RADIANCE CACHING WITH ENVIRONMENT MAPS

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Michael Buerli

June 2013
### COMMITTEE MEMBERSHIP

<table>
<thead>
<tr>
<th>TITLE: Radiance Caching With Environment Maps</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTHOR: Michael Buerli</td>
</tr>
<tr>
<td>DATE SUBMITTED: June 2013</td>
</tr>
<tr>
<td>COMMITTEE CHAIR: Zoë Wood, Ph.D., Associate Professor, Computer Science Department</td>
</tr>
<tr>
<td>COMMITTEE MEMBER: Foaad Khosmood, Ph.D., Assistant Professor, Computer Science Department</td>
</tr>
<tr>
<td>COMMITTEE MEMBER: Aaron Keen, Ph.D., Associate Professor, Computer Science Department</td>
</tr>
</tbody>
</table>
Abstract

Radiance Caching With Environment Maps

by

Michael Buerli

The growing demand for realistic renderings in both film and games has led to a number of proposed solutions to the Global Illumination problem. In order to imitate natural lighting, it is necessary to gather indirect illumination of the surrounding environment for lighting computations. This is a computationally expensive problem, requiring the sampling or rasterization of the hemisphere surrounding each ray intersection, to which there is no standardized solution.

In this thesis we propose a new method of approximation using environment maps for caching radiance. The proposed method leverages a voxelized scene representation for storing direct illumination and a cache of environment maps for integrating indirect illumination. By using a voxelized scene to gather indirect lighting contributions and caching these contributions spatially, we are able to achieve fast and convincing renders of large complex scenes.

The result of our implementation produces images comparable to those of existing Monte Carlo integration methods with render speeds a magnitude or more faster.
## Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Global Illumination .................................................. 1

1.2 Geometric Complexity .................................................. 3

1.3 Environment Maps ..................................................... 6

1.4 Our Contribution ...................................................... 7

2 Background and Previous Work 8

2.1 Octrees ................................................................. 8

2.2 Voxelization ............................................................. 9

2.3 Radiance ................................................................. 10

2.3.1 Irradiance ............................................................ 12

2.3.2 The Lighting Equation .............................................. 13

2.4 Ray Tracing ............................................................... 14

2.5 Global Illumination .................................................... 15

2.5.1 Monte Carlo Integration ............................................ 16

2.5.2 Photon Mapping ..................................................... 16

2.5.3 Point-Based Color Bleeding ....................................... 17

2.5.4 Virtual Point Lights ................................................. 18

2.5.5 Voxels ................................................................. 19

2.6 Caching ................................................................. 19

2.6.1 Irradiance Caching .................................................. 20
3 Radiance Caching With Environment Maps

3.1 Overview

3.2 Caching Schemes
   3.2.1 Single Map Caching
   3.2.2 Nearest Map Caching
   3.2.3 Neighborhood Caching
   3.2.4 Compound Caching

3.3 Concurrency

3.4 Voxelization
   3.4.1 Refinement Criteria
   3.4.2 Voxel Shading
   3.4.3 Node Structure
   3.4.4 Voxel Structure
   3.4.5 Octree Traversal

3.5 Environment Map
   3.5.1 Spherical Mapping
   3.5.2 Direction Adjustment
   3.5.3 Environment Map Structure

3.6 Indirect Illumination using Environment Maps
   3.6.1 Environment Map Integration
   3.6.2 Interpolating Environment Maps

3.7 Performance Trade-offs

4 Results

4.1 Implementation
   4.1.1 Threading
   4.1.2 Memory
   4.1.3 System
4.2 Test Scenes ................................................. 50
4.3 Performance Metrics ................................. 51
4.4 Render Comparison .................................. 52
  4.4.1 Sampling Error .................................. 52
  4.4.2 Environment Map Resolution .................. 54
  4.4.3 Environment Map Sampling .................... 55
  4.4.4 Caching Schemes ................................. 56
4.5 Analysis ................................................... 58
  4.5.1 Memory ............................................. 59
  4.5.2 Speed .............................................. 59
  4.5.3 Accuracy .......................................... 60
4.6 Conclusion ............................................... 61

5 Future Work ............................................. 63
  5.1 Voxels ................................................... 63
  5.2 Memory Usage ....................................... 64
  5.3 Irradiance Gradients ............................... 64
  5.4 Adaptive Environment Maps .................... 65

Bibliography ............................................... 66
List of Tables

2.1 Radiant Units .......................................................... 12
3.1 Octree Node Structure ............................................. 34
3.2 Environment Map Attributes ...................................... 43
4.1 Sampling Performance Comparison .............................. 53
4.2 Environment Map Resolution Comparison ..................... 54
4.3 Multi-sampling Comparison ....................................... 55
4.4 Caching Scheme Performance .................................... 57
## List of Figures

1.1 Example scene with direct illumination (left) and one bounce global illumination (right). .................................................. 2

1.2 A scene rendered with single sampling (left) and with supersampling (right). ................................................................. 4

1.3 Geometry coarsely represented as point samples [2]. ............ 5

2.1 Octree representation. Credit: Nü (Creative Commons License) . 9

2.2 Voxelized bunny. ................................................................. 10

2.3 Radiance at a point $\mathbf{x}$ with respect to the surface and its normal $\mathbf{n}$. 11

2.4 Example of ray tracing a scene. Credit: Henrik (Creative Commons License). ........................................................... 15

3.1 The scene is voxelized (1), then during the render phase (2) radiance is cached and integrated for indirect illumination. ....... 24

3.2 Radiance from the scene is stored in voxels (blue) from which radiance is sampled and cached in environment map texels (green) to later be integrated into irradiance. ........................... 25

3.3 Voxelized representation of the Sponza Atrium. .................. 31

3.4 Subdivision and shading of a single voxel (blue). Samples are averaged to provide a more accurate approximation. ............. 33

3.5 Environment visualized as a sphere. ................................. 36

3.6 Adjustment of direction $\tilde{\mathbf{w}}_0$ using an environment map at location $\mathbf{x}_m$. ................................................................. 40

4.1 Sponza Atrium Scene. ................................................... 50
4.2 Error and noise due to varying numbers of samples (2, 10, 20, 40, 100, 200). 53

4.3 Three caching schemes and reference image of Sponza Atrium; Single Map Caching, Nearest Map Caching, Neighborhood Caching, Reference Image (top left to bottom right) 56

4.4 Caching artifacts of environment map, nearest map and neighborhood caching schemes. 57

4.5 Three caching schemes and reference image; Single Map Caching, Nearest Map Caching, Neighborhood Caching, Reference Image (top left to bottom right) 58

4.6 Comparison of direct, indirect and combined illumination. Top rendered using only Monte Carlo methods. Bottom rendered with neighborhood environment map caching. 59

4.7 Direct only render. Monte Carlo render (1045s). Neighborhood environment map caching render (68s, 58s subsequent). 60

4.8 Sibenik Atrium render comparison. 62
Chapter 1

Introduction

In computer graphics, software is used for the realization of images from virtual environments. Physical and abstract objects are represented as geometry with numeric values and equations. This scene geometry is then rendered into pixels which make up the images that we see. Rendering, while achieved by a variety of techniques, comes down to the basic sampling of light and light’s interactions in a scene. Over the past years, these techniques have rapidly grown in complexity and efficiency producing many advances in real-time graphics and film. This paper proposes a new strategy for more efficiently approximating the lighting of a scene.

1.1 Global Illumination

With the aim of creating more photo-realistic images, Global illumination algorithms provide a more complex simulation of light in order to more accurately model illumination in a scene. For the most basic lighting situation, direct illumination, each visible fragment of geometry is shaded by modeling only the
light that comes directly from a light source. Natural lighting, however, takes into account light not only from light sources but the whole environment. *Global illumination* algorithms model this by taking contributions not only from direct illumination but also from multiple bounces of indirect illumination. An example of the difference achieved from incorporating just one bounce of indirect illumination can be seen in Figure 1.1. The pitfall of gathering indirect illumination is that it creates almost exponentially more computational work, leading to much longer render times.

This increase in computational complexity is mitigated by approximating and memoizing indirect illumination. A bulk of these methods use a two pass algorithm. Where the first pass concentrates on creating a data structure to represent light in the scene, and the second pass (the rendering pass) gathers indirect illumination from the structure. This limits the amount of work done shading indirect lighting contributions by more efficiently gathering it from a data structure. Much of the indirect lighting can also be approximated due to the gradual or smooth variance in indirect lighting over a surface. However, integrating the
indirect illumination still requires gathering a hemisphere of samples for each
directly shaded fragment. For homogeneous regions and regions with mostly dis-
tant indirect lighting contributions, this becomes inefficient with the redundant
gathering of like samples.

1.2 Geometric Complexity

Another factor that is limiting performance in current rendering systems is ge-
ometric complexity. As scenes strive for greater realism, the amount of geometry
and detail used to represent a scene increases. Render times, in turn, are directly
affected by the amount of geometry; linearly and sometimes quadratically. With
the addition of more geometric primitives, renderings of the scene are required
to iterate or search through a larger amount of data. For indirect illumination,
this is especially costly as a view of the entire scene needs to be sampled for each
shaded fragment. This problem is boiled down to being able to efficiently sample
a scene to get the most accurate approximation. Four different approaches can
be employed to reduce this cost.

**Sampling:** The first approach to reduce the cost of high geometric detail is
through better sampling. One can accurately approximate a region of geometry
at a given resolution through an appropriate sampling distribution. According
to the Nyquist-Shannon sampling theorem [19], this distribution needs to match
at least twice the highest signal frequency in order to be accurate. In the case of
a scene represented by geometric primitives, twice the size of the smallest prim-
itive needs to be sampled in order to be completely accurate. If the sampling
of geometry is not at a high enough resolution, the result will introduce aliasing
artifacts like those seen in Figure 1.2. *Supersampling*, or multi-sampling, of the geometry resolves the aliasing problem, however greatly increases the amount of work. Increasing the sampling rate in general is extremely inefficient in scenes with highly varying geometric complexity. This is because of the high sampling resolution that is needed for only small complex portions of what is overall sampled. While adaptive and stochastic sampling methods have been developed to limit this, the overall scalability of sampling is still not very good.

![Figure 1.2: A scene rendered with single sampling (left) and with supersampling (right).](image)

**Level Of Detail:** Another approach for approximating geometric complexity relies on geometric Level Of Detail (LOD) algorithms, namely mesh simplification. Mesh simplification modifies the geometry itself by removing and averaging primitives from a model giving a less precise representation with fewer total primitives. Limiting the number of geometric primitives for a given resolution will not only give better sampling performance, but also deal with the aforementioned
aliasing issues. However, another issue introduced with the rasterization of surface geometry is *overshading*. Because rasterizing geometry requires each polygon to be rendered to pixels, rasterization of micropolygons (polygons smaller than a pixel) leads to many of the overlapping pixels to be rendered more times than necessary. While this issue is again resolved by using LOD, the disadvantage of such an approach is that it does not always provide an accurate approximation especially for sharp geometry or handles. While many approaches allow the LOD representations to be tuned and made more accurate, the overall usefulness of this approach is limited to single models and objects.

![Figure 1.3: Geometry coarsely represented as point samples [2].](image)

**Point clouds:** Moving away from geometry completely, the geometry can be pre-filtered and represented as a set of point samples instead (Figure 1.3). This approach, employed in many global illumination techniques, removes the high variance of complexity allowing for faster gathering with less aliasing. Organizing the point cloud of samples into a data structure improves the performance of gathering such samples while also allowing for hierarchical approximations of the samples to be made. While this technique requires an additional setup phase, the overall performance gain is worthwhile. The only other disadvantage to pre-filtering is the inherent view dependence that results from sampling the scene a certain way. More adaptive sampling and perhaps tuning may be required for
Volumetric rendering: One can approximate geometry using volumetric rendering. This is similar to pre-filtered sampling, converting geometry into a different representation, only this time geometry is sampled in terms of volumetric units (voxels). Much like samples, voxels store lighting attributes at a given resolution in volumetric space. The advantage of using voxels is that they describe axis-aligned regions of space which approximate all the contained space, not just the sampled surface representation. This allows us to gather information on the surrounding neighborhood and better approximate larger regions of space. Unlike geometric LOD, volumes provide a better LOD approximation by preserving illumination through a much quicker averaging. Overall, volumetric rendering can provide a quick alternative to point clouds.

1.3 Environment Maps

Environment maps are virtual structures that provide a 360 degree view at a given location. This basically serves as a snapshot of the surrounding environment at a given point in a scene. In rendering, these structures have been used to both emulate a much larger environment past the scene’s edge as well as approximate reflections and other environmental contributions on geometry. These structures lend themselves well to the storage and mapping of directional light, providing a mechanism for lighting approximations.
1.4 Our Contribution

This paper presents a ray tracing algorithm to pre-filter and cache light with environment maps. By limiting the amount of redundant light gathering and dealing with high geometric complexity using voxels, we decrease render times by more than an order of magnitude. The proposed algorithm provides a fast approximation to indirect illumination by using a cache to compute indirect lighting contributions. Additionally, a volumetric representation of the scene is used to approximate the light of the scene during the pre-filtering pass, improving gather performance.

The main contributions of this thesis are:

1. A rendering algorithm for fast approximate indirect illumination.
2. An environment map structure for radiance caching.
3. Radiance approximation and mapping algorithm extensions.
4. New environment map caching strategies.

The rest of the document is organized as follows. Section 2 provides details on the background and previous work. Section 3 describes the environment caching algorithm and caching strategies. Section 4 shows the results of our implementation and Section 5 concludes the work and provides insight on future work.
Chapter 2

Background and Previous Work

The radiance caching and irradiance caching techniques used in this thesis build upon a wide range of techniques. Many of these fundamental algorithms and techniques provide insight into methods of caching and performance improvements.

2.1 Octrees

Octrees [17], the 3D equivalent of a quadtree, provide a spatial data structure for more efficient querying of geometry. Space is broken up into axis aligned boxes, where the head of the tree contains the whole scene and each subsequent node is a partition thereof. As seen in Figure 2.1, the number of subdivisions varies depending on where more detail is required. Nodes are partitioned by each axis into eight equal blocks, much like a quadtree breaks 2D space into four squares. This data structure is widely used for its $O(\log N)$ search time and spatial locality information associated with the space partitioning.
2.2 Voxelization

Building upon the octree structure, octrees can also be used to represent and store geometry. In volumetric rendering, instead of representing only the surfaces of geometry, volumes of space must be taken into account. A voxel represents an axis aligned block of space with geometric or lighting properties. Voxelization is the conversion of scene geometry into a representation of voxels. The benefit of this conversion, as mentioned earlier, is the independence from geometric complexity and fast approximation introduced by using axis aligned blocks.

Just as octrees can be sparse, only containing leaf nodes for filled space, so too can voxel octrees. Additionally octrees can be adaptive, representing more complex regions of geometry with finer voxel resolution (Figure 2.2). Sparse voxel octrees [16] provide an efficient and adaptive structure with near constant lookup time. The achievements of voxel rendering can be seen in the Crassin et al. GigaVoxel architecture [6]. Voxels provide a speedy alternative to point clouds, with only a small decrease in accuracy.
2.3 Radiance

The interaction of light and geometry is captured in terms of radiance. This model is obtained from radiometry, the measurement of radiant energy in the real world. Radiant energy $Q$, observed in the world, provides a measurement of energy for light. In order to use this measurement for rendering a single image, the energy at a specific moment in time is used. Radiant flux $\Phi$, the unit of energy per unit time, is derived from the radiant energy ($\Phi = \frac{dQ}{dt}$). While flux can be used to describe light in a scene, a more convenient measurement relies on flux’s distribution across a surface. To discretize radiant flux over geometry, the relation of solid angle to the area on a surface is used to to derive our base unit radiance $L$. Given the flux at point $x$ with normal $\vec{n}$, solid angle $d\vec{w}$, and projected area $\cos \theta dA$ the radiance can be visualized as in Figure 2.3.
Figure 2.3: Radiance at a point \( \mathbf{x} \) with respect to the surface and its normal \( \mathbf{n} \).

and calculated as such:

\[
L(\mathbf{x}, \mathbf{\hat{w}}) = \frac{d^2 \Phi}{\cos \theta dA d\mathbf{\hat{w}}} \tag{2.1}
\]

The projection of \( dA \), which can also be written as the dot product \( \mathbf{n} \cdot \mathbf{\hat{w}} \), compensates for the weakening factor of total incoming radiance with regard to the angle at which the radiance hits the surface. This can be likened to the area of a surface seen from different angles. The direction perpendicular to the surface, or aligned with the normal, receives the maximum amount of radiance which tapers off along the hemisphere away from the normal. Radiance simplifies the integration of light interacting with geometry by separating light into cones. The units of radiance can be found in Table 2.1. Important properties of radiance worth noting:

1. Radiance, like light, is additive. The combined effect of two samples of radiance is the same as the sum of the two.

2. The radiance from a given point \( \mathbf{a} \) to a point along its direction vector \( \mathbf{b} \)
does not change, unless acted upon by geometry. Therefore the radiance from \( b \) to \( a \) is also equivalent.

### 2.3.1 Irradiance

The total incoming radiance, or *irradiance*, is computed for a given point by integrating all incoming radiance over the surrounding hemisphere. This can be computed with the following equation:

\[
E(x) = \int_{\Omega} L_i(x, \vec{w}) \cos\theta d\vec{w}.
\]

(2.2)

This integral again takes into account the weakening factor introduced by the projection of \( dA \) for the given solid angle \( d\vec{w} \), in this case affecting the amount of incoming radiance absorbed. The units of irradiance (Table 2.1) are similar to that of radiance, however independent of the solid angle. Here we can see that the conservation of radiance holds. For all outgoing radiance discretized from the radiant flux, the reverse incoming radiance integrates to the same value for a given surface. Irradiance is particularly important for computing the lighting of a point and becomes an even greater concern with global illumination.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Property</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Q )</td>
<td>Radiant Energy</td>
<td>J</td>
</tr>
<tr>
<td>( \Phi )</td>
<td>Radiant Flux</td>
<td>W</td>
</tr>
<tr>
<td>( L )</td>
<td>Radiance</td>
<td>( Wsr^{-1}m^{-2} )</td>
</tr>
<tr>
<td>( E )</td>
<td>Irradiance</td>
<td>( Wm^{-2} )</td>
</tr>
</tbody>
</table>

Table 2.1: Radiant Units
2.3.2 The Lighting Equation

The interaction of radiance with different types of surfaces can result in different outcomes of shading. This concept of material properties is supported by Bidirectional Reflectance Distribution Functions (BRDFs) [12] as pioneered by Kajiya in 1986. BRDFs provide a model for which the lighting at a given point can be integrated. Represented as a function, the BRDF takes into account the position, direction, and intensity of incoming light for the direction at which the point is being viewed. For example, a shiny material will reflect more radiance at the angle of reflection than a strictly Lambertian (diffuse) material. Additionally, materials alter the spectrum of incoming radiance to produce a given color for the surface. As seen in equation 2.3 below, the BRDF returns the portion of radiance reflected from a given value of incoming radiance. The change in irradiance, seen in the denominator is simply the derivative of projected incoming radiance (derived from equation 2.2). This formal description defines how a BRDF can be found:

\[
f(x, \vec{w}_i, \vec{w}_r) = \frac{dL_o(x, \vec{w}_r)}{dE(x, \vec{w}_i)}.
\] (2.3)

Defining a BRDF ranges through a variety of approximations. Flat shading, the simplest approximation returns a constant value for all values. More realistic lighting considers ambient, diffuse, and specular contributions for incoming radiance. Phong shading [22] provides a more realistic model, by providing coefficients for each of these lighting factors. Beyond this, the BRDF can be expressed functionally with different effects to the spectrum and reflectance of light. Ultimately,
the shading of a fragment can be compounded to a single lighting equation:

\[ L_o(x, \vec{w}_r) = \int_\Omega f(x, \vec{w}_i, \vec{w}_r) L_i(x, \vec{w}_i) \cos \theta_i \, d\vec{w}_i. \]  

(2.4)

Where for a position \( x \) and viewing direction of \( \vec{w}_r \), a radiance value can be obtained. In summary, radiometry introduces three new properties used to model light: radiant flux, radiance, and irradiance. These gathered properties culminate in the calculation of the lighting for a given intersection, which in rendering is repeated many times.

### 2.4 Ray Tracing

Ray tracing is the discretized sampling of the radiance in a scene using rays. First introduced by Appel in 1968 [1], ray casting involves casting a ray for every pixel in an image originating from a single camera position. A ray is defined as a position - in this case the camera - and a direction, determined by the intersecting pixel. Each ray’s intersection with the scene’s geometry is then computed mathematically to determine the correct shading for the corresponding pixel.

This fundamental task of casting rays is improved upon with the casting of secondary rays [33] and eventually recursive ray casting. The primary use of a secondary ray is to determine where a shadow lies. Once the intersection of the primary ray is computed, a secondary ray can be cast from the intersection point towards the light source. If this ray intersects geometry before the light source is reached, the fragment is shadowed and therefore shaded differently. This method of casting rays is visualized in Figure 2.4, with the intersections
of rays on a sphere. In addition to shadow rays, secondary rays can be used to emulate reflections and refractions by casting an additional ray from the point of intersection. This core idea has lead to what is now a large part of ray tracing, finding ways to improve, approximate and circumvent sending secondary rays.

A great deal more detail and resources for ray tracing are provided in Physically Based Rendering: From Theory to Implementation [20].

2.5 Global Illumination

For more realistic lighting, shading takes into account not only direct light sources but radiance from the entire scene. This process of computing irradiance from the surrounding environment, requires integrating a magnitude more secondary ray casts on the hemisphere about an intersection. The reason for this integration stems from the fact that in nature light bounces of all objects. So in actuality almost everything is a source of radiance. By casting secondary rays across the scene, we integrate the incoming radiance. However, each subsequent
ray requires additional rays in order to compute its lighting contribution. To mitigate this need to cast an infinite amount of rays from every ray intersection, a number of methods and techniques have been developed and built upon to help approximate this.

2.5.1 Monte Carlo Integration

Monte Carlo integration and related Monte Carlo techniques have many applications throughout math and science. The technique proposes using a random sampling heuristic in order to better approximate complex systems. In terms of global illumination, Monte Carlo integration is used to approximate the irradiance integral. Randomly sampling or stochastically sampling the hemisphere for a given distribution gives an accurate approximation with an order of magnitude fewer samples.

The downside to using this technique manifests itself as noise when too few samples are taken. While the reduction in sample size needed is substantial, the method is still a significant hit to performance and is not as scalable as other methods.

2.5.2 Photon Mapping

Deviating from the idea of casting ray after ray in order to compute the total lighting contribution, light instead can be stored and gathered. Photon mapping [11, 10], introduced by Jensen, does just that. The method breaks rendering into two phases, constructing the photon map and a render phase.

During construction of the photon map, photons of light are cast into the scene
from all of the light sources. Photons, representing a fraction of the flux radiating from each light source, are stored in a k-d tree at the point of intersection. A portion of the flux is also cast off from the intersection point, determined by a Russian roulette heuristic. In this way, photons originating from a light source are bounced around the scene randomly dying off with more certainty as they go. This structure of photons represents the radiant flux in the scene which can now be queried much faster than casting additional rays.

The render phase performs the usual ray casting, however for indirect lighting contributions it instead queries the photon map. Nearby photons are gathered and interpolated resulting in an approximate irradiance value. Photon mapping is also well known for simulating caustics. The same method of gathering is used, only with a separate caustic map focused on reflections and refractions. Overall this method fairs well for approximating global illumination, limited only by the accuracy and distribution of photons in the scene.

2.5.3 Point-Based Color Bleeding

Similar to photon mapping, in 2008 Christensen proposed Point-Based Approximate Color Bleeding (PBACB)[3]. This two phase method, instead of using photons, stores a point cloud of surfels with the direct lighting contribution. Instead of casting samples from each light and bouncing light around the scene, the scene is sampled from a pulled back location to more accurately represent the geometry as surfels. Surfels are a disk-like representation of outgoing radiance, storing an area, color, and normal. They are used to sample the geometry uniformly by overlapping these disks so there are no visible gaps. The surfel area is determined by the solid angle to the camera. Surfels closer to the camera have a
smaller area, effectively making the sampling more adaptive.

The gather or setup phase constructs the point cloud of surfels and stores it in an octree. Later, spherical harmonics are used to approximate higher levels in the octree, spanning multiple surfels. This way, surfels that are further away can be approximated as groups rather than sampling over all surfels.

As with photon mapping, the render phase starts with the usual rendering, however differs in gathering indirect illumination. For PBACB, instead of interpolating between nearby samples, the surfels are instead rasterized to a cube-map. Each axis aligned cube-map represents the hemisphere surrounding a given intersection. The surfels are rasterized onto each face and then the radiance is integrated.

This approach offers the same advantages of photon mapping with less noise. Because of the adaptive sampling of the geometry as a point cloud, the radiance values are more continuous and concentrated at areas of more detail.

2.5.4 Virtual Point Lights

A different approach altogether is to create Virtual Point Lights (VPLs) to represent sample locations of radiosity. First introduced in 1997 by Keller’s Instant Radiosity [13], the method skips the irradiance integration by instead computing the lighting multiple times for different VPLs. This method, however, still requires a setup pass where rays are cast from all light sources, creating new virtual point lights at each bounce. No gathering is required during rendering, however intersections must compute lighting for all VPLs.

The use of VPLs has since been built upon with new methods in real time [28] and screen space approximations [18]. Most recently, Christensen et al. has
introduced a method of multi-resolution radiosity caching [5]. This single pass algorithm caches radiosity in each surface patch structure, much like a map VPLs, and computes indirect illumination from these patches.

2.5.5 Voxels

Most recently, interactive techniques have been introduced that make use of voxels. Voxels represent volumetric regions of space and their properties. Using voxel cone tracing, Crassin et al. [7] was able to achieve interactive indirect illumination as an approximation. VGI [30], a similar approach, uses path tracing with voxels to achieve performance improvements.

These methods, again require two phases. First the scene is voxelized, pre-computing direct lighting contributions. Then, when rendering, the voxel octree is queried, either by cone tracing or path tracing and radiance is integrated. While not as accurate as the previous methods, the use of a voxelized scene provides the best improvement in performance. For this reason, a voxelized scene is used in our proposed algorithm.

For more detail on global illumination methods, Ritschel et al. [26] provide a comprehensive survey of current techniques.

2.6 Caching

In addition to, or perhaps to complement, the above global illumination techniques, caching methods have been developed to boost gathering performance. The motivation for caching indirect illumination was first explored by Ward et al. [32]. Irradiance values across a surface vary only gradually, with little high
frequency contributions. This is due to the fact that irradiance is integrated from the scene’s radiance visible at a point. Translations or rotations to this hemisphere alter the integrated value only slightly. Therefore, it is safe to approximate or reuse parts of the irradiance integration.

2.6.1 Irradiance Caching

Irradiance Caching [32] takes advantage of the low frequency of irradiance by caching values in the surface geometry. Instead of integrating irradiance for every pixel, only a subset of pixels compute irradiance values. These values are then integrated over to create smooth indirect illumination. Ward et al. improves their method with the addition of irradiance gradients [31]. Irradaince gradients improve the accuracy of interpolating irradiance values, by providing a way to align an irradiance distribution both through translation and rotation.

In 2004, Christensen et al. [4] introduced a similar method of caching irradiance through an irradiance atlas. Irradiance values precomputed from a photon map, are stored in a 3D MIP map representation as voxels. Each MIP map is represented as a brick of $8^3$ voxels and stores the approximate irradiance values.

2.6.2 Radiance Caching

A similar method to irradiance caching, Radiance Caching [15] was developed by Krivanek et al. with more focus on glossy surfaces. Instead of caching irradiance values at a surface, directional radiance is stored using hemispherical harmonics or spherical harmonics. This method also incorporates an adapted version of irradiance gradients in order to align the radiance representation. Further papers [14] expand on this idea, addressing problems introduced by ray leaking,
and improving neighborhood interpolation with neighborhood clamping.

Newer techniques have also been proposed that leverage discrete representations of radiance. Micro-Rendering [25], proposed by Ritschel et al., is a final gathering approach focused on GPU parallelization. Radiance is stored in a point cloud of surfels and later rasterized in hemispherically mapped images, weighted by the BRDF. Based on a photon map, radiance is first gathered into a binary tree of surfels. During rendering, the surfels are micro-rendered based off a solid angle distribution provided by the BRDF. While radiance is not cached in this approach, the use of micro-rendered images does provide a technique for speeding up irradiance integration. This method is similar to Point-Based Color Bleeding [3], where radiance is rasterized to a cube-map, only with a warped spherical mapping of pixels for importance sampling the BRDF.

Most recently, Pre-convolved Radiance Caching [27] introduces the idea of using MIP mapped textures to cache radiance. Radiance values from the scene are gathered and pre-convolved per cache. The result is a set of MIP mapped textures representing different levels of glossy irradiance. This method achieves constant per-pixel lookup, by interpolating irradiance values of a given glossiness from a neighborhood of MIP maps.

### 2.6.3 Radiosity Caching

As mentioned before, methods in radiosity caching also exist. For instance, there is one method developed by Christensen et al. [5]. The goal of this type of caching is to instead cache radiance at its source. Much like the use of dozens of VPLs, radiosity (outgoing radiance) is stored and MIP mapped in source control patches. This alleviates the need for a two pass method, however requires
geometry to be more well defined.

### 2.7 Environment Maps

An environment map is a virtual structure that represents incoming light at a given position. This structure was first described as a projection of the world onto a cube by Greene [9] in 1986. The framework proposed, projects a 3D viewpoint onto a 2D image. Later applications vary in both the structure and mapping of pixels to spherical and other warped mappings.

Environment maps may also be represented with spherical harmonics [23]. Instead of discrete colors or values, coefficients to a set of continuous functions are used to represent the same mapping of color. Here resolution is determined by the number of coefficients provided.
Chapter 3

Radiance Caching With Environment Maps

The goal of many global illumination techniques centers around the efficient approximation of irradiance for all shaded fragments. In order to minimize the amount of rays cast into the scene, radiance and radiant flux are precomputed, stored, or cached. Environment maps lend themselves well to the storage of radiance and integration of irradiance. They provide an efficient, discretized representation of spheres or hemispheres of radiance. This paired with the fast approximation of radiance values using a sparse voxel octree provides the basis for efficient irradiance approximation. The remainder of this chapter describes our approach to approximate irradiance by caching radiance with environment maps.
3.1 Overview

The algorithm is broken into two phases, a setup phase and a render phase (Figure 3.1). In the setup phase, the scene geometry is voxelized and stored in a sparse voxel octree. This voxelization provides a representation of the radiance in the scene, storing the direct lighting contributions of volumetric regions of space. Next, in the render phase, the voxel representation is used to gather incoming radiance into our cache of environment maps. For a given ray intersection the cache of environment maps is queried, adding a new environment map if one does not already exist for the neighborhood. The gathered radiance, now stored in or retrieved from our environment map cache, can now be integrated for the indirect illumination of our intersection. Because radiance is stored spatially in environment maps, only the first intersection of a given region needs to gather radiance from the voxel octree. All subsequent intersections can go directly to the cache, allowing for much faster integration of incoming radiance. Additionally, as neighboring environment maps are filled, irradiance values can be interpolated to provide smoother results.
3.2 Caching Schemes

Our method proposes four different caching schemes to approximate indirect illumination, focusing on three for final implementation. The caching scheme defines the manner in which the environment map cache is created and how irradiance is integrated from it.

3.2.1 Single Map Caching

Our first caching scheme localizes regions to a single environment map for radiance caching. During rendering, after a ray intersection has computed its direct lighting contribution, the indirect illumination is computed by consulting the sparse voxel octree. First, we find the leaf octree node that contains the intersection position. If this octree node contains an environment map cache, we use it to integrate irradiance, otherwise one is created. In this way, the first intersection in a given octree node caches the incoming radiance which can then be used by the remainder of intersections. Initializing and gathering radiance for
a given cache is done using the voxelized scene and the position of the first given intersection. Because the environment map only represents a certain resolution, a course voxelization is sufficient to construct the approximate incoming radiance. Gathering radiance from this structure, rather than the actual scene, improves gather time significantly, especially in cases of high geometric complexity.

<table>
<thead>
<tr>
<th>Data: intersection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result: irradiance value</td>
</tr>
</tbody>
</table>

find octree node for intersection;

if no environment map at current octree node then

create environment map;

sample environment map to gather radiance;

end

integrate irradiance from environment map, given intersection’s normal;

Algorithm 1: Single Map Caching

To sample an environment map, after initialization, we iterate over all the pixels in the map. For a given pixel, we cast multiple sample rays into the voxelized scene (Figure 3.2). This multi-sampling, much like anti-aliasing, is done stochastically. The lighting contributions for these rays are computed and then averaged. The normal of the incident surface is also stored, along with the distance from the environment map origin. Once the sampling of the environment map is complete, the irradiance value is integrated with respect to the intersection position. Integration, as described in Section 3.6.1, results in the indirect lighting contribution for the given surface.
3.2.2 Nearest Map Caching

Determining the indirect illumination by using the environment map of the given intersection’s octree node, may produce blocky results. This is due to the fact that the map’s location is not guaranteed to be at the center of the node. Rather, environment maps are randomly sampled across octree nodes due to the order in which intersections are evaluated. One effort to increase the accuracy of irradiance values is to use instead the nearest environment map for integration. Essentially, this means sample locations close to the edge of a given octree node have the opportunity to use a more accurate neighboring node’s environment map.

Data: intersection
Result: irradiance value

find octree node for intersection;

if no environment map at current octree node then
    create environment map;
    sample environment map to gather radiance;
end

enumerate neighbor nodes that contain an environment map;
find closest environment map;
integrate irradiance from environment map, given intersection’s normal;

Algorithm 2: Nearest Map Caching

For this caching scheme we not only search the encapsulating octree node for the given intersection, but also look to its neighbors for a closer environment map. As before, we start out by finding the octree node corresponding to our
intersection. Again, we create an environment map for our node, if one has not already been created. Next, we use our octree neighborhood traversal algorithm (Section 3.4.5) to enumerate all neighboring octree nodes and children that have an associated environment map. These nodes are then iterated to find the closest environment map. Once the closest map is found, the indirect illumination is determined by integrating the cached radiance values.

3.2.3 Neighborhood Caching

Neighborhood caching takes our first two approaches a step further, by interpolating irradiance values from multiple environment maps. Similar to the

```
Data: intersection

Result: irradiance value

find octree node for intersection;

if no environment map at current octree node then
    create environment map;
    sample environment map to gather radiance;
end

enumerate neighbor nodes that contain an environment map;

for each neighbor, including self do
    integrate irradiance from node’s environment map;
    add the weighted irradiance to a sum;
end

return the normalized sum of irradiance;
```

**Algorithm 3:** Neighborhood Caching
previous two schemes, an environment map is always created for our intersection’s leaf octree node. Then, like the previous approach, we enumerate the neighboring nodes and their children. Each node’s map is then integrated for the indirect illumination, multiplying by a weight (Section 3.6.2), and adding it to a sum. The following pseudo code summarizes this caching scheme:

\section{3.2.4 Compound Caching}

Lastly, a compound caching approach can be applied to these caching schemes. Instead of sampling the entire environment map on creation, we can add radiance values as we go. The main difference in this extension, is that environment map texels have to first check if the number of samples taken is high enough before a radiance value can be returned. For a given texel, if the sample weight (number of samples) is not above a given threshold, additional samples rays are cast into the voxelized scene and added to the existing value. The benefit of this approach is that the whole map does not need to be sampled by the first integration. Instead, portions of the map are filled as irradiance is integrated. Additional mechanisms can be added to compound caching in order to improve accuracy. For instance, a Russian roulette strategy can be adapted to the sample weight threshold. This would allow for greater accuracy in subsequent renders, as a portion of gathers would result in additional radiance values being gathered.

\section{3.3 Concurrency}

With the aim of achieving the greatest speedup for environment map caching, our caching schemes must also handle concurrency well. This not only affects the
performance of our algorithm, but the potential scalability of its implementation. For this reason, compound caching (Section 3.2.4) was not considered for our implementation. In order for concurrency to work for this approach, a cache must block access for all other threads each time the map is modified. Due to the volatile state of the environment map, accesses to the map may require additional sampling, thereby modifying the map. The map allows radiance values to be modified at the same time that radiance is retrieved, which requires synchronous access of the radiance value. This is seen as a significant hit to scalability, rendering compound caching viable mainly for synchronous implementations.

The three earlier defined schemes translate to much more thread-safe implementations. To support concurrency, a mutex is used to block all calls to an environment map when it is being constructed. This provides a way of stopping all threads trying to access the map until initialization is complete. For retrieving radiance values from the map, a ‘sampled’ flag determines whether the map has been initialized. This allows for multiple intersections to access the map concurrently, without having to wait on other threads.

3.4 Voxelization

The first step in caching radiance is to create the data structure to hold the radiance from which we will gather. During the setup phase of our algorithm, we construct a voxelized representation of the scene geometry. This representation is stored as a sparse octree, leveraging fast lookup and the additional neighborhood information provided by a relational octree structure. Voxelization is achieved through an adaptive top-down approach. Starting with the bounds containing the entire scene, space is subdivided into new octree nodes each containing
subset of the node’s primitives. Once a desired depth is achieved a leaf node is created pointing either to a voxel or NULL. To shade the voxel, the space is again subdivided to sample and find an average shading for its intersecting geometry. This process of subdividing octree nodes and creating voxels is repeated until all nodes contain children, voxels, or have been truncated. Figure 3.3 shows a voxelization of the Sponza Atrium scene.

3.4.1 Refinement Criteria

For a given voxelization, a minimum and maximum depth is provided. This limits the resolution of geometric detail while still providing adequate resolution for sampling. The minimum depth controls how adaptive the voxelization will be, limiting the sampling resolution of mostly homogeneous regions of space. If this
minimum depth is set too high we are not able to take advantage of homogeneous regions of space. Conversely, if the minimum depth is too small, the irradiance reflected off these regions will have less resolution and may appear blocky. The maximum depth, on the other hand, controls the resolution of detailed or complex regions. Setting the maximum depth too high or too low trades off performance for resolution.

During voxelization, octree nodes are subdivided until both of the following criteria are met:

1. The node’s depth is greater than the minimum depth.
2. Subdivision of the node results in all eight octants occupying the same primitive.

Or:

1. The node’s depth has exceeded the maximum depth.

If at any point a node contains no geometry, that node is truncated and replaced with a NULL pointer.

These criteria result in a sparse voxel octree independent of geometric complexity. The tree can efficiently be sampled at lower resolutions while still obtaining smooth approximations, due to the comprehensive approximation already built into the sampling.

3.4.2 Voxel Shading

The lighting contributions of a given voxel are approximated by averaging the computed lighting of the contained geometry. This is done by first sub-sampling
the voxel, into eight octants. Using the subregions’ bounds, intersection tests are
done to determine the subset of geometry that is intersected. The primitives’
normals provide a means of determining whether a set of bounds is considered
inside or outside of a given object. Once an intersection is found, the closest
surface point is determined and used to computing lighting attributes. From
these sample locations, the normals and diffuse lighting are each averaged and
stored in the voxel’s properties (Figure 3.4).

3.4.3 Node Structure

The structure of our octree nodes, determines the size and functionality of our
octree. We aim for a lightweight representation with the necessary functionality
for neighborhood queries. Our structure is based on the space efficient representa-
tion set forth by Laine et al. [16]. Each node can be represented by a single
pointer and status byte. The byte determines whether the node is a leaf node.
The pointer points to the starting location of eight consecutive child nodes or, in
the case of a leaf node, either a voxel or null.

In order to support fast relational and neighborhood queries, some attributes
have been added. In particular, the neighborhood traversal algorithm suggested by Frisken et al. [8] requires the use of location identifiers on a per node basis. These identifiers can be used to traverse in a given direction from a node. They represent the position of the node normalized to the octree resolution. For traversal, this structure also requires the addition of a parent pointer so that traversals do not have to originate from the head of the tree. Lastly, in order to determine the resolution or delta of location identifiers, each node must also keep track of its depth.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>pointer</td>
</tr>
<tr>
<td>parent</td>
<td>pointer</td>
</tr>
<tr>
<td>object</td>
<td>pointer</td>
</tr>
<tr>
<td>loc</td>
<td>integer[3]</td>
</tr>
<tr>
<td>depth</td>
<td>byte</td>
</tr>
<tr>
<td>isLeaf</td>
<td>byte</td>
</tr>
</tbody>
</table>

Table 3.1: Octree Node Structure

For the purposes of caching, we have included an additional object pointer in the node structure to point to our environment map caches. In total, the octree node structure will look similar to Table 3.1.

An additional optimization would be to combine isLeaf and depth, where leaf nodes are represented by a negative depth.

### 3.4.4 Voxel Structure

The voxel structure, pointed to by leaf octree nodes, contains the stored representation of outgoing radiance. Here we store the direct lighting contribution (or color), the object normal, and an optional array of the primitives represented by the voxel.
3.4.5 Octree Traversal

To traverse the octree structure a number of different methods are used. First, to find the closest node corresponding to a given position a simple top-down method is employed. This is used to determine the lowest octree node that contains a given intersection. Starting at the head of the tree, the bounds are subdivided for each octant, recursing upon the child node that contains the given point. The recursion keeps track of the current bounds and continues to subdivide as it goes. This query can also be done using the location identifiers [8]. Normalizing the point’s coordinates to location identifiers, the top down recursion can instead follow the bits of the provided location identifiers. This is done by masking the identifier with the current depth bit, resulting in three bits that determine which child is to be traversed. In practice, the first of the two methods performed better.

Next, for traversals of the tree based on ray casting, the parametric method introduced by Revelles et al. [24] is used. This traversal is used for gathering radiance values from the voxel octree. The algorithm is a top-down parametric traversal of the octree. The main benefit of this method is the initial sub-node selection, which allows a recursive traversal to start at the smallest starting octree node on the ray’s path and recurse through the tree without starting from the head of the tree again. This also avoids the need for neighborhood searches and repetitive tree recursion.

Last, to collect an octree node’s neighbors, we adapt the 2D neighborhood traversal method of Frisken et al. [8] to 3D. Using the location identifiers we can find neighboring nodes of the same depth by incrementing the identifiers with the correct delta along the desired axis. Using this new identifier, a common
ancestor can be found using the difference (xor) of the current node and the desired node’s identifiers. We first traverse up the tree until a common ancestor is found (the highest location bit that differs). Next we traverse back down the tree, choosing the sub-node based on the desired location identifier until the same depth is achieved. This traversal is done for all directions surrounding a given node, enumerating all the children of the found neighbors. The result is a set of octree nodes, that account for all environment maps within a node’s distance.

3.5 Environment Map

Figure 3.5: Environment visualized as a sphere.

Now that a voxel octree has been constructed with the radiance of the scene,
we can gather radiance values from it in order to compute irradiance. However, in order to bypass redundant gathering of radiance for a given region, we instead gather and store radiance values in a cache of environment maps. Each environment map is responsible for storing the incoming radiance at a given sample location. As such, an environment map represents a sphere of incoming radiance at a given resolution which can later be used to integrate irradiance for a given direction.

Environment maps traditionally represent a cube of textures surrounding a scene or object. Similar to a cube-map, this structure has lent itself well to the sampling of a hemisphere of radiance by rasterization of each face of the cube. In our approach, because rasterization has been replaced with the sampling of our sparse voxel octree, we deviate from this structure. Treating every sample with the same importance, we find that using a cube-map structure gives higher weight to the corners of the cube. Because we are looking for an accurate representation of our sphere of radiance, we instead want our structure to give uniform importance to all samples across the sphere. This will not only allow for a more accurate uniform sampling of the sphere, but also provide a more efficient use of our environment map texels. For this reason, our approach represents environment maps as uniformly weighted spherical maps (Figure 3.5).

### 3.5.1 Spherical Mapping

To remedy the non-uniformity of a cube-map representation, radiance is instead mapped to a 2D coordinate plane using polar coordinates. The conversion from Cartesian space into polar coordinates gives a way of mapping our 3D direction vector onto a sphere represented by a longitude, latitude, and radius.
Because the magnitude of our unit direction vectors is always one, we can discard the radius to make our polar coordinates a 2D coordinate plane. While our mapping now distributes our 2D samples more evenly over a sphere, the distribution is still not uniform. Due to the interaction of the longitude with the latitude, the 2D mapping has a heavier concentration of values at each axis of the sphere. For more uniform sampling, the spherical mapping has to be normalized with respect to area. This is done by considering the Jacobian determinant of the Cartesian mapping and weighting the mapping to account for this. Consider the following spherical to Cartesian conversion:

\[
\begin{align*}
    x &= r \sin \phi \cos \theta \\
    y &= r \sin \phi \sin \theta \\
    z &= r \cos \phi
\end{align*}
\]  

(3.1)

The Jacobian determinant can be found to be \( r^2 \sin \phi \). This shows that for all values of longitude \( \theta \) the associated area or volume is invariant. The colatitude \( \phi \), on the other hand, exhibits a change in area across its values. The probability distribution function (PDF) with respect to \( \phi \) is derived by normalizing the Jacobian. For \( \sin \phi \) on the range of 0 to \( \pi \), the PDF is \( \frac{\sin \phi}{2} \). This distribution function represents the change in area with respect to the colatitude. To account for this non-uniformity, the quantile (inverse cumulative distribution function) can be applied to the mapping. Integrating the PDF and finding the inverse we get a quantile function of \( 2 \arcsin \sqrt{v} \). Using the composite of this function and our polar mapping we can create a new uniform mapping of Cartesian space. Our
uniform mapping can be explained in terms of $u[0 \rightarrow 2]$ and $v[0 \rightarrow 1]$:

$$\theta = \pi u$$

$$\phi = 2 \arcsin \sqrt{v}$$

$$x = \sin \phi \cos \theta$$

$$y = \sin \phi \sin \theta$$

$$z = \cos \phi$$

(3.2)

The inverse mapping, used for retrieving samples can be found with the following:

$$\theta = \arctan \frac{y}{x} + \pi$$

$$\phi = \arccos z$$

$$u = \frac{\theta}{\pi}$$

$$v = \sin^2 \frac{\phi}{2}$$

(3.3)

Note that the arctan must be suitably defined. In our C++ implementation atan2 is used which returns a range of $-\pi$ to $\pi$. This range is shifted by the addition of $\pi$ in our first step.

### 3.5.2 Direction Adjustment

Another source of inaccuracy brought about by caching radiance with environment maps is the retrieval of radiance values from offset locations. Because
each environment map is sampled with respect to a location, radiance values are
dependent upon both the sample direction and the environment map location for
an accurate approximation. Therefore, the mapping of direction \((x,y,z)\) to uv co-
ordinate is only accurate for rays originating from the map’s center. The further
the origin of a given ray deviates from the map’s center, the less accurate the
sampled radiance value will be. To account for this, the direction of a radiance
lookup is first adjusted before it is mapped to uv coordinates.

![Diagram](image)

**Figure 3.6: Adjustment of direction \(\mathbf{w}_0\) using an environment map at
location \(x_m\).**

Using information provided by each radiance value sampled, we know for a
given direction both the distance to the source of the radiance and the normal
of the geometry at that radiance source. Estimating an adjusted direction for a
ray with an origin that is offset, requires using the stored radiance information
of the initial direction in order to approximate a new one.

In a method similar to the recursive strategy proposed by Szirmay-Kalos et
al. [29], in which distance values are stored in environment map texels, we pro-
pose a recursive planar approximation. Incorporating the radiances’ distance and
normal, we are able to implement a recursive solution that linearly approximates the adjusted direction. The normal of the radiance is used to create a planar approximation, which is then used to calculate the next approximate direction. The plane is defined by the distance, direction, and normal of the given radiance sample. From the distance and direction from the map’s origin, we define the position of the plane with the normal of the radiant surface. Next, we find the offset sample ray’s intersection with the plane. By normalizing the direction from the map’s origin to ray’s intersection we are given the next approximate direction. This algorithm can be recursed until the desired accuracy is achieved.

In practice, the recursive algorithm introduced errors due to approximations of steep or disjoint normals. Even limiting the amount of displacement from each approximation lead to some error. Therefore, much like Szirmay-Kalos et al. [29], we instead use a simplified approximation. Instead of using the radiance normals as planar approximations, we simply create a plane perpendicular to the initial ray direction. Then using the same logic as before, we approximate a new direction. This can be simplified to the following algorithm:

\[
\begin{align*}
\vec{d}x &= \mathbf{x} - \mathbf{x}_m \\
\mathbf{x}_r &= \vec{w}_0 \ast d_i \\
d_p &= (\mathbf{x}_r - \vec{d}x) \cdot \vec{w}_0 \\
\vec{w} &= \vec{w}_0 \ast d_p + \vec{d}x \\
\hat{\mathbf{w}} &= \frac{\vec{w}}{\|\vec{w}\|} 
\end{align*}
\]

Here we use the position and the direction to find the adjusted direction for a given environment map. First we determine the offset of the position from the environment map. Next, the estimated origin of the radiance is found by walk-
ing the given direction from the environment map times the cached approximate
distance. Projecting the vector from the offset location to the estimated radi-
ance origin onto the given direction vector we find the projected distance. This
distance represents the distance from the offset location to a point on the plane
perpendicular to the direction, defined by the estimated radiance origin. Finally,
the adjusted direction is computed as the normalized combination of the scaled
direction and offset vector (Figure 3.6). The direction is scaled by the projected
distance, essentially representing the vector from the environment map origin to
the new estimated radiance location.

An alternative to adjusting the sample direction to account for changes in
radiance, is to instead account for the difference in direction when interpolating
irradiance. Methods like irradiance gradients [31, 15] provide a way of aligning
radiance values to the new offset position and direction for smoother interpola-
tion. These methods, however, require a continuous representation of radiance
rather than the discretized samples we are working with.

### 3.5.3 Environment Map Structure

Based on the requirements of our spherical mapping and direction adjustment
algorithms we propose a new structure of environment maps for radiance caching.
Radiance gathered from our sparse voxel octree is mapped to 2D coordinates
using its sample direction. As a result, the color can be stored as a pixel in a
texture. Additionally, each environment map texel must also store information
on the radiant source’s normal and distance for direction adjustment.

A lightweight representation of our environment maps consists only of the
necessary 2D arrays for color, distance and normals. Upon initialization, the
whole sphere is sampled storing the average color, distance and normals in the map. The only other information needed to represent an environment map is the environment map location and a flag to determine if the sphere has been sampled (Table 3.2).

Second, to handle the requirements of compound caching (Section 3.2.4), some additional attributes are needed. In this representation, radiance samples are compounded over multiple samplings. To support this mechanism, the previous texel arrays instead represent the sum of all samples. An additional array of sample weights is added to keep track of the total weight of each texel location. Then, in place of the sample flag from the first approach, an additional mutex is required for concurrency. A summary of environment map structure is shown in Table 3.2.

### 3.6 Indirect Illumination using Environment Maps

The main performance increase achieved by our proposed method, comes from the integration of cached radiance instead of casting additional rays or gathering
radiance from a structure. Our use of environment maps bypasses the need for additional sampling or gathering.

### 3.6.1 Environment Map Integration

When shading a fragment, lighting contributions can be separated into direct illumination and indirect illumination. It is common among Global Illumination techniques to compute the two contributions separately and add them together later. In our method direct illumination is computed using ray tracing and the indirect illumination is integrated from our environment map cache.

In order to compute indirect illumination, it is necessary to integrate the irradiance for the given position. However, as seen in the lighting equation 2.4, we must also incorporate the BRDF of our material while integrating incoming radiance values. This integral over all incoming radiance is approximated using the Monte Carlo integration method of randomly sampling the hemisphere. The resulting equation can be rewritten as:

\[
L_{\text{indirect}}(x, \vec{w}_r) = \frac{1}{N} \sum_{i=1}^{N} L_i(x, \vec{w}_i) f(x, \vec{w}_i, \vec{w}_r) \vec{w}_i \cdot \vec{n}.
\] (3.5)

Each Monte Carlo sample finds the incoming radiance for a direction and applies the BRDF and weakening factor individually. All the samples are then averaged, effectively partitioning the sample weight. This results in the indirect lighting contribution, which can now be added to the direct illumination.

In the case of environment maps, the incoming radiance \( L_i \) is the value being stored. Rather than gathering radiance from the scene or casting secondary rays,
the value is retrieved from the environment map in constant time. Additionally, this can be done from positions offset from the center of the environment map. Because the radiance, distance, and surface normals are being stored, the mapped direction can be adjusted to provide a more accurate approximation when integrating.

### 3.6.2 Interpolating Environment Maps

Regions of space share the same cached environment map when integrating. Though integration is adjusted for the offset and direction, large changes in irradiance appear as noise between cache locations. To remedy this, irradiance is interpolated between caches (as described in Neighborhood Caching 3.2.3) to provide smoother and more accurate values.

Indirect illumination calculated from environment maps is interpolated using Gaussian weights. For a given sample location, we iterate through all neighboring environment maps, summing up their contributions. The weight of a particular environment map is determined by the distance from the environment map to the sample location. A Gaussian distribution is applied to give higher importance to nearby environment maps and to create a smooth falloff. Our Gaussian distribution is created with respect to the environment map spacing, with a falloff just beyond neighboring environment maps. The following equation describes our weight function for a sample location $\mathbf{x}$ with environment map location $\mathbf{p}$:

$$w(\mathbf{x}, \mathbf{p}) = e^{-\frac{||\mathbf{x} - \mathbf{p}||^2}{\sigma^2}}$$

(3.6)
The distance $d$ represents the average spacing of environment map samples. For our method, this value is determined by the voxel resolution, as the environment maps are cached in the same octree nodes as the voxels.

Finally, by summing the weighted indirect illumination values of all nearby environment maps and dividing by the total weight, we return our interpolated indirect illumination value.

### 3.7 Performance Trade-offs

Three accuracy vs. performance trade-offs are introduced by environment map caching:

1. Number of Monte Carlo samples.
2. Environment map resolution.
3. Environment map sampling.

In order to optimize performance and accuracy of indirect illumination, some tuning is necessary. Each variable controls the accuracy vs. performance trade-off. Additionally, the environment map resolution and voxelization depth control the amount of memory usage. Regardless of tuning, the improvement in performance from using caching is seen in subsequent renders from caching radiance.

The amount of samples gathered in Monte Carlo integration regulates the speed of integration. As with any type of sampling, the fewer samples cast the less accurate the result. Monte Carlo integration, in particular, introduces more noise as the amount of samples is decreased. While this sampling is independent
of our proposed caching method, the sample size affects the effectiveness of our environment map resolution.

The environment map resolution defines the accuracy at which radiance will be stored. Not only does this control the memory usage of our cache but also the time it takes to sample a single environment map. In general, the resolution does not have to be high in order to achieve an accurate approximation. Because the radiance values are integrated together anyway, the main effect of low resolution is the accuracy of irradiance over varying directions. Too high resolution, on the other hand, will result in longer sampling times. However, this increased sampling time is limited to initial rendering passes. All subsequent renders will not be affected by the resolution and will run at much faster rates.

Lastly, the number of samples cast, in order to gather radiance into each environment map, also affects the accuracy of our approximation. While multi-sampling each environment map texel costs more initial work, the radiance value stored better represents the total radiance of the solid angle. Ultimately, radiance caching with environment maps allows the user to tune their algorithm either for speedy approximations or more accurate rendering.
Chapter 4

Results

In this chapter, we will analyze the performance of radiance caching with environment maps. Here we will compare our implementation of Monte Carlo ray tracing against one using radiance caching with environment maps.

4.1 Implementation

The implemented radiance caching architecture provides a convenient interface to a CPU ray tracer. Written in C++, the ray tracer makes use of custom math, geometry and parser classes to provide more legible code. The structure of the ray tracer is loosely based off of the PBRT ray tracer [20], where aggregates and their primitives provide intersection information for a given ray. Our implementation is much more lightweight, focusing only on objects relevant to radiance caching, with all borrowed classes reimplemented for readability.
4.1.1 Threading

The system is multi-threaded using Thread Building Blocks [21] for concurrency. TBB is used in rendering by partitioning up blocks of rays and executing the ray intersection tests in parallel. For this we have constructed a renderer class that uses the given scene and ray tracer to render a partition of rays.

Octree construction is also parallelized using asynchronous task blocks. Even though the top-down construction is inherently recursive, each level has the ability to execute its possible eight child nodes in parallel. This is done by adding each node construction to a pool of asynchronous tasks, with the parent task being that of the parent node. Once all subtasks have completed for the head node, octree construction is complete.

4.1.2 Memory

In order to more efficiently allocate and free memory, a memory pool construct was added. A memory pool is a large chunk of memory from which partitions are distributed. This speeds up the allocation process by allocating chunks of memory less frequently, and returning incremented pointers to free memory instead. This not only reduces the number of system calls but insures byte alignment.

4.1.3 System

All render times are collected running our implementation on OS X (10.8.3) with 1.8 GHz Intel Core i7 and 4 GB of system RAM.
4.2 Test Scenes

For testing accuracy, a simple scene similar to a Cornell Box was used (Figure 4.5). While this scene does not take advantage of much of the performance gains seen by using a voxelized scene, it does show our ability to approximate irradiance with environment maps.

For performance testing, Dubrovnik’s Sponza Atrium is used (Figure 4.1). Due to its varying amounts of geometric complexity, it provides an ideal scene for testing the accuracy and performance of our caching algorithm. The scene consists of 66,000 triangles along with textures for texture mapping. For more distinct differences in indirect illumination, texture mapping has been disabled for indirect lighting contributions.

![Figure 4.1: Sponza Atrium Scene.](image)

Additional tests are run using the Sibenik Atrium (Figures 4.6, 4.7 and 4.8). This scene, similar to the Sponza Atrium, contains varying amounts of geometric
The scene is made up of 75,000 triangles with texture mapping.

All renderings are computed at a resolution of 800x600 with 2x anti-aliasing.

### 4.3 Performance Metrics

The performance of our algorithm is measured with respect to three metrics:

**Render Time**, our primary goal, is measured on the order of seconds over the elapsed render phase. We also include the subsequent render time, to show the performance increase gained from an already cached scene. This is useful for subsequent renders, which share visible regions of the scene, as they can leverage portions of the scene already cached. For our results, subsequent renderings are rendered from the same viewpoint.

**Image difference** is the perceived difference between our approximation and the naive Monte Carlo approximation. This metric is calculated using the Perceptual Image Diff Tool [35] created by Hector Yee. The tool outputs the number of pixels that are visibly different, from this we create the percentage value seen.

**Memory overhead** is measured as the amount of additional memory required by our caching method. This is computed as the difference in total memory use at the end of rendering from the total memory used at the beginning of rendering.


4.4 Render Comparison

To provide an accurate comparison of our method’s performance, compared to the performance of renders not using our caching method, we first establish the error associate with different samplings of the base rendering. This distribution of sampling error is based off our ground truth image. Our ground truth image, or basis, is the Monte Carlo integrated, indirect illumination image at a sample size of twice the desired accuracy (200 secondary rays). Image differences are computed using this image, showing that our method’s resulting images approach the image difference achieved by the desired Monte Carlo sampling.

Radiance caching with environment maps introduces multiple variables and multiple caching schemes. To compare the performance gains of all these variables, we break our data comparison into five parts. First we show the sampling error introduced by different sampling frequencies. Then keeping that constant, we move on to show the effect of different environment map resolutions. Last, we show the performance characteristics of our different caching schemes.

4.4.1 Sampling Error

As described in earlier sections, the benefit of using Monte Carlo integration methods can be seen in vast decreases in render time over casting even more secondary rays. The side-effect of fewer samples, however, introduces a great deal of noise. Here we show the render time and image difference resulting from different numbers of indirect illumination samples (Table 4.1). Our ground truth image, is an image created using two hundred indirect illumination samples.

As expected, render times increase linearly with the number of samples. Im-
age difference, on the other hand, is affected logarithmically, asymptotically approaching the ground truth. The goal of our proposed method is to achieve a high level of accuracy with render times better than simply decreasing the amount of samples. However, even this method of comparison is not a completely fair assessment of the performance of our method. Images sampled at lower rates can show higher values of accuracy, even if they contain much more noise than a compared image (Figure 4.2). Here we must clarify the divergence of ‘visually similar’ from ‘visually accurate’. The image difference measures only the perceived accuracy of the scene’s pixels, not the similarity of the images characteristics. Therefore we must take into account the smoothness of our images separately through visual inspection.

<table>
<thead>
<tr>
<th>Samples</th>
<th>Render Time</th>
<th>Image Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>17 sec</td>
<td>6.203%</td>
</tr>
<tr>
<td>10</td>
<td>82 sec</td>
<td>1.583%</td>
</tr>
<tr>
<td>20</td>
<td>152 sec</td>
<td>0.8260%</td>
</tr>
<tr>
<td>40</td>
<td>315 sec</td>
<td>0.3923%</td>
</tr>
<tr>
<td>100</td>
<td>765 sec</td>
<td>0.1158%</td>
</tr>
<tr>
<td>200</td>
<td>1444 sec</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 4.1: Sampling Performance Comparison

Figure 4.2: Error and noise due to varying numbers of samples (2, 10, 20, 40, 100, 200).
4.4.2 Environment Map Resolution

The accuracy of radiance integrated from the scene is determined by our environment map resolution. To measure the effect of varying the environment map resolution, we keep constant the number of indirect illumination samples. Here we use one hundred samples, which eliminates most of the sampling noise and produces a high level of accuracy. The compared render time, not using caching (Table 4.1), is 765 seconds which achieves an image difference of 0.1158%.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Render Time</th>
<th>Subsequent Render Time</th>
<th>Image Difference</th>
<th>Memory Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x8</td>
<td>23 sec</td>
<td>21 sec</td>
<td>14.61%</td>
<td>15.0 MB (6.98%)</td>
</tr>
<tr>
<td>8x16</td>
<td>30 sec</td>
<td>23 sec</td>
<td>4.519%</td>
<td>53.2 MB (21.0%)</td>
</tr>
<tr>
<td>16x32</td>
<td>40 sec</td>
<td>25 sec</td>
<td>1.064%</td>
<td>255 MB (56.0%)</td>
</tr>
<tr>
<td>32x64</td>
<td>78 sec</td>
<td>25 sec</td>
<td>0.7208%</td>
<td>900 MB (81.8%)</td>
</tr>
<tr>
<td>64x128</td>
<td>199 sec</td>
<td>30 sec</td>
<td>0.3965%</td>
<td>3570 MB (94.7%)</td>
</tr>
</tbody>
</table>

Table 4.2: Environment Map Resolution Comparison

We quickly see that increasing the resolution affects the memory overhead exponentially. Even though the accuracy and render speed achieved by higher resolutions is significant, the memory is a severe limiting factor. Scalability of our method must therefore depend on other variables to be viable. However, focusing on the lower resolutions, we see the potential speed increase of radiance caching with environment maps. Across the board, we have achieved a speed increase between 3 and 33 times for initial renders and between 25 - 35 times for subsequent renders.
4.4.3 Environment Map Sampling

In order to achieve better accuracy and scalability of our caching method we instead focus on the sampling resolution of each environment map texel. Working off a constant environment map resolution of 16x32, which had the lowest image difference for memory used, we test the payoff of different levels of multi-sampling when radiance is initially gathered into our environment maps.

<table>
<thead>
<tr>
<th>Multi-sampling</th>
<th>Render Time</th>
<th>Subsequent Render Time</th>
<th>Image Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>34 sec</td>
<td>22 sec</td>
<td>1.063%</td>
</tr>
<tr>
<td>2x</td>
<td>64 sec</td>
<td>24 sec</td>
<td>0.7923%</td>
</tr>
<tr>
<td>3x</td>
<td>112 sec</td>
<td>22 sec</td>
<td>0.5565%</td>
</tr>
<tr>
<td>4x</td>
<td>163 sec</td>
<td>23 sec</td>
<td>0.5402%</td>
</tr>
</tbody>
</table>

Table 4.3: Multi-sampling Comparison

Initially only casting one sample ray per environment map texel, we see a great deal of improvement in accuracy from casting more samples. Not only does this improve the accuracy of our incoming radiance approximation, but it also serves as a form of pre-integration. Since the radiance is eventually integrated to compute the irradiance at a point, the averaging of more radiance values effectively creates larger quantized chunks to integrate over. Multi-sampling our environment maps improves accuracy by almost two times, while keeping initial render times still within 4 to 20 times faster and the memory overhead constant. Subsequent render times also remain more than 30 times faster.
4.4.4 Caching Schemes

Finally, since a good accuracy has been achieved within a given amount of error, we must consider the visual characteristics of our results. The single environment map caching scheme used up until now, while producing accurate results in comparison to our reference image, contains some visible artifacts. Because each intersection only references the most convenient environment map, irradiance values between environment maps are disjoint. To remedy this, the other caching schemes take into account the neighborhood of environment maps around each intersection. Table 4.4 shows the performance of the proposed caching schemes.


<table>
<thead>
<tr>
<th>Caching Method</th>
<th>Render Time</th>
<th>Subsequent Render Time</th>
<th>Image Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Map Caching</td>
<td>34 sec</td>
<td>22 sec</td>
<td>1.063%</td>
</tr>
<tr>
<td>Nearest Map Caching</td>
<td>39 sec</td>
<td>28 sec</td>
<td>1.057%</td>
</tr>
<tr>
<td>Neighborhood Caching</td>
<td>69 sec</td>
<td>57 sec</td>
<td>1.054%</td>
</tr>
</tbody>
</table>

Table 4.4: Caching Scheme Performance

using a map resolution of 16x32 with no multi-sampling.

While the image difference between caching schemes remains almost identical, a visual inspection of the images (Figure 4.3) shows a major improvement in image quality from the single environment map caching to the neighborhood caching scheme. Looking in particular at the artifacts (Figure 4.4), the neighborhood caching scheme is the only approach that removes all the artifacts and produces the most visually similar results. The performance of the nearest map caching scheme, provides better results but still shows artifacts. Currently our construction of environment maps is somewhat structured; dependent on how the image plane is sampled. Improvements for this caching scheme could potentially

![Figure 4.4: Caching artifacts of environment map, nearest map and neighborhood caching schemes.](image)
be seen through more randomly sampling or constructing the environment maps. This can again been seen in Figure 4.5 with distinctly smoother results from the neighborhood caching approach.

4.5 Analysis

A further analysis of the data provides insight into what performance improvements can be gained through radiance caching.
4.5.1 Memory

Memory, the limiting factor of our algorithm, caps the resolution of our radiance representation. The more space available, the more accurate integrated irradiance values will be. However, in most cases it is beneficial to stick to smaller representations. Scenes like that in Figure 4.6 require a large number of environment maps due to the vast environment visible in the scene. Both the scalability and speed rely on the memory overhead of the environment map cache. We have found in practice that as long as the environment map resolution is not too small, accuracy can be made up elsewhere.

(a) Direct Illumination  (b) Indirect Illumination  (c) Combined Illumination

Figure 4.6: Comparison of direct, indirect and combined illumination. Top rendered using only Monte Carlo methods. Bottom rendered with neighborhood environment map caching.

4.5.2 Speed

Speed, the driving factor of caching radiance, showed great improvements in all subsequent renders. The idea of a pre-cached scene, much like voxelizing a scene, provides great performance benefits with speedups of more than 20X. In
addition, the initial render times also provided substantial improvements to speed due to already voxelized radiance. Additional testing of indirect illumination seen in Figure 4.7, shows an accurate approximation of indirect illumination at more than a 15X speedup. Furthermore, a 42X speedup was achieved (Figure 4.8) using an already full environment map cache.

4.5.3 Accuracy

Accuracy, throughout all the tests, showed improvement over Monte Carlo ray tracing. Increased rendering time, it seems, can always be traded for better accuracy. Because speed increases achieved by our base method are substantial, one can easily give up some render time for better accuracy. Our results reached visual accuracy nearing that of the base Monte Carlo method, at a fraction of the time. Again, an important fact to mention is the visual characteristics achieved by our neighborhood caching method compared to just using a single environment
map. Even at lower levels of accuracy, our results produced smoother images with fewer artifacts than the Monte Carlo equivalent.

4.6 Conclusion

In conclusion, this paper introduced a new method to approximate indirect illumination using environment maps. Global illumination introduces the problem of computing, gathering, and integrating incoming radiance for each shaded fragment. Previous work has found that storing the radiance or radiant flux of a scene can significantly improve the gather performance for indirect illumination. Building upon this, our method leverages a voxelized scene representation to store radiance. However, instead of gathering radiance at each intersection, our method instead caches the incoming radiance spatially in a cache of environment maps. Future irradiance calculations then make use of this cache and bypass radiance gathering completely.

From our resulting implementation, we found that this method of caching does indeed achieve significant (10X - 30X) improvements in render time, with accuracy comparable to Monte Carlo sampling. While more accurate results require an initial hit to performance, subsequent renders all show a performance increase of at least an order of magnitude. The proposed method’s applications include pre-visualization and potential interactive ray tracing environments. Ultimately, our method provides a fast approximation of indirect illumination, with focus on the case of sequential rendering.
(a) Direct illumination.

(b) Monte Carlo render: 975 sec.

(c) Neighborhood Environment Map Cached render: 130 sec, 23 sec subsequent.

Figure 4.8: Sibenik Atrium render comparison.
Chapter 5

Future Work

In our work, we focused on creating a fast approximation of indirect illumination by caching radiance in environment maps. The topics covered in our algorithm touch many different components seen in graphics and global illumination. As such, many of these subcomponents can be built upon and improved past our naive implementation.

5.1 Voxels

In particular, more recent Global Illumination methods using voxels [7, 30] make use of real-time voxelization methods. A faster voxelization, as well as a more adaptive voxel octree, would provide significant performance gains, allowing our method to be more interactive. Additionally, techniques like voxel cone-tracing could be implemented to greatly improve environment map sampling. In place of the current multi-sampling of environment map texels, a single cone could be traced to gather radiance for the given solid angle. This would not only improve environment map sampling performance, but provide a more
comprehensive and accurate approximation of radiance.

5.2 Memory Usage

For storing radiance over a sphere, the growing trend is to use Spherical Harmonics (SH). SH provides a light-weight representation of a sphere by storing a vector of coefficients. This is much more space efficient, when compared to storing discretized radiance values. Some problems are, however, introduced by using SH. The level of detail achieved by environment maps becomes much harder to represent with SH as resolution increases. Increasing the number of coefficients requires a great deal more work, as the computation to convolve radiance values becomes increasingly more lengthy. Additionally, in the case of integrating irradiance from an offset location, the discretized normals from the sources of radiance may still need to be integrated, reducing gains seen by convolution. Overall, a spherical harmonic representation is worth investigating for the promising gains in space efficiency.

5.3 Irradiance Gradients

A related topic, seen in irradiance caching [31] and radiance caching [15], is the use of irradiance gradients. This is used for more accurate interpolation of irradiance values. By determining a translational and rotational gradient for our spherical maps, later interpolation can be more efficiently warped for the given offset and direction.
5.4 Adaptive Environment Maps

Lastly, a more adaptive approach could be taken to environment map caching. Our current method only considers environment maps at levels offset from the leaf nodes. Because our voxel octree is already adaptive, we leverage the ability to approximate larger homogeneous regions. However, a more adaptive approach could make use of the octree hierarchy to approximate similar environment maps higher up in the tree. This could take into account not only the homogeneity of geometry but also of lighting. To do this, we could create an importance sampling of the scene to be used for our environment maps. Then, instead of placing environment maps randomly during rendering, we could use this sampling to ensure better placement of our environment maps. Additionally, performance benefits could be seen from creating an environment map cache with a view-dependent sampling. Overall, one could look into making environment map caching more adaptive and efficient for general purpose rendering.
Bibliography


