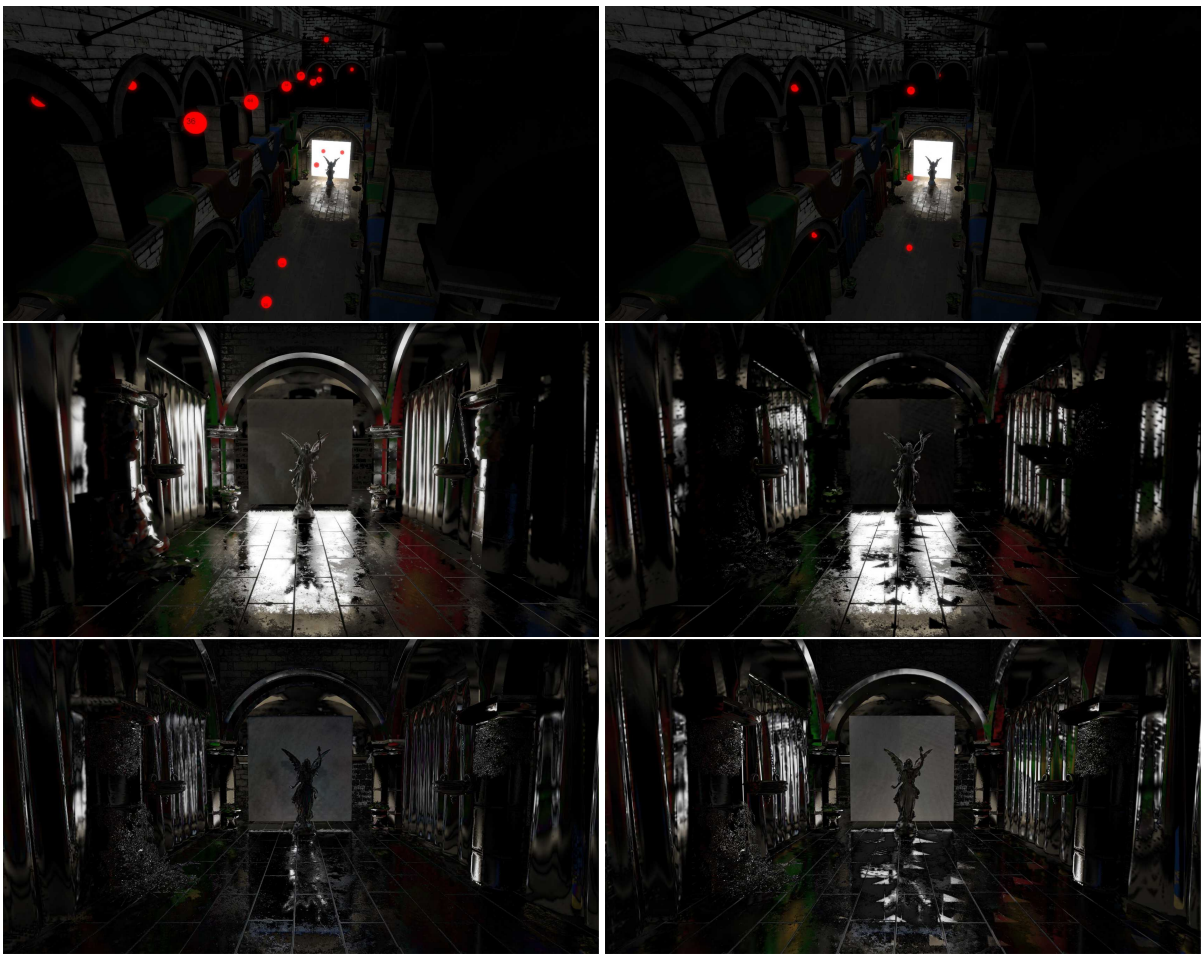
To achieve the same performance as our sparsest non-uniform probe setting (minimal number of 54 probes), we reduced the grid resolution to $3 \times 3 \times 3$. As shown in Figure 5.9, there are unacceptable artifacts due to poor coverage in regular grids.

5.3 Memory Footprint

Generally, our algorithm uses more memory than the original algorithm. This is because probe positions cannot be computed by a simple division of the grid size for non-uniform placement. We store probe indices in a sparse voxel octree, which takes most of the memory usage of our algorithm. Depending on the resolution and depth of octree, memory usage varies. For storing probe indices in sparse voxel octree, we use the RGBA8 format and each voxel has four probe candidates baked.

For probe textures we use a 512×512 resolution for all test scenes. The original algorithm encodes radial distance into the R16F format. In order to use hi-Z ray tracing, we mipmap distance probes into the RG16F format. The R channel for the minimum depth and the G channel for the maximum depth. The memory usage is listed in Table 5.2. We show distance probes and sparse voxel octree in separate columns since these are differences between the two algorithms. Other probe textures such as radiance and irradiance probes are shown in the *Other Textures* column. The texture format is the same, the difference of memory usage is due to different number of probes.

We also baked probe positions into a 1D texture. This is usually a few KB that we added to *Other Textures*.



(A) Non-uniform placement
54 probes,
render time 20.6ms,
MSE 1.27×10^{-2}

(B) Regular grids
$3 \times 3 \times 3 = 27$ probes,
render time 20.0ms,
MSE 2.41×10^{-2}

FIGURE 5.9: Comparison of visual quality between using non-uniform and in a regular grid placement with the same performance, in filtered glossy component (second row) and error visualization (third row). Note that our non-uniform placement generates higher quality results and fewer artifacts with similar performance.

Chapter 6

Conclusion and Future Work

We conclude our work and discuss directions for future work.

6.1 Conclusion

Our contributions can be summarized in two parts.

Firstly, we provided a new approach to place radiance probes in a non-uniform way, which can identify important positions that maximize visibility and is good for light field ray tracing. This approach can be extended to open space with a bounding volume intersected.

For run-time, we provided a way to organize probes that is convenient for our light-field ray tracer, and adapted light-field ray tracing to non-uniform placement. We also improved run-time performance for ray tracing using non-uniform probes by using hi-Z tracing.

In all test scenes, our solution achieves higher rendering quality with fewer probes (in comparison with path traced references) and maintains better performance than regular grids. Our non-uniform probe settings can capture much more precise indirect glossy reflection compared with the same number of probes in regular grids.

6.2 Future Work

We would like to continue this work for both pre-computation and run-time performance.

6.2.1 Probe Placement and Optimization

Analytic Visibility

In order to better identify interesting positions for placement, we can compute the visibility from probe to surface analytically. Arvo [1995] introduces an analytic method

that can be used to compute visibility for any given surface to the light source.

Visibility Atlas

In order to evaluate the visibility of each probe, we unwrap the geometry of a 3D scene into an atlas. However, for scenes at large scales with complex geometry, mapping all surfaces into a single texture atlas will lose a lot of details. Therefore we would like to separate geometry, and map each part into a local atlas so that all surfaces can be fully represented at a unified scale and preserve the details of geometry in an atlas.

6.2.2 Run-time Rendering Algorithm

Run-time Probe Selection

Selecting the most promising probe for any given ray is important at run-time. During a ray tracing process, there could be multiple probe selection queries, depending on the complexity of the scene. Therefore we need this strategy to be efficient and GPU friendly. In our current algorithm, we use a sparse voxel tree that encodes light probe indices in each voxel. However, the sparse voxel tree is stored in a RGBA 3D texture, and due to the sparsity of storage, a lot of space is left empty. We need to further optimize the memory usage of an octree since we only store data in leaves. For run-time, traversing down an octree involves reading children index buffer and computing correct nodes, this operation can be expensive for large scenes because the depth of a tree is greater if we want to keep the same resolution as our testing scenes. We are seeking a different representation of probe structures that can save all the memory, for example, bounding volume hierarchy (BVH) can be used to approximate the range of each probe for selection.

Cone Tracing

Although we applied hi-Z ray tracing, we do not apply the cone tracing algorithm. In probe texture, the squared mapped octahedral texture is sampled from six faces of a cubemap. Because we bake radial distances to probes, which cannot be used to pre-convolve visibility, we constructed visibility from real depths for each cube face. However, constructing a mipmap of a radiance octahedral texture for run-time sampling is not trivial, since we need to sew the seams of an octahedron together to have the correct cone projected, as shown in Figure 6.1. The cone constructed at the hit texel requires a large radius, where we need to mipmap the value correctly based on which edges are connected.

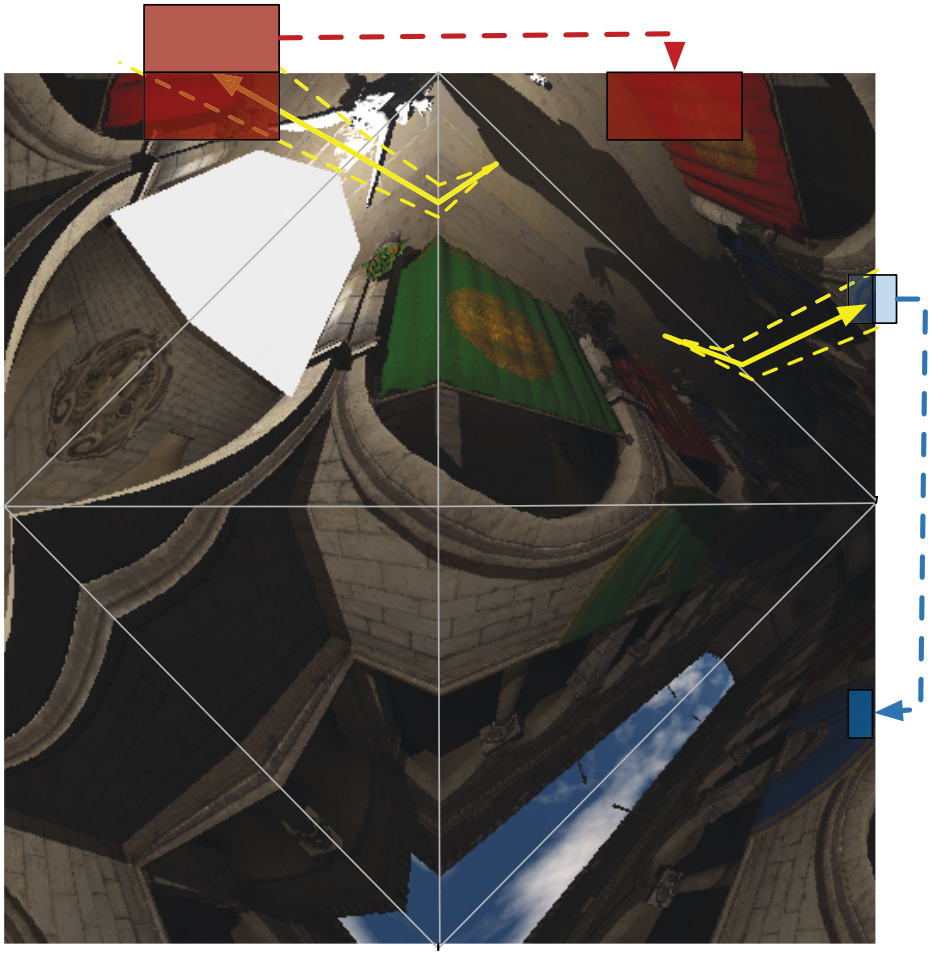


FIGURE 6.1: Correct mipmap radiance probe texture. Two rays (in yellow arrows) traced in the probe texture of Crytek Sponza, and the hit texels lie on the edges of the octahedron. To correctly mipmap the texels on boundaries of an octahedral texture, we need to consider the pixels that are reflective symmetric with respect to the midline of the boundary, as shown in red and blue.

Appendix A

Appendix

A.1 Probe Placement Settings

The following probe positions are generated from clustering medial axes. With different cluster parameters, we get probe positions with different densities.

A.1.1 Iconic Temple

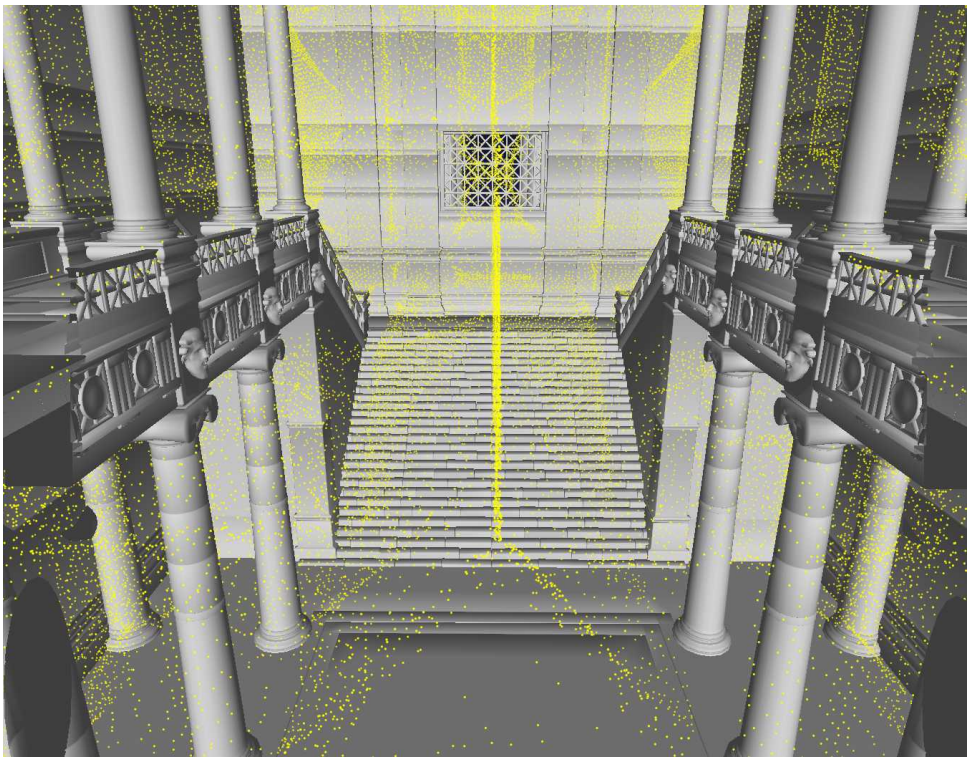


FIGURE A.1: Medial axis of the iconic temple Ubisoft©.

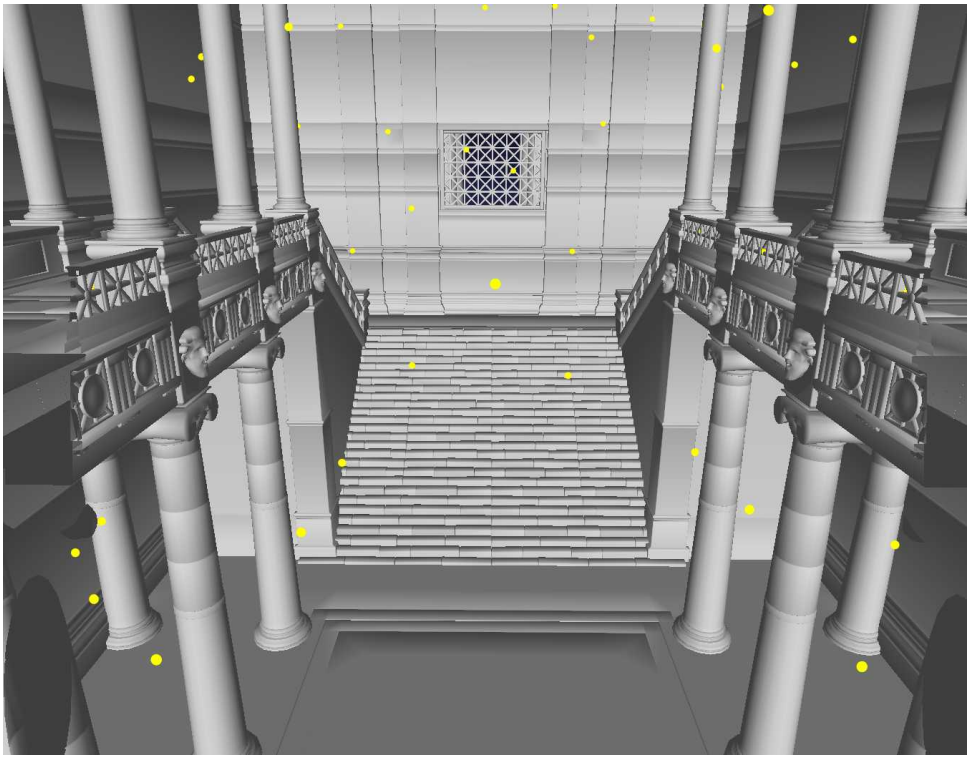


FIGURE A.2: Visualization of non-uniform 63 probes in the iconic temple Ubisoft©

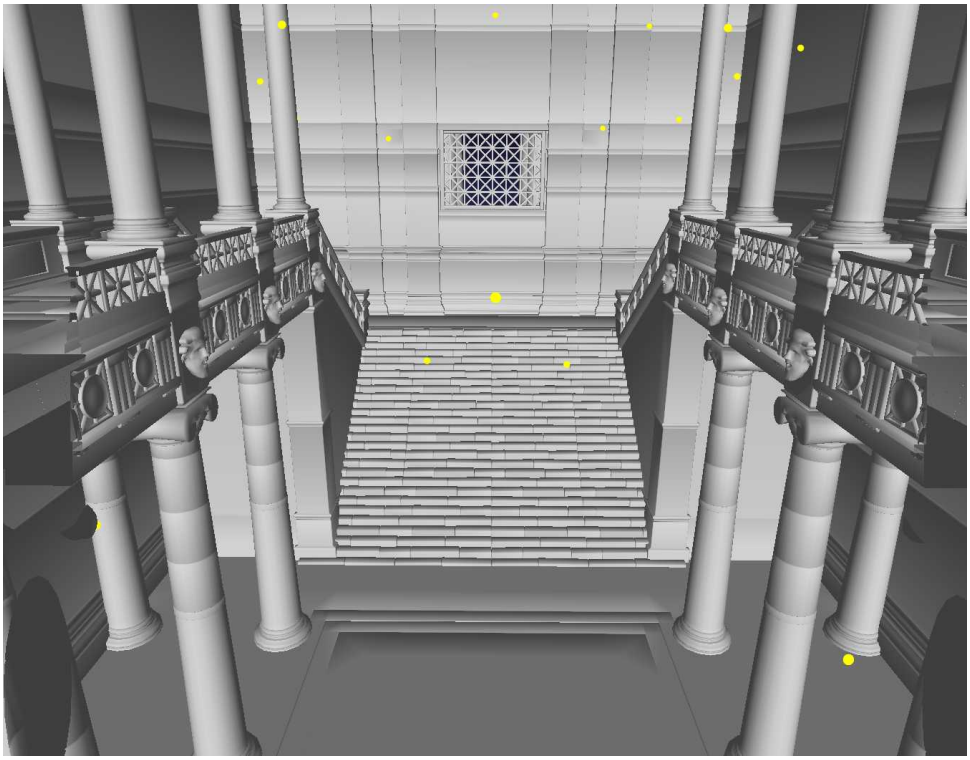


FIGURE A.3: Visualization of non-uniform 48 probes in the iconic temple Ubisoft©.

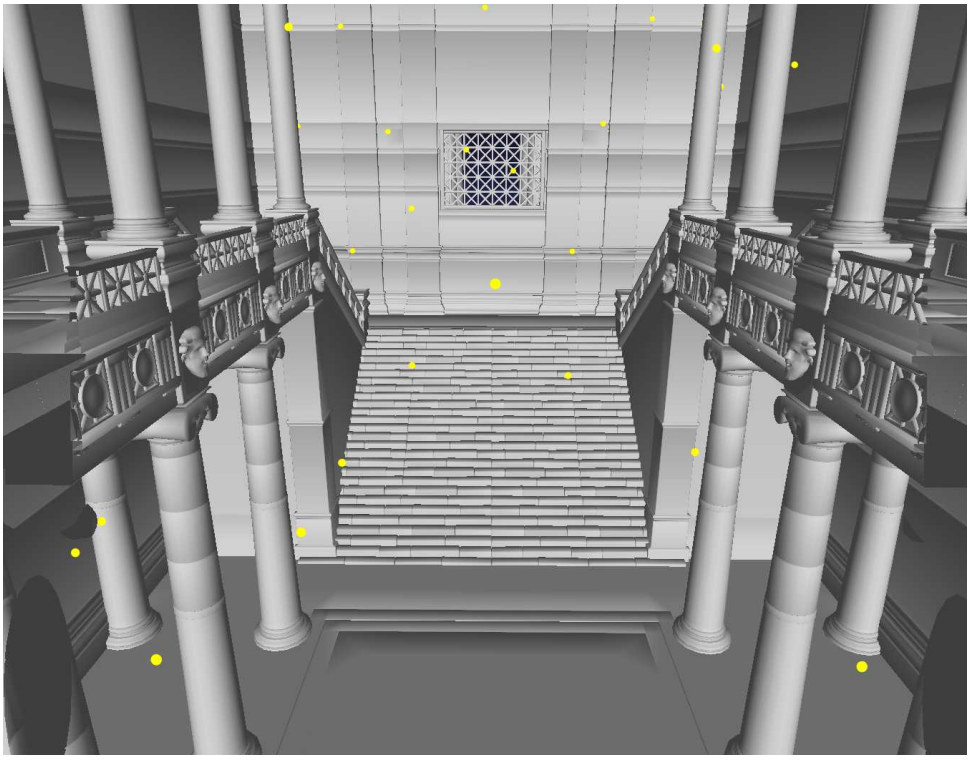


FIGURE A.4: Visualization of non-uniform 43 probes in the iconic temple Ubisoft©.

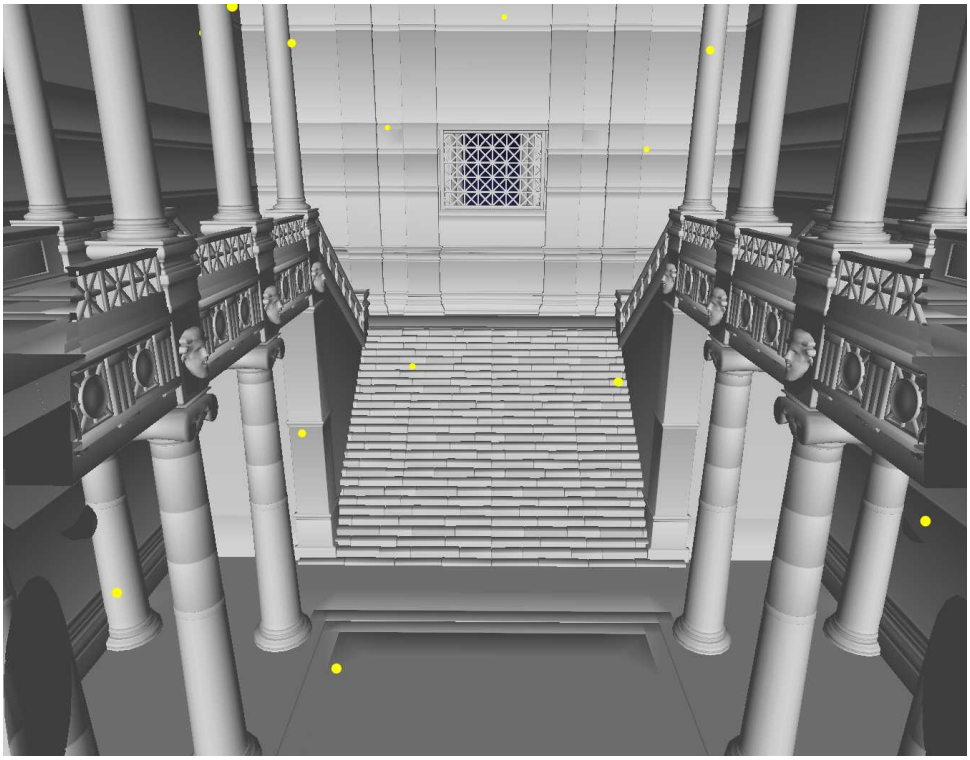


FIGURE A.5: Visualization of non-uniform 36 probes in the iconic temple Ubisoft©.

A.1.2 Crytek Sponza

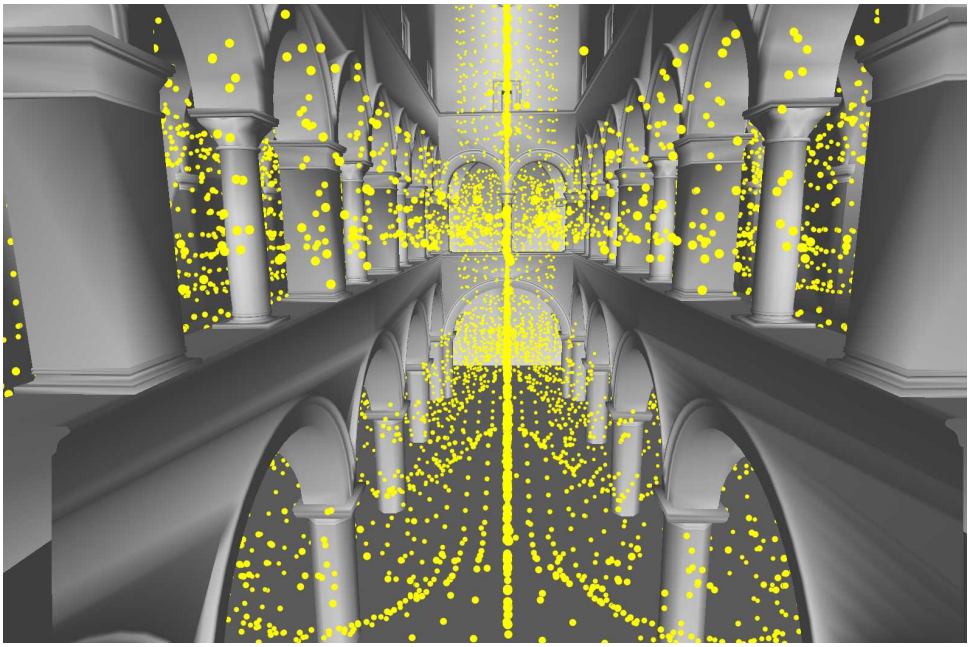


FIGURE A.6: Medial axis of Crytek Sponza.

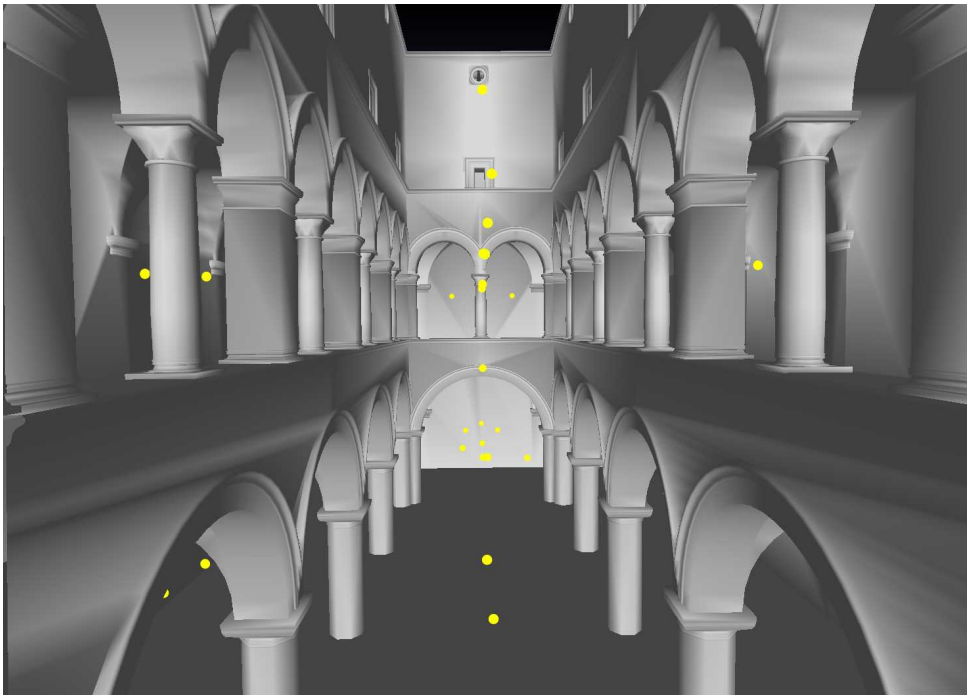


FIGURE A.7: Visualization of non-uniform 87 probes in Sponza

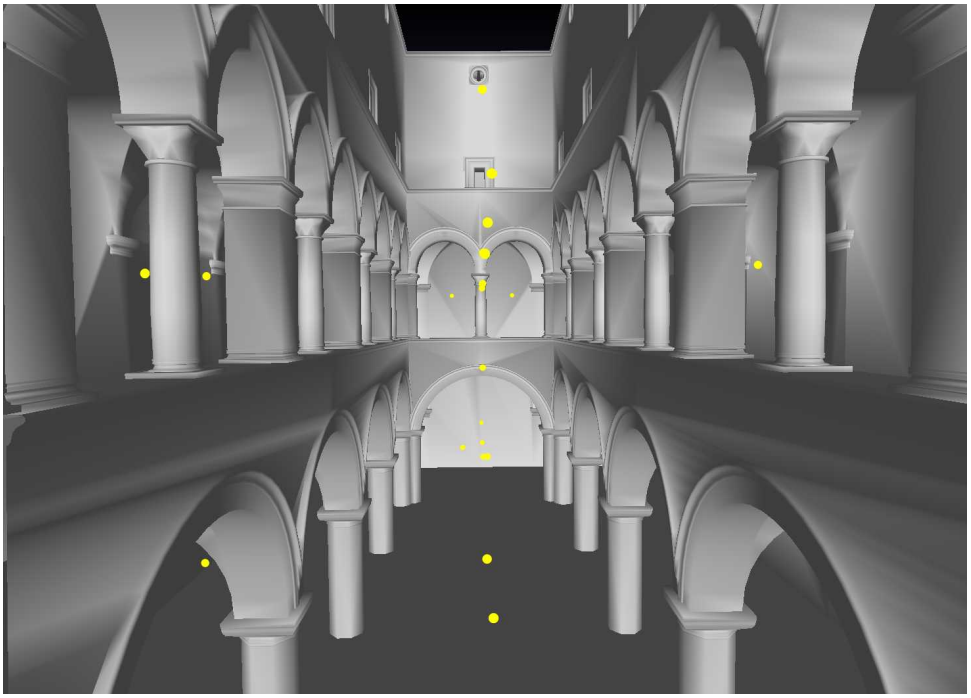


FIGURE A.8: Visualization of non-uniform 74 probes in Sponza.

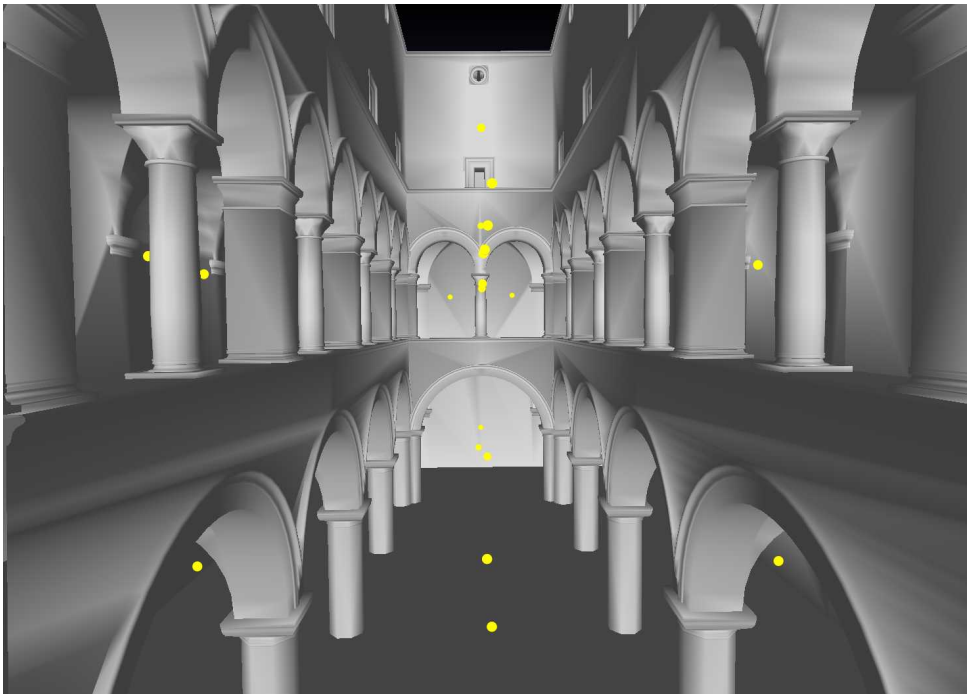


FIGURE A.9: Visualization of non-uniform 64 probes in Sponza.

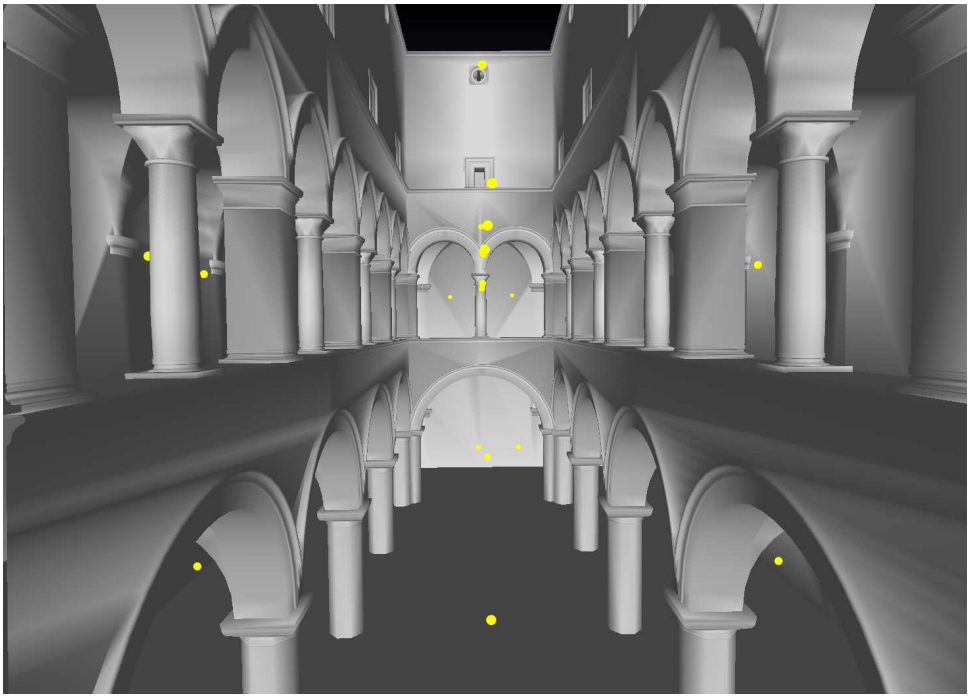


FIGURE A.10: Visualization of non-uniform 54 probes in Sponza.

A.2 Ground truth

A.2.1 Cornell Box

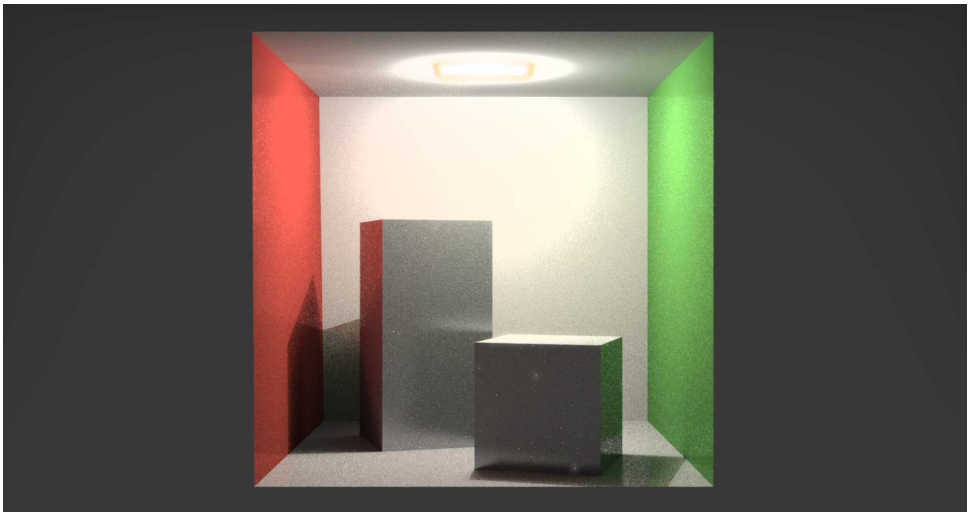


FIGURE A.11: Global illumination with one-bounce indirect light.

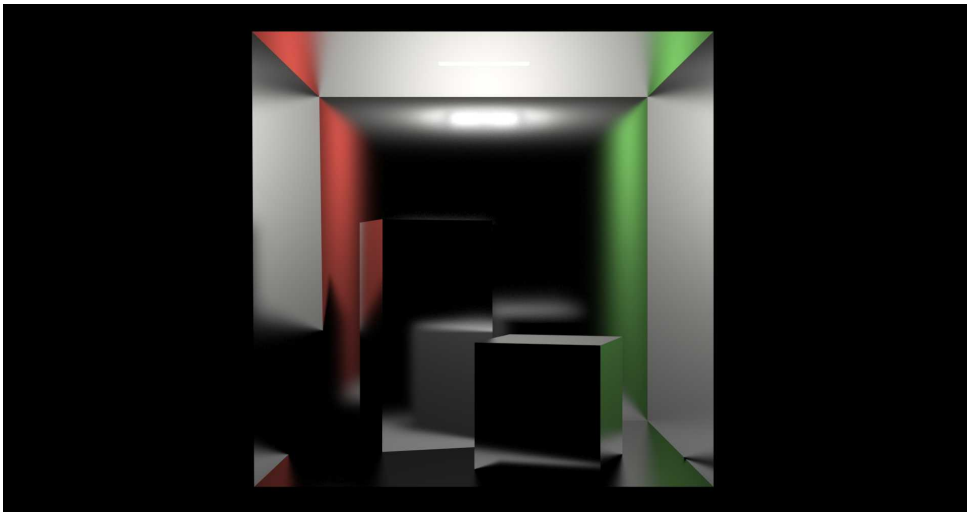


FIGURE A.12: One-bounce indirect glossy component.

A.2.2 Iconic Temple

FIGURE A.13: Global illumination with one-bounce indirect light.

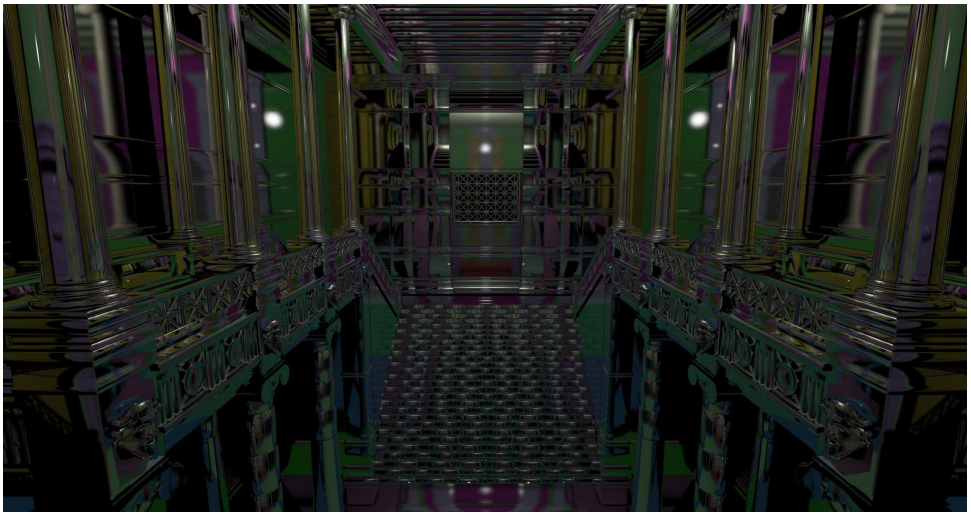


FIGURE A.14: One-bounce indirect glossy component.

A.2.3 Crytek Sponza

FIGURE A.15: Global illumination with one-bounce indirect light.

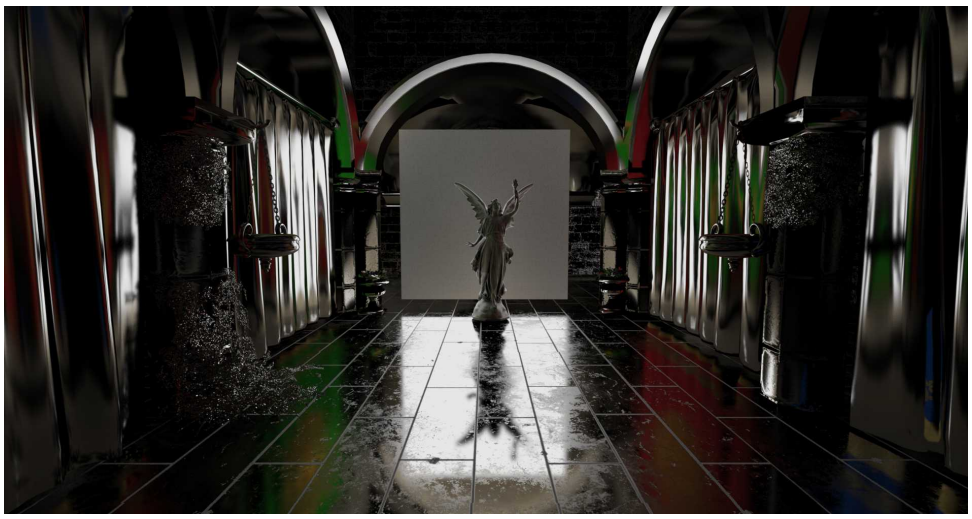
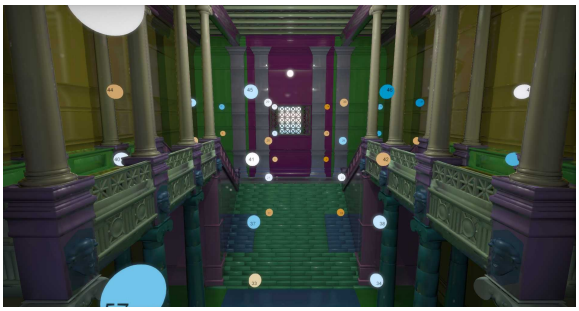


FIGURE A.16: One-bounce indirect glossy component.

A.3 Comparison With Dense Grids

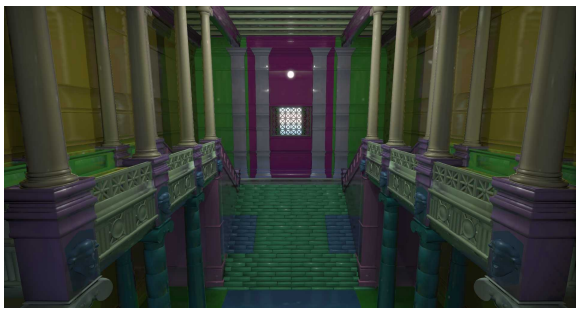
A.3.1 Iconic Temple

(A) visualization of 16 probe positions.

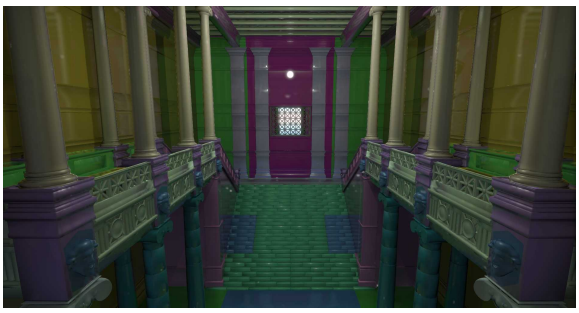


(B) visualization of 4 x 4 x 4 probe positions.

FIGURE A.17: Visualization of the probe positions, non-uniform and dense grid in the iconic temple.

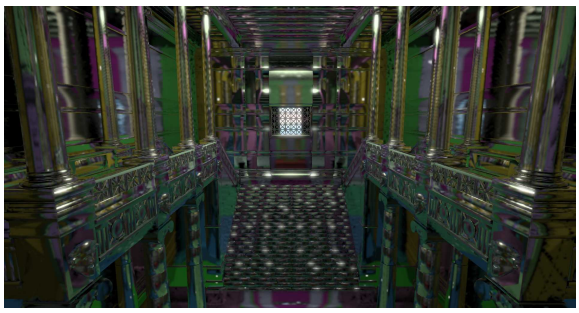


(A) Rendering result using non-uniform 16 probes.

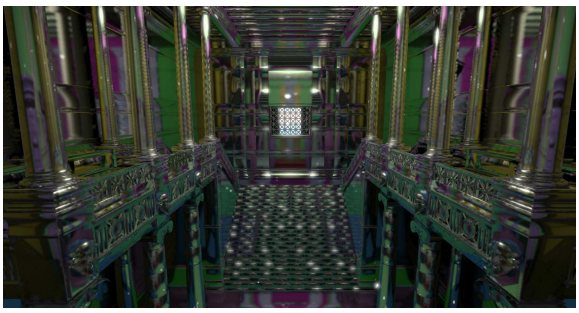


(B) Rendering results using regular grids $4 \times 4 \times 4$.

FIGURE A.18: Comparison of rendering results using non-uniform placed probes and a dense grid in the iconic temple.

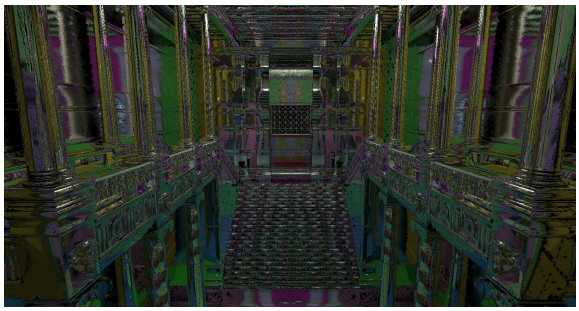


(A) Filtered glossy component, non-uniform 54 probes.

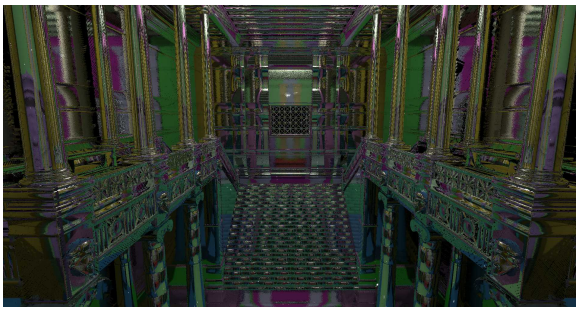


(B) Filtered glossy component, regular grids $4 \times 4 \times 4$.

FIGURE A.19: Comparison of the filtered glossy component using non-uniform placed probes and dense grid in the iconic temple.



(A) Glossy component, one sampler per pixel, non-uniform 16 probes

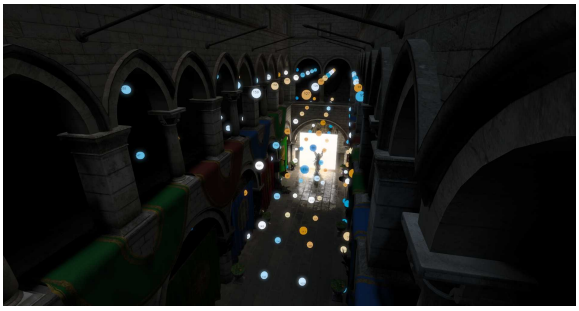


(B) Glossy component, one sample per pixel, regular grid $4 \times 4 \times 4$

FIGURE A.20: Comparison of per-frame glossy component using non-uniform placed probes and dense grid in the iconic temple

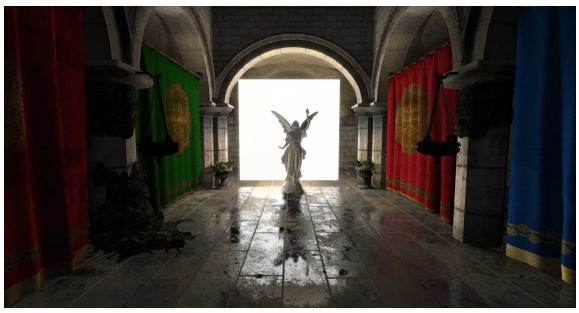
A.3.2 Crytek Sponza

(A) Visualization of 54 probe positions.



(B) Visualization of $16 \times 8 \times 4$ probe positions.

FIGURE A.21: Visualization of the probe positions, non-uniform and dense grid in Sponza.



(A) Rendering result using non-uniform 54 probes.

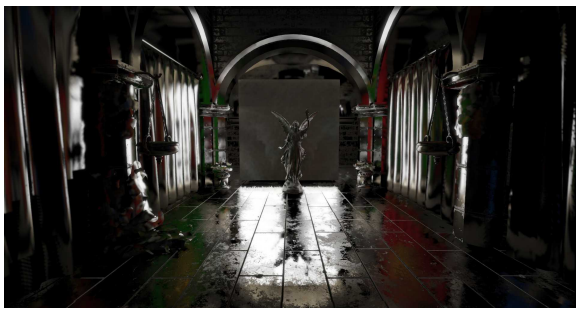
(B) Rendering result using regular grids $16 \times 8 \times 4$.

FIGURE A.22: Comparison of rendering result using non-uniform placed probes and dense grid in Sponza.

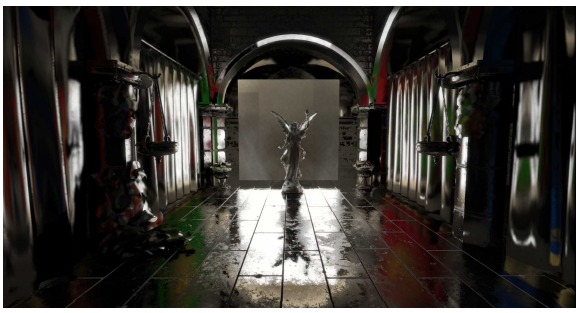
(A) Indirect lighting, one sampler per pixel, non-uniform 54 probes.

(B) Indirect light, one sample per pixel, regular grid $16 \times 8 \times 4$.

FIGURE A.23: Comparison of the indirect component using non-uniform placed probes and dense grid in Sponza.

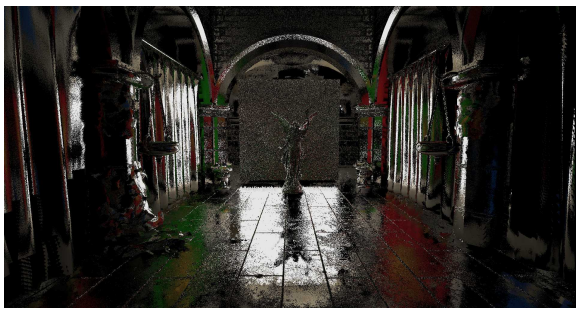


(A) Filtered glossy component, non-uniform 54 probes.

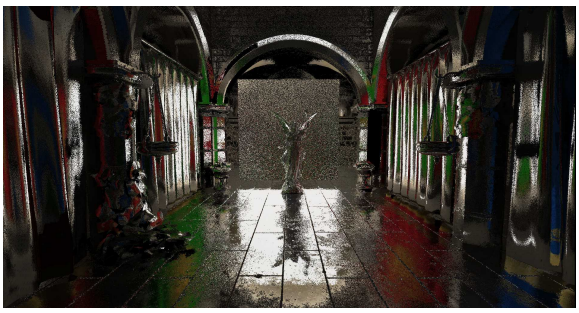


(B) Filtered glossy component, regular grids $16 \times 8 \times 4$.

FIGURE A.24: Comparison of filtered glossy component using non-uniform placed probes and dense grid in Sponza.



(A) Glossy component, one sampler per pixel, non-uniform 54 probes.



(B) Glossy component, one sample per pixel, regular grid $16 \times 8 \times 4$.

FIGURE A.25: Comparison of per-frame glossy component using non-uniform placed probes and dense grid in Sponza.

References

J. Arvo. *Analytic methods for simulated light transport*. PhD thesis, PhD thesis, Yale University, 1995.

A. Bagheri and M. Razzazi. Drawing free trees inside simple polygons using polygon skeleton. *Computing and Informatics*, 23(3):239–254, 2012.

G. Barequet, D. Eppstein, M. T. Goodrich, and A. Vaxman. Straight skeletons of three-dimensional polyhedra. In *European Symposium on Algorithms*, pages 148–160. Springer, 2008.

J. Bowald. Global illumination for static and dynamic objects using light probes. Master's thesis, Chalmers University of Technology, 2016.

N. Burger. Realtime interactive architectural visualization using unreal engine 3.5, 2013.

G. Casella and E. I. George. Explaining the gibbs sampler. *The American Statistician*, 46 (3):167–174, 1992.

M. G. Chajdas, A. Weis, and R. Westermann. Assisted environment map probe placement. In *Proceedings of SIGRAD 2011. Evaluations of Graphics and Visualization - Efficiency; Usefulness; Accessibility; Usability; November 17-18; 2011; KTH; Stockholm; Sweden*, number 65, pages 17–25. Linköping University Electronic Press; Linköpings universitet, 2011.

H. Cheng, S. L. Devadoss, B. Li, and A. Risteski. Skeletal configurations of ribbon trees. *Discrete Applied Mathematics*, 170:46–54, 2014.

S. Chib and E. Greenberg. Understanding the metropolis-hastings algorithm. *The american statistician*, 49(4):327–335, 1995.

C. Crassin and S. Green. Octree-based sparse voxelization using the gpu hardware rasterizer. In *OpenGL Insights*, pages 303–318. CRC Press, Patrick Cozzi and Christophe Riccio, 2012. URL http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-SparseVoxelization.pdf,ChapterPDF.

C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, volume 30, pages 1921–1930. Wiley Online Library, 2011.

R. Cupisz. Light probe interpolation using tetrahedral tessellations. In *Game Developers Conference (GDC)*, 2012.

M. Drobot. Quadtree displacement mapping with height blending. In *GPU Pro 360 Guide to Rendering*, pages 1–32. AK Peters/CRC Press, 2018.

M. Gilabert and N. Stefanov. Deferred radiance transfer volumes, global illumination in Far Cry 3. In *Game Developers Conference*, 2012.

G. Greger, P. Shirley, P. M. Hubbard, and D. P. Greenberg. The irradiance volume. *IEEE Computer Graphics and Applications*, 18(2):32–43, 1998.

J. Jendersie, D. Kuri, and T. Grosch. Precomputed illuminance composition for real-time global illumination. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '16, pages 129–137, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4043-4. doi: 10.1145/2856400.2856407. URL http://doi.acm.org/10.1145/2856400.2856407.

T. Kelly, J. Femiani, P. Wonka, and N. J. Mitra. Bigsur: Large-scale structured urban reconstruction. *ACM Trans. Graph.*, 36(6):204:1–204:16, Nov. 2017. ISSN 0730-0301. doi: 10.1145/3130800.3130823. URL http://doi.acm.org/10.1145/3130800.3130823.

J. Ma, S. W. Bae, and S. Choi. 3d medial axis point approximation using nearest neighbors and the normal field. *The Visual Computer*, 28(1):7–19, 2012.

M. McGuire and M. Mara. The G3D innovation engine, 01 2017. URL https://casual-effects.com/g3d/. https://casual-effects.com/g3d/.

M. McGuire, M. Mara, D. Nowrouzezahrai, and D. Luebke. Real-time global illumination using precomputed light field probes. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '17, pages 2:1–2:11, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4886-7. doi: 10.1145/3023368.3023378. URL http://doi.acm.org/10.1145/3023368.3023378.

J. Mortensen. Global illumination in unity 5. *Haettu osoitteesta: http://blogs. unity3d. com/2014/09/18/globalillumination-in-unity-5/(luettu 17.03. 2016)*, 2014.

R. Ramamoorthi and P. Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of the 28th Annual Conference on Computer Graphics and*

Interactive Techniques, SIGGRAPH '01, pages 497–500, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: 10.1145/383259.383317. URL http://doi.acm.org/10.1145/383259.383317.

L. Sébastien and A. Zanuttini. Local image-based lighting with parallax-corrected cubemaps. In *ACM SIGGRAPH 2012 Talks*, page 36. ACM, 2012.

A. Silvennoinen and J. Lehtinen. Real-time global illumination by precomputed local reconstruction from sparse radiance probes. *ACM Trans. Graph.*, 36(6):230:1–230:13, Nov. 2017. ISSN 0730-0301. doi: 10.1145/3130800.3130852. URL http://doi.acm.org/10.1145/3130800.3130852.

A. Silvennoinen and V. Timonen. Multi-scale global illumination in quantum break. In *SIGGRAPH '15 ACM SIGGRAPH 2015 Courses*. ACM, 2015.

P.-P. Sloan, J. Kautz, and J. Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Trans. Graph.*, 21 (3):527–536, July 2002. ISSN 0730-0301. doi: 10.1145/566654.566612. URL http://doi.acm.org/10.1145/566654.566612.

N. Stefanov. Global illumination in tom clancy's the division. In *Game Developers Conference*, 2016.

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

Y. Uludag. Hi-z screen-space cone-traced reflections. *GPU Pro*, 5:149–192, 2014.

J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz. Real-time concurrent linked list construction on the gpu. In *Computer Graphics Forum*, volume 29, pages 1297–1304. Wiley Online Library, 2010.