Light Probe Selection Algorithms for Real-Time Rendering of Light Fields

by

Samuel Donow

Professor Morgan McGuire, Advisor

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts

May 22, 2016

Contents

List of Figures

List of Algorithms

Abstract

This thesis presents the data structures and algorithms necessary for approximating the plenoptic function in real-time using samples at a low spatial density called light probes. By computing the light into a few points in a scene in many directions, as well as distance from those key points to the scene, we can interpolate in order to reconstruct lighting information for the entire scene. The algorithm presented here is tunable with various heuristics for achieving desired results, as well as extensible, allowing other techniques from Computational Graphics to be brought to bear upon this new framework for answering general ray queries.

Acknowledgments

I would like to thank all those who helped bring this work to fruition. I would like to thank my advisor, Morgan McGuire, for introducing to computer graphics research in general and the questions that led to this thesis in particular. I would like to thank Daniel Evangelakos (Williams '15) for being a helpful partner in research over two Summers, and giving insight from his experiences with his thesis to aid me in mine. Additionally, I would like to thank my second reader Bill Lenhart for his helpful feedback, as well as the other eyes and ears that have been lent to this information, including Michael Mara (Williams '12) and Ward Lopes' 2014 summer research group. I would also like to thank my friends and family, without whom I could not have gotten where I am now, particularly my parents Bart and Renee Donow, and my friends Mia Knowles and Rinna Caldieraro. Finally, I would be remiss if I did not thank Hudson River Trading LLC, without whose offer of employment, the mystery of the future would have made focusing on this thesis more difficult.

Chapter 1

Introduction

Computational Graphics seeks to provide algorithms which take as inputs a description of a 3D scene, which is typically given as geometry, materials, and lights, and information about a camera's position and rotation, and produces as output a 2D image which represents what view the camera has of the scene.

One class of algorithms that have been used to solve this problem are called *Global Illumination* solutions. These tend to use Monte Carlo methods in order to approximate an integral which comes from physical models which gives the light in the scene. Such methods include Metropolis Light Transport [Veach and Guibas, 1997] and Path Tracing [Kajiya, 1986]. The integral below is what Kajiya in 1986 called the Rendering Equation (in a form given here by Immel et al Immel et al. [1986]):

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}) = L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}) + \int_{\Omega} f(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}) |\omega_{\mathrm{i}} \cdot \mathrm{n}| d\omega_{\mathrm{i}}$$

where $L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}})$ is the outgoing light (units of irradiance) at point $\mathbf{x}$ in direction ω_{o}, L_{e} is emitted light at a point in a direction, $\mathbf{n}$ is the normal to the surface at point $\mathbf{x}$, Ω is the unit hemisphere centered at x, f_r is the Bidirectional Reflectance Distribution Function (BRDF), and L_{i} is incident light at point, and ω_{i} is the direction of incident light.

Even for rendering techniques which do not attempt to directly approximate the integral, its form brings to mind the fundamental quantities involved. The function we wish to find values of is $L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}})$. For global illumination, it is often necessary to calculate this function for many points in a scene as the recursion inherent in the integral is expanded. However, always the most critical values occur when $\mathbf{x}$ is the location of the camera and ω_{o} is a direction from the camera through the center of a pixel in the desired output image. Then $L_{\mathrm{o}}(\mathbf{x}, \omega_o)$ gives (after appropriate unit conversions), the color of the pixel corresponding to ω_{o} in the final output image.

This paper presents an algorithm for real-time rendering of 3D graphics scenes through light field rendering [Gortler et al., 1996], [Levoy and Hanrahan, 1996]. The fundamental problem of light field rendering for which our algorithm is a solution is to, given a description of $L_{\mathrm{o}}(X, \omega)$ for only a collection of fixed points X, but all $\omega \in S^2$ up to a certain level of discretization, reconstruct approximations to the value of $L_{\mathrm{o}}(Y, \omega)$ for other points Y in the scene.

Despite the fact that our methods are being here applied to the problem of light field rendering, that term is often used specifically for referring to the data structures used by Gortler in his papers on the Lumigraph or Levoy and Hanrahan's work. Therefore, throughout this document, we will often phrase the problem

in the more general framework of trying to compute the plenoptic function $L_o(X, \omega)$ which determines the irradiance incident to all points from all directions.

In this framework, what we are attempting to do is reconstruct $L_o(X, \omega)$ using discrete samples, where the spatial variable is heavily discretized, while the angular variable is given a large amount of resolution. This formulation is one of sampling and reconstruction as is commonly used with methods that attempt to do direct sampling of the integral in the rendering equation (such as Monte Carlo methods, like Metropolis Light Transport). However, our method of reconstruction is independent of the Rendering Equation, although our estimates of L_o could be used as samples in a more traditional estimate of the integral, allowing an extension that exclusively computed direct illumination, or one which is suitable for path tracing, if desired [Evangelakos, 2015].

While attempting to approximate the plenoptic function $L_o(X, \omega)$, we will refer to our samples, which are functions $L_P(\omega)$ with $L_P(\omega) = L_o(P, \omega)$ as light probes.

The principal tasks of this thesis will be to address the following tasks required for the construction of an algorithm for approximation of the plenoptic function.

- Choose and construct a data structure for the representation of the plenoptic function approximation. Our representation will be as a set of light probes with depth (defined formally in chapter 4). Then, the further steps left to picking out a data structure is choosing a method of representation for these light probes, which at an abstract level are functions on continuous domains. The choice of representation must be optimized not only for efficiency in size and speed, but also to reduce distortion of the data.

- A method for choosing which probe(s) to gather information from. Naively, one might assume that using lighting information from the nearest probe would produce the most accurate results, but indeed more sophisticated methods are needed in order to make sure that lighting results are accurate, both in terms of computing more accurate lighting information based on the resolution of probe data, and in terms of ensuring that a probe has visibility of the point in the scene with the correct data. Further, probe placement issues must be temporally and spatially coherent and be chosen to minimize the creation of artifacts in the rendering process.

- A method for correctly finding answers to ray queries using the data of a single probe must be found. This amounts to being a root-finding algorithm on the distance function represented by a depth probe. Previous light-probe based rendering methods tended to perform this operation either using analytical methods or rough approximations, but as we will be using the light probe for global illumination, we will require doing a full ray trace to obtain accurate information from our probes. Therefore, the choice of this algorithm has large trade-offs between speed and accuracy: environment mapping performs constant time probe lookups with extremely large error, and the optimal algorithm will be one which provides acceptable time costs for real-time rendering, without impermissibly large (or unpredictable) error.

- Develop an algorithm for using multiple probes to answer a single query, which not only computes an ordering of probes to use after a visibility loss occurs, but which can combine the results from multiple probes in a more complex way that simply having each trace on a probe throw out the work done by previous probes.

- Applying the information from the probe lookups in order to efficiently compute various lighting effects commonly used in real-time rendering, such as cone tracing for efficient computation of glossy

reflections.

Chapter 2

Related Work

In this chapter, we will briefly summarize the work of others upon which this thesis builds.

Environment Maps and object based lighting Blinn and Newell first introduced the the notion of an environment map in 1976 [Blinn and Newell, 1976]. They were initially presented as a mechanism for texturing objects (i.e, by computing how reflections map to a surface), in a situation where only one object needs to be textured, and an environment map can be generated for that one object. An environment map is simply a radial map of directions to irradiance. More generally, an environment map can be viewed as a light probe, but upon which queries assume that all light came in from a sphere of infinite radius. This assumption is equivalent to the fact that environment maps are functions only of direction, and are invariant under translation. Formally for a light probe, the incident irradiance L_{o}, for all points X, Y and directions ω, has the property that $L_{\mathrm{o}}(X, \omega) = L_{\mathrm{o}}(Y, \omega)$.

Later Image-Based Lighting techniques [Bjorke, 2004] extended the idea of using an environment map as a light probe, but used a likely more accurate geometric proxy than an infinite sphere for real scenes, such as a fixed side-length cube or fixed-radius sphere. Such an approach leads to more accurate reflections in roughly cube-shaped or sphere-shaped scenes, but will be clearly inaccurate in most scenes, and is thus not suitable if correct results are needed from light probes. This work was extended further [Szirmay-Kalos et al., 2005] to incorporate rich depth information into the light probes, but the methods of Szirmay-Kalos et. al used a fast and heuristic method of root-finding, which is not guaranteed to produce accurate ray-queries, and which, even when given infinite time, may converge upon incorrect results, and so which again is not amenable to a discipled approach to global illumination.

Global illumination Global illumination solutions generally are attempts to directly approximate the Rendering Equation integral [Kajiya, 1986] through Monte-Carlo sampling. Metropolis Light transport does this by reformulating the Rendering Equation as a path integral and sampling whole paths through the scene [Veach and Guibas, 1997], while path tracing samples points in the scene to create paths from light sources to the camera. Rasterization, which is a popular method in real-time rendering, sacrifices the information to do such sampling, as it is an algorithm that operates over pixels, then checks which triangles intersect that pixel, etc, so without specialized techniques it is difficult to capture complex light phenomena that depend on the long, multi-bounce paths than can be captured through Monte Carlo approximation. Global illumination solutions like Path Tracing and Metropolis Light Transport, however, have performance characteristics that make them impractical for real-time rendering at the present time.

Grid Tracing [Musgrave, 1988; Donnelly and Lauritzen, 2006; Amanatides et al., 1987] Grid Tracing,

first introduced by Musgrave in 1988 but extended by later work such as that by Donnelly and is similar to work on voxel traversal by Amantides and Woo, is an algorithm initially designed for ray tracing heightfields. That is, it is an algorithm for determining the intersection between a heightfield (a mesh represented by an array of heights) and a ray. The algorithm is based on the DDA algorithm for efficiently stepping along a ray, and runs in $O(\sqrt{N})$ time, where N is the number of values in the heightfield. Applying this to a light field, we can perform Musgrave Tracing to accurately ray trace our scenes, as a depth probe is essentially a spherical heightfield, and in that case the N above is the resolution of the depth probe image. Musgrave's original paper proposes a hierarchical variant of the algorithm which runs in $O(\log \sqrt{N})$. Such a hierarchical method has been applied to previous Ray-Tracing problems [Stachowiak, 2015; Evangelakos, 2015], though hierarchical methods are not naively well-suited to today's graphics hardware due to the requirement of maintaining some sort of a stack while traversing a tree.

Per-Pixel Displacement Mapping [Donnelly, 2005] Displacement mapping is a technique similar to heightfield tracing. However, it is applied to more generalized distance map that compute the distance from any point to a surface, rather than just height from a plane. Donnelly's work here expressed the problem as a ray-tracing problem, and applies a technique which bears great resemblance to Musgrve Grid Tracing in the more general setting of tracing arbitrary depth maps such as the depth probes considered in this text.

Screen Space Ray Tracing Screen-space ray-tracing [McGuire and Mara, 2014] is a method of ray-tracing that attempts to avoid the under- or oversampling of pixels. A typical ray-tracing algorithm will march along a ray in 3D space using a DDA rasterization algorithm. However, SSRT instead performs a 2D ray trace on the color and depth buffers produced by rasterization. This has the advantage of only sampling the ray at at most once per pixel along the ray, preventing gross oversampling that can occur if a 3D ray trace occurs mostly through empty space.

An optimized data structure for SSRT, which is analogous to the hierarchical optimization of Musgrave Tracing, maintains a hierarchy of axis-aligned bounding boxes and planes, and stores multiple layers of depth against which to trace, in case a ray needs to be traced in a region that would otherwise be occluded [Widmer et al., 2015] . Widmer et al, in their discussion of their optimized data structure, discuss that performing SSRT on a cube map may be ideal for being able to compute reflections that come from outside the screen; the method they develop for this is similar to what would be necessary for light probe tracing on a cube map representation.

Stochastic Screen-Space Reflections Stachowiak's 2015 SIGGRAPH presentation shows how SSRT can be used to accurately capture more general lighting effects than just simple reflections [Stachowiak, 2015]. By stochastically sampling in multiple directions using importance sampling and cone tracing, and weighing the contributions of various pixels using BRDFs, SSRT techniques can be used to calculate glossy reflections for various materials.

Light Field Representation Much of the groundwork for light field rendering was laid out in 1996 by Gortler's paper on the Lumigraph [Gortler et al., 1996], and Levoy and Hanrahan's paper on light field rendering [Levoy and Hanrahan, 1996]. These papers established the general framework of light field rendering, and the general notion of light probe-like data structures used that are applied in the algorithm presented in this paper, though the algorithms in these papers do not closely resemble ours.

A Light Field Representation for Global Illumination. The light field representation used for the algorithm of this paper grew out of previous work in Dan Evangelokos' 2015 undergraduate thesis [Evangelakos, 2015]. In that thesis, the representation is discussed, as are possible parameterizations of the light field data, such as the octahedral mapping [Cigolle et al., 2014]. The paper also gives an algorithm for

performing a hierarchical Musgrave trace on a light field in the octahedral mapping on the CPU, but which is impractical for real-time application on the GPU, and which is set up only as a method of answering ray queries, not as a complete system for global illumination.

View-Based Rendering Pulli's method of view-based rendering takes as input images and a very simple geometric model, and combines the two to render 3D images. Their method involves constructing an actual simple geometric model (a simple triangular mesh), and mapping points from the input images onto this mesh and using traditional computer graphics techniques such as blending and z-buffering in order to improve the result image. Kari Pulli and Stuetzle [1990]

IRay VR Soon before the finalization of this thesis, NVIDIA has announced IRay VR [NVIDIA, 2016], their system for real-time ray tracing for virtual reality (VR) applications. Though the details of their algorithm are not public, what is clear is that it is a light-probe based ray tracing algorithm, and thus bears resemblance to the sort of algorithm proposed in this text. However, based on the only demo of the technology available, it seems as though the algorithm may not be resilient to a large amount of motion of the camera; the demo involves a nearly stationary VR viewer looking around him in a very open room, so it is unknown to what extent the algorithm is capable of effective interpolation while moving across a scene. From this demo, we can infer that the kinds of algorithm presented in this paper are, as hypothesized, well-posed to be used in industry VR applications, and also that there is plenty of work left to be done until such algorithms are ideal, and this thesis hopes to contribute to that end.

Chapter 3

Notation

In this chapter we will briefly go over the notation used in this paper for formulas, algorithms, and diagrams. Most of this notation is either standard or has precedents in other graphics contexts which will be discussed when necessary.

$L_\mathrm{o}, L_\mathrm{i}, L_\mathrm{e}$ have their definitions from the Rendering equation given in the introduction.

S^1, S^2 denote the unit circle and unit sphere, respectively. We will also use these spaces functionally, i.e $S^2(P)$ for some $P \in \mathbb{R}^3$ is the projection of P onto the unit sphere (or equivalently, a unit vector in the direction of P from the origin).

$\mathscr{M}$ represents the set of points which lie on a surface in a scene.

As much as possible, we attempt to use script letters ($\mathscr{M}, \mathscr{L}, \mathscr{S}$) for generalized sets or spaces, capital Latin letters ($X, Y, P, \ldots$) for spatial points, lowercase Greek letters ($\omega, \alpha, \beta, \ldots$) for angles or directions, which in many contexts will be viewed as interchangeable (formally, we will use Greek letters for elements of S^2 or S^1).

Also, we will use lower case Latin letters ($a, b, c, \ldots$) for scalar quantities. Though we will generally attempt to represent points in space abstractly, we will perform certain operations that assume certain properties of space. $d(X, Y)$ represents the Euclidean distance (L^2 norm) from point X to point Y, and $Y - X$ is the displacement vector from X to Y. The visibility function $v_{\mathscr{M}}(X, Y)$ is 1 if there is a line segment with endpoints X, Y which does not otherwise intersect the collection of surfaces $\mathscr{M}$. Typically, when this function is used the set of surfaces is clear, so the subscript will be suppressed.

A notable exception to the above case identifiers is that we will use ϵ for the standard meaning of a very small scalar value used for thresholds of closeness or approximate equality, and use of boldface letters such as $\mathbf{x}$ for vectors that we need to treat as elements of a vector space with explicit components

A ray from point X in direction ω will be denoted by the tuple (X, ω).

Lastly, while the above is formal notation which we will attempt to never break from throughout this document, we also will adopt a heuristic for naming common variables. A typical situation is to have a ray query (X, ω) being answered by a light probe (L_P, r_P) (this notation is defined at the beginning of chapter 4). In this case, we will always use X for the base of the ray, ω for the ray direction, P for the point associated to the probe, Y for the point given as an answer to the ray query by the probe, and Z will be used for points along the ray.

Chapter 4

Light Probes

The algorithm of this paper depends fundamentally on the notion of a light probe. Here we give a formal definition of a light probe, as it will be used in this paper.

Definition. A **Light Probe** (with depth[1]) in a scene with surfaces $\mathscr{M}$ is a tuple (L_P, r_P), where P is a point, and L_P, r_P are functions with domain S^2. L_P has the property that $L_P(\omega) = L_o(P, \omega)$, and the function r_P is defined as $r_P(\omega) = \min\{d(P, X) : X \in \mathscr{M}\}$.

Conceptually, a light probe stores the solutions to ray queries for all rays with origin at X. The function L_P stores irradiance values. It is equivalent to the more common graphics data structure of an environment map. The function r_P, in words, for each $\omega \in S^2$ gives the distance to the nearest point in a scene from point P in direction ω.

The first detail to discuss with these light probes is how we store the functions (L_P, r_P) efficiently, as functions with domain S^2 cannot be trivially represented in memory, as S^2 is a continuous and not necessarily easily discretized domain.

4.1 Probe Representation

An early mapping of the sphere commonly used in graphics applications what is known as a cubemap [Blinn and Newell, 1976]. A cubemap contains 6 textures[2], each of which represent the face of a cube. Based on the maximal (x, y, z) components of a direction ω, one of the 6 faces of the cube are chosen, and the other two coordinates are used as an index into the two-dimensional texture. Cubemaps are ideal for their simplicity of lookup, the fact that the data, at a glance, looks like an appropriate and easy to understand visualization, and due to their direct support in current graphics hardware.

We note that the common geometric interpretation of S^2 is the unit sphere using the L^2 norm, which yields the standard Euclidean distance in 3-space. A cubemap represents S^2 using the L^∞ norm (in which S^2 is indeed a cube, and the cubemap lookup is precisely the mapping between the norms). The representation we shall use for the representation of light probes is the one promoted by Dan Evangelokos' thesis, and which

[1] Formally it is more accurate to refer to the entities discussed here as light probes with depth, but for brevity, throughout this paper, we refer to them simply as light probes.

[2] Here and elsewhere in the thesis, texture simply refers to a 2D image stored on a GPU

has been shown in earlier work to be an efficient and accurate representation of the sphere, is the Octahedral Mapping.[Evangelakos, 2015; Cigolle et al., 2014].

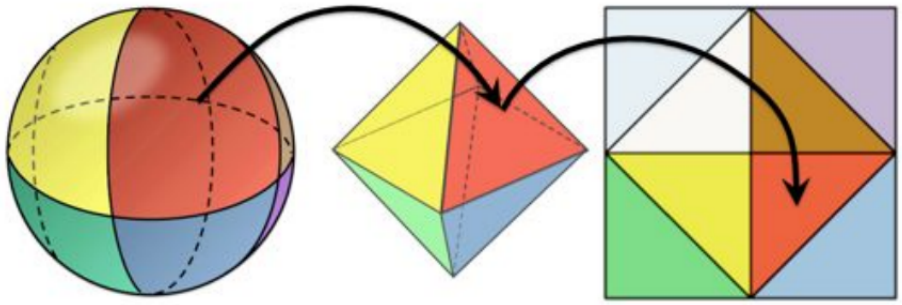


Figure 4.1: A visual representation of the octahedral mapping.

In the octahedral mapping, S^2 is interpreted using the L^1 norm, yielding an octahedron. Then, in order to map an octahedron to a square, we "unfold" the octants as shown in the diagram above. This representation is both easy to compute, and distorts the original data very little, on the order of 0.01 radian deviation when storing unit vectors in 32 bits [Cigolle et al., 2014].

Though our algorithm does not hinge fundamentally on the details of this representation, we will be leveraging the fact that our light probes can be efficiently and accurately represented by a 2D texture; a property that would be maintained across a number of other spherical parameterizations. However, the octahedral mapping also has the desirable property that rays in 3D space map to piecewise lines in octahedral space, bending only at edges of the octahedron [Evangelakos, 2015]. This allows for the relatively easy transference of previous ray tracing techniques to octahedral maps, as discussed in section 7.6, as well as the aforementioned benefits of being a low-distortion compression method.

4.2 Probe Usage

We need to be able to use our light probes in order to produce answers to ray queries. By definition, we know that $L_P(\omega) = L_o(P, \omega)$, and thus our light probe can accurately answer all ray queries with origin P. However, we need to be able to answer ray queries of other forms as well.

In order to use L_P to exactly compute queries of the form $L_o(X, \omega)$ for $X \neq P$, we will first make the assumption that the surfaces in our scene are perfectly diffuse. That is, the light scattered from the surface in one direction equals the light scattered in all other directions, so that two viewers of the same point see the same irradiance value. This is in some cases a highly unrealistic assumption, the effects of which can be decreased (as discussed later in this chapter), or avoided (as we will look into in chapter 5).

However, given this assumption, so long as we can determine what point $Y \in \mathscr{M}$ is the first intersection of a ray (X, ω) with $\mathscr{M}$, and P has visibility of Y (that is, $r_P(S^2(P - Y)) = d(Y, P)$, or equivalently based on the visibility function defined in the section on notation, $v(Y, P) = 1$), then we will have

$$L_o(X, \omega) = L_o(P, S^2(Y - P)) = L_P(S^2(Y - P))$$

and thus our light probe does indeed store the information necessary to compute this irradiance value.

Therefore, we have reduced answering the irradiance query $L_o(X, \omega)$ to answering the ray intersection query: What is the first intersection of (X, ω) with $\mathscr{M}$? We first note that by "first intersection" we mean the point in the intersection with minimal distance to X, or equivalently, the first intersection that would be reached by stepping along the ray. This latter characterization gives rise to the algorithm we use for computing the intersection, ray marching.

Here, we discuss how to apply a simple ray marching algorithm given the depth information stored in our probes. Later we shall discuss ways of performing such marches accurately and efficiently.

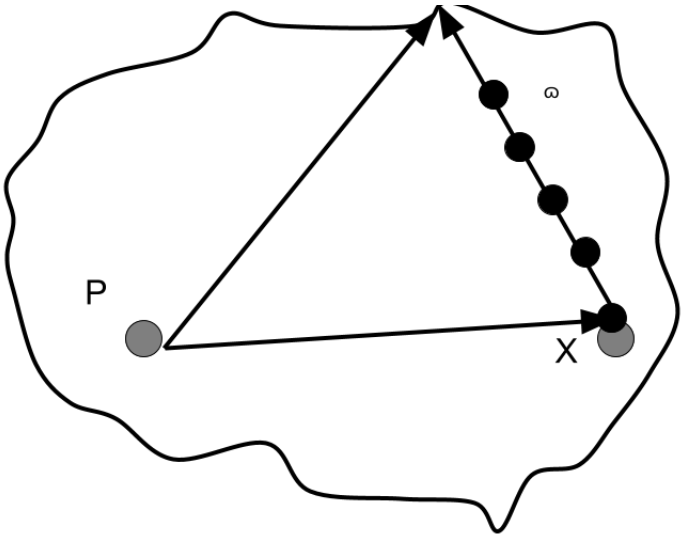


Figure 4.2: Ray marching along a ray: We have a probe (L_P, r_P) and a 2D slice of the image of r_P illustrated. In order to ray march the ray (X, ω), we begin by computing $r_P(X - P)$. If $r_P(X - P) = d(P, X)$, then the ray travels zero distance. Otherwise, we continue to step along the ray to points of the form $X + ks\omega$ for $k = 1, 2, 3, \ldots$ and some step size s until $r_P(X - P) \approx d(P, X + ks\omega)$. When this occurs, we know we have an intersection with the scene (up to the precision of our step size). The main flaw of this algorithm is its sensitivity to your choice of s for step size and an ϵ for the threshold of the fuzzy equality above, as well as not having a fixed termination condition, making the runtime unbounded. Both of these issues will be addressed in section 5.6.

4.3 Probe Selection

Now that we have a method for answering ray queries given a light probe that satisfies certain conditions, we must answer the question of how we choose a light probe to use to answer any particular query.

The vast majority of scenes are such that a single probe is incapable of having visibility of all points in the scene (for a simple example, one probe cannot see all sides of a column). Therefore, we will need to have multiple probes placed in the scene, and we require some method of selecting which probes to perform ray queries with.

The minimal theoretic restriction that must be placed on the probe we select for the ray query, as discussed above, is that, if Y is the first intersection of (X, ω) with $\mathscr{M}$, then we must pick a probe (L_P, r_P) such that $v(P, Y) = 1$. However, this restriction is impossible to implement in our framework as it is circular: we want to use the probe to compute what Y is, so we cannot easily check a condition involving Y.

So, instead, we must impose a restriction based on our choice of algorithm: we compute Y by ray marching the ray (X, ω), so in order for a probe (L_P, r_P) to be guaranteed to produce the correct solution

from ray marching, we must have that $v(L_P, Z) = 1$ for all points Z that lie on (X, ω) between X and Y. Informally, the probe must have visibility of the entire line segment that would be traced out in a ray march; this is necessary so that the probe cannot only see the hit point, but can also guarantee the correctness of the answer it gives.

Such a restriction can be implemented by only attempting to trace rays with visibility of the origin of the ray ($v(P, X) = 1$), and throwing out the result of a ray march if the probe loses visibility at any point in the ray march. Presently, we assume that on a loss of visibility, we would have to throw out the work done in the current ray march and begin tracing using another probe, but methods for preventing redundant computation will be discussed in section 5.4. An ideal probe selection algorithm would not only be able to recommend a single probe with high probability of being useful for a trace, but would also be able to provide alternatives after other probes have failed, as is discussed in section 5.5.

Now that we have formalized the problem of probe selection, we discuss ways in which it can be solved. Clearly, one probe selection algorithm is to order the probes randomly or in a fixed order independent of the ray query, and trying each probe in this fixed order for a ray query, falling back to the next probe if a probe fails to yield a solution to a query. Such an algorithm, while it would work, has terrible worst case behavior (needing to check every probe), and in many scenes will have very low probability of finding a hit on the first probe tried, and this algorithm would also not prefer probes that may potentially be able to yield more accurate answers to ray queries.

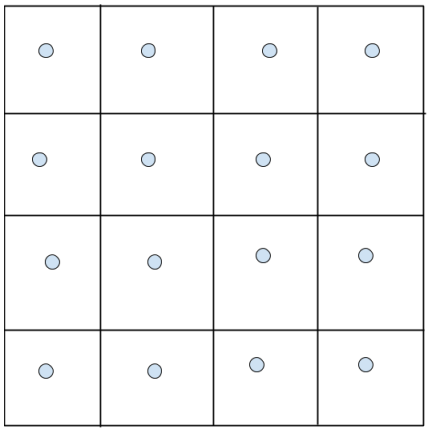


Figure 4.3: An example of a scene that makes clear the need for a probe heuristic based on distance. If the lines are viewed as walls and the circles as probes, we see that any ray we could ever want to trace would be visible to exactly one probe (the one in the same room), and if we picked that probe, we could properly trace the ray. Thus in this scene, if we picked probes at random, if there are n probes, the probability of a random probe being suitable for a ray trace would be $1/n$.

In looking for an alternative probe selection algorithm to the check-every-probe approach, there are two clear heuristics that arise. One is to optimize for the position of the probe: That is, prioritize checking those probes such that, for a ray query (X, ω) for probe (L_P, r_P), we want $d(P, X)$ to be small for the probes we check. This heuristic will be useful in many cases, as in general scenes, the likelihood of there being an obscuring object between the probe center P and the ray (X, ω) will increase with distance.

The other heuristic is to optimize for the direction ω. If the hit point Y of a ray query (X, ω) is very far from X, then picking probes near X may fail drastically at providing good results, by not only not being capable of finding the solution through ray marching, but also being likely to have the visibility to fall off a

great distance along the ray, increasing the amount of work wasted.

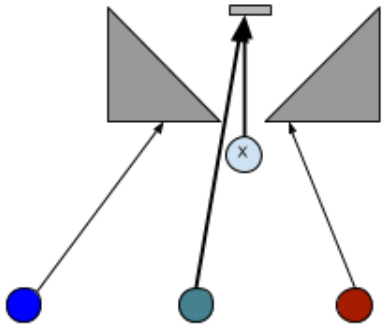


Figure 4.4: An example of a scene that makes clear the need for a parallelness heuristic. If we seek to trace the ray from X in the indicated direction, the blue and red probes, and any probes which similarly are displaced from the probe in a direction not close to parallel to the lookup direction, would lose visibility as the ray from X goes through a "corridor." However, the green probe has visibility of the entire ray from X, and would thus be the one we would want a heuristic to pick out.

However, optimizing for ω gives a high probability that, if the origin X of the ray is visible, then the rest of the ray will be visible as well. Whereas optimizing for X had a very clear definition (minimizing $d(P, X)$), optimizing for ω is a bit less clear, but as the schematic above illustrates, what we want is for the displacement vector from the probe center to the ray origin, $S^2(X-P)$, to be nearly parallel to ω. Formally, we want to maximize the quantity $S^2(X-P)\cdot\omega$. While parallelness would imply we should take the absolute value of this quantity, a discussion of the reason for taking the signed version is given in section 5.7 . When this quantity is close to 1, the entire ray (X, ω) will occupy a small portion of the field of view of the probe (L_P, r_P), which increases likelihood of finding a hit quickly. In addition, nearly parallel probe queries will alleviate the issue with light probe queries not taking into account surfaces with glossy or reflective BRDFs. Indeed, if $S^2(X - P) = \omega$, the value returned by the probe query is exactly correct. Lastly, as will be discussed in chapter 9, ray queries of more parallel rays will, with certain algorithms, take less time than less parallel rays.

Further analysis of probe selection methods will be discussed in the algorithm section

4.4 Probe Placement

The last parameter of a light probe-based rendering algorithm that remains to be discussed is how the light probes are arranged in the scene. Their are many approaches to this problem, though for the purpose of this text, we will be taking probably the simplest approach possible to this aspect.

An ideal theoretical solution to probe selection would be be to, given a geometric model of our scene, divide the scene into a union of convex regions of space, and then placing a light probe in the center of each of these convex regions. The convexity restriction will ensure that within a particular convex region, the light probe placed in that region will be capable of answering all possible ray queries. Indeed, we could reduce the number of necessary regions by loosening the restriction on regions from being convex to being "star-shaped." We define this term as follows:

Definition. A region $\mathscr{S} \subset \mathbb{R}^n$ is **star-shaped** if there exists a point $P \in \mathscr{S}$ such that $v_{\mathscr{S}}(P, X) = 1$ for all points $X \in \mathscr{S}$.

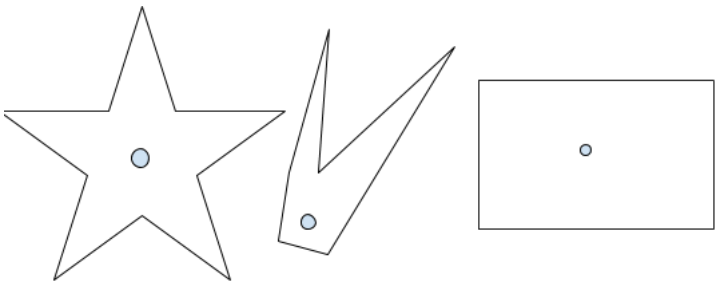


Figure 4.5: An example of some 2D-Star Shaped Regions. An important realization is that all convex shapes are star-shaped, so very simple scenes (like rectangular rooms) can be handled simply with one probe. The 1-probe box case is handled well by the analytic method of Bjorke [Bjorke, 2004], however that method would not be able to accurately render all views of, for example, a 3D extrusion of the 5-pointed star in the above image.

It follows immediately from the definition of star-shaped that if we have a probe (L_P, R_P), then this will be sufficient for all ray queries contained entirely in $\mathscr{S}$.

While the star-shaped placement method is theoretically interesting and ensures that there is as little overlap in the points seen by probes as possible, it will have flaws overcoming relaxation of the assumption that all surfaces are lambertian that we made above, and it is computationally intensive, both for determining where to place the probes and in that it requires storage of where the probes actually are as probe placement, using this algorithm, is highly scene-dependent. We shall not discuss algorithms for this procedure in further detail, but we note that the star-shaped probe placement problem this is the 3-dimensional analog of Chvátal's Art Gallery problem Chvatal [1975], where it has been proved (in the language of our problem) that at most $\lfloor n/3 \rfloor$ probes would be needed for visibility of an n-gon scene. Also, in the language of vertex guards, a star-shaped region is precisely a region that can be guarded by one point.

A problem which is related to the the star-shaped region divisibility problem is the problem of dividing 3D space into a collection of convex regions: all convex regions are star-shaped, though a star-shaped region may be the union of a large number of convex regions (for example, an n-pointed star cannot always be expressed as the union of fewer than n convex regions, despite being a single star-shaped region). The problem of dividing space into convex regions is also solved by generation of Binary Space Partioning Trees (or BSPTrees) [Fuchs et al., 1980]. Thus, we can bound above the minimum number of probes required for perfect visibility of a scene by the size of a BSPTree for that scene, though this discussion will often be generous.

An algorithm which has the benefit of not requiring computation but has the same drawback of needing to explicitly store and lookup probe locations, is the method which most closely resembles how light probes are used in various algorithms today with Image-Based Lighting techniques in games such as Remember Me Lagarde [2012]. This common method is to have artists manually place all of the probes with considerations in mind so that they will likely be good for lookups.

However, in this paper, we will mostly consider a far simpler probe placement algorithm: we place the probes in a scene in a uniform grid. In general, if we pick an orthogonal (but not necessarily orthonormal) basis for $\mathbb{R}^3$ $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$, and some integers A, B, C we place probes at the lattice points:

$$\mathcal{L}(\mathbf{e}_1, \mathbf{e}_1, \mathbf{e}_3) = \{a\mathbf{e}_1 + b\mathbf{e}_2 + c\mathbf{e}_3 : a, b, c \in \mathbb{N}_0, a \leq A, b \leq B, c \leq C\}$$

These lattice points give rise to a regular grid, which is convenient for $O(1)$ runtime computations of where

a given probe is, or what the nearest probes to a given point are. In practice, and for the duration of the discussion in the rest of this thesis, we will let $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ be scalar multiples of the standard basis $\mathbf{i}, \mathbf{j}, \mathbf{k}$. We use these directions in order to simplify formulas, and also because the vast majority of graphics scenes are arranged in ways that roughly align with these axes (for example, rooms and buildings tend to be arranged to have walls and ceilings in these orthogonal directions), and any scene for which such a grid was not appropriate could be rotated such that they were appropriate.

We allow scalar multiples of the orthogonal directions so that we can scale the density of the grid based on what units are being used and so that we can have different grid resolution in different directions. One could imagine that a room with high ceilings would need much less resolution in the y direction than the x or z directions, using standard computer graphics conventions of the meaning of these directions: y going upwards, z going towards the camera, and x pointing to the right. Such a hypothesis has been born out in practice on common scenes.

Chapter 5

Algorithm

In this section, we describe an accurate and efficient light field rendering algorithm which incorporates the light probe data structure discussed in the previous section. Much of the work presented here moves from a general framework of algorithms down towards specifics, in order to make it clear that various features of the algorithm could be changed or extended as is fit for a particular use case.

5.1 Past Work

Past work on tracing light probes often used analytic methods to determine what irradiance value to use. These analytic methods can be characterized as using some function $\phi(P, X, \omega)$ such that the algorithm can be described as making the approximation

$$L_o(X, \omega) \approx L_P(\phi(P, X, \omega)$$

The simplest analytic method is to use a light probe as an environment map [Blinn and Newell, 1976], which corresponds to choosing $\phi(P, X, \omega) = \omega$.Later methods such as Image Based Lighting [Bjorke, 2004] use geometric proxies such as cubes or spheres, and in this case ϕ corresponds to computing the intersection of the ray (X, ω) with a cube or sphere centered at P. Szirmay et.al [Szirmay-Kalos et al., 2005] used a root-finding method, in particular the secant method, in order to determine intersection points, but such a method is not guaranteed to converge, and there is no guarantee if it converges that it converges to a correct point.

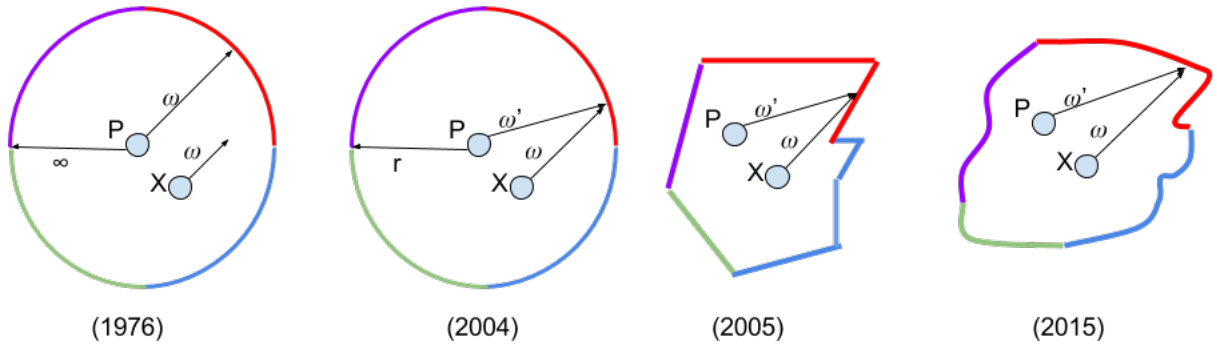


Figure 5.1: A brief history of the progression of depth proxies in light probes over time. Moving from left to right, an environment map essentially views the scene as an infinite sphere, so it makes the assumption that $L_P(\omega) = L_o(X, \omega)$ for all X, ω [Blinn and Newell, 1976]. The work of Bjorke uses static depth proxies, such as a finite sphere (as illustrated) or a cube (as most graphics scenes are, roughly, cubic rooms) to produce a lookup direction analytically [Bjorke, 2004]. Szirmay's 2005 method of Depth Imposters uses depth information to form successive planar approximations of scene geometry in order to converge to an intersection point (which is not guaranteed to be a bounded distance away) [Szirmay-Kalos et al., 2005]. Lastly on the right is the ray marching approach with high resolution data used here and first proposed in [Evangelakos, 2015], which is guaranteed to yield correct results if given sufficient data. This is the approach used here for tracing with a single probe, though this thesis extends the ray marching to use multiple probes.

All of these algorithms were appropriate to their use of light probes (exclusively for the computation of second-order lighting effects such as glossy reflections), but are not robust enough, or able to work accurately at high enough resolutions, to accommodate for global illumination. In Dan Evangelokos' thesis [Evangelakos, 2015], he showed that light probes can produce highly accurate ray queries suitable for global illumination, but such results were given in the context of a expensive CPU algorithm which cannot be run in real-time.

In the foregoing section, we will discuss an algorithm which has been shown to able to produce accurate images while running in real time on modern graphics hardware. This implementation was produced using the G3D Graphics Innovation Engine, an open source graphics engine headed by my advisor Morgan McGuire and which I have contributed to both before and during the course of the development of this algorithm.

5.2 Probe Generation

The first consideration for a rendering algorithm which relies upon light probes is the construction of the light probes themselves. We store light probes as 2D Textures which map to the sphere through the aforementioned octahedral mapping.

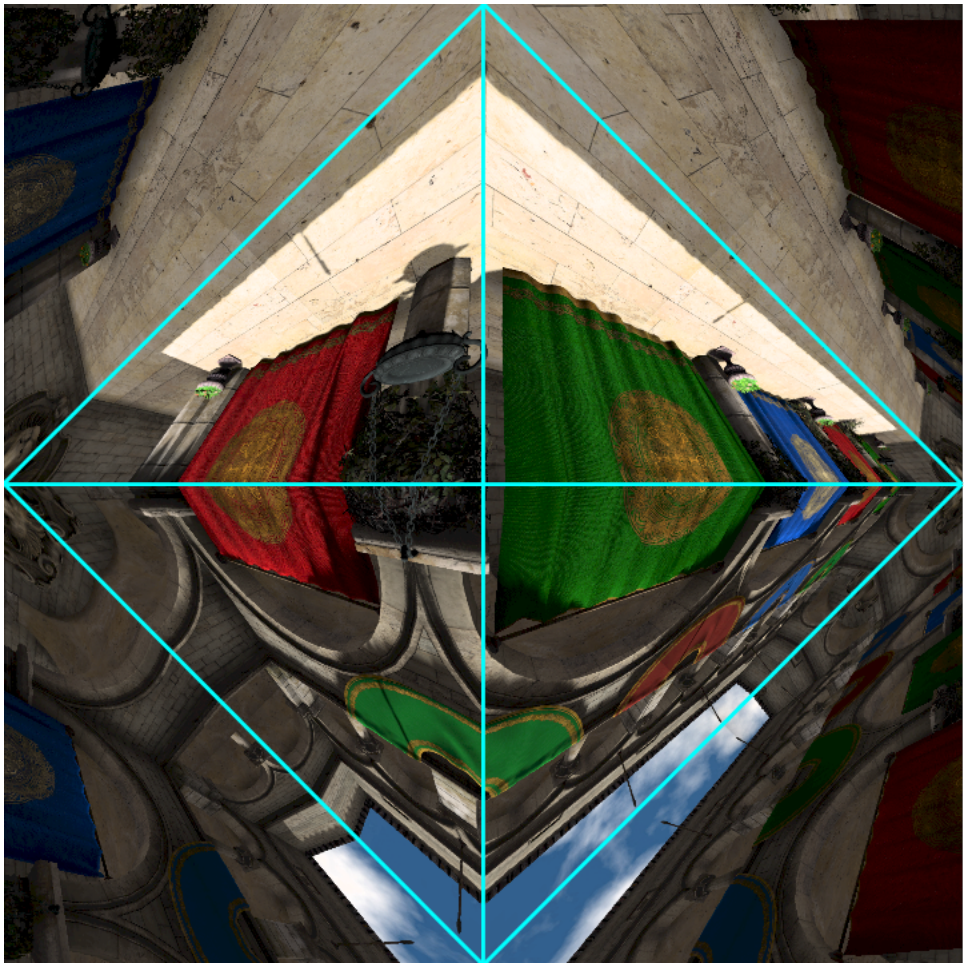


Figure 5.2: An example of an octahedral map texture for a light probe taken from the standard graphics Sponza scene. The cyan lines illustrate the borders of the cube faces.

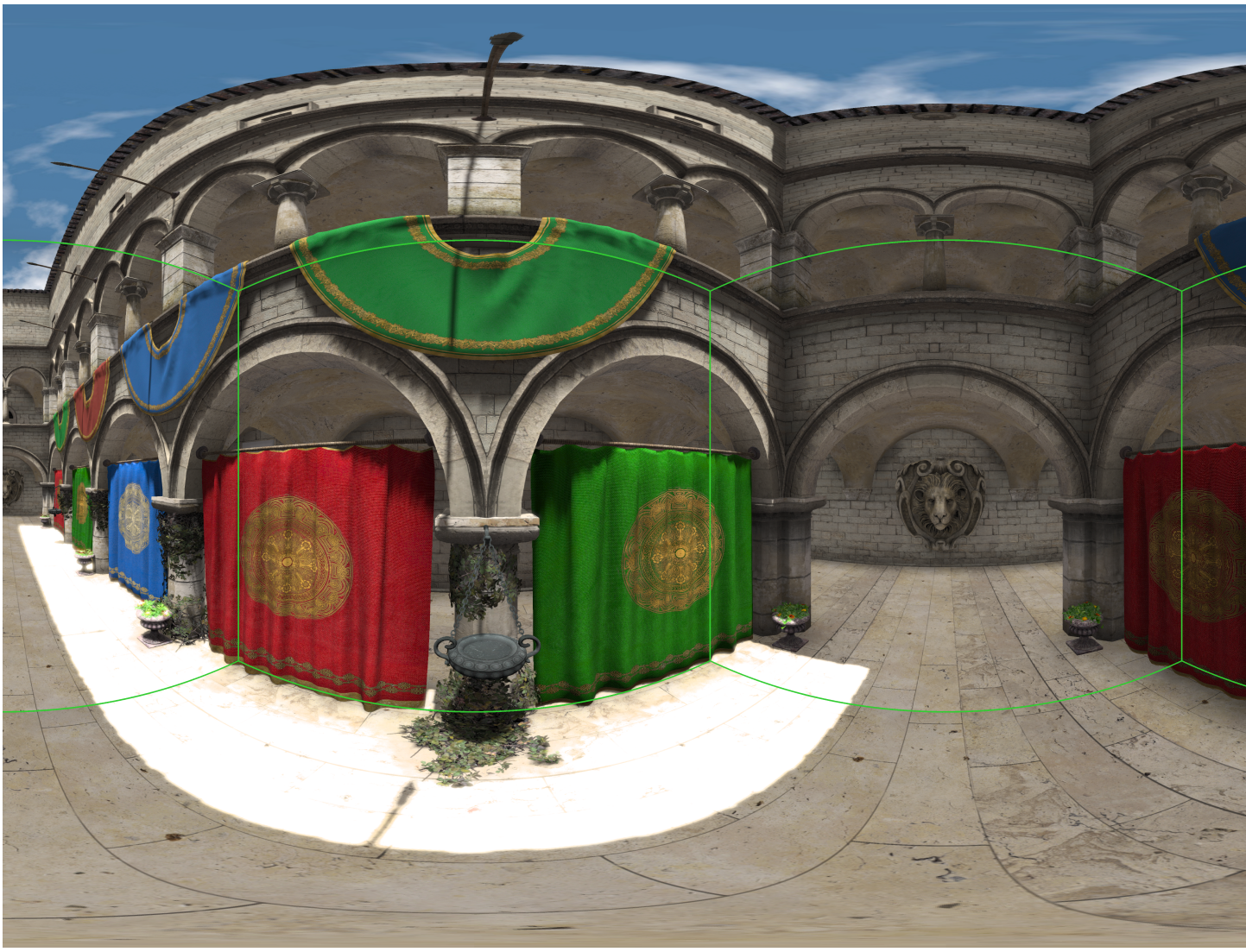


Figure 5.3: For comparison,the cube map similar to the one from which that octahedral map was generated, with the light green lines indicating cube boundaries (and the top and bottom faces stretched out to create a coherent image).

Each light probe is stored as a texture in an RGB format, and stores true irradiance values (not γ-corrected). Paired with each light probe texture is a Depth Probe texture, which is stored in a single channel format. At each pixel with texture coordinate (u, v), the depth probe r_X stores the distance from the origin of the probe, X to the point in the scene in the direction ω which corresponds to the octahedral mapping of (u, v). As is discussed in chapter 7, it is also useful to store normals, which can be represented as another texture with few bits per pixel, or as a channel in one of the other textures.

Ideally, the probes could be generated offline with standard physical cameras and depth cameras, in order to capture real world data, or a scene could be original specified only in terms of a set of light probes produced computationally or directly by artists. However, most of today's 3D graphics scenes are represented as collections of objects with geometry represented as triangle meshes. Therefore, in our experiments, our light probes were generated by first loading a scene using G3D's rasterizer, moving the camera to the points that are to be probe origins, rendering a cube map (which is easy as we can rasterize the six faces), and then

converting this cube map into an octahedral map. The depth probes are generated similarly, but by using the world space position buffer from the rasterization, and converting this position to a distance from the probe center. This algorithm is described by the following pseudocode:

Algorithm 1 Rendering Probes

```
1: procedure ComputeLightProbes
2:     LightProbes ← ∅
3:     DepthProbes ← ∅
4:     for 0 ≤ i < numberOfProbes() do
5:         P ← probeOrigin(i)
6:         camera.position ← P
7:         cubeFaces ← ∅
8:         cubeFacesDepth ← ∅
9:         for cf ∈ CubeFaceRotations() do
10:            camera.rotation ← cf
11:            cubeFaces[cf] ← rasterizeImage()
12:            cubeFacesDepth[cf] ← wsPosition()
13:        end for
14:        LightProbes[i] ← convertToOct(cubeFaces)
15:        DepthProbes[i] ← convertToDepthOct(cubeFacesDepth, P)
16:    end for
17: end procedure
```

Using this algorithm, we can generate an array of 2D textures stored as octahedral maps. These can then be bound as a 2DTextureArray in OpenGL when running the shader which will implement the rendering portion of the algorithm.

5.3 Ray Marching

The basic algorithm is implemented most simply on modern graphics hardware as a pixel shader: it runs for each pixel on the screen, and runs the algorithm in order to compute what color should be seen on a surface using our light field rendering algorithm at that point, and its depth from the camera. The depth is required to be able to compose this rendering algorithm with other approaches. For example, if most surfaces in a scene are rendered using the light field algorithm, but some other surface that is difficult to ray march on, such as a tree, is rasterized, so long as we have the depth information of the light field we can properly compose the results of the two algorithms into a proper image. This will be discussed further in section 8.1.

We now present a very simplistic form of the rendering algorithm, assuming that at each pixel we have some oracle function $findBestProbe$ which returns the "best" probe to perform our ray march against for the given ray, and that we only check one probe per pixel, and we have some method of ray marching a single probe.

Algorithm 2 First Overview of Algorithm

```
1: procedure ComputePixel(p)
2:     (X, ω) ← rayThroughPixel(p)
3:     (L_P, r_P) ← findBestProbe((X, ω))
4:     outColor, outDepth ← rayMarchProbe((L_P, r_P), (X, ω))
5: end procedure
```

Now, clearly the above version of the algorithm only works if we are guaranteed to be able to find a best probe. However, our initial criterion for a probe being "good" for a ray trace is that the probe has visibility of the entire ray, a property which cannot be verified without performing the ray march. In chapter 4, we discussed the simplest version of the ray marching algorithm. No matter what advancements are made, all ray marching algorithms essentially obey the following abstract structure:

Algorithm 3 General Ray Marching

procedure RAYMARCHPROBE($(L_P, r_P), (X, \omega)$)
  $Z \leftarrow X$
  while $\left|d(Z,P) - r_P(S^2(pZ - P))\right| > \epsilon$ **do**
    $Z \leftarrow nextPointOnRay(X, Z, \omega)$
    if $d(Z,P) - r_P(S^2(Z - P)) > \epsilon$ **then**
      $failure$
    end if
  end while
  $outColor, outDepth \leftarrow L_P(S^2(Z - P))$
  $outDepth \leftarrow r_P(\omega)$
end procedure

Here we see that the algorithm marches along the ray (picking the next point on the ray to check is here left open, but in the simplest ray marching algorithms, this is simply stepping along by some fixed distance). The other important part of this version of the ray marching algorithm is that if, at any point while marching along the ray, the probe loses visibility of the ray, it returns failure (indicating that this probe cannot yield an accurate result for the ray march). In the next section, we will discuss how we can loosen our assumptions on the existence of an oracle function, and reduce the degree of catastrophic failure when we do lose visibility.

5.4 Using Multiple Probes

There are two senses in which it is possible to use multiple light probes in order to compute the result in one pixel. One is by performing traces on multiple probes, and then either using one result based on some measurement of confidence on the probes or somehow blending the results together, and the other is by using multiple probes to perform a single trace. We begin by developing the theory in the first context, and then show how it can be extended to the second.

If we want to be able to trace multiple probes to contribute to one pixel, the most naive possible thing would be to perform a ray march on every probe in a scene, and then either use the result from the first probe which gave a valid result, or using some measurement of success use the "best" result. Both of these have very bad worst case performance. However, if we use some heuristic that gives a large degree of confidence that we will find a hit in the first few probes, then the average case performance of the algorithm will be quite good.

Therefore, our goal is to establish some ordering on the probes, that gives a fair deal of confidence that in most scenes a hit will be found in the first few probes. Given some ordering of the probes defined by $firstProbe()$ and $nextProbe()$ functions, we can express the general ray tracing algorithm by:

Algorithm 4 Compute Pixel with Visibility

```
1: procedure ComputePixel(p)
2:     (X, ω) ← rayThroughPixel(p)
3:     (L_P, r_P) ← firstProbe((X, ω))
4:     hit ← false
5:     while ¬hit do
6:         outColor, outDepth, hit ← rayMarchProbe((L_P, r_P), (X, ω))
7:         f ← nextProbe((X, ω), (L_P, r_P))
8:     end while
9: end procedure
```

where a trace will return $false$ for hit if the probe lost visibility at some point during the ray march, causing the trace to be thrown out.

However, we note that such an algorithm is, in some sense, inefficient. If we have to trace along multiple probes which all have a failure, we will end up tracing along the same portion of the ray multiple times. It is important to note that most of the process of ray marching is merely marching through empty space, so marching along any probe in order to eliminate portions of empty space from candidacy will give the same information, and it is entirely redundant to obtain it from multiple probes.

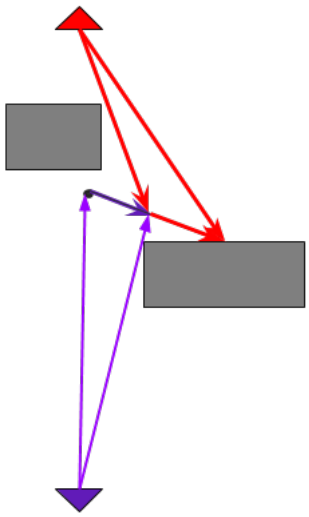


Figure 5.4: An illustration of a situation where a multi-probe trace is necessary. The purple probe can see the first part of the ray, but then loses visibility, at which point the red probe can take over, and trace the rest of the ray where the purple probe left off. This particular ray in this 2D schematic of a scene could not have been traced without using both. However, if the top box were removed, and we had chosen to trace with the purple probe first, we note that when we switched to the red probe, it would still be prudent to start where the purple probe left off, as otherwise we would be redundantly tracing the part of the ray we just traced.

Using this information, we can improve our algorithm by always performing exactly one ray march, but potentially switching between probes while moving along the ray when a probe loses visibility. This is the formulation we use in the below algorithm, though it can also be formulated as successive calls to a ray marching method that has three possible return values – a hit or miss, with their typical meanings, and a return value of "unknown," indicating that partial progress has been made along the ray, but neither a hit nor a miss has yet been concluded.

Algorithm 5 Multi-Probe Algorithm

```
1: procedure ComputePixel(p)
2:     Z ← X
3:     (X, ω) ← rayThroughPixel(p)
4:     (L_P, r_P) ← firstProbe((X, ω))
5:     while |d(Z, P) − r_P(S²(Z − P))| > ε do
6:         if d(Z, P) − r_P(S²(Z)) > ε then
7:             (L_P, r_P) ← nextProbe((X, ω), (L_P, r_P))
8:             if ¬(L_P, r_P) then
9:                 failure
10:            end if
11:        else
12:            Z ← nextPointOnRay(X, ω)
13:        end if
14:    end while
15:    outColor ← L_P(S²(Z − P))
16:    outDepth ← r_P(S²(Z − P))
17: end procedure
```

The "failure" case in the above algorithm will occur when we have searched through either all probes in a scene, or all the probes that we believe are reasonable candidates. In this case, a fall-back of some sort can be used. One option is to use the closest probe as an environment map, which should hopefully yield a "somewhat reasonable" result in many cases, especially if (as is likely, this fall-back is only needed for a few pixels.

Now that we have formulated an algorithm that works well given some heuristic ordering of probes, we will investigate how one might establish such an ordering.

5.5 Probe Ordering

In this section, we focus on how to define the previously undefined $firstProbe()$ and $nextProbe()$ functions discussed in the previous section. As discussed in chapter 4, heuristics for probe ordering have two parameters which can be optimized for: given a ray query (X, ω), we can either optimize for the position X or the direction ω.

Optimizing for position X intuitively makes sense at a coarse level: in a very large scene, probes far away are likely to have obstacles to visibility. While this will not always be the case (one could imagine a long, thin corridor, where all the nearest probes are at the other side of the wall and the only probes that carry useful information are at the ends of the hallway), it will be true for most queries in most scenes. Also, at large distances, due to the discretization of direction, probes will have a small amount of visibility. In the above corridor case, a painting at the end of the corridor might fall entirely within a single pixel of the faraway probe, which would not be useful for rendering views near the painting.

However, once we have narrowed our scope to a set of probes with spatial locality, it is unlikely that using closer probes will yield significantly better results (except in the extreme case where $X = P$ for some light probe (L_P, r_P)), due to objects in the scene having the potential to be in arbitrary locations. Therefore, at this point, it is logical to optimize for direction ω.

When we say optimize for ω, we mean that we want a probe (L_P, r_P) such that $|\omega \cdot (X - P)|$ is maximized; i.e the displacement vector $X - P$ is nearly parallel to the lookup direction ω. In the extreme case (where

the vectors are exactly parallel), as long as $v(P, X) = 1$, it must be the case that $L_P(\omega) = (L_o(X, \omega)$, i.e our light probe yields exactly correct results.

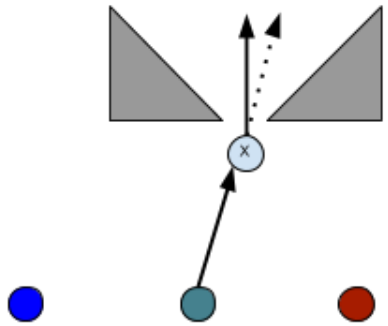


Figure 5.5: When looking to trace the ray from X in the indicated (solid) direction, we clearly want to use the green probe; the other probes have smaller values of $|\omega \cdot (X - P)|$, and in this case that indeed translates to them seeing the walls and not being able to trace the whole ray. Further, if we were tracing a ray in the dotted direction, then the trace direction would be exactly parallel to the displacement vector from the probe, so we could immediately conclude that $L_P(\omega = L_o(X, \omega)$

We can imagine solving for this analytically, and a discussion of such solutions is presented in the next section, as for practical purposes for this algorithm, constructing a full and fully accurate ordering is not essential, especially as the accuracy of the ordering is only based upon our heuristics, which are only heuristics and could still fail spectacularly in certain adversarial situations.

As the above problem is computationally intensive, we instead seek a more efficient solution to the problem. When we optimize for X, assuming an orthogonal grid of probes, we select the eight probes that are the vertices of a cube containing X – this is the least number of probes that can be chosen without introducing a directional bias. With other probe grid shapes, we would similarly choose an "element" of the grid (i.e a bounding tetrahedron in a tetrahedral grid or a bounding parallelpiped in a skew rectangular grid), but at this point the technique cannot be well-generalized to grids with probes whose positions cannot be determined analytically, as then computing the nearest eight probes becomes a non-trivial problem. Also we note that this cage of eight probes does not exist if X lies outside of the grid: in this case, we can take the eight-cage probe on the boundary of the grid nearest to X. In practice, however, it is likely to be convenient to restrict views to ones that fall within the grid of probes.

With a rectangular grid, determining the nearest eight probes is simple: given a ray query (X, ω), and a grid of probes generated with origin O, and spacings ℓ_x, ℓ_y, ℓ_z between probes in the standard orthogonal directions $\hat{\imath}, \hat{\jmath}, \hat{k}$, we can find the "normalized probe space" coordinates of X as (where the vector division below represents component-wise division as is common in programming languages with vector support such as GLSL):

$$nps(X) = (X - O)/(\ell_x, \ell_y, \ell_z)$$

Assuming X did not lie exactly on a probe origin (an assumption which would be true with probability 1 if we were working with actual real numbers, but which remains incredibly unlikely using 32-bit floats), this will give a 3-tuple of floating point coordinates for X: the nearest probes will be located at the lattice points corresponding to the eight ways of rounding these coordinates to integer values (taking floors or ceilings of each coordinate).

Now that we have established that we can easily find these nearest eight probes, the next issue is how

we can optimize for ω. Intuitively, it would make sense to sort the eight candidate probes based on their values of $|\omega \cdot (X - P)|$. However, such a sort of 8 elements running on a current GPU implemented in GLSL would require many accesses to global memory, and would be most efficiently implemented with a sorting exchange network, requiring approximately $n^2 = 64$ comparisons.

Not only is sorting potentially expensive, it also wastes computation in most cases: we expect to find a hit in the first one or two probes tried in the vast majority of cases, assuming sufficient probe density. So, instead of performing a full sort, we find the probe which maximizes the desired quantity (max can be found efficiently using only 7 comparisons and no global memory). If that probe, which based on our heuristics appears to be the "best" probe fails to yield a complete result, we could then compute the second best probe, then third, etc. However, as we move away from the "best" probe, our heuristics become ever more questionable, especially as the "next most parallel probe" may be looking in a similar direction as the previous best probe, and lose visibility due to the same obstacle.

Therefore, we propose that, if the first probe fails, we then traverse the remaining probes in an order that is in some sense "random." Here, by "random," we mean that it should not be globally biased by direction (as would be the case if we always checked the lower-left-bottom corner of the cube first, for example), and it should move around the cube in a manner that maximizes reflections across axes of symmetry in order to vary viewing angles. If we assume the x, y, z coordinates of our eight probes are all 0 or 1 (in other words, we map the situation to looking at the unit cube in the first octant), then this maximization of rotations requirement can be encoded as maximizing the average xor of the encodings of vertices checked in sequence.

So, formally, we want the following: a permutation π of the numbers $0, 1, 2, \ldots, 7$ that encode the 8 vertices of the cube of probe locations, such that

$$\sum_{i=0}^{7} \pi_i \oplus \pi_{i+1}$$

is maximized (where $\oplus$ denotes xor and we assume wrapping of indices, i.e $\pi_8 = \pi_0$). It can be checked by routine calculation that the optimal π based on the above criterion can be described by the mapping obtained by addition of 3 mod 8 between successive vertices. Therefore, if the first probe checked was 0, then we will check the remaining probes in the order $3, 6, 1, 4, 7, 2, 5$. To see that this does really maximize the rotations across faces of the cube, see the following visualization:

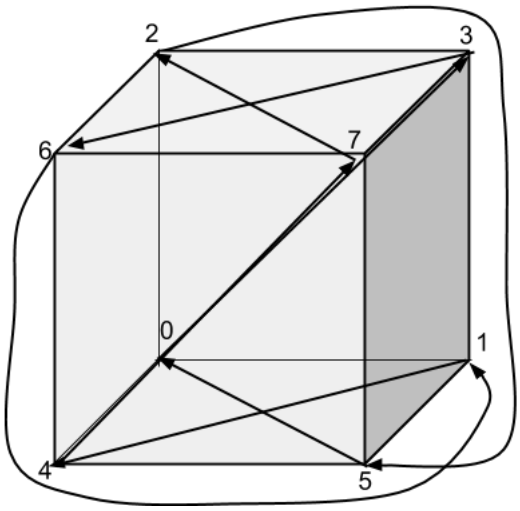


Figure 5.6: A diagram of the traversal of the cube in a pseudorandom, maximally mixing, deterministic order. The vertices are labeled 0 to 7, with the origin labeled 0, and then every other vertex located using the 3 bits of the label as the zyx coordinates of the next vertex, in that order from most to least significant. As can be seen, every transition between two vertices is either a reflection across a face or a diagonal.

So, with this, we can now write down pseudocode for the $firstProbe()$ and $nextProbe()$ functions, as desired:

Algorithm 6 Probe Ordering Methods

```
1: procedure FIRSTPROBE((X, ω))
2:     candidates ← nearestEightProbes((X, ω))
3:     maxDotProduct, bestProbe ← 0, null
4:     for (L_P, r_P) ∈ candidates do
5:         if |ω · (X − P)| > maxDotProduct then
6:             bestProbe ← (L_P, r_P)
7:             maxDotProduct ← |ω · (X − P)|
8:         end if
9:     end for
10: end procedure
11: procedure NEXTPROBE((X, ω), (L_P, r_P))
12:     i ← normalizedIndex((X, ω), (L_P, r_P))
13:     return(i + 3) mod 8
14: end procedure
```

5.6 2D and Hierarchical Traces

In the above, we discussed only the most simple ideas of a ray tracing algorithm (we now use "ray tracing" to distinguish the general problem from the particular algorithm of ray marching), as an algorithm which steps along a ray, returning a sequence of points which can be tested against. The most naïve form of ray tracing was discussed in chapter 4. Here, we will discuss what sorts of ideas can be applied to develop more advanced ray tracing techniques that have the potential to be at least an order of magnitude more efficient than the most naive form in many cases.

First, we will give a name to the previously described naïve strategy. We will call this a 3D ray trace.

This name derives from the fact that each step is a step along the ray as it is represented in world space, which is 3-dimensional, which contrasts with the first optimization.

A 2D ray trace involves tracing, rather than in 3-dimensional world space, in 2-dimensional texture space. That is, the radiance function L_P is defined in terms of a certain 2-dimensional texture. Each possible output of the function is represented by a pixel in this texture. Therefore, it does not make sense to, in a ray trace, check two points which fall within the same pixel. A 2D ray trace projects the ray being traced onto the 2D texture, and proceeds along that ray, pixel by pixel, typically using a DDA rasterization algorithm.

2D ray tracing comes from the method of screen-space ray tracing, an algorithm for efficient tracing of secondary rays for reflections within a rasterization rendering scheme [McGuire and Mara, 2014]. This screen-space ray trace is generally performed using the z-buffer from rasterization, and is thus performed on a rectangular projection of the scene. We wish to perform a ray trace on a the textures which represent our light probes with depth, and thus we must perform a ray trace on an octahedral map.

The fundamental calculations that enable an octahedral-space ray trace were carried out by Dan Evangelakos in his undergraduate thesis [Evangelakos, 2015]. There, he showed that straight lines in world space project to piecewise straight lines in octahedral space, bending exactly when the ray crosses from one face of the octahedron to another. In that thesis he also sketched an algorithm for performing an actual ray trace, but the algorithm was designed for a serial CPU architecture, and is not efficient for real time rendering, as the current screen-space ray tracing techniques are.

We will do further analysis on the efficiency of 2D ray tracing in chapter 9. However, the immediate observation is that the maximal number of samples that need to be tested is at worst proportional to the resolution of the texture, whereas for a 3D ray trace, the maximal steps is effectively unbounded – a maximal number of steps needs to be set so that the ray trace will terminate when tracing a ray that goes off to infinity, not intersecting any object within the scene.

A further optimization of 2D ray tracing is performing a hierarchical trace. The idea of a hierarchical trace is to skip over large sections of empty space along a ray by doing a 2D ray trace on lower MIP-levels of a texture, where a single pixel covers numerous pixels in the original image. The methods for such a hierarchical trace come from heightfield ray tracing, as described by Musgrave [Musgrave, 1988]. Though we reserve the implementation of these algorithms for highly efficient ray tracing to further work, we note that such methods are essentially for enhancing the performance characteristics of the algorithm, and we analyze the worst-case run time of the algorithm in terms of these 2D traces in chapter 9.

5.7 Signed or Unsigned Parallelness Heuristic

In the previous sections, we have discussed the parallelness heuristic in terms of wanting to maximize $\left|\omega \cdot S^2(X - P)\right|$. This formulation is the most literal interpretation of the geometric underpinnings of the algorithm as attempting to maximize how parallel the lookup direction ω is to the direction of displacement $S^2(X - P)$. However, we note that for further analysis, it will often be desirable to attempt to maximizes only the signed dot product $\omega \cdot S^2(X - P)$.

If the dot product is negative, this means in effect that the probe P is in front of the lookup point X; this allows for a very natural situation for viewing backfaces (as discussed in chapter 7). Further, additional memory lookups will be necessary along the ray – indeed, a negative dot product close to one could cause almost twice as many lookups as a dot product of 0, as will be shown in chapter 9. In short, we will from now on assume that we select based on the signed value of the dot product, not the absolute value. In practice,

the probes which had an initially negative dot product may become more reasonable selections if moving to a new probe, and a distance has been traversed along a ray such that the dot product with the new ray origin Z has become nonnegative.

Chapter 6

The Analytic Probe Ordering Problem

In the previous section, we introduced the problem of ordering the probes in a bounding cube around a ray query using the heuristic of straightness of ray. Before, we asserted that computing a full order that is optimal in terms of parallelness is likely to not be worthwhile, and the optimally mixing order was preferred. However, here we will briefly study the problem of computing the ordering analytically. Formally, we which to solve this problem:

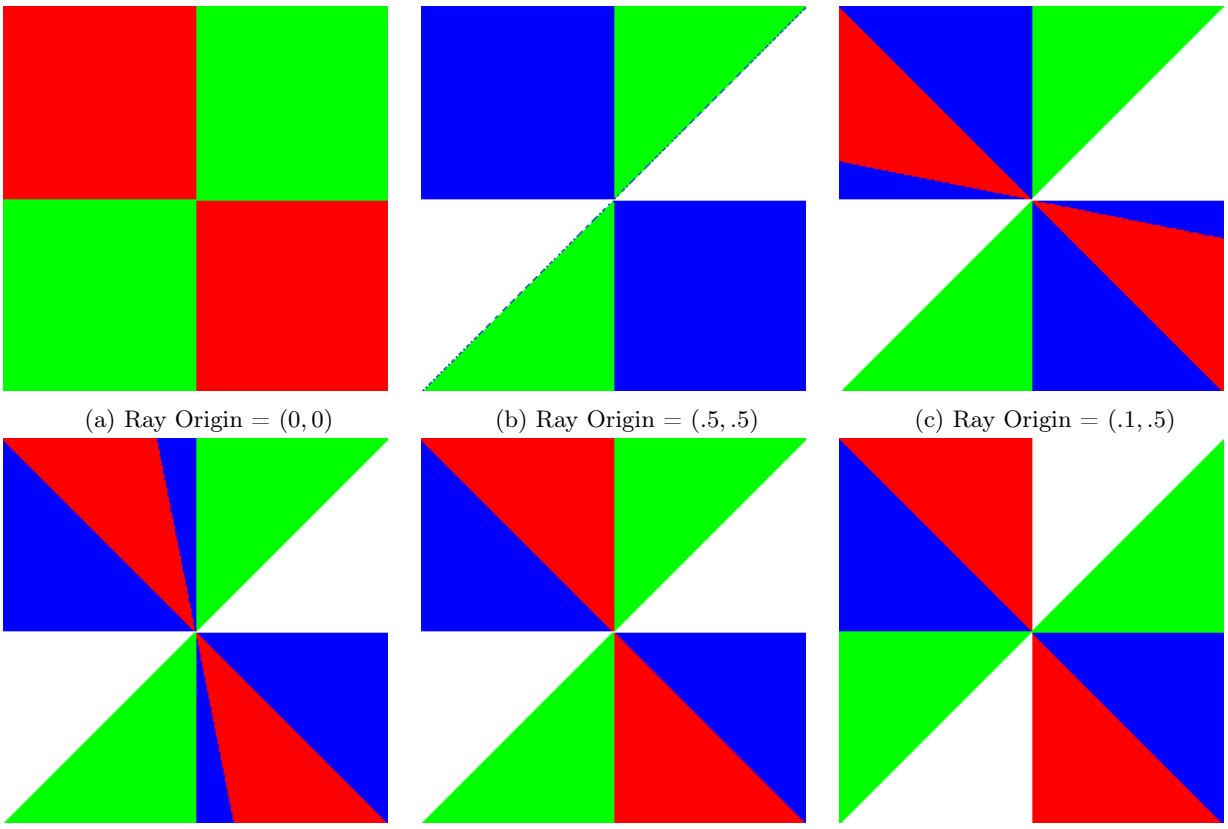
Probe Ordering Problem In some fixed number of dimensions d, given a ray $(X, \omega) \in \mathbb{R}^d \times S^d$ and the corners of a bounding cube $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{2^d}$, produce a permutation π of the points sorted from least to greatest values of $\left|S^d(\mathbf{x}_1 - X) \cdot \omega\right|$.

While we have explained this heuristic previously in this text, we first note that the problem above is expressed in an arbitrary number of dimensions, while our algorithm only needs to apply it in dimension 3, where most of graphics occurs. However, we will find it illustrative to look at the $d = 2$ case as well, which is far simpler. Also, we note that we take the absolute value here as we primarily care about how parallel the given lines are, though in section 5.7 we discuss why it is probably better to take the signed dot product into account in practice

6.1 The 2D case

We will begin by analyzing the two dimensional case of the probe ordering problem. First, we note that there are two inputs to the probe ordering problem: the position X and the direction ω of the ray. We will first view solving this problem as the process of solving it for all ω given a fixed X.

First we note that as we begin with the 2D case, the 2-cube is also simply known as a square. We first will solve (through brute force) what probe orderings are appropriate when the point X is in the center of the square. It is important to note that when X is in the center, the displacement vectors from the opposite corners are parallel, so we only have to order the pairs of opposite corners here, so only two different probe orderings are possible. With X at the center of the square, $(0, 0$, it is easy to see what the optimal probes are, but we can also see what effect changing the origin is



(a) Ray Origin $= (0, 0)$ (b) Ray Origin $= (.5, .5)$ (c) Ray Origin $= (.1, .5)$

(d) Ray Origin $= (.5, .1)$ (e) Ray Origin $= (.9, 0)$ (f) Ray Origin $= (0, -.9)$

Figure 6.1: In these figures, the center of the square is $(0, 0)$, and the edge is the unit square, so the corners are the points $(-1, -1), (-1, 1), (1, 1), (1, -1)$ in clockwise order starting in the bottom left. In that order, we associate the colors red, green, clue, white to the vertices. The color of a pixel is the "best" probe in the direction from the origin to that pixel (and thus, the color of a pixel only depends on angle from the origin, not distance, but the entire square is drawn for clarity). Further, in each picture, the ray origin that we are considering is indicated below the image.

So we can see that, given a point, the angles for optimal probes can be described by some collection of regions bounded by straight lines. Similarly in the 3D case, we can imagine that the regions associated to each choice of optimal probe would be associated to regions bounded by planes in 3-space. However, we leave solving for a closed form solution to the analytical problem to further workers on this problem, as in our experiments finding the optimal probe by computing the dot product for each potential probe and finding the max is a sufficient and efficient procedure.

Chapter 7

Aliasing and Backfaces

In this chapter, we will discuss some pessimal cases for the rendering algorithm as it has been stated, and some solutions to these issues.

7.1 Definition of an ϵ

In the above pseudocode formulations of our algorithm, we introduced an ϵ which represents the minimal distance between the point on the ray in a ray march and the hit point found by the probe which will be considered a hit. Given a fixed resolution of the textures which define the values of a probe (L_P, r_P), ϵ cannot be made arbitrarily small, as the hit points represented by adjacent pixels in the texture representing by L_P will necessarily be separated by some distance in a scene, and if ϵ is smaller than the distance from the actual hit point to the points represented by each of the pixels surrounding that hit point, then a ray march of the probe will report a miss even though there was data which would have represented a maximally reasonable hit point.

So, ϵ cannot be made arbitrarily small. What are the disadvantages to making it large? Making ϵ large effectively increases the observed thickness of objects in a scene. In the worst case (when all ray marches are accepted with distances that are equal to ϵ), this will have the effect of extruding all visible objects in the scene by ϵ in the direction from the probe. Now, this thickness-increasing effect does not cause uniform degradation of the scene. If far away objects (say, 10 meters away) appear, for instance, $\epsilon = 2$cm closer to the camera, this will not cause a noticeable (and likely not even representable at the output resolution) difference in the appearance of the scene. However, if an archway 5cm from the viewer is extruded 2c in a direction perpendicular from the direction from the camera, this will be highly noticeable and recognizable as an error

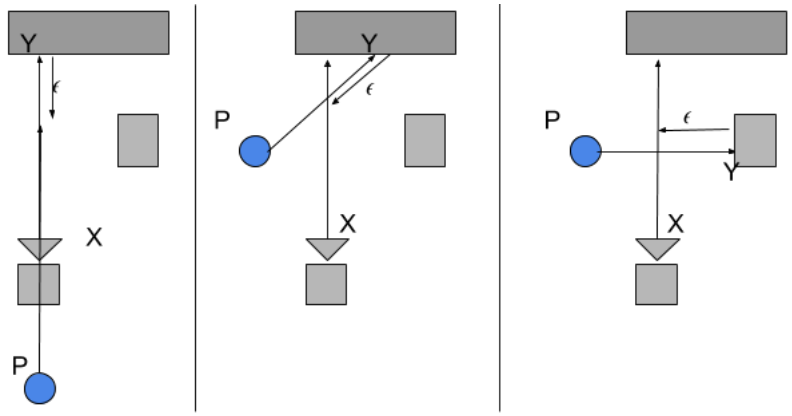


Figure 7.1: An illustration of three different effects that can result from the ϵ of thickness. In all cases, the blue circle is a probe at point P, the camera is at point X, and the point returned by the query is Y. In the first (leftmost) case, the probe was looking parallel to the query ray, and the ϵ caused early termination of the trace, but this still resulted in the correct point being returned (as should always happen if looking exactly parallel). In the second case, the thickness caused a point from nearby to be chosen as the hit point, shifting details over slightly, though, for reasonable ϵ, this error should lead to a hit point within the same pixel as the correct answer. In the third case, an entirely different object is seen due to an object being extruded in a direction perpendicular to the camera. This is a worse-case scenario, as explained below, and it would be desirable to not allow such a scenario, unless the objects are sufficiently distant that the erroneous hit point would still lie within the same pixel as the correct hit point.

So, how can these issues with ϵ be addressed? The immediate realization from the above argument is that a static ϵ is unlikely to be sufficient, and a dynamic ϵ that is some function of the parameters of the ray query being answered is likely to be desirable.

So, what parameters might we want to consider? Let (L_P, r_P) be the light probe under consideration, (X, ω) the ray query, and the hit point be a point Y on a surface with outward unit normal ν. Further, let $\alpha = S^2(X - Y)$ be the direction of displacement from the camera to Y, and similarly $\beta = S^2(X - P)$ the direction of displacement from X to P.

We will now discuss some of the effects at hand. If $|\alpha \cdot \beta| \approx 1$, then our light probe is looking in almost exactly the same direction of the camera, so we expect a hit point to be quite accurate regardless of distance away, so ϵ can be allowed to be large without being likely to give a misleading result. So in this way, we can have $\epsilon \propto |\alpha \cdot \beta|$. We note that if the dot product is 0, and thus the probe direction is perpendicular to the looking direction, it would be impossible for a hit to occur (except a perfect "glancing angle" situation, such as hitting precisely a vertex of a mesh, which should cause no issues to rule out).

From the earlier example, allowing $\epsilon \propto d(X, Y)$ seems reasonable, as at points near to the camera, small ϵ can cause catastrophic events, while at far away points, the value of ϵ does not matter particularly much

If $\alpha \cdot \nu \approx 1$, then the object is directly facing the camera, so ϵ changes in thickness are unlikely to have much of an effect, so ϵ can be large, while perpendicularity would again be contradictory if glancing cases are ignored, so we can have $\epsilon \propto \alpha \cdot \nu$.

Combining these equations, we come to the conclusion that:

$$\epsilon \propto (|\alpha \cdot \beta|)(1 - \alpha \cdot \nu)d(X, Y))$$

But how should a proportionality constant be determined? It seems reasonable to choose a proportionality constant in order to minimize the impact of the most noticeable error: extrusion nearly perpendicular to

the camera near to the camera. So, we should choose ϵ so that the effect of extrusion is limited to a 1-pixel extrusion. To solve for ϵ, we look at the following situation:

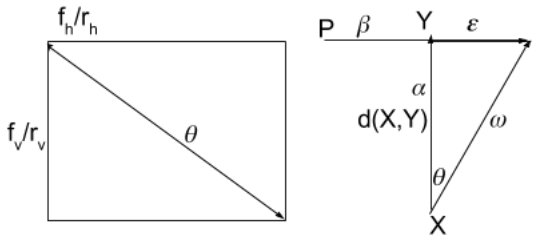


Figure 7.2: Assuming our camera has resolution (r_h, r_v) and field of view (f_h, f_v), where the subscripts denote horizontal and vertical, respectively, the maximal angle between two directions going through the same pixel is, as illustrated on the left, the diagonal of an angular rectangle, $\theta = \frac{\sqrt{f_h^2+f_v^2}}{\sqrt{r_x^2+r_y^2}}$. On the right, we have a drawing of the situation in the pixel, and on the right we have a drawing illustrating a situation where we would have the maximal allowable ϵ to restrict the error to one pixel. From this, we see that we want ϵ such that $\tan\theta \geq \frac{\epsilon}{d(X,Y)}$, so $\epsilon \leq \frac{d(X,Y)}{\tan\theta}$

$$\epsilon = k \tan\theta d(X,Y)(|\alpha \cdot \beta)(1 - \beta \cdot \nu)|$$

We allow still having a constant of proportionality (k), to allow for slackness in the ϵ. Reasons for choosing a $k > 1$ might include having low resolution probes which cause finding low ϵ hits. Also, one might heuristically want to increase k when tracing additional probes after initial failures, in order to increase the probability of some hit.

While the above gave us an answer for ϵ in terms of our earlier formulation, we note that this analysis has led us to another potential characterization of the thresholding of accepted values. Instead of establishing a threshold ϵ in terms of distance in the scene, we can express a threshold in terms of angle between α and ω – namely, the discrepancy between the direction from the query direction, ω and the actual direction from X to the result α. In terms of this, as the previous figure illustrated, we would want to accept only results such that

$$\cos^{-1}(\alpha \cdot \omega) < k\theta$$

where k is the number of pixels of error we want to allow. This formulation implicitly includes all of the factors discussed above except for the angle between ω and β; however, the angle between ω and β is constrained by our probe-finding heuristic (this is precisely the quantity we seek to minimize in our angle-based heuristic discussed earlier), so the combination of constraints on the angle between β and ω and α and ω indirectly leads to a constraint on the angle between β and α

7.2 The Problem of Backfaces

S The characterization of backfaces within the context of our rendering algorithm is the typical one: if the surface normal ν of a point and the viewing direction from the camera α are such that $\alpha \cdot \nu > 0$, then the surface is pointing away from the camera, and thus it is impossible for it to be seen. It is important to note that this is the definition of backface we use, based on the camera, (or, more generally, the origin of a query ray), *not* a backface with respect to a light probe. A light probe's function L_P should see only correct

geometry, as we assume it to have been generated from some ground truth. However, from the values of this L_P, we could yield lighting results corresponding to a query ray (X, ω) intersecting a backface.

Without the explicit use of surface normals, it is impossible to distinguish between slightly thick (within ϵ) objects which are continuously visible to the camera and objects of such thickness, one side of which is visible to the camera, and one side of which is facing away from the camera. Thus, if thin 2-sided objects exist in a scene (for example, a curtain which is lit differently from both sides), then surface normals must be stored, if for this check only.

Any ray query which returns a hit point which lies on a backface must be rejected, as the fact that the hit point lies on a backface implies that the probe in question does not have visibility of the correct side of the correct hitpoint, and will not be capable of yielding a correct result. However, it should be noted that this rejection is of a different sort than a typical ray rejection: when a trace is rejected for loss of visibility, we can have an additional probe continue the trace where the previous probe left off in order to eventually find the correct result. When a trace is rejected because of seeing a backface, it will necessarily have already traced too far, and found an object outside of the visibility of the camera. So, either the trace will have to be restarted from another probe, or the trace on the next probe will have to proceed *backwards*, until the ray has "gone back through" the object whose backface was being seen.

The additional overhead that comes from storing these normals is not particular high. From the results of [Cigolle et al., 2014], using 16 bits per unit normal, a mean angular error of about .33 degrees can be achieved, which is more than accurate enough for our use of testing against 0 (angle between two vectors $> 90°$). Even though this is a binary test, so any amount of discrepancy will lead to improper rejections, this will not produce significant additional error, as we will only reject "glancing" views, which occur with low probability and may themselves be erroneous. So indeed, an even more extreme compression scheme could likely be used, if normals aren't brought into other parts of the algorithm (such as for the optimized adaptable ϵ discussed in the previous section).

Chapter 8

Application Domains and Failure Cases

In this chapter, we will discuss areas where the light field rendering algorithm discussed in this thesis can be best applied, as well as cases where it cannot be applied effectively.

8.1 Algorithmic Extensions

Throughout this thesis, the light field rendering algorithm has mostly been discussed as an algorithm that would be used in isolation in order to compute images from a 3D graphics scene. In this section, we will discuss how it can be used as a general tool for the computation of ray queries, and how it could be extended by accounting for more general phenomena or could be mixed with other rendering techniques in order to yield practical results.

8.1.1 Dynamic Regeneration of Light Probes

At first, it may be seen as major flaw of the algorithm discussed in this paper that light fields are assumed to be static, and then rendering of the light field is always interpolation of the static data. However, it could be practical in some environments to dynamically regenerate light probes as the data in them becomes out of date. A light probe can be computed from within a traditional 3D graphics scene graph renderer (which represents a scene as a collection of models with triangle meshes) through rasterization. Modern GPUs support rasterization directly into a cube map.

Currently, the bottleneck in dynamic probe regeneration would then be compressing this cube map into an octahedral map for efficient storage and computation. If later GPUs allow rendering directly into the octahedral format, then a probe could be computed in exactly the same amount of time that it takes to rasterize a scene, making it a very efficient operation, and potentially practical to regenerate probes with great frequency (say, once a second), so as long as a dynamic scene changes slowly enough that using slightly stale information would not be catastrophic, this could a practical solution to using light field rendering in dynamic contexts. If only a subset of probes needs to be regenerate, this would further decrease the overhead of dynamic regeneration.

8.1.2 Mixed Static Light Fields and Dynamic Objects

In many graphics applications (such as games and virtual reality experiences), the majority of a scene is static, with only a few objects moving around. An example would be a game where dynamic characters move around a world which is largely static. In such a situation, the light field rendering algorithm presented in this document could be used to render the scenery, and as long as the renderer outputs both the color of points to return and the proper z-value (depth, i.e distance from the camera), these results could be properly mixed with a rasterization of the characters in a scene.

As rasterization algorithms scale with geometric complexity (number of triangles) of a scene, such an approach would mean that the complexity of the static elements of the scene can increase without having an impact on the rasterization, potentially allowing for an increased complexity in the dynamic models.

Figure 8.1: A screen-shot from the game The Last of Us by Naught Dog Dog [2014]. In this image, the scenery is entirely static in the game, and could thus be rendered with precomputed light probes, and indeed, the openness of the geometry makes it suitable to the light probe algorithm. The characters in the foreground move around the scene, so they could be rendered using rasterization or other techniques; and as the rasterization of the characters would be separate from the methods used on the background, they could have added geometric complexity or other advanced rendering effects (such as accurate hair) without having the cost of this be paired with the cost of rendering the rest of the scene – the complexity of the characters and the background could scale independently.

8.1.3 More Complex Rendering Effects

Throughout this paper, the light field rendering algorithm has mostly been discussed in the context of computation of primary rays from a camera, given sample data that can be arbitrarily photorealistic. However, certain physical phenomena are not modeled well through the interpolation used in this algorithm. For example, if there are not enough probes in a scene for it to be likely for one with the probe lookup vector and the camera viewing vector to be nearly parallel, then highly inaccurate results could be given for the rendering of a mirror, as the appearance of a mirror (and other glossy objects) depends greatly upon viewing angle.

There are methods for computation of glossy reflections in real-time settings [McGuire et al., 2013] which

use environment maps, so the textures used for the color portion of the light probe functions can be reused for this purpose. For more general rendering effects, which may involve the computation of secondary rays, our light field algorithm can continue to be used for this purpose, as it is a general-purpose ray tracing framework. The light field rendering algorithm yields results for $L_o(X, \omega)$. Though in previous discussion it has been assumed in many cases that X is the location of the camera and ω was a direction through a pixel, this restriction is not necessary, and thus the light-probe based ray tracing scheme can be extended to other frameworks, such as path tracing, which has been discussed in previous work [Evangelakos, 2015]. Further, while we have assumed that our light probe functions L_P stored approximations to global illumination, we could instead store just direct illumination in each probe, enabling the extension to path tracing

Another common mechanism for various rendering effects is cone tracing, which can be viewed as performing a ray trace of a low-resolution environment map (or a texture in screen-space [Stachowiak, 2015]). By performing these low-resolution traces, one gets an effect similar to computing an average of randomly sampled rays as would happen using direct sampling. As mentioned before, these computations are usually performed using screen-space ray-tracing or environment maps, but the textures for light probes are effectively more powerful environment maps that also allow for ray marching to compute values of arbitrary rays, so they could be used for algorithms such as this as well

8.2 Virtual Reality Applications

One potentially appealing application of the light-probe based light field rendering algorithm is to rendering for virtual reality (VR) experiences. There are multiple characteristics which make the algorithm applicable to this domain.

For one, as is discussed in chapter 9, the algorithm can run in time independent of scene complexity. This fact, combined with the fact that even a naïve implementation has been shown to run easily at 60fps on modern graphics hardware, implies that it may be appropriate for very efficient rendering. Further, as the algorithm is low-bandwidth, due to most of its data being static and capable of fitting in memory on a graphics card, it can be implemented in a low-latency setting, which is very important for VR applications, as the human sensory system demands rendering that reacts quickly to changes in head and body position.

Further, virtual reality content is likely to be well-suited to the light-probe algorithm. The Vive, developed by Valve, enables users to walk around within a virtual environment, but the user is naturally constrained by the size of the real-world room in which they walk. Naturally, this would lead to virtual reality content which caters to single-room scenes, with motion between small rooms done through teleportation or other non-continuous means. Single rooms can often be captured accurately with very few light probes (see the rendering of the Holodeck scene using a single probe in the Analysis section), and, as discussed in the next section, indoor rooms do not have any of the characteristics that would typically lead to failure of the algorithm.

Lastly, there is precedent for application of similar techniques to virtual reality rendering. NVIDIA's IRay VR [NVIDIA, 2016], about which few details have been released a the time this thesis is going to print, appears to use statically rendered light probes in order to perform real-time ray tracing within a Virtual Reality scene. Indeed, the IRay VR demo involves a person standing still while looking around a room (presumably with many light probes rendered nearby) with the lighting computed by ray tracing on the probes. Thus, the direction of the industry shows that similar approaches are desirable in VR, so too may the algorithm presented in this paper. The ideas in this paper will be followed up a paper to be submitted

to Transactions on Graphics written in collaboration with additional co-authors, which will expound on the benefits of the algorithm within a VR context.

8.3 Failure Cases

The light probe rendering algorithm is not ideal for all rendering situations. However, most situations in which it fails either have reasonable workarounds or are pathological for many rendering algorithms.

As it is assumed that all light probes are available in memory at all times, it may not apply directly to huge, sprawling scenes that would require vast numbers of probes to cover, though this could be solved by partial loading of the probes in some locality, similar to how for rasterization surfaces are culled and only the remaining triangles are sent to the GPU for computation.

Huge amounts of local geometric complexity can cause problems for this algorithm: for example, fractals have huge amounts of geometric complexity at any level of detail, which is unlikely to be easily captured by static light probes and interpolation for a certain amount of complexity. Similarly, trees (such as the ones in the standard graphics San Miguel scene) can cause difficulty as paths through the tree will be seen by probes with very low probability, and could lead to misses even though there were points that could be seen through the tree. Despite this apparent issue, for sufficient probe resolution, trees (at least when viewed from a distance) have been shown experimentally to cause fewer difficulties than first suspected (see chapter 9). One solution to this pathological case would be to render the trees using a separate rendering method, and any rays that pass through the trees applying the light probe based algorithm.

Lastly, there are certain extremely pathological cases that may require the necessity for hand-tweaking of probe placement, even though this is undesirable. In cases where the lattice placement of probes would be particularly useless at a given resolution, certain probes could have their actual origins tweaked, and this value used in computation of actual ray marching, while the probes could still be assumed to be on a uniform grid in order to keep the probe selection algorithm simple and not dependent on variations in the probes locations

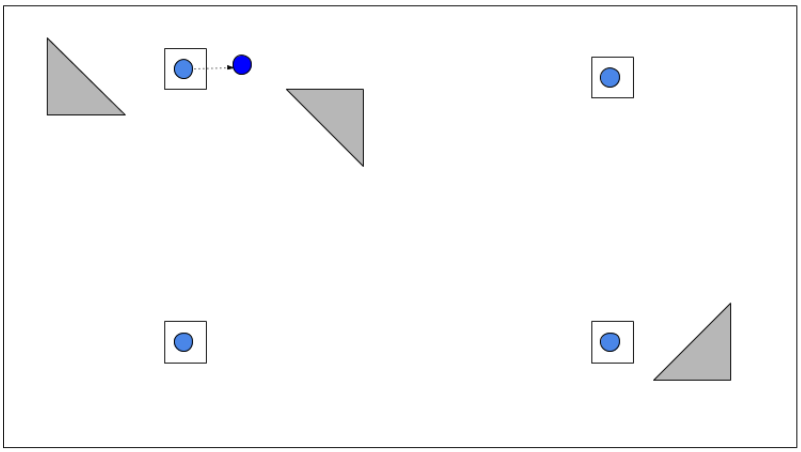


Figure 8.2: An example of a failure case. Here, the triangles and squares represent geometry of the scene, while the blue circles are light probes. In this scene, placing the light blue probes on a uniform grid leads to all probes being placed inside of tiny boxes, so that the probes cannot see much of a scene. One could imagine a pathological scene with a patterned placement of boxes so that at any grid resolution above a certain high density, this would always be the situation. However, in this situation, one could manually move the probes just outside of the boxes (indicated by the dark blue probe). Because the displacement from the lattice points is small, assuming the translated probe is at the original lattice point will not make the heuristics for probe selection too erroneous, so the simplicity of the grid could be used up until use of the actual probe location for accurate computation of the ray trace using the probe

Chapter 9

Analysis

In this chapter, we will conduct analysis of our algorithm, both theoretical analysis on the runtime characteristics of the algorithm, and experimental results from running the algorithm on real scenes.

9.1 Theoretical

In this section, we will analyze the complexity of our algorithm. For the duration of this analysis, we will only study the run time per pixel of the algorithm. This is a practical consideration for two reasons. For one, the algorithm runs independently per pixel, so if the run time per pixel is T, and we run the algorithm on p pixels, with m independent threads, a proportion c of which are capable of running coherently, then the overall run-time will be

$$O\left(\frac{pT}{mc}\right)$$

While our algorithm has been structured in order to have c close to 1, the form of this run-time is effectively the same for any graphics per-pixel algorithm in a multi-threaded environment. Therefore, the quantity of interest, that is uniquely dependent on our algorithm, is T, the run time per pixel.

Now we can begin to establish some bounds on T. For a given pixel, we must first find the probes we will be checking, and then we must trace the ray. Whether we have a visibility failure or intersect a backface, we can always continue tracing along a ray when visibility is lost. So then the run time of the algorithm will be

$$T = O(\text{Time to find probes} + \text{Number of queries along ray} + \text{Number of partial failures})$$

Now, the time to find the probes, assuming a rectangular grid, is $O(1)$. The number of queries along a ray is unbounded with a 3D ray trace, but in a 2D trace, it is bounded by the length of a diagonal of the texture representing a probe – (this is the bound as a ray of a greater length would cover more than 180°, and thus at some point become unusable), so if we check up to k probes, we may require $\sqrt{r}k$ (where r is the resolution of the probe texture) queries, and the number of possible losses of visibility or backface failures is k, this gives a total "asymptotic" run time of

$$T = O(k\sqrt{r})$$

We note that $k = 8$ in the version of our algorithm we presented, so this reduces to $O(\sqrt{r})$ for the current

suggested version of our algorithm. However, with a hierarchical 2D trace, the runtime is bounded by the log of the length in pixels of a ray, i.e we can replace the $\sqrt{r}$ with a $\log r$, yielding a maximal runtime of

$$T = O(\log r)$$

Now, we have established an upper bound on T, which assumed that the worst case occurs when the projection of a ray into probe space is an entire diagonal of the probe texture. However, in general, this worst case will not occur, so we shall now discuss how we can get this run-time even lower.

Thus, we now have that the runtime of our algorithm runs in time proportional to the log of the length, in pixels, of the projection of a queried ray from world space to the octahedral space of a probe. This length is in turn related to the degree of parallelness of the displacement from the probe and the queried ray – which is measured by the $S^2(X-P)\cdot\omega$ quantity studied earlier (for a ray query (X,ω) and probe (L_P, r_P)). We note here that it is pivotal that we want to maximize the signed dot product, as a dot product of -1 would correspond to the pessimal case of a 180° projection..

The other extreme case is when $S^2(X-P)\cdot\omega = 1$, in which case $X-P$ and ω are parallel, then the projection of the ray (X,ω) into the probe space of P is exactly one pixel, and that pixel has exactly the correct value.

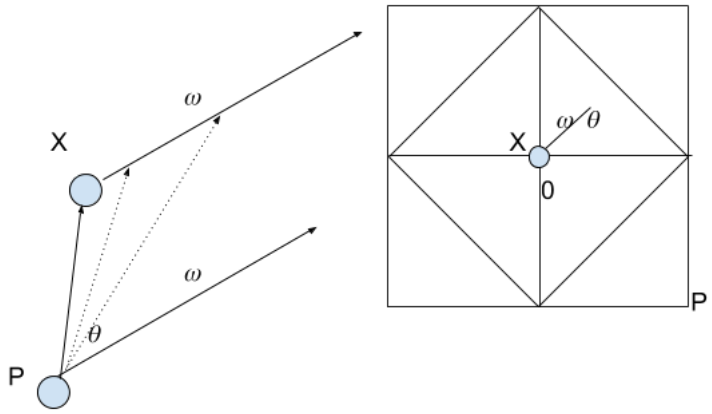


Figure 9.1: An Illustration of the situation. Here, $\theta = \cos^{-1}\left|S^2(X-P)\cdot\omega\right|$. The queried points from P will correspond to the point X itself, which we let correspond to angular deviation 0. Then, we can also receive query points along the ray, where the angle between the query ray and $X-P$ is less than θ. However, once we reach θ, the ray will be either parallel or pointing away from the ray (X,ω), and an intersection will never occur

The above illustration shows how the ray queries can be at an angle at most $\cos^{-1}(S^2(X-P)\cdot\omega)$ from $X-P$. Now, if we could have ray queries from $-90°$ to $90°$ degrees, this would correspond to a ray which occupies $\sqrt{r}$ pixels. In a hierarchical trace, we now have that the number of pixels that would need to be sampled in the worst case is

$$T = O\left(\log r\left(1+\cos^{-1}(S^2(X-P)\cdot\omega)\right)\right)$$

The constants that are being suppressed here include a factor of $\frac{1}{\pi}$ outside of the cosine, and a factor of $\frac{1}{2}$ as the quantity for the hierarchical run-time is really $\log\sqrt{r} = \frac{1}{2}\log r$. However, now we make an observation about this last version of the runtime: as we increase the number of probes, the expected worst case measure

of the parallelness metric will decrease, and thus both the expected number of samples necessary and the maximal number of samples necessary will decrease.

Thus, as we increase the density of a grid of probes (let n represent the number of probes), we decrease the amount of time for the algorithm to run, not only by reducing visibility failures (as was the original motivation for the parallelness heuristic), but also by guaranteeing the existence of a probe which will allow sufficiently parallel queries. Indeed, this analysis shows that, given a fixed probe resolution r, for any number of samples $k \geq 1$, there exists some ϵ such that, if we could generate a probe grid with n sufficiently large that there is always a selected probe with $S^2(X - P) \cdot \omega > (1 - \epsilon)$, and $v(P, (X, \omega)) = 1$, then each ray query can be answered with at most k samples. This theorem shows that, not only do we have the limiting case of having a probe at each point means that we can require exactly one query (as then using probes as environment maps is sufficient), but indeed as we increase our probe density from 1 probe we will steadily shrink the maximal number of required queries (assuming a non-fractal scene so that the visibility condition can be reasonably expected to be satisfied for some nearby probes). Formally we can write the run-time per pixel as

$$T = \left(\frac{\log r}{f(n)}\right)$$

Where n is the number of probes, and f is some nondecreasing function of n such that $T \to 1$ as $n \to \infty$ (for scenes with finite complexity).

We will now discuss the implications of the fact that in terms of input size the run-time of our algorithm (in a model of computation where high-speed memory is free in arbitrary quantities) *decreases* with input size in terms of number of probes n, and scales slowly with respect to probe resolution r. Therefore, while at any given point in time, available graphics hardware will restrict what densities of probe grids are practical, as available high-speed memory on graphics cards increases, this algorithm will become more and more efficient to implement. It is important to note, however, that the efficiency of this algorithm is affected only by the *amount* of memory available on a graphics device, not the bandwidth with which that memory can be shuttled between a GPU and a CPU, as is the case with many modern rasterization algorithms and the OptiX ray tracing algorithm [Parker et al., 2010]. This is because the information about the light probes provides is constant (assuming they do not need to be regenerated), and the only streaming input information is the camera's coordinate frame (rather than collections of triangles), and the only output information is the output image (possibly with depth and normals if additional rendering effects are to be applied).

The last note we make for this section is that, in the above discussion, we briefly discussed the existence of backface failures and grouped them together with visibility failures in terms of impact on run time. Intersection with backfaces is unlikely with nearly parallel probes, so in the limiting case it becomes less of a factor, and in addition, it can be solved simply by retracing the ray if a backface is found – thus, the run time becomes $O(\text{steps per probe} \times \text{Number of backface failures})$, but there can be at most 8 backface failures, so even if we have worst-case results with backfaces and do not handle them intelligently, the worst-case run time of the algorithm still only increases by a constant factor. However, with intelligent handling, by doing something of a "backwards trace" on the new probe, this could be bettered to a situation where we need to retrace only a small amount of a ray after a backface intersection; and we can indeed bound the amount of tracing necessary, as the improper backface intersection must have been within distance ϵ of the correct hit point.

9.2 Experimental

In this section, we will produce empirical evidence of the correctness and efficiency of our algorithm. The main purpose of these results is to show that the algorithm is capable of rendering complex standard 3D graphics scenes in real-time in modern hardware. The version of the algorithm which produced the results in this section used probes with 4K resolution (4096x4096), and used a 2D hierarchical trace to answer ray queries.

First we present a rudimentary proof of concept of the algorithm: in the Holodeck scene, which is a simple rectangular room, whose only variation is the inset where the door lies, our algorithm can generate the below image at a rate of over 60 fps using only a single probe.

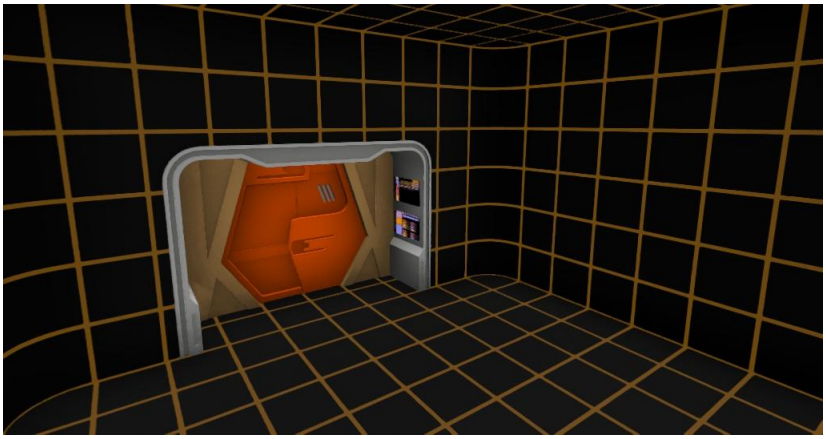


Figure 9.2: Holodeck rendered with 1 probe at 60 fps. Note that the probe origin is towards the center of the room, so this image requires interpolation from that point.

We note that though the above image seems relatively artifact-free, zooming up to an object does cause notable visual artifacts. This is, however to be expected, and is a common feature of most graphics algorithms. Just as algorithms which depend on a geometric model of a scene will reveal triangles at some resolution, so too does this algorithm reveal the pixels where probe samples were taken at some resolution.

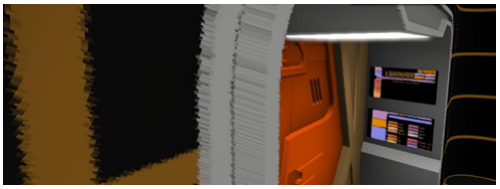


Figure 9.3: A closeup of the door in the Holodeck scene, showing individual pixels of the probe at the given resolution

Next, we show our algorithm running on a more complex common graphics scene, namely the San Miguel scene. This scene is not optimal for our algorithm, due to the complex visibility caused by the tree and tables, but the basic algorithm can still capture a number of key features, while running in real-time:

Figure 9.4: An image from of San Miguel rendered in real time using 4K probes in a relatively dense (about 30) probe grid. Though there are artifacts, the complex shadows from the tree, metallic look of the chairs, and other features are still visible.

Though the results images produced above do not match the state-of-the-art for other rendering algorithms, we note that they were produced with a relatively naïve version of the algorithm, and with further work to refine the algorithm by adding appropriate filtering, blending, and using other methods for computation of light (such as path tracing), can be vastly improved in further work.

Further, we note that the rendering algorithm, using 2D ray tracing techniques, even when all 8 probes around a point need to be sampled, can run at 60fps on modern hardware. This is promising for the goal of having an optimized version the algorithm which could be used for high frame-rate (such as virtual reality) environments, even when combined with additional techniques.

Chapter 10

Future Work

In this document, we have outlined the data structures, algorithms, and techniques necessary for approximation of the plenoptic function using light probes with depth. Much of the work presented here was in as general a framework as possible, so that many variations of the technique would be possible in future work. Open areas of research include doing the systems work to yield a highly accurate and efficient algorithm using all of the methods described in this document, which my collaborators, including my advisor Morgan McGuire, Michael Mara (Stanford), David Luebke (NVIDIA), and Derek Nowrouzezahrai (U Montreal) are focusing on.

Other open areas include investigating those parts of the algorithm for which we picked a simple method, but for which other options are available. For instance, all of our work assumes placement of the light probes on a uniform grid, but how easily can the results be extended to other grids? Should different heuristics be used in a tetrahedral grid for probe gathering? Can a totally non-uniform solution such as one that comes from partition into star-spaced regions be made efficient?

My results prompt new questions for the probe ordering problem. We did a small amount of empirical investigation of the Analytic Probe Ordering problem, but there may be interesting theoretical results there for anyone who wishes to study it. It is also not known how a partition of a space into star-shaped regions could be done efficiently.

I have focused on the probe selection problem, and the problem of handing off the trace across probes when visibility constraints change. My collaborators have been working on implementing this in the context of a high-performance, parallel and hierarchical tracing algorithm for path tracing. Our early results show that it is possible to trace primary visibility in real time for virtual reality applications, and full path tracing can be computed at interactive rates. These results are being readied for submission to ACM Transactions on Graphics in 2016.

This thesis has further developed the framework for real-time light probe based algorithms for rendering of light fields. However, the landscape of such algorithms remains largely unexplored. While it has been hypothesized that results could be obtained using real-world cameras and depth cameras, this has not been tested, and while it is further conjectured that a variety of previous rendering techniques can be rephrased in terms of the light probe based framework, there still remains plenty of work to be done on how to perform those re-phrasings. Thus this thesis ends at the point of establishing a foundation, and encourages further research in exploring the diversity of algorithms which could be built out of the framework discussed here.

Bibliography

John Amanatides, Andrew Woo, and others. 1987. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, Vol. 87. 10.

Kevin Bjorke. 2004. Image-based lighting. *GPU Gems* (2004), 307–322.

James F Blinn and Martin E Newell. 1976. Texture and reflection in computer generated images. *Commun. ACM* 19, 10 (1976), 542–547.

Vasek Chvatal. 1975. A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory, Series B* 18, 1 (1975), 39–41.

Zina H. Cigolle, Sam Donow, Daniel Evangelakos, Michael Mara, Morgan McGuire, and Quirin Meyer. 2014. A Survey of Efficient Representations for Independent Unit Vectors. *Journal of Computer Graphics Techniques (JCGT)* 3, 2 (17 April 2014), 1–30. `http://jcgt.org/published/0003/02/01/`

Naughty Dog. 2014. The Last of Us Remastered. Video Game. (2014).

William Donnelly. 2005. Per-pixel displacement mapping with distance functions. *GPU gems* 2, 22 (2005), 3.

William Donnelly and Andrew Lauritzen. 2006. Variance shadow maps. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. ACM, 161–165.

Daniel Evangelakos. 2015. A Light Field Representation for Real Time Global Illumination. Williams College Undergraduate Thesis. (5 2015).

Henry Fuchs, Zvi M Kedem, and Bruce F Naylor. 1980. On visible surface generation by a priori tree structures. In *ACM Siggraph Computer Graphics*, Vol. 14. ACM, 124–133.

Steven J Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F Cohen. 1996. The lumigraph. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM, 43–54.

David S Immel, Michael F Cohen, and Donald P Greenberg. 1986. A radiosity method for non-diffuse environments. In *ACM SIGGRAPH Computer Graphics*, Vol. 20. ACM, 133–142.

James T Kajiya. 1986. The rendering equation. In *ACM Siggraph Computer Graphics*, Vol. 20. ACM, 143–150.

Tom Duchamp Hugues Hoppe Linda Shapiro Kari Pulli, Michael Cohen and Werner Stuetzle. 1990. View based Rendering: Visualizing Real Objects from Scanned Range and Color Data. `http://graphics.stanford.edu/~kapu/vbr/webslides`. (1990).

Sébastien Lagarde. 2012. Image-based Lighting approaches and parallax-corrected cubemap. https://seblagarde.wordpress.com/2012/09/29/image-based-lighting-approaches-and-parallax-corrected-cubemap/. (2012).

Marc Levoy and Pat Hanrahan. 1996. Light field rendering. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM, 31–42.

Morgan McGuire, Daniel Evangelakos, James Wilcox, Sam Donow, and Michael Mara. 2013. *Plausible Blinn-Phong reflection of standard cube MIP-maps*. Technical Report. Citeseer.

Morgan McGuire and Michael Mara. 2014. Efficient GPU Screen-Space Ray Tracing. *Journal of Computer Graphics Techniques (JCGT)* 3, 4 (9 December 2014), 73–85. http://jcgt.org/published/0003/04/04/

F Kenton Musgrave. 1988. *Grid tracing: Fast ray tracing for height fields*. Technical Report. Research Report No. RR-639, Dept. of Computer Science, Yale Univ.

NVIDIA. 2016. IRay VR. https://blogs.nvidia.com/blog/2016/04/05/vr-with-iray/, https://blogs.nvidia.com/blog/2016/04/05/vr-with-iray/. (2016).

Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and others. 2010. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics (TOG)* 29, 4 (2010), 66.

Tomasz Stachowiak. 2015. Stochastic Screen-Space Reflections. SIGGRAPH.

László Szirmay-Kalos, Barnabás Aszódi, István Lazányi, and Mátyás Premecz. 2005. Approximate Ray-Tracing on the GPU with Distance Impostors. In *Computer graphics forum*, Vol. 24. Wiley Online Library, 695–704.

Eric Veach and Leonidas J Guibas. 1997. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co., 65–76.

S Widmer, D Pająk, A Schulz, K Pulli, J Kautz, M Goesele, and D Luebke. 2015. An adaptive acceleration structure for screen-space ray tracing. In *Proceedings of the 7th Conference on High-Performance Graphics*. ACM, 67–76.