# Global Illumination For Point Models

**Third Progress Report**

Submitted in partial fulfillment of the requirements
for the degree of

**Ph.D.**

by

**Rhushabh Goradia**
**Roll No: 04405002**

under the guidance of

**Prof. Sharat Chandran**

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai
August 14, 2007

## Acknowledgments

I would like to thank Prof. Sharat Chandran for devoting his time and efforts to provide me with vital directions to investigate and study the problem.

I would also like to specially thank Anil Kumar Kanankanti who supported me all through my work, and Prekshu Ajmera, Ankit Agrawal and all the members of ViGIL for their participation in discussions and valuable support.

Rhushabh Goradia

# Contents

**1 Introduction** **1**

1.1 Introduction . . . . . 1

1.1.1 Point Based Modelling and Rendering . . . . . 2

1.1.2 Global Illumination . . . . . 3

1.1.3 Fast computation with Fast Multipole Method . . . . . 6

1.1.4 Parallel computations on GPU . . . . . 7

1.1.5 Visibility between Point Pairs . . . . . 8

1.2 Problem Definition . . . . . 8

1.3 Overview of the Report . . . . . 8

**2 Visibility Maps in Point Clouds for Global Illumination** **10**

2.1 Introduction . . . . . 10

2.2 FMM-based Global Illumination . . . . . 12

2.3 Visibility Maps . . . . . 13

2.4 Previous Approach: Visibility in Point Models . . . . . 15

2.4.1 Point–Point Visibility . . . . . 15

2.4.2 Hierarchical Visibility . . . . . 16

2.5 Limitations . . . . . 18

2.6 New Approach: Visibility Maps in Point Models . . . . . 20

2.6.1 Point–Pair Visibility Algorithm . . . . . 20

2.6.2 Octree Depth Considerations . . . . . 21

2.6.3 Construction of Visibility Maps . . . . . 22
2.7 Extending Visibility Maps to Adaptive Octrees . . . . . 24
2.8 Experimental Results . . . . . 27
2.8.1 Visibility Validation . . . . . 27
2.8.2 Quantitative Results . . . . . 27
2.8.3 Discussion . . . . . 27

**3 Discussion: Parallel FMM on GPU 33**
3.1 Introduction . . . . . 33
3.1.1 Fast computation with Fast Multipole Method . . . . . 33
3.1.2 Parallel computations on GPU . . . . . 35
3.2 Spatial Locality Based Parallel Domain Decomposition . . . . . 35
3.2.1 Space Filling Curves . . . . . 36
3.2.2 Parallel Compressed Octrees . . . . . 37
3.2.3 Spatial Domain Decomposition Methods on GPU . . . . . 41
3.2.4 Parallel Compressed Octrees on GPU . . . . . 44
3.3 Parallel FMM Algorithm . . . . . 48
3.3.1 Constructing Parallel Compressed Octree . . . . . 49
3.3.2 Near Field Computations . . . . . 49
3.3.3 Building Interaction Lists . . . . . 49
3.3.4 Computing Multipole Expansions . . . . . 49
3.3.5 Computing Multipole to Local Translations . . . . . 50
3.3.6 Computing Local Expansions . . . . . 50
3.3.7 Parallel FMM on GPU . . . . . 51

**4 Discussion: Specular Inter-reflections and Caustics in Point based Models 52**
4.1 Introduction . . . . . 52
4.2 Photon Mapping . . . . . 53
4.2.1 Photon Tracing (First Pass) . . . . . 53
4.2.2 Preparing the Photon Map for Rendering . . . . . 55
4.2.3 Rendering (Second Pass) . . . . . 56
4.2.4 Radiance Estimate . . . . . 58
4.2.5 Limitations of Photon Mapping . . . . . 58
4.3 Our Approach . . . . . 59

4.3.1 Splat-Based Ray Tracing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
4.3.2 Ray Tracing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
4.3.3 Optimizing Photon Generation and Sampling . . . . . . . . . . . . . . . . . . . . . . 65
4.3.4 Optimized Photon Traversal and Intersection tests . . . . . . . . . . . . . . . . . . . 65
4.3.5 Fast Photon Retrieval using Optimized $kNN$ Query Algorithm . . . . . . . . . . . . 66

**5 Conclusion and Future Work** **69**

# List of Figures

1.1 Impact of photorealistic computer graphics on filmed and interactive entertainment. Left: A still from the animated motion picture ‘Final Fantasy : The Spirits Within’. Right: A screenshot from the award-winning first person shooter game ‘Doom III’ . . . . . . . . . . . . . . . . . . . 1
1.2 Example of Point Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.3 Global Illumination. Top Left[KC03]: The ‘Cornell Box’ scene. This image shows local illumination. All surfaces are illuminated solely by the square light source on the ceiling. The ceiling itself does not receive any illumination. Top Right[KC03]: The Cornell Box scene under a full global illumination solution. Notice that the ceiling is now lit and the white walls have color bleeding on to them. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.4 Complex point models with global illumination [WS05] [DYN04] effects like soft shadows, color bleeding, and reflections. Bottom Right: “a major goal of realistic image synthesis is to create an image that is perceptually indistinguishable from an actual scene”. . . . . . . . . . . 4
1.5 Specular (Regular) and Diffuse Reflections . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
1.6 Left: Colors transfer (or ”bleed”) from one surface to another, an effect of diffuse interreflection. Also notable is the caustic projected on the red wall as light passes through the glass sphere. Right: Reflections and refractions due to the specular objects are clearly evident . 5

2.1 Grottoes, such as the ones from China and India form a treasure for mankind. If data from the ceiling and the statues are available as point samples, can we capture the interreflections? . . . 11

2.2 Views of the visibility map (with respect to the hatched node in red) is shown. Every point in the hatched node at the first level is completely visible from every point in only one node (the extreme one). At level 2, there are two such nodes. The Figure on the left shows that at the lowest level, there are three visible leaves for the (extreme) hatched node; on the other hand the Figure on the right shows that there are only two such visible leaves, for the second son (hatched node). The Figure also shows invisible nodes that are connected with dotted lines. For example, at level 1, there is one (green) node $G$ such that no point in $G$ is visible to any point in the hatched node. Finally the dashed lines shows "partially visible" nodes which need to be expanded. Partial and invisible nodes are not explicitly stored in the visibility map since they can be deduced. . . . . . . 13

2.3 Leaf nodes (or cells, or voxels) are at level three. . . . . . . 14

2.4 Only $x_2$ and $x_4$ will be considered as occluders. We reject $x_1$ as the intersection point of the tangent plane lies outside the line segment $\overline{pq}$. $x_3$ has earlier been rejected because it is more than a distance $\Delta$ from the line segment $\overline{pq}$. . . . . . . 16

2.5 Only $x_2$ and $x_3$ will be considered as occluders. We reject $x_1$ as the intersection point of the tangent plane lies outside segment $\overline{pq}$, $x_4$ because it is more than a distance $R$ away from $\overline{pq}$, and $x_5$ as its tangent plane is parallel to $\overline{pq}$. . . . . . . 20

2.6 Point-Point visibility is obtained by performing a number of tests. Now its extended to Leaf-Leaf visibility . . . . . . . 22

2.7 (a) The Buddha model and the cornell room both contain $500000$ points each. However, the density of points on Buddha is very high as compared to density on the walls of the room. (b) Bresenham Algorithm applied for computing visibility between two leaves in an adaptive octree structure. Wrong step-length might miss out the **potential occluding cell(cyan colored) in between cell $p$ and cell $q$ leading to wrong computation of visibility status between the two cells.** . . . . . . . 25

2.8 Ray-Sphere intersection algorithm to determine point-point visibility . . . . . . . 26

2.9 Various visibility tests where purple color indicates portions visible to the candidate eye (marked cyan/brown). . . . . . . 29

2.10 Various visibility tests where purple color indicates portions visible to the candidate eye (marked green/cyan). . . . . . . 30

2.11 Various visibility tests where purple color indicates portions visible to the candidate eye (marked cyan/brown). The left column shows the output of the *Bresenham's Line Algorithm* applied to the scene when divided in an adaptive octree structure. Note the artifacts which are clearly visible due to errors introduced by the step-length criteria of Bresenham's Algorithm. The step-length here was chosen to be 1 unit. Decrease in the step-length leads to drastic increase of running times *(in hours)*. The right column shows the output of the new *Sphere-Ray Intersection Algorithm* applied on similar models and similar input environment. The notable artifacts have reduced quite a lot. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

2.12 Use of V-Maps for GI effects. The hierarchy was constructed till we had roughly $75$ points per leaf. The images rendered using a custom point-based renderer. Soft shadows, color bleeding and parts of models indirectly visible to the light source being *lit* can be observed. Five different set of images, corresponding to different point models, are shown. Each set shows a front view, a back view and a close up of the point model/s placed in the cornell room. . . . . . . . . . . 32

3.1 (a) The z-curve for $k$ = 1 and 2. (b) A $8 \times 8$ decomposition of two dimensional space containing 7 points. The points are labeled in the order in which the cells containing them are visited using a $Z$-SFC. (c) Bit interleaving scheme for generating index of a cell as per $Z$-SFC. . . . . . . . 36

3.2 A quadtree built on a set of 10 points in 2-D. . . . . . . . . . . . . . . . . . . . . . . . . . . 38

3.3 A compressed quadtree corresponding to the quadtree of Fig. 3.2 . . . . . . . . . . . . . . . . . 38

3.4 Bit interleaving scheme for a hierarchy of cells. . . . . . . . . . . . . . . . . . . . . . . . . . . 39

3.5 Cell locations stored as an indirection grid in texture memory and the points as an 1-D array. The *RGBA* value of the cell will index the location of the point (in texture memory) contained in it. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

3.6 A simple parallel Bitonic Merge Sort of eight elements requires six passes. Elements at the head and tail of each arrow are compared, with larger elements moving to the head of the arrow. The nal sorted sequence is achieved in $O(\log_2 n)$ passes. . . . . . . . . . . . . . . . . 43

3.7 Computing the scan of an array of 8 elements . . . . . . . . . . . . . . . . . . . . . . . . . . 45

3.8 Storage in texture memory. The indirection pool encodes the tree. Indirection grids are drawn with different colors. The grey cells contain data. . . . . . . . . . . . . . . . . . . . . . . . . 46

3.9 At each step the value stored within the current node's indirection grid is retrieved. If this value encodes an index, the lookup continues to the next depth. Otherwise, the value is returned. . . 48

4.1 Photon paths in a scene (a Cornell box with a chrome sphere on left and a glass sphere on right): (a) two diffuse reflections followed by absorption, (b) a specular reflection followed by two diffuse reflections, (c) two specular transmissions followed by absorption. . . . . . . . . . 55

4.2 Building (a) the caustics photon map and (b) the global photon map. . . . . . . . . . . . . . . 56

4.3 Example output of Photon Mapping Algorithm [Jen96] showing reflection, refractions and caustics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

4.4 (a) Generation of splat $S_j$ starts with point $\mathbf{p}_i$ and grows the splat with radius $r_j$ by iteratively including neighbors $\mathbf{q}_l$ of $\mathbf{p}_i$ until the approximation error $\delta_\epsilon$ for the covered points exceeds a predefined error bound. (b) Splat density criterion: Points whose distance from the splats center $\mathbf{c}_j$ when projected onto splat $S_j$ is smaller than a portion *perc* of the splats radius $r_j$ are not considered as starting points for splat generation. (c) Generation of linear normal field (green) over splat $S_j$ from normals at points covered by the splat. Normal field is generated using local parameters $(u, v) \in [1, 1]$X$[1, 1]$ over the splats plane spanned by vectors $\mathbf{u}_j$ and $\mathbf{v}_j$ orthogonal to normal $\mathbf{n}_j = \mathbf{n}_i$. The normal of the normal field at center point $\mathbf{c}_j$ may differ from $\mathbf{n}_i$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

4.5 (a) Octree generation: In the first phase, the octree is generated while inserting splats $S_j$ into the cells containing their centers $c_j$ (red cell). In the second phase, splat $S_j$ is inserted into all additional cells it intersects (yellow cells). (b)(c) The second test checks whether the edges of the bounding square of splat $S_j$ intersect the planes *E* that bound the octree leaf cell. (b) $S_j$ is inserted into the cell. (c) $S_j$ is not inserted into the cell. This second test is only performed if the first test (bounding box test) was positive. . . . . . . . . . . . . . . . . . . . . . . . . . . 64

4.6 Merging the results from multiple hash tables. (a) the query point retrieves different candidates sets from different hash tables, (b) the union set of candidates after merging, and (c) the two closest neighbors selected. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

## Abstract

Advances in scanning technologies and rapidly growing complexity of geometric objects motivated the use of *point-based geometry* as an alternative surface representation, both for efficient rendering and for flexible geometry processing of highly complex 3D-models. Traditional geometry based rendering methods use triangles as primitives which make rendering complexity dependent on complexity of the model to be rendered. But point based models overcome that problem as points don't maintain connectivity information and just represents surface information. Based on their fundamental simplicity, points have motivated a variety of research on topics such as shape modeling, object capturing, simplification, rendering and hybrid point-polygon methods.

*Global Illumination for point models* is an upcoming and an interesting problem to solve. The lack of connectivity touted as a plus, however, creates difficulties in generating global illumination effects. This becomes especially true when we have a complex scene consisting of several models, the data for which is available as hard to segment aggregated point-models. Inter-reflections in such complex scenes requires knowledge of visibility between point pairs. Computing visibility for points is all the more difficult (as compared to polygonal models), since we do not have any surface/object information. In this report we present, a novel, hierarchical, fast and memory efficient algorithm to compute a description of mutual visibility in the form of a *visibility map*. Ray shooting and visibility queries can be answered in sub-linear time using this data structure. We evaluate our scheme analytically, qualitatively, and quantitatively and conclude that these maps are desirable.

We use the *Fast Multipole Method (FMM)*, a robust technique for the evaluation of the combined effect of pairwise interactions of $n$ data sources, as the light transport kernel for inter-reflections, in point models, to compute a description – *illumination maps* – of the diffuse illumination. Parallel computation of the FMM is considered a challenging problem due to the dependence of the computation on the distribution of the data sources, usually resulting in dynamic data decomposition and load balancing problems. We present, in this report, an algorithm [HAS02] for parallel implementation of FMM, which does not require any dynamic data decomposition and load balancing steps. We, further, also provide necessary hints to implement a similar algorithm on a *Graphics Processing Unit (GPU)* as a "GPGPU" application.

A complete global illumination solution for point models should cover both diffuse and specular (reflections, refractions, and caustics) effects. Diffuse global illumination is handled by generating *illumination maps*. For specular effects, we use the *Splat-based Ray Tracing* technique for handling reflections and refractions in the scene and generate *Caustic Maps* using optimized Photon generation and tracing algorithms. We further discuss a time-efficient $kNN$ query solver required for fast retrieval of caustics photons while ray-traced rendering.

Chapter 1

# Introduction

## 1.1 Introduction

The pixel indeed has assumed mystical proportions in a world where computer assisted graphical techniques have made it nearly impossible to distinguish between the real and the synthetic. Digital imagery now underlies almost every form of computer based entertainment besides serving as an indispensable tool for fields as diverse as scientific visualization, architectural design, and as one of its initial killer applications, combat training. The most striking effects of the progress in computer graphics can be found in the filmed and interactive entertainment industries (Figure 1.1).

Figure 1.1: Impact of photorealistic computer graphics on filmed and interactive entertainment. Left: A still from the animated motion picture ‘Final Fantasy : The Spirits Within’. Right: A screenshot from the award-winning first person shooter game ‘Doom III’

The process of visualizing a virtual three dimensional world is usually broken down into three stages:

- **Modeling.** A geometrical specification of the scene to be visualized must be provided. The surfaces in the scene are usually approximated by sets of simple surface primitives such as triangles, cones, spheres, cylinders, NURBS surfaces, **points** etc.

- **Lighting.** This stage involves ascribing *light scattering properties* to the surfaces/surface-samples composing the scene (e.g. the surface may be purely reflective like a mirror or glossy like steel). Finally, a description of the *light sources* of the scene must be provided - those surfaces that spontaneously emit light.

- **Rendering.** The crux of the 3D modeling pipeline, the rendering stage accepts the three dimensional scene specification from above and renders a two dimensional image of the same as seen through a camera. The algorithm that handles the simulation of the light transport process on the available data is called the *rendering algorithm*. The rendering algorithm depends on the type of primitive to be rendered. For rendering points various rendering algorithms like QSplat, Surfel Renderer etc are available.

*Photorealistic* computer graphics attempts to match as closely as possible the rendering of a virtual scene with an actual photograph of the scene had it existed in the real world. Of the several techniques that are used to achieve this goal, *physically-based* approaches (i.e. those that attempt to simulate the actual physical process of illumination) provide the most striking results. The emphasis of this report is on a very specific form of the problem known as *global illumination* which happens to be a photorealistic, physically-based approach central to computer graphics. This report is about capturing interreflection effects in a scene when the input is available as point samples of hard to segment entities. Computing a mutual visibility solution for point pairs is one major and a necessary step for achieving good and correct global illumination effects.
Before moving further, let us be familiar with the terms point models and global illumination.

### 1.1.1 Point Based Modelling and Rendering

Figure 1.2: Example of Point Models

In recent years, point-based methods have gained significant interest. In particular their simplicity and total independence of topology and connectivity make them an immensely powerful and easy-to-use tool for both modelling and rendering. For example, points are a natural representation for most data acquired via measuring devices such as range scanners [LPC+00], and directly rendering them without the need for cleanup and tessellation makes for a huge advantage.

Second, the independence of connectivity and topology allow for applying all kinds of operations to the points without having to worry about preserving topology or connectivity [PKKG03, OBA+03, PZvBG00]. In particular, filtering operations are much simpler to apply to point sets than to triangular models. This allows for efficiently reducing aliasing through multi-resolution techniques [PZvBG00, RL00, WS03], which is particularly useful for the currently observable trend towards more and more complex models: As soon as triangles get smaller than individual pixels, the rationale behind using triangles vanishes, and points seem to be the more useful primitives. Figure 1.2 shows some example point based models.

### 1.1.2 Global Illumination

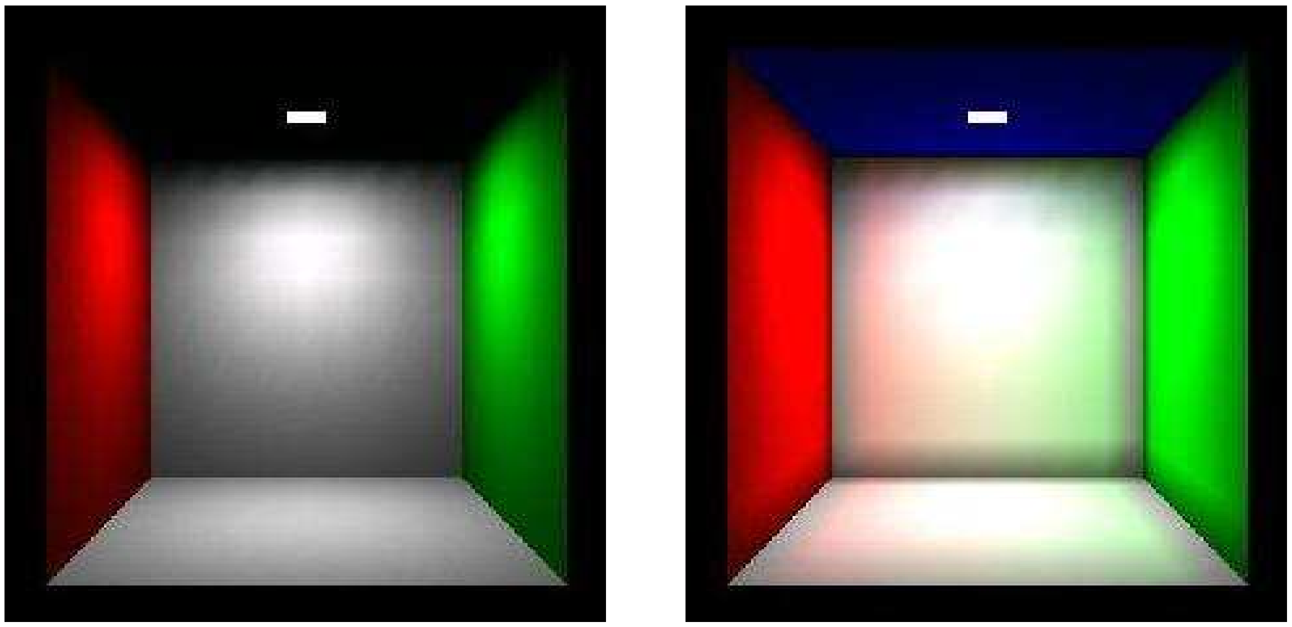


Figure 1.3: Global Illumination. Top Left[KC03]: The 'Cornell Box' scene. This image shows local illumination. All surfaces are illuminated solely by the square light source on the ceiling. The ceiling itself does not receive any illumination. Top Right[KC03]: The Cornell Box scene under a full global illumination solution. Notice that the ceiling is now lit and the white walls have color bleeding on to them.

*Global illumination algorithms are those which, when determining the light falling on a surface, take into* *account not only the light which has taken a path directly from a light source (direct illumination), but also light which has undergone reflection from other surfaces in the world (indirect illumination).*

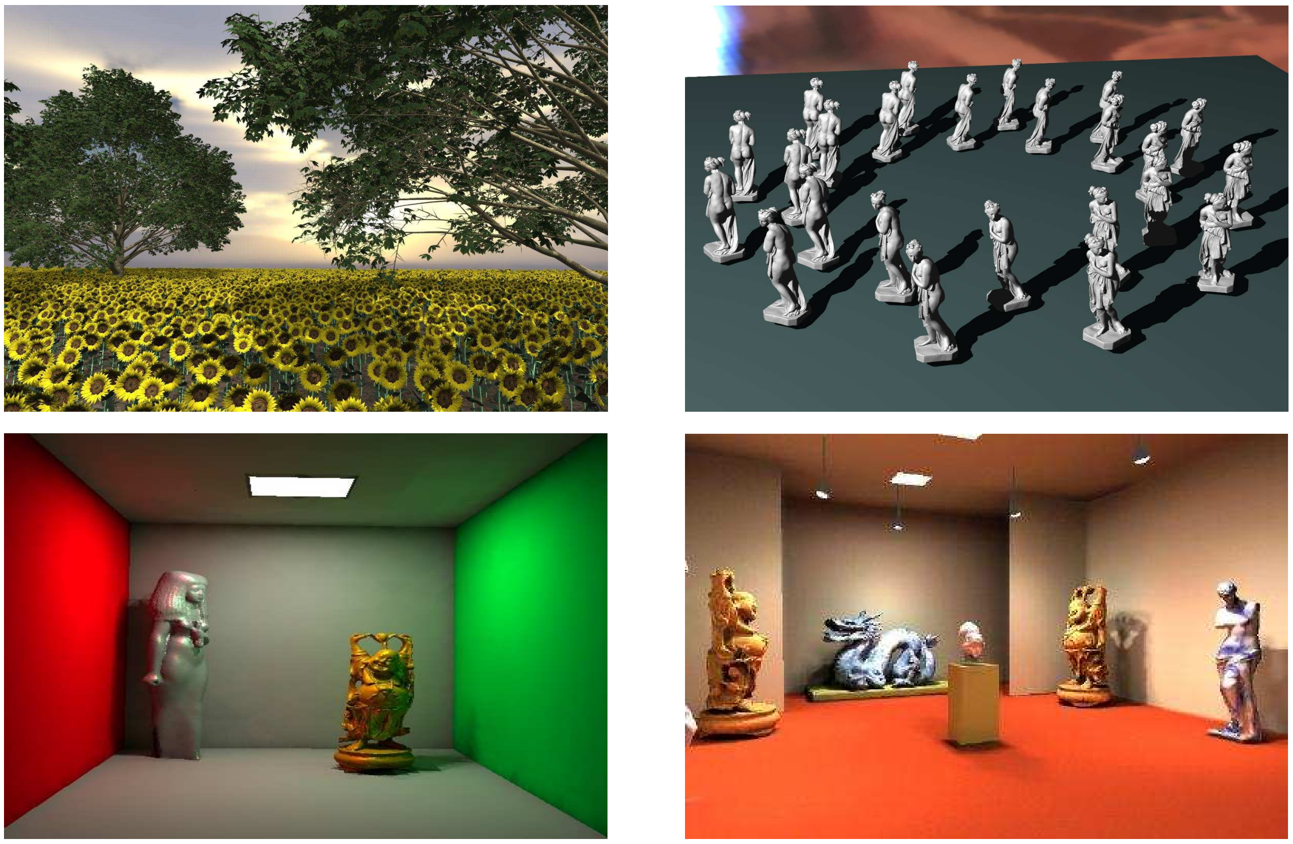


Figure 1.4: Complex point models with global illumination [WS05] [DYN04] effects like soft shadows, color bleeding, and reflections. Bottom Right: “a major goal of realistic image synthesis is to create an image that is perceptually indistinguishable from an actual scene”.

Figures 1.3 and 1.4 gives you some examples images showing the effects of *Global illumination*. It is a simulation of the physical process of light transport. Global Illumination effects are the results of two types of light reflections and refractions, namely Diffuse and Specular.

#### 1.1.2.1 Diffuse and Specular Inter-reflections

*Diffuse reflection* is the reflection of light from an uneven or granular surface such that an incident ray is seemingly reflected at a number of angles. The reflected light will evenly spread over the hemisphere surrounding the surface ($2\pi$ steradians).

*Specular reflection*, on the other hand, is the perfect, mirror-like reflection of light from a surface, in which light from a single incoming direction (a ray) is reflected into a single outgoing direction. Such behavior is described by the law of reflection, which states that the direction of incoming light (the incident ray), and the direction of outgoing light reflected (the reflected ray) make the same angle with respect to the surface normal, thus the angle of incidence equals the angle of reflection; this is commonly stated as $\theta_i = \theta_r$.

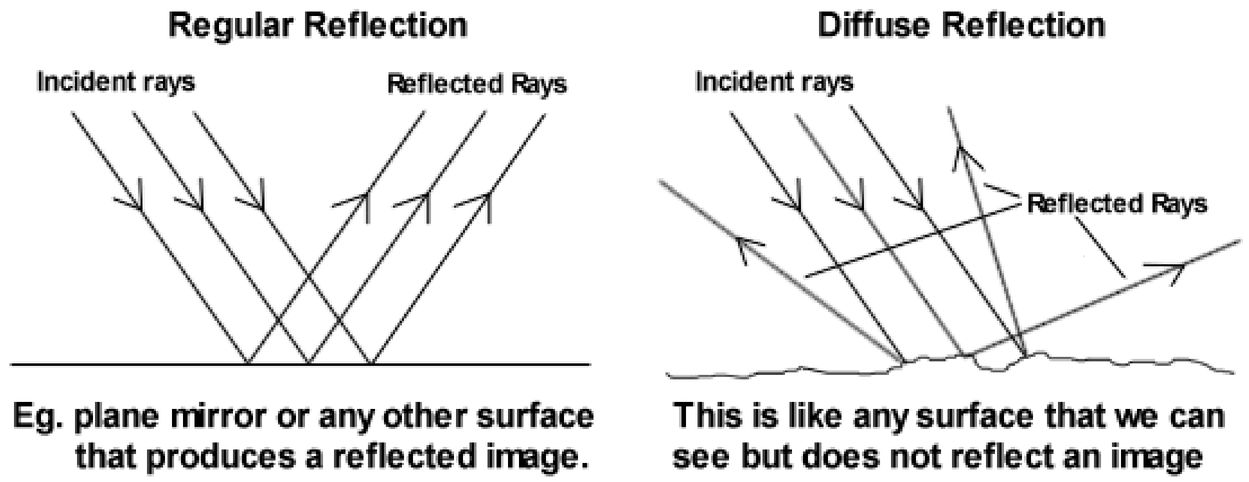

Regular Reflection
Incident rays
Reflected Rays
Eg. plane mirror or any other surface that produces a reflected image.
Diffuse Reflection
Incident rays
Reflected Rays
This is like any surface that we can see but does not reflect an image

Figure 1.5: Specular (Regular) and Diffuse Reflections

The most familiar example of the distinction between specular and diffuse reflection would be matte and glossy paints as used in home painting. Matte paints have a higher proportion of diffuse reflection, while gloss paints have a greater part of specular reflection.

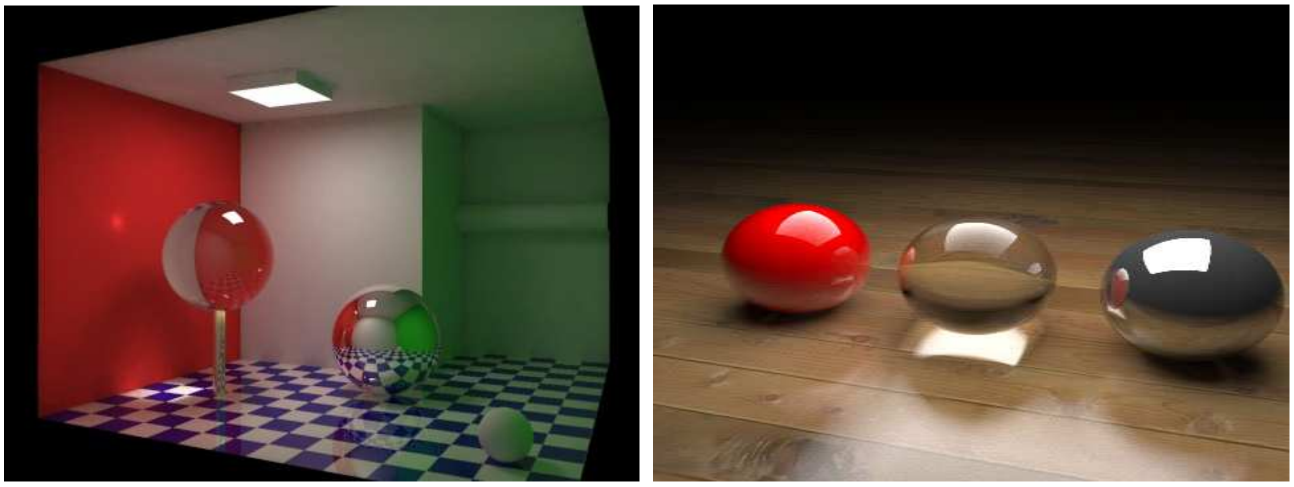


Figure 1.6: Left: Colors transfer (or ”bleed”) from one surface to another, an effect of diffuse inter-reflection. Also notable is the caustic projected on the red wall as light passes through the glass sphere. Right: Reflections and refractions due to the specular objects are clearly evident

Due to various specular and diffuse inter-reflections in any scene, various types of global illumination effects may be produced. Some of these effects are very interesting like color bleeding, soft shadows, specular highlights and caustics. *Color bleeding* is the phenomenon in which objects or surfaces are colored by reflection of colored light from nearby surfaces. It is an effect of diffuse inter-reflection. *Specular highlight* refers to the glossy spot which is formed on specular surfaces due to specular reflections. A *caustic* is the envelope of light rays reflected or refracted by a curved surface or object, or the projection of that envelope of rays on another surface. Light coming from the light source, being specularly reflected one or more times before being diffusely reflected in the direction of the eye, is the path traveled by light when creating caustics. Figure 1.6 shows color bleeding and specular inter-reflections including caustics.

Interesting methods like statistical photon tracing [Jen96], directional radiance maps [Wal05], and wavelets based hierarchical radiosity [GSCH93] have been invented for computing a global illumination solution. A good global illumination algorithm should cover both diffuse and specular inter-reflections and refractions, *Photon Mapping* being one such algorithm. Traditionally, all these methods *assume a surface* representation for the propagation of indirect lighting. Surfaces are either explicitly given as triangles, or implicitly computable. The lack of any sort of connectivity information in point-based modeling (PBM) systems now *hurts* photo-realistic rendering. This becomes especially true when it is not possible to correctly segment points obtained from an aggregation of objects (see Figure 2.1) to stitch together a surface.
There have been efforts trying to solve this problem [WS05], [Ama84, SJ00], [AA03, OBA$^+$03] , [RL00]. Our view is that these methods would work *even better* if fast pre-computation of diffuse illumination could be performed. *Fast Multipole Method* (FMM) provides an answer.

### 1.1.3 Fast computation with Fast Multipole Method

Computational science and engineering is replete with problems which require the evaluation of pairwise interactions in a large collection of particles. Direct evaluation of such interactions results in $O(N^2)$ complexity which places practical limits on the size of problems which can be considered. Techniques that attempt to overcome this limitation are labeled N-body methods. The N-body method is at the core of many computational problems, but simulations of celestial mechanics and coulombic interactions have motivated much of the research into these. Numerous efforts have aimed at reducing the computational complexity of the N-body method, particle-in-cell, particle-particle/particle-mesh being notable among these. The first numerically-defensible algorithm [DS00] that succeeded in reducing the N-body complexity to $O(N)$ was the Greengard-Rokhlin Fast Multipole Method (FMM) [GR87].

The algorithm derives its name from its original application. Initially developed for the fast evaluation of potential fields generated by a large number of sources (e.g. the gravitational and electrostatic potential fields governed by the Laplace equation), this method has been generalized for application to systems described by the Helmholtz and Maxwell equations, and to name a few, currently finds acceptance in chemistry[BCL$^+$92], fluid dynamics[GKM96], image processing[EDD03], and fast summation of radial-basis functions [CBC$^+$01]. For its wide applicability and impact on scientific computing, the FMM has been listed as one of the top ten numerical algorithms invented in the 20th century[DS00]. The FMM, in a broad sense, enables the product of restricted dense matrices with a vector to be evaluated in $O(N)$ or $O(N \log N)$ operations, when direct multiplication requires $O(N^2)$ operations.

Besides being very efficient ($O(N)$ algorithm) and applicable to a wide range of problem domains, the FMM is also highly parallel in structure. Thus implementing it on a parallel, high performance multi-processor cluster will further speedup the computation of diffuse illumination for our input point sampled scene. Our interest lies in a design of a parallel FMM algorithm that uses static decomposition, does not require any explicit dynamic load balancing and is rigorously analyzable. The algorithm must be capable of being efficiently implemented on any model of parallel computation.

### 1.1.4 Parallel computations on GPU

The graphics processor (GPU) on today's video cards has evolved into an extremely powerful and flexible processor. The latest GPUs have undergoing a major transition, from supporting a few fixed algorithms to being fully programmable. High level languages have emerged for graphics hardware, making this computational power accessible. Architecturally, GPUs are highly parallel streaming processors optimized for vector operations, with both MIMD (vertex) and SIMD (pixel) pipelines. With the rapid improvements in the performance and programmability of GPUs, the idea of harnessing the power of GPUs for general-purpose computing has emerged. Problems, requiring heavy computations, like those dealing with huge arrays, can be transformed and mapped onto a GPU to get fast and efficient solutions. This field of research, termed as *General-purpose GPU (GPGPU) computing* has found its way into fields as diverse as databases and data mining, scientific image processing, signal processing etc.

Many specific algorithms like bitonic sorting, parallel prefix sum, matrix multiplication and transpose, parallel Mersenne Twister (random number generation) etc. have been efficiently implemented using the GPGPU framework. One *such* algorithm which can harness the capabilities of the GPUs is *parallel adaptive fast multipole method*.

### 1.1.5 Visibility between Point Pairs

Even a good and efficient global illumination algorithm would not give us correct results if we do not have information about mutual visibility between points. An important aspect of capturing the radiance (be it a finite-element based strategy or otherwise) is an object space *view-independent* knowledge of visibility between point pairs.*Visibility calculation between point pairs is **essential** as a point receives energy from other point only if it is **visible** to that point.* But its easier said than done. Its complicated in our case as our input data set is a point based model with *no connectivity* information. Thus, we do not have knowledge of any intervening surfaces occluding a pair of points. Theoretically, it is therefore impossible to determine exact visibility between a pair of points. We, thus, restrict ourselves to **approximate visibility**.

## 1.2 Problem Definition

After getting a brief overview of the topics, let us now define the problem we pose in this report.

**Problem Definition:** *Capturing interreflection effects in a scene when the input is available as point samples of hard to segment entities.*
There are four things to look out for:

- Computing a mutual visibility solution for point pairs is one major and a necessary step for achieving good and correct global illumination effects.
- Inter-reflection effects include both diffuse and specular effects like reflections, refractions, and caustics.
- We compute diffuse inter-reflections using the **Fast Multipole Method**(FMM).
- We desire parallel implementation of visibility and FMM algorithms on Graphics Processing Units(GPUs) so as to achieve speedups for generating the global illumination solution.

## 1.3 Overview of the Report

Having got a brief overview of the keyterms, let us review the approach in detail in the subsequent chapters. The rest of the report is organized as follows. We present a novel, hierarchical, fast, and memory efficient algorithm to compute a description of mutual visibility, in the form of a *visibility map* (V-Map), for point models, especially for the global illumination problem, in Chapter 2. We evaluate our scheme analytically, qualitatively, and quantitatively by providing results for the same. As discussed above, we use FMM for solution to diffuse global illumination in point sampled scenes. An efficient algorithmic design for fast, parallel FMM (yet to be implemented) is detailed in Chapter 3 along with necessary hints to implement a similar algorithm on GPU. Also, an overview of parallel GPU implementation of elementary operations like parallel perfix sum, bitonic sort and construction of simple, parallel octree textures is given which are eventually required for implementing parallel FMM on GPU. Further, Chapter 4 discusses efficient algorithms required for computing specular effects (reflections, refractions, caustics) for point models, so as to give a complete and fast global illumination package for point models. We conclude our report with our concluding remarks in Chapter 5.

Chapter 2

# Visibility Maps in Point Clouds for Global Illumination

**Overview and Contribution:** Point-sampled geometry has gained significant interest due to their simplicity. The lack of connectivity touted as a plus, however, creates difficulties in generating global illumination effects. This becomes especially true when we have a complex scene consisting of several models, the data for which is available as hard to segment aggregated point models.

Inter-reflections in such scenes requires knowledge of visibility between point pairs. Computing visibility for points is all the more difficult (as compared to polygonal models), since we do not have any surface or object information. The visibility function is highly discontinuous and, like the BRDF, does not easily lend itself to an analytical FMM formulation. Thus the nature of this computation is $\theta(n^2)$ for $n$ primitives, which depends on the geometry of the scene. We, in this chapter, present a new visibility algorithm (Section 2.6.1) for Point Based Models. We further extend this algorithm to an efficient hierarchical algorithm (implemented using Octrees) to compute mutual visibility between points, represented in the form of a *visibility map*(V-Map). Thus the key features are twofold. First, we have a basic point-to-point visibility function that might be useful in its own right. Second, we have a hierarchical version of aggregated point clouds. Ray shooting and visibility queries can be answered in sub-linear time using this *V-Map* data structure. The scheme is then evaluated analytically, qualitatively, and quantitatively and it concludes with the desirability of *visibility maps*.

## 2.1 Introduction

In this section, I will describe the details of the papers we wrote [RGed].

Points as primitives have come to increasingly challenge polygons for complex models; as soon as triangles get smaller than individual pixels, the raison d'etre of traditional rendering can be questioned. Simultaneously, modern 3D digital photography and 3D scanning systems [LPC+00] acquire both geometry and appearance of complex, real-world objects in terms of (humongous) points. More important, however, is the considerable

Figure 2.1: Grottoes, such as the ones from China and India form a treasure for mankind. If data from the ceiling and the statues are available as point samples, can we capture the interreflections?

freedom points enjoy. The independence of connectivity and topology enable filtering operations, for instance, without having to worry about preserving topology or connectivity [PKKG03, OBA$^+$03, PZvBG00]. Further, points are a natural representation for most data acquired by range scanners [LPC$^+$00], and directly rendering them without the need for cleanup and tessellation makes for a huge advantage.

Such three-dimensional scanned point models of cultural heritage structures are useful for a variety of reasons – be it preservation, renovation, or simply viewing in a museum under various lighting conditions. We wish to see the effects of Global Illumination (GI) – the simulation of the physical process of light transport that captures inter-reflections – on point clouds of not just solitary models, but an environment that consists of such hard to segment (see Figure 2.1) entities.

Interesting methods like statistical photon tracing, directional radiance maps, and wavelets based hierarchical radiosity have been invented for this purpose. Traditionally all these methods *assume a surface* representation for the propagation of indirect lighting. Surfaces are either explicitly given as triangles, or implicitly computable.The absence of connectivity between points inherent in point based models now *hurts* computation, especially in such hard to segment models.

Moreover, an important aspect of capturing the radiance (be it a finite-element based strategy or otherwise) is an object space view-independent knowledge of visibility between point pairs. A view-independent visibility solution cannot (in general) use the popular hardware-based z-buffering technique. Since points are zero-dimensional, only approximate invisibility between points can be inferred.

This chapter presents a atomic point-pair visibility algorithm. The visibility problem, in general, has a *worst-case* $\Theta(n^2)$ time complexity for $n$ primitives. Given the fact that point models are complex, dense and consists of millions of points, visibility algorithms are highly time consuming. In real scenes, we might have *partitions* that are completely unoccluded, or hopelessly obscured. Hierarchical visibility is often used to discover unoccluding portions, and prune uninteresting parts of the mutual-visibility problem. We define the *visibility map* for this purpose. With this map, *visibility queries are answered quickly* whether we have lots of unoccluded space (such as the Cornell room without any boxes) or densely occupied space (the same room packed with boxes). Although the atomic algorithm is modelled from [DTG00], there are several performance enhancements. Further, the explicit use of hierarchy in the new proposed method is to be noted.

Many global visibility (both view-dependent and view-independent) solutions for *polygonal* models have previously been designed [DDP97]. [Bit02] provides a visibility map for an input scene given in terms of polygons. However, the V-Map structure presented here is different from the visibility map of [Bit02], which specifies the *Potential Visible Set* from a given view (unlike *view-independent* in our case) and uses it specifically for occlusion culling. Visibility has been considered by [WS05] for radiosity on point models, but their primary focus was on computing radiance than visibility. The algorithm presented, on the other hand, focuses on computing mutual visibility in the context of point clouds.

*In summary, the chapter presents a hierarchical, approximate, fast, and memory efficient visibility determination algorithm suitable for point based models, especially for the global illumination problem.*

Before introducing the core algorithm for constructing V-Maps, we first give a brief introduction of the Fast Multipole Method (FMM), the algorithm used to compute diffuse global illumination solution for input point cloud in Section 2.2 followed by an overview of what a V-Map is and what all queries can it entertain in Section 2.3. We then review our previous approaches for computing mutual visibility between point pairs and constructing V-Maps in Section 2.4, as described in [Gor06]. We first describe our basic “primitive” visibility algorithm for point to point visibility in subsection 2.4.1. Next, we extend this primitive in the FMM context to build a hierarchical visibility algorithm for V-Maps in subsection 2.4.2. We follow it up highlighting the problems (Section 2.5) the above algorithms had and describe the necessary changes we made to produce an efficient (both in memory and time) and optimized core algorithm for constructing V-Maps in Section 2.6 and Section 2.7. We evaluate our scheme by providing results for the same in Section 2.8.

## 2.2 FMM-based Global Illumination

The FMM [GR87], in a broad sense, enables the product of restricted dense matrices with a vector to be evaluated in $O(N)$ or $O(N \log N)$ operations, when direct multiplication requires $O(N^2)$ operations. A global illumination version of FMM (albeit for polygonal models) was given in [KGC04]. However, the inherent notion of points in FMM blends very well with hierarchical point models as input. We therefore devised a point-based version which serve as a test bed for the proof of our concept of V-maps. For every node in an octree, FMM defines an “interaction” list consisting of all possible nodes which can contribute energy to this node. The V-map data structure is therefore needed to identify the *visible* nodes in the interaction list.

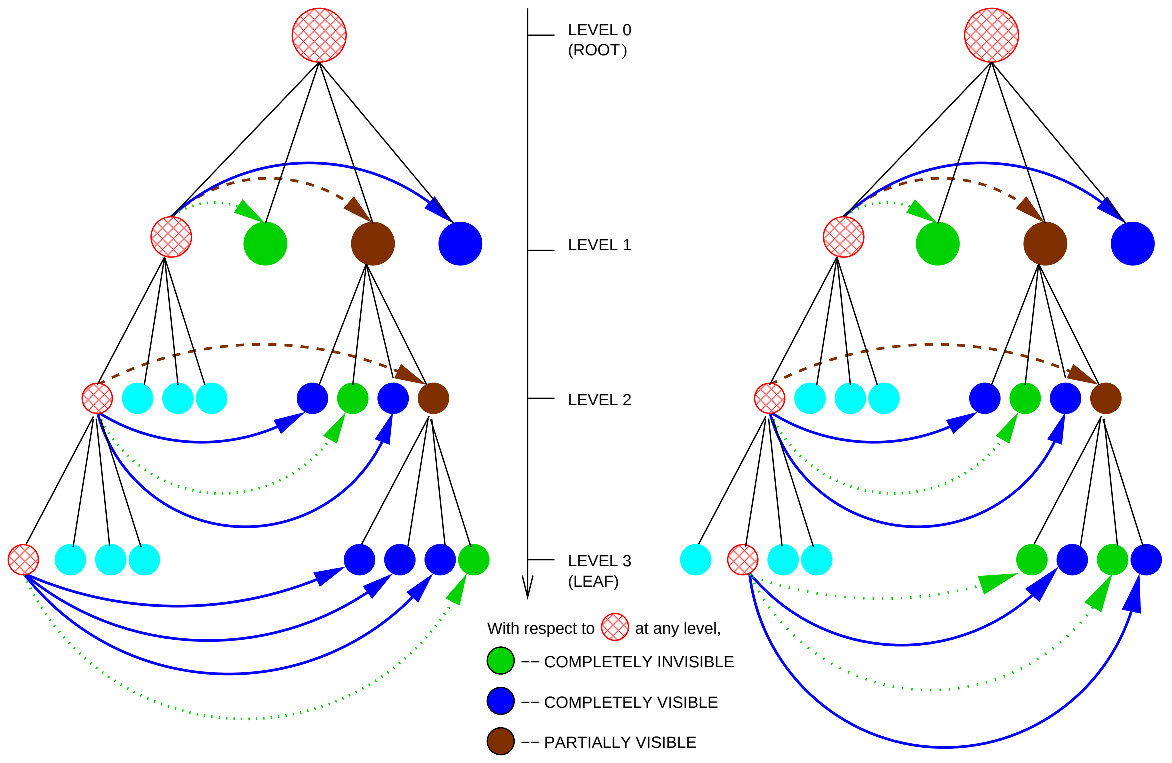


Figure 2.2: Views of the visibility map (with respect to the hatched node in red) is shown. Every point in the hatched node at the first level is completely visible from every point in only one node (the extreme one). At level 2, there are two such nodes. The Figure on the left shows that at the lowest level, there are three visible leaves for the (extreme) hatched node; on the other hand the Figure on the right shows that there are only two such visible leaves, for the second son (hatched node). The Figure also shows invisible nodes that are connected with dotted lines. For example, at level 1, there is one (green) node $G$ such that no point in $G$ is visible to any point in the hatched node. Finally the dashed lines shows "partially visible" nodes which need to be expanded. Partial and invisible nodes are not explicitly stored in the visibility map since they can be deduced.

Details on the theoretical foundations of FMM, requirements subject to which the FMM can be applied to a particular domain and discussion on the actual algorithm and its complexity as well as the mathematical apparatus required to apply the FMM to radiosity are available in [Gor06] and [KC03]. Five theorems with respect to the core radiosity equation are also proved in this context.

Note that usage of the V-map is not restricted to FMM based GI but can also be incorporated in existing hierarchical GI algorithms for point models.

## 2.3 Visibility Maps

The construction of the visibility map starts assuming a hierarchy is given. For the purpose of illustration of our method, we use the standard *regular* octree-based subdivision of space. Figure 2.3 shows a two dimensional version to illustrate the terminology.

The *visibility map* for a tree is a collection of visibility links for every node in the tree. The *visibility link* for any node $p$ is a list $L$ of nodes; every point in any node in $L$ is guaranteed to be visible from every point in $p$. Figure 2.2 provides different views of the *visibility map*. (The illustration shows directed links for clarity; in fact, the links are bidirectional.)

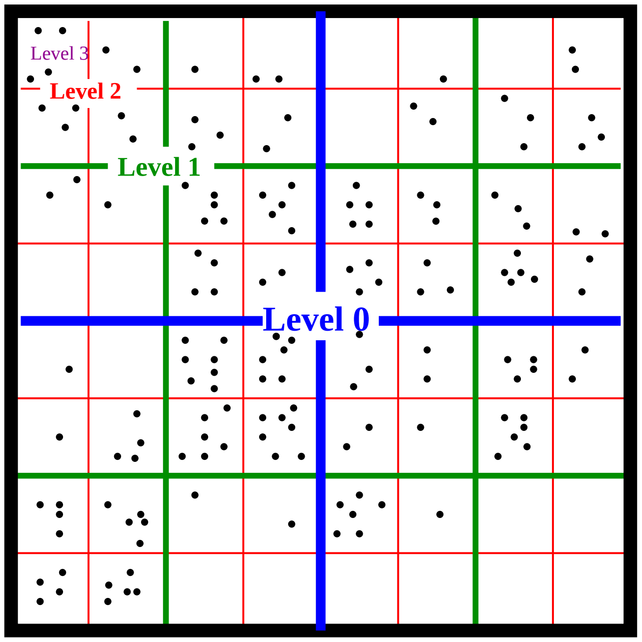


Figure 2.3: Leaf nodes (or cells, or voxels) are at level three.

Visibility maps entertain efficient answers to the following queries.

1. Is point $x$ visible to point $y$? The answer may well be, "Yes, and, by the way, there are a whole bunch of other points near $y$ that are also visible." This leads to the next query.

2. What is the visibility status of $u$ points around $x$ with respect to $v$ points around $y$? An immediate way of answer this question is to repeat a "primitive" point-point visibility query $uv$ times. With a visibility map, based on the scene, the answer is obtained more efficiently with $O(1)$ point-point visibility queries.

3. Given a point $x$ and a ray $R$, determine the first object of intersection.

4. Is point $x$ in the shadow (umbra) of a light source?

All the above queries are done with a simple traversal of the octree. For example for the third query, we traverse the root to leaf $O(\log n)$ nodes on which $x$ lies. For any such node $p$, we check if $R$ intersects any node $p_i$ in the visibility link of $p$. A success here enables easy answer to the desired query.

## 2.4 Previous Approach: Visibility in Point Models

Visibility is not considered in the original FMM algorithm. For our purposes it is complicated in that occlusion is a point to point based phenomenon and not a node to node phenomenon where the bulk of the computation occur. In this section we first give a point to point visibility algorithm. Later we incorporate it in the FMM context by constructing the V-Maps.

### 2.4.1 Point–Point Visibility

Since our input data set is a point based model with *no connectivity information*, we do not have knowledge of any intervening surfaces occluding a pair of points. Theoretically, it is therefore impossible to determine exact visibility between a pair of points. Thus, we restrict ourselves to *approximate visibility* with a value between 0 and 1. Consider two points *p* and *q* (as in Figure 2.4) in the input scene on which we run a number of tests to efficiently produce $O(1)$ possible occluders.

First we apply the culling filter to straightway eliminate backfacing surfaces.

$$n_p \cdot \overline{pq} > 0 \quad \text{and} \quad n_q \cdot \overline{qp} > 0$$

where $n_p$ and $n_q$ are normals at point *p* and *q* respectively.

**Algorithm 1** Visibility between Point 'p' and Point 'q'

```
procedure PointtoPointVisibility(p,q)
 1: Define threshold $t_1$, $visible_{p,q} = 1$
 2: if FacingEachOther($p$,$q$) then
 3:    Find $k$ closest points in region $\Delta$ around $\overline{pq}$
 4:    Prune based on tangent plane
 5:    for $i = 0$ to $k$ do
 6:       $contributeVis_i = visibility\_look\_up(distance_i)$
 7:       $visible_{p,q} = visible_{p,q} * contributeVis_i$
 8:    end for
 9:    if $visible_{p,q}) >$ threshold $t_1$ then
10:       return(visible)
11:    end if
12: end if
```

If the above condition is satisfied, we then determine the possible occluder set *X* $(\mid X \mid = k)$. This is a set of points in the point cloud which are close to $\overline{pq}$ and thus can possible affect the visibility. These points lie in a cylinder around $\overline{pq}$. In Figure 2.4, for example, $x_3$ is dropped. This set is further pruned by considering the tangent plane at each potential occluder. If the tangent plane does not intersect $\overline{pq}$ the occluder is dropped (for example, $x_1$ in Figure 2.4). A final pruning happens by measuring the *distance along the tangent* to $\overline{pq}$. We pick the smallest $O(1)$ occluders (equal to 3 in our implementation) using this distance metric. We compute a visibility fraction based on this distance. This results in Algorithm 2.4.1.

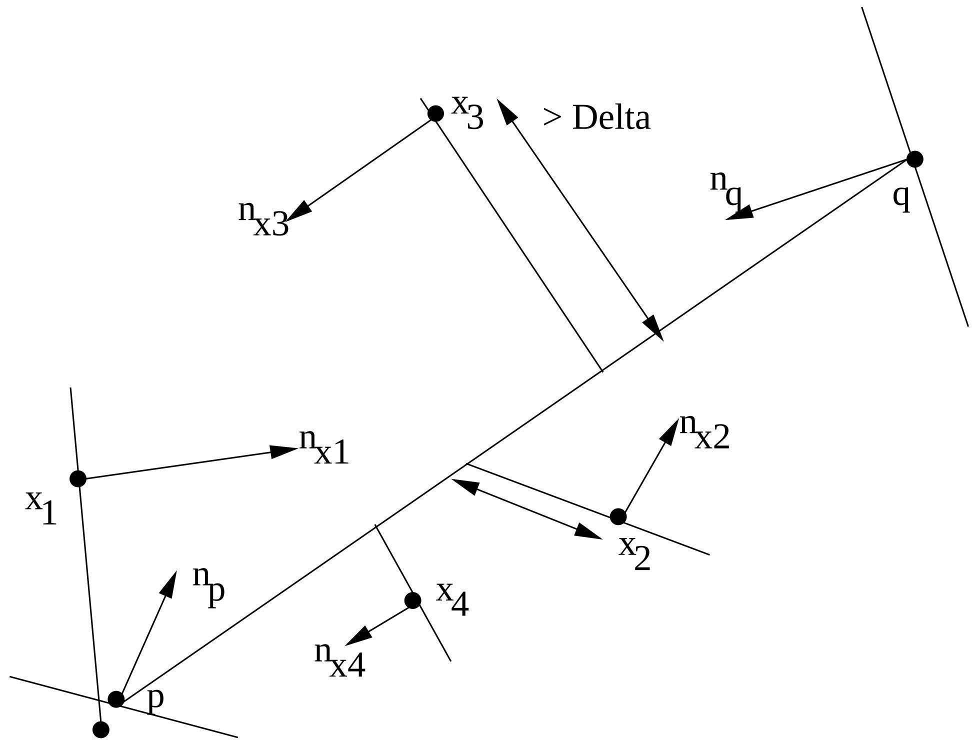


Figure 2.4: Only $x_2$ and $x_4$ will be considered as occluders. We reject $x_1$ as the intersection point of the tangent plane lies outside the line segment $\overline{pq}$. $x_3$ has earlier been rejected because it is more than a distance $\Delta$ from the line segment $\overline{pq}$.

### 2.4.2 Hierarchical Visibility

In this section, we now incorporate visibility into the FMM algorithm by constructing the visibility links to form a V-Map for the point modelled scene. The object space composed of points was was initially divided into an *non-adaptive octree*. Note that each point receives energy from every other point either directly, or through the points in the interaction list of the ancestor of the leaf it belongs to. The key idea is to modify the interaction list

If the points in a node $c$ in the interaction list of node $b$ are completely visible from *every* point in $b$, then the *visibility state* of the pair (b,c) is said to be *valid*. If, on the other hand, no point in $c$ is visible from any point in $b$, the visibility state of the pair (b,c) is said to be *invalid*. The node $c$ is dropped from the interaction list since no exchange of energy is permissible. Finally, when the visibility state is *partial*, we *postpone* the interaction. In the sequel, we ensure that the postponed interaction happens at the lowest possible depth (the root is at depth 0) for maximum efficiency. This is done by extending the notion of point–point visibility to the node level as follows.

#### 2.4.2.1 Point–Leaf Visibility

In this section, we determine the visibility between a leaf node $L$ and a point $p$. We start by making point to point visibility calculations between point $p$ and every point $p_i \in L$. This results in Algorithm 2.4.2.1.

**Algorithm 2** Visibility between Point 'p' and Leaf 'L'

```
procedure PointtoLeafVisibility(p,L)
1:  Define threshold $t_2$, Visi_point_$L$ = 0
2:  for each point $p_i \in L$ do
3:      state = PointtoPointVisibility($p, p_i$)
4:      if equals(state,visible) then
5:          $Visi\_point\_L = Visi\_point\_L + 1$
6:          if $Visi\_point\_L >$ threshold $t_2$ then
7:              return(visible)
8:          end if
9:      end if
10: end for
11: return(invisible)
```

#### 2.4.2.2 Leaf–Leaf Visibility

Similarly we determine visibility between two leaf nodes $C$ and $L$. For every point $p_i \in C$, we start by calculating *Point–Leaf Visibility* between point $p_i$ and $L$. This results in Algorithm 2.4.2.2.

**Algorithm 3** Visibility between Leaf 'L' and Leaf 'C'

```
procedure LeaftoLeafVisibility(L,C)
1:  Define threshold $t_3$, Visi_point_L = 0
2:  for each point $p_i \in C$ do
3:      state = PointtoLeafVisibility($p_i, Leaf\ L$)
4:      if equals(state,visible) then
5:          Visi_point_L = Visi_point_L + 1
6:      end if
7:  end for
8:  if $Visi\_point\_L >$ threshold $t_3$ then
9:      return(visible)
10: end if
11: return(invisible)
```

#### 2.4.2.3 Node–Node Visibility

In this section, we determine the visibility between nodes $A$ and $B$ of the octree. We start by computing visibility of all $b \in Leafnodes(B)$ to all $a \in Leafnodes(A)$. If all are visible, the status is valid. If none are visible, the state is invalid. Otherwise, we have partial visibility. In this scenario, we repeat the procedure

*Node–Node Visibility* for all the child nodes of $A$ and $B$. Note that there is no case of partial visibility between leaf nodes. The algorithm 2.4.2.3 is summarized below.

**Algorithm 4** Visibility between Node 'A' and Node 'B'

```
procedure NodetoNodeVisibility(A,B)
 1: Declare vis_cnt = 0
 2: for each a ∈ leafcell(A) do
 3:     for each b ∈ leafcell(B) do
 4:         state = LeaftoLeafVisibility(a, b)
 5:         if equals(state,visible) then
 6:             vis_cnt = vis_cnt + 1
 7:         end if
 8:     end for
 9: end for
10: if equals(vis_cnt,LeafNode(A).size*LeafNode(B).size) then
11:     return(valid)
12: else if equals(vis_cnt,0) then
13:     return(invalid)
14: else
15:     return(partial)
16: end if
```

#### 2.4.2.4 Constructing Visibility Map

We now are in a position to compute the interaction list and construct the Visibility Map as in Algorithm 2.4.2.4. We start by initializing an interaction list of every node to be its seven siblings. This default list ensures that every point is presumed to interact with every other point. The V-Map is then constructed by calling Algorithm 2.4.2.4 initially for the $root$ node. It then recursively sets up the relevant visibility links in the interaction list. The complexity of this algorithm is around $O(N^2 logN)$, assuming point-point visibility takes $O(1)$ time.

## 2.5 Limitations

Having looked at the previous approach to the constuction of the Visibility Maps, we now look at some of the limitations the above described algorithms possess.

- We use three different thresholds in the whole process of construction of V-Maps. Getting a proper combination of all the three threshold values so as to produce *correct* results is a difficult task. Further, these threshold values are somewhat dependent on the input scene complexity and hence finding the correct thresholds for the given scene calls for lots of trial and error tests.

**Algorithm 5** Construct Visibility Map

```
procedure OctreeVisibility(Node A)
 1: for each node B ∈ interactionlist(A) do
 2:    if notLeaf(A) then
 3:       state=NodetoNodeVisibility(A,B)
 4:    else if Leaf(A) then
 5:       state=LeaftoLeafVisibility(A,B)
 6:    end if
 7:    if equals(state,valid) then
 8:       Retain B in interactionlist(A)
 9:    else if equals(state,partial) then
10:       for each a ∈ children(A) do
11:          for each b ∈ children(B) do
12:             interactionlist(a).add(b)
13:          end for
14:       end for
15:       interactionlist(A).remove(B)
16:    else if equals(state,invalid) then
17:       interactionlist(A).remove(B)
18:    end if
19: end for
20: for each R ∈ child(A) do
21:    OctreeVisibility(R)
22: end for
```

- The process of finding the $k$ nearest occluders, out of millions of input points, for determining mutual visibility between points is a time consuming task and a major bottleneck in terms of speedups desired. We should have an efficient algorithm for finding the potential occluders for a fast point–pair visibility algorithm.

- In the point–pair visibility algorithm, we dont use any conditions which helps us to exit instantaneously as soon as an invisibility case is detected. We, thus, perform unnecessary time-consuming calculations reducing the speed further.

- If noted carefully, Algorithm 2.4.2.4 does *high amount* of extra computations in case a partial visibility case is detected at a particular level. To elaborate, we compute visibility between two nodes by computing visibility between all leaf–pairs of both the nodes. If a partial visibility case is detected, we postpone the computations to the next level, i.e. between the children of the two nodes. We then *again* compute (re-compute) the visibility between the same leaf–pairs when we visit the children of those nodes in future. This extra computation introduces a factor of at least $O(log(N)$, assuming the leaf–pair visibility takes $O(1)$ time (which is not generally the case. Leaf–Pair visibility generally takes *a lot more* then $O(1)$). Hence, if we can reduce these extra computations, high speed-ups can be achieved.

- The algorithms described above has been implemented on non-adaptive octrees constructed on input point model. But many a times, non-adaptive division is not preferable, especially when the data in the scene is of non-uniform density. Thus we would desire to scale this algorithm to suit the adaptive sub-division structure of the octree.

## 2.6 New Approach: Visibility Maps in Point Models

We now present a modified, hierarchical, fast and memory efficient visibility determination algorithm suitable for point based models, which overcomes all the limitations described in Section 2.5. We first explain the modified *point–pair visibility algorithm* in subsection 2.6.1 and follow it up by extending it to construct the V-Maps in the *most efficient* manner in subsection 2.6.3.

### 2.6.1 Point–Pair Visibility Algorithm

Since our input data set is a point model with *no connectivity information*, we don't have knowledge of any intervening surfaces occluding a pair of points. Theoretically, it is therefore impossible to determine exact visibility but only approximate visibility. Albeit, for practical purposes we restrict ourself to boolean visibility ($0$ or $1$) based on results of the following visibility tests. This algorithm is motivated by work done in [DTG00].

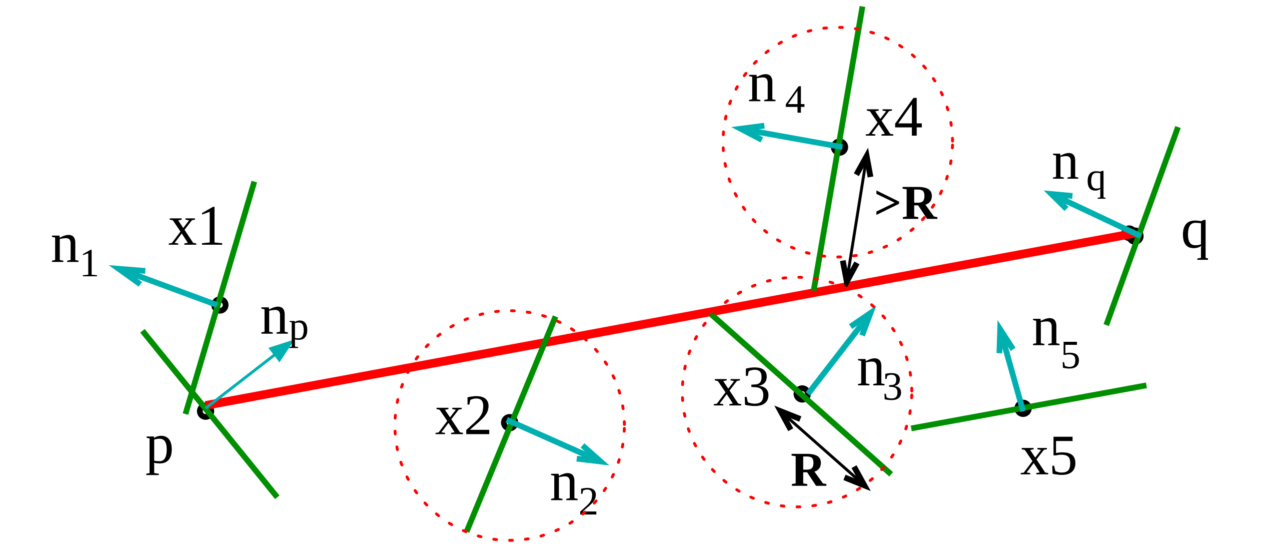


Figure 2.5: Only $x_2$ and $x_3$ will be considered as occluders. We reject $x_1$ as the intersection point of the tangent plane lies outside segment $\overline{pq}$, $x_4$ because it is more than a distance $R$ away from $\overline{pq}$, and $x_5$ as its tangent plane is parallel to $\overline{pq}$.

Consider two points *p* and *q* with normals $n_p$ & $n_q$ as in Figure 2.5. We run the following tests to efficiently produce $O(1)$ possible occluders.

1. Cull backfacing surfaces *not* satisfying the constraint $n_p \cdot \overline{pq} > 0$ **and** $n_q \cdot \overline{qp} > 0$

2. Determine the possible occluder set *X*, of points close to $\overline{pq}$ which can possibly affect their visibility. As an example, in Figure 2.5, points $(x_1, x_2, x_3, x_4, x_5) \in X$. An efficient way to obtain $X$ is to start with the output of a 3D Bresenham line segment algorithm [E.62] between *p* and *q*. Bresenhams algorithm will output a set $Y$ of points which are co-linear with and between *p* and *q*. Using the hierarchical structure, add to $X$, all points from the leaves containing any point from $Y$.

3. Prune $X$ further by applying a variety of tangent plane intersection tests as shown in Figure 2.5.

Any point from $X$ which fails any of the tangent tests is considered an occluder to $\overline{pq}$. If we find $K$ such occluders, $q$ is considered invisible to $p$.
Elimination of thresholds as compared to previous point–pair visibility approach simplifies the tasks for the user and also helps in achieving better results. Also, the bresenham's algorithm used, gives us an efficient way to find the potential occluders between given point–pair, thereby providing us with necessary speedups.

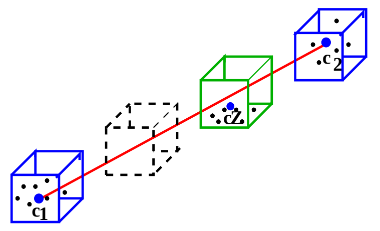
### 2.6.2 Octree Depth Considerations

In a hierarchical setting, and for sake of efficiency, we may terminate the hierarchy to some level with several points in a leaf. A simple extension of our point–pair visibility algorithm to a *leaf–pair* would be to compute visibility between their centroids ($p$ and $q$, Figure 2.5). Set $X$ now comprises of centroids, each corresponding to a intersecting leaf (ILF). Our occlusion criteria is then:

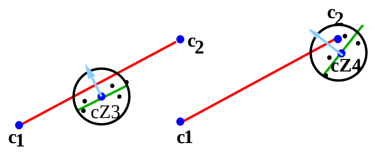
- If the ILF contains no point, it is dropped.

- Likewise, if the tangent plane of the centroid of ILF is parallel to $\overline{pq}$(x5), intersects outside segment $\overline{pq}$(x1), or intersects outside distance $R$ (distance between centroid to its *farthest* point in the leaf)(x3), we drop the leaf (See Figure 2.6).

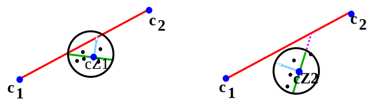
Any ILF which fails any of the above tests is deemed to be an occluder for point-pair $p - q$. We consider $p - q$ as invisible, if there exists *at least* one occluding ILF. Although this algorithm involves approximation, the high density of point models results in no significant artifacts. (See Section 2.8.1).
Extending point–pair visibility determination algorithm to the leaf level (although is an approximation) makes it much more faster. The strict condition of concluding a leaf–pair as *invisible*, in a presence of just a *single* occluder balances the approximation done. Further, finding just a single occluder makes us exit instantaneously (as soon as an invisibility case is detected) and thereby avoids making unnecessary computations, making it much more time efficient.



(a) A potential occluding set of voxels are generated given centroids $c_1$ and $c_2$. The dotted voxel contains no point and is dropped.



(b) $cZ_3$ is rejected because the tangent plane is parallel to $c_1c_2$. Similarly, we reject $cZ_4$ as the intersection point of the tangent plane lies outside the line segment.



(c) $cZ_2$ is rejected because the line segment $c_1c_2$ doesn't intersect the tangent plane within a circle of radius determined by the farthest point from the centroid. Only $cZ_1$ is considered as a potential occluder.

Figure 2.6: Point-Point visibility is obtained by performing a number of tests. Now its extended to Leaf-Leaf visibility

### 2.6.3 Construction of Visibility Maps

We now extend the leaf–pair algorithm (subsection 2.6.2) to determine visibility between nodes (non-leaf) in the hierarchy. In addition, the new algorithm presented is the most efficient and optimized algorithm for constructing the V-Maps for the given point model. *No extra computations between node and leaf pairs are are performed*, thereby reducing much of the time complexity as compared to the original algorithm(Section 2.4.2). We also give a brief overview of how the constructed V-Map can be applied to compute a global illumination solution.

In constructing a visibility map, we are given a hierarchy and, optionally for each node, a list of interacting nodes termed o-IL (a mnemonic for Old Interaction List). If the o-IL is not given, we initialize the o-IL of every node to be its seven siblings. This default o-IL list ensures that every point is presumed to interact with every other point. The V-Map is then constructed by calling Algorithm 2.6.3 initially for the $root$ node, which sets up the relevant visibility links in New Interaction List(n-IL). This algorithm invokes Algorithm 2.6.3 which constructs the visibility links for all descendants of $A$ w.r.t all descendants of $B$ (and vice-versa) at the best (i.e. highest) possible level. This ensures an optimal structure for hierarchical radiosity as well as reduces redundant computations.

**Computational Complexity:** The visibility problem provides an answer to $N = \Theta(n^2)$ pair-wise queries, $n$ being the number of points in input model. As a result, we measure the efficiency w.r.t $N$ especially since the V-Map purports to answer *any* of these $N$ queries. We shall see later that `NodetoNodeVisibility()` is linear w.r.t $N$. `OctreeVisibility()` then has the recurrence relation $T(h) = 8T(h-1) + N$ (for a Node $A$ at height $h$) resulting in an overall linear time algorithm (w.r.t. $N$), which is as far the best possible for *any* algorithm that builds the V-Map.

**Algorithm 6** Construct Visibility Map

```
procedure OctreeVisibility(Node A)
1:  for each node B in old interaction list (o-IL) of A do
2:      if NodetoNodeVisibility(A,B) == VISIBLE then
3:          add B in new interaction list (n-IL) of A
4:          add A in new interaction list (n-IL) of B
5:      end if
6:      remove A from old interaction list (o-IL) of B
7:  end for
8:  for each C in children(A) do
9:      OctreeVisibility(C)
10: end for
```

**Algorithm 7** Node to Node Visibility Algorithm

```
procedure NodetoNodeVisibility(Node A, Node B)
1:  if $A$ and $B$ are leaf then
2:      return the status of leaf–pair visibility algorithm for $A$ & $B$ (subsection 2.6.2)
3:  end if
4:  Declare s1=children(A).size
5:  Declare s2=children(B).size
6:  Declare a temporary boolean matrix $M$ of size($s1 * s2$)
7:  Declare count=0
8:  for each a $\in$ children(A) do
9:      for each b $\in$ children(B) do
10:         state=NodetoNodeVisibility(a,b)
11:         if equals(state,visible) then
12:             Store true at corresponding location in $M$.
13:             $count = count + 1$
14:         end if
15:     end for
16: end for
17: if count == $s1 * s2$ then
18:     free $M$ and return VISIBLE
19: else if count == 0 then
20:     free $M$ and return INVISIBLE
21: else
22:     for each a $\in$ children(A) do
23:         for each b $\in$ children(B) do
24:             Update n-IL of $a$ w.r.t every visible child $b$ (simple look up in $M$) & vice-versa, free $M$
25:         end for
26:     end for
27:     return PARTIAL.
28: end if
```

The complexity for `NodetoNodeVisibility(A,B)` is determined by the calls to point-pair visibility algorithm. Assuming the latter to be $O(1)$, the recurrence relation for the former is $T(h) = 64T(h-1) + O(1)$ (for a node $A$ at height $h$). The resulting equation is linear in $N$.

The overall algorithm consumes a small amount of memory (for storing $M$) during runtime. The constructed V-Map is also a memory efficient data structure as (apart from the basic octree structure) it requires to store only the link structure for every node.

**Visibility Map $+$ GI algorithms:**

1. Given a V-Map, ray shooting queries are reduced to searching for primitives in the *visibility set* of the primitive under consideration, thereby providing a *view-independent* preprocessed visibility solution. An intelligent search (using kd trees) will yield faster results.

2. Both diffuse and specular passes on GI for point models can use V-Maps and provide an algorithm (similar to photon mapping), which covers both the illumination effects.

## 2.7 Extending Visibility Maps to Adaptive Octrees

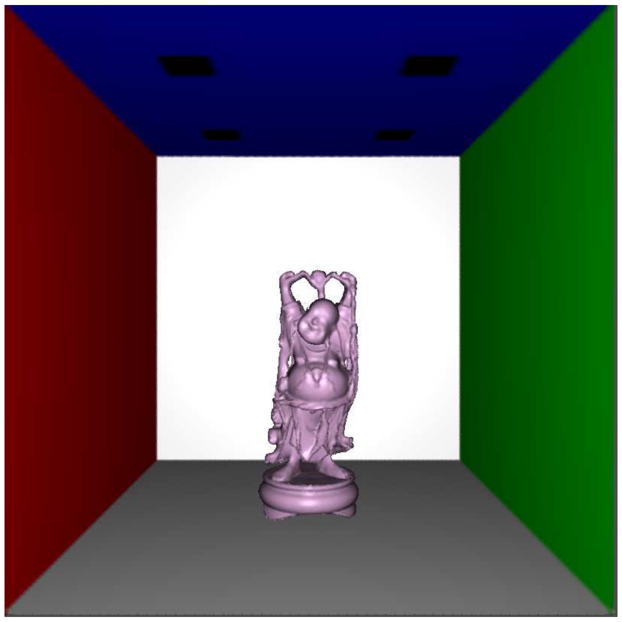
We now have a clear picture of what a V-Map signifies. An important thing to keep in mind while computing mutual visibility among points and constructing V-Maps would be the way spatial sub-division of the scene is done. The input scene can be divided in two ways using the octree-based data structure.

1. Non-Adaptive Subdivision of space, where all the leaves of the tree are at the same level and of same size (refer Figure 2.3). Figure 2.3 is divided till level 3, assuming root node to be at level 0.

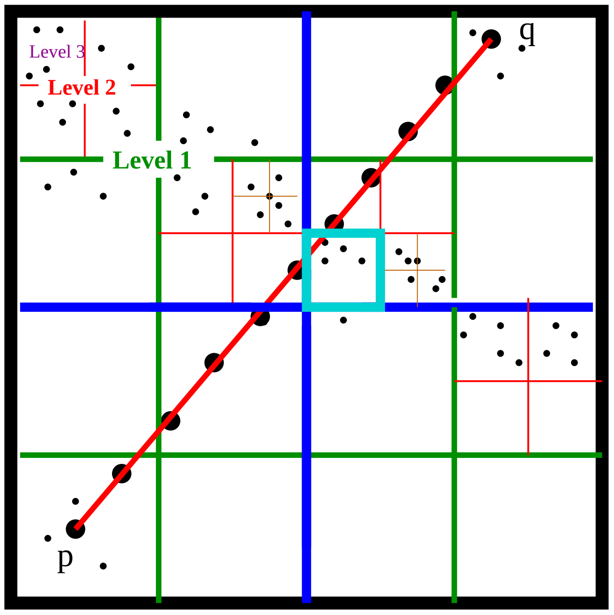
2. Adaptive sub-division of space based on input model density. The sub-division stops when a node in a tree contains some $k$ points or less. Leaves in the tree can now be present at different levels and hence will be of different sizes (See Figure 2.7(b)).

Till now, we saw the mutual point–pair visibility and V-Map construction algorithms being applied on *non-adaptive* octrees. But many a times, non-adaptive division is not preferable, especially when the data in the scene is of non-uniform density (see figure 2.7(a)).

However, the 3D Bresenhams algorithm used in determining visibility between points (or centroids for leaf–pair) is not suitable for irregular sub-division of space. Also, the Bresenhams algorithm output highly depends on the step-length selected (step-length is kept equal to leafcell side-length in non-adaptive octree due to its regular sub-division of space). A wrong step-length might many a times produce false negatives or false positives (refer Figure 2.7(b)). Interestingly, one way to get around this situation is to decrease the step-length



(a)



(b)

Figure 2.7: (a) The Buddha model and the cornell room both contain $500000$ points each. However, the density of points on Buddha is very high as compared to density on the walls of the room. (b) Bresenham Algorithm applied for computing visibility between two leaves in an adaptive octree structure. Wrong step-length might miss out the **potential occluding cell(cyan colored) in between cell $p$ and cell $q$ leading to wrong computation of visibility status between the two cells.**

used in Bresenham Algorithm to be equal to side-length of the leaf at greatest depth. But this, on the other hand increases the running times *drastically*.

We now present a different approach to computing visibility between points (or centroids for leaf–pair) in an adaptive octree sub-division of space. Here, instead of using the *3D Bresenham's Line Algorithm*, we use *Sphere-Ray intersection* for finding the list of *Potential occluders*. Like in Section 2.6, a V-Map is constructed assuming an input hierarchy using the Algorithm 2.6.3. The new, efficient and correct mutual leaf–pair visibility algorithm is presented below.

**Algorithm 8** Visibility between Leaf Cell 'A' and Leaf Cell 'B'

```
procedure LeaftoLeafVisibility(A,B)
1: if $A$ and $B$ face each other then
2:     return isIntersectingOctree(root,A,B)
3: else
4:     return INVISIBLE
5: end if
```

In simple words,

**Algorithm 9** Check whether line joining A and B intersects any octree node

```
procedure isIntersectingOctree(Node,A,B)
 1: if $Node$ is a leaf then
 2:     state = $true$ IF
 3:     // $Node$ is equal to $A$ or $B$ OR
 4:     // Tangent plane passing through centroid of $Node$ is parallel to line segment $\overline{AB}$) OR
 5:     $Node$ belongs to same surface as ($A$ or $B$) or line segment intersects outside $Node$ OR
 6:     $\overline{AB}$ intersects line at a distance more than the bounding sphere radius $R$)
 7:     if STATE == $true$ then
 8:         return VISIBLE
 9:     else
10:         return INVISIBLE
11:     end if
12: end if
13: if $\overline{AB}$ intersects $Node$ then
14:     for each child $NodeC$ of $Node$ do
15:         state = isIntersectingOctree(NodeC, A, B)
16:         if state == INVISIBLE then
17:             return INVISIBLE
18:         end if
19:     end for
20:     return VISIBLE
21: end if
```

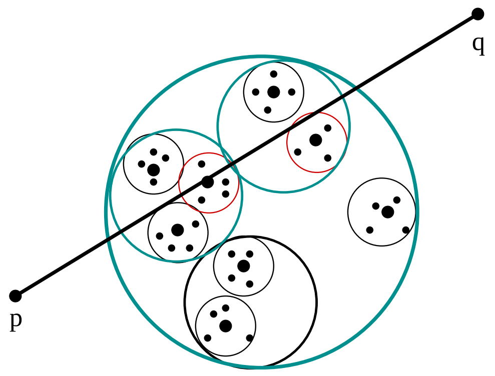


Figure 2.8: Ray-Sphere intersection algorithm to determine point-point visibility

- If node is not a leaf and $\overline{pq}$ intersects the node then traverse its children
- If node is a leaf then check whether tangent plane of that node intersects $pq$ within radius $R$ then node $p$ and $q$ are invisible otherwise declare $p$ and $q$ visible

Bounding sphere radius $R$ for a leafcell is set to the distance between the centroid and its farthest point in the leaf. Radius $R$ for any non-leaf node in the octree is set equal to maximum of the sum of distance between its center and the child's center and $R$ of that particular child.

## 2.8 Experimental Results

In this section, we discuss the validity and application of the proposed method to various point models. All examples shown are calculated using a Pentium4 2.6 GHz sans any GPU.

### 2.8.1 Visibility Validation

We validate our proposed method here. We have ran the code on point scenes taken as input and divided using an *Adaptive Octree Division* to form a hierarchical structure. We used our new *Sphere-Ray Intersection* method to compute the visibility in the following examples. Figure 2.9(a) shows a point model of empty Cornell room. Note the default colors of the walls. We now introduce a Stanford bunny. In Figure 2.9(b), the eye (w.r.t. which visibility is being computed) is on the red wall (on the left), marked with a *cyan* colored dot. The violet (purple) colour indicates those portions of the room that are visible to this eye. Notice the "shadow" of the bunny on the green wall and on the floor. The same idea is repeated with the eye placed at different locations for the Bunny and for different models like the Buddha and the Dragon.
Figure 2.10(a) shows a point model of a different, empty Cornell room. Note the default colors of the walls. We repeat similar tests for point model of an Indian god Ganesha (Figure 2.10(b) and 2.10(c)), an Indian goddess Satya (Figure 2.10(d) and 2.10(e)) and a Stanford Blue Bunny placed (Figure 2.10(g), 2.10(h) and 2.11(a)) in a Cornell room.
We now do a small comparison of the quality of the output of our new *Sphere-Ray Intersection Algorithm* with the output of the *Bresenham's Line Algorithm* being applied to the adaptive structure. We use the point models of Stanford Bunny, the Dragon and the Buddha placed in the cornell room for the same. Figure 2.11 shows the difference.

### 2.8.2 Quantitative Results

We note, in Table 2.1, that the number of visibility links (column 5) is a small fraction of the quadratic possibilities. (For example, the decrease in the empty Cornell room is the fraction (1.4 - 0.27)/1.4 (in millions), roughly $80\%$). This situation persists whether the scene is sparse, or dense.

### 2.8.3 Discussion

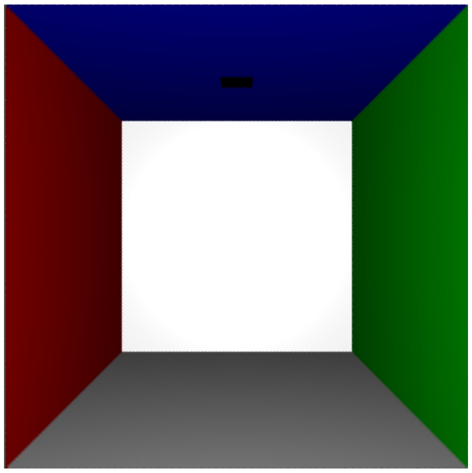
We then use the V-Map in our FMM-based GI algorithm. Figure 2.12 shows results where the subdivision of hierarchy is performed till 75 points per leaf. The figure shows the front view, the back view and the close up of the point models placed in the cornell room. The color bleeding effects and the soft shadows are clearly visible. Also notable is the back of the Ganpati and Goddess (See Fig 2.12(e) and Fig. 2.12(h)) being *lit*, due to global illumination, even though they are not directly visible to the light source.

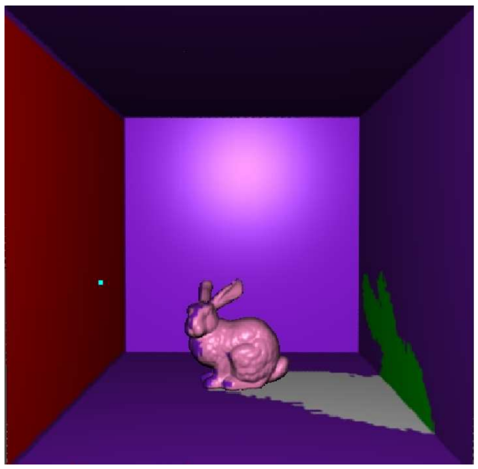
| Model | Points (millions) | $N^2$ possible links (millions) | V-Map Links (millions) | % Decrease | Memory(MB) $N^2$ links | Memory(MB) V-Map links | Build V-Map Time(mins) |
|---|---|---|---|---|---|---|---|
| ECR | 0.1 | 1.4 | 0.27 | 79.5% | 5.35 | 1.09 | 20.6 |
| PCR | 0.14 | 3.85 | 0.67 | 82.62% | 15.43 | 2.68 | 23.8 |
| BUN | 0.15 | 1.53 | 0.38 | 74.64% | 6.09 | 1.5 | 21.7 |
| DRA | 0.55 | 2.75 | 0.43 | 84.54% | 11.0 | 1.7 | 23.5 |
| BUD | 0.67 | 1.58 | 0.39 | 74.75% | 6.33 | 1.6 | 23.9 |
| GAN | 0.15 | 1.56 | 0.38 | 75.64% | 6.2 | 1.55 | 22.0 |
| GOD | 0.17 | 1.62 | 0.4 | 75.31% | 6.4 | 1.63 | 22.9 |

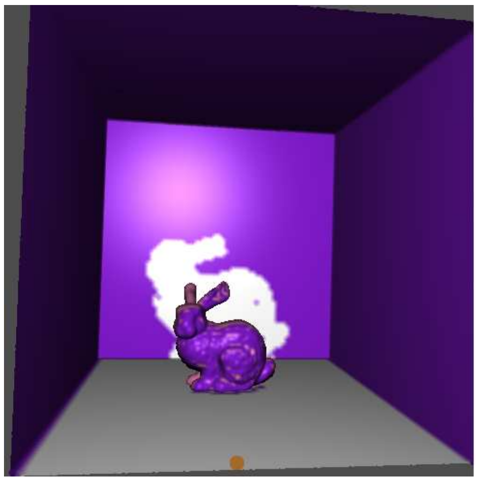
Table 2.1: V-Map details for sparse scenes such as the Empty Cornell Room(ECR), dense scenes such as the Cornell Room Packed(PCR) to capacity with a large box, or 'typical' scenes such as the Bunny(BUN), the Dragon(DRA), the Buddha(BUD), one of the Indian God's, Ganesha(GAN), and an Indian Goddess(GOD), placed in Cornell room. $N$ represents no. of leafcells. The reduced number of visibility links essentially signify less computations for GI radiosity algorithm. Maximum memory required for V-Map was 2.68MB for PCR, but was still very less compared to storage for $N^2$ links.

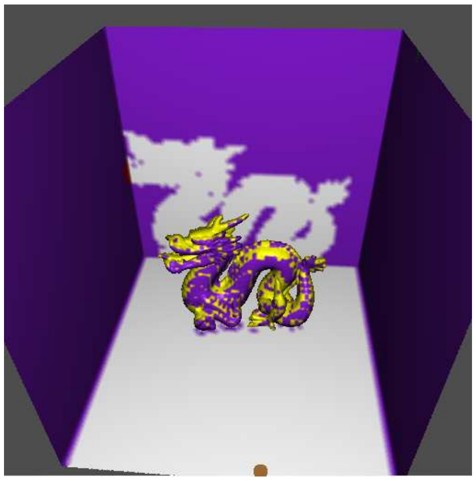
The models are not very detailed because of a pre-processing step of point-based simplification being applied on them. Also, some *roughness and discretization* appears due to low level of subdivision used (Octree is divided roughly till level 7). More sub-division can be done but the processing time increases quite a bit (*and hence we require a faster, parallel FMM radiosity kernel solver on GPU so as not to trade off quality for time*). The V-Map computation takes approximately $20 - 25\ minutes$ for about a *million points*, which can further be improvised when implemented in parallel (if analyzed properly, the visibility algorithm is *embarrassingly parallel*). Note that the *non-adaptive* version with *6 levels* in the octree took *more than 10 hours* for the Visibility Map computations.
The videos of the results are present at the following locations:

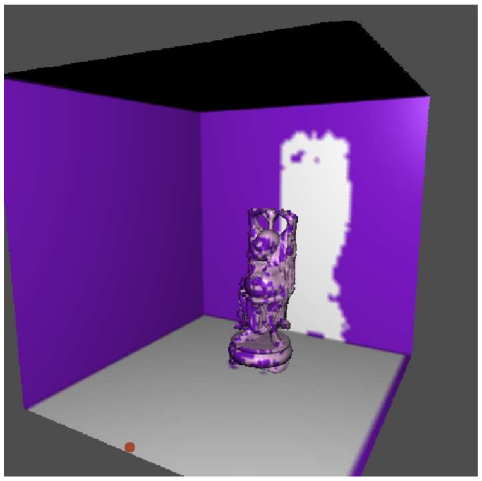
- http://trellis.cse.iitb.ac.in/ rhushabh/aps3/resultVideos/1.mpeg
- http://trellis.cse.iitb.ac.in/ rhushabh/aps3/resultVideos/2.mpeg
- http://trellis.cse.iitb.ac.in/ rhushabh/aps3/resultVideos/3.mpeg

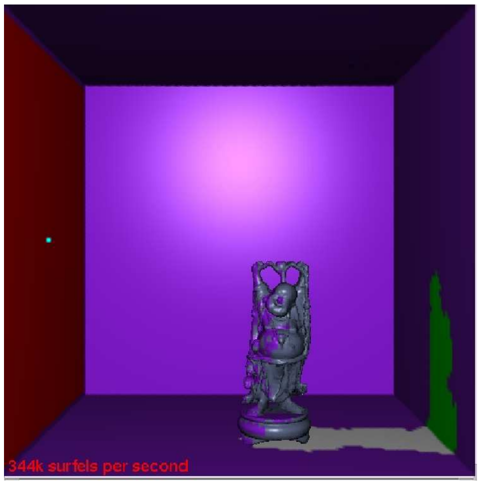

(a) An empty Cornell room.


(b) Visibility test with eye on the red wall.


(c) The Bunny (eye on the floor)

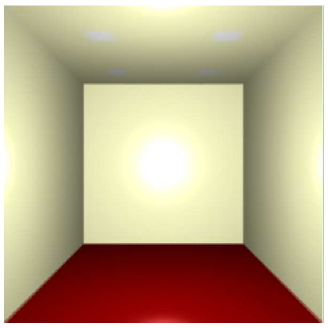

(d) The Dragon (eye on the floor)

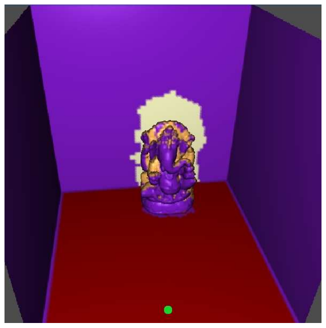

(e) The Buddha (eye on the floor)

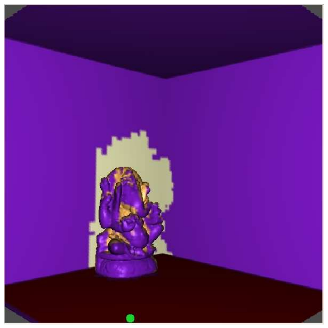


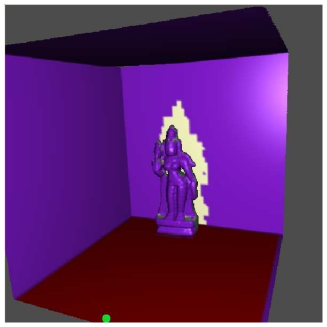
(f) The Buddha (eye on the red wall)

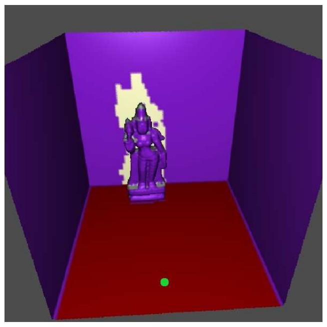
Figure 2.9: Various visibility tests where purple color indicates portions visible to the candidate eye (marked cyan/brown).

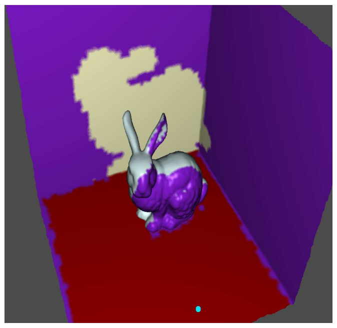

(a) An empty Cornell room.

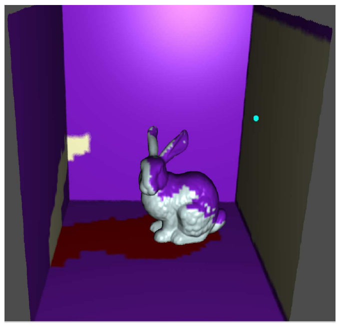

(b) Visibility test with eye on the red floor.


(c) Ganesha(viewed from a different eye position)


(d) The Goddess (eye on the floor)


(e) The Goddess(viewed from a different eye position)


(f) A Blue Bunny (eye on the floor)


(g) A Blue Bunny (eye on the right wall)

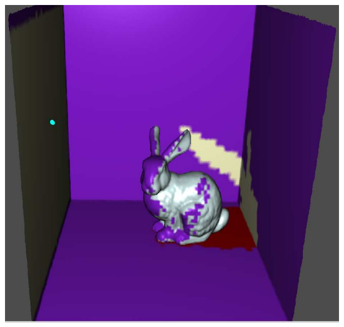
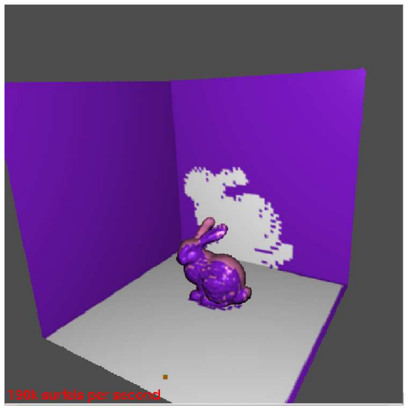
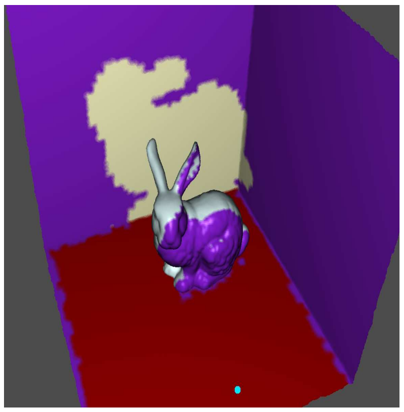
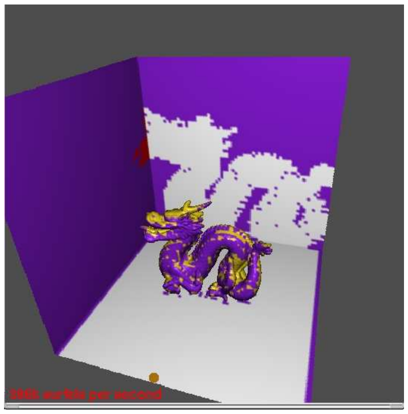
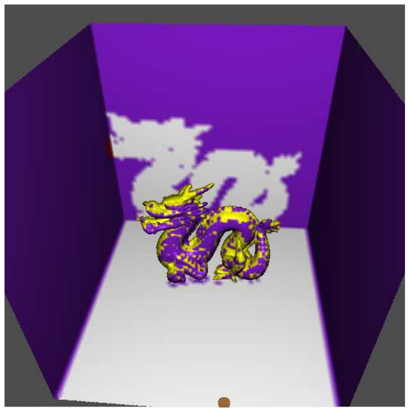

(h) A Blue Bunny (eye on the left wall)

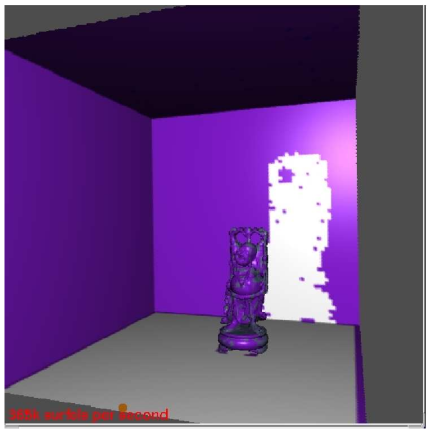
Figure 2.10: Various visibility tests where purple color indicates portions visible to the candidate eye (marked green/cyan).


(a) The Bunny (eye on the floor).


(b) A Blue Bunny (eye on the floor)


(c) The Dragon (eye on the floor)


(d) The Dragon (eye on the floor)


(e) The Buddha (eye on the floor)

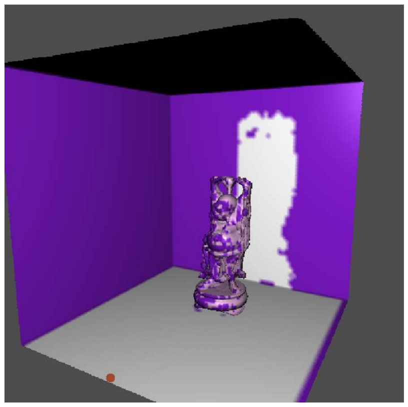

(f) The Buddha (eye on the floor)

Figure 2.11: Various visibility tests where purple color indicates portions visible to the candidate eye (marked cyan/brown). The left column shows the output of the *Bresenham's Line Algorithm* applied to the scene when divided in an adaptive octree structure. Note the artifacts which are clearly visible due to errors introduced by the step-length criteria of Bresenham's Algorithm. The step-length here was chosen to be 1 unit. Decrease in the step-length leads to drastic increase of running times *(in hours)*. The right column shows the output of the new *Sphere-Ray Intersection Algorithm* applied on similar models and similar input environment. The notable artifacts have reduced quite a lot.

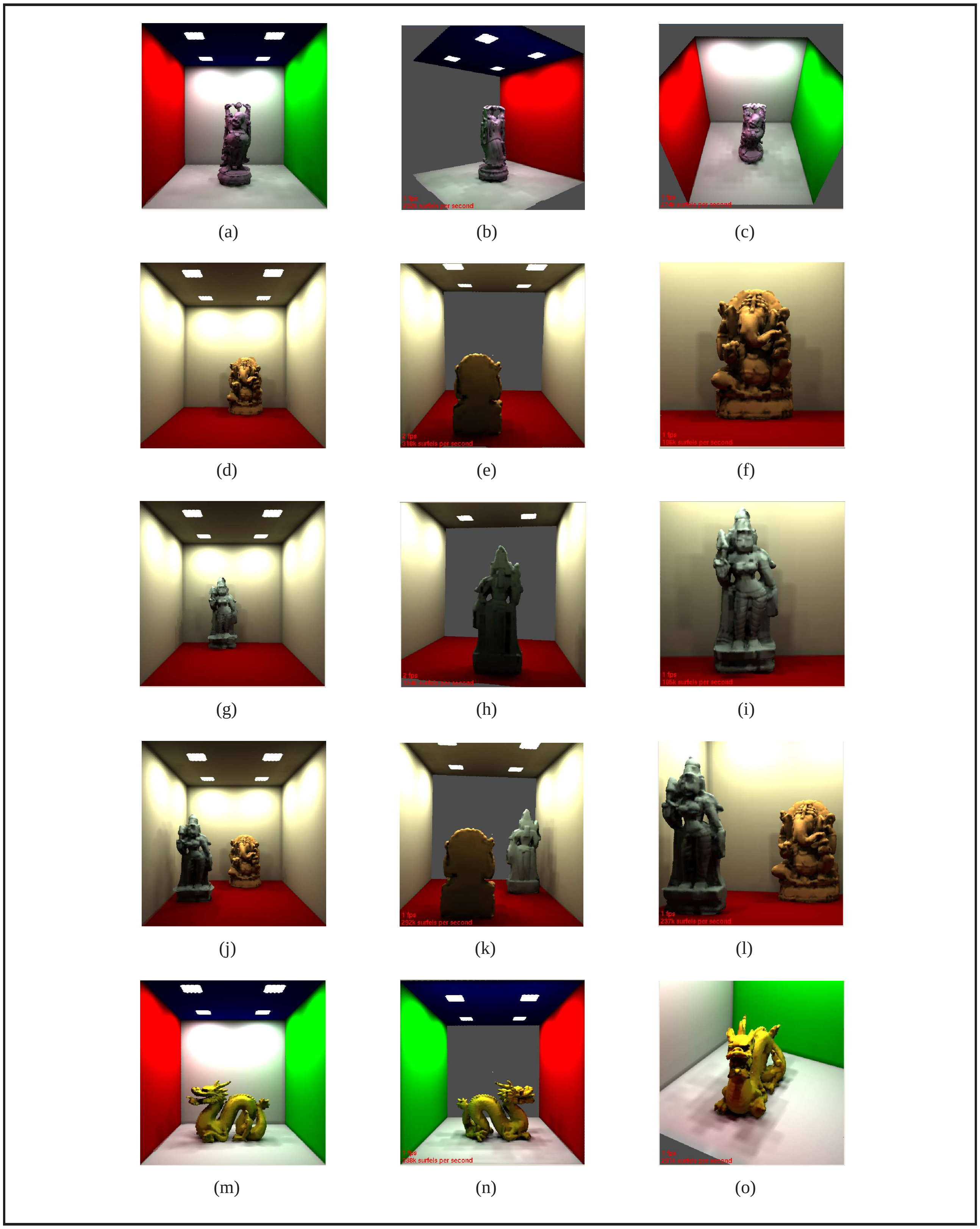


Figure 2.12: Use of V-Maps for GI effects. The hierarchy was constructed till we had roughly 75 points per leaf. The images rendered using a custom point-based renderer. Soft shadows, color bleeding and parts of models indirectly visible to the light source being *lit* can be observed. Five different set of images, corresponding to different point models, are shown. Each set shows a front view, a back view and a close up of the point model/s placed in the cornell room.

Chapter 3

# Discussion: Parallel FMM on GPU

## 3.1 Introduction

### 3.1.1 Fast computation with Fast Multipole Method

Computational science and engineering is replete with problems which require the evaluation of pairwise interactions in a large collection of particles. Direct evaluation of such interactions results in $O(N^2)$ complexity which places practical limits on the size of problems which can be considered. Techniques that attempt to overcome this limitation are labeled N-body methods. The N-body method is at the core of many computational problems, but simulations of celestial mechanics and coulombic interactions have motivated much of the research into these. Numerous efforts have aimed at reducing the computational complexity of the N-body method, particle-in-cell, particle-particle/particle-mesh being notable among these. The first numerically-defensible algorithm [DS00] that succeeded in reducing the N-body complexity to $O(N)$ was the Greengard-Rokhlin Fast Multipole Method (FMM) [GR87].

The FMM, in a broad sense, enables the product of restricted dense matrices with a vector to be evaluated in $O(N)$ or $O(N \log N)$ operations, when direct multiplication requires $O(N^2)$ operations. The Fast Multipole Method [GR87] is concerned with evaluating the effect of a "set of sources" $\mathbb{X}$, on a set of "evaluation points" $\mathbb{Y}$. More formally, given

$$\begin{aligned} \mathbb{X} &= \{x_1, x_2, \ldots, x_N\}, \quad x_i \in \mathbb{R}^3, \quad i = 1, \ldots, N, & (3.1) \\ \mathbb{Y} &= \{y_1, y_2, \ldots, x_M\}, \quad y_j \in \mathbb{R}^3, \quad j = 1, \ldots, M & (3.2) \end{aligned}$$

we wish to evaluate the sum

$$f(y_j) = \sum_{i=1}^{N} \phi(x_i, y_j), \quad j = 1, \ldots, M \tag{3.3}$$

The function $\phi$ which describes the interaction between two particles is called the "kernel" of the system (e.g. for electrostatic potential, kernel $\phi(x, y) = |x - y|^{-1}$). The function $f$ essentially sums up the contribution from each of the sources $x_i$.

Assuming that the evaluation of the kernel $\phi$ can be done in constant time, evaluation of $f$ at each of the $M$ evaluation points requires $N$ operations. The total complexity of this operation will therefore be $O(NM)$. The FMM attempts to reduce this seemingly irreducible complexity to $O(N+M)$ or even $O(N\log N+M)$. Three main insights that make this possible are:

1. **Factorization** of the kernel into source and receiver terms
2. Most application domains do not require that the function $f$ be calculated at very high accuracy.
3. FMM follows a **hierarchical structure** (*Octrees*)

Details on the theoretical foundations of FMM, requirements subject to which the FMM can be applied to a particular domain and discussion on the actual algorithm and its complexity as well as the mathematical apparatus required to apply the FMM to radiosity are available in [KC03] and [Gor06]. Five theorems with respect to the core radiosity equation are also proved in this context. *In our case, this highly efficient algorithm is used for solving the radiosity kernel and getting a diffuse global illumination solution.*
Besides being very efficient and applicable to a wide range of problem domains, the FMM is also highly parallel in structure. There are two versions of FMM: the *uniform* FMM works very well when the particles in the domain are uniformly distributed, while the *adaptive* FMM is used when the distribution is non-uniform. It is easy to parallelize the uniform FMM effectively: A simple, static domain decomposition works perfectly well. However, typical applications of FMM are to highly non-uniform domains, which require the adaptive algorithm. Obtaining effective parallel performance is considerably more complicated in this case, and no static decomposition of the problem works well. Moreover, certain fundamental characteristics of the FMM translate to difficult challenges for efficient parallelization. For eg. the FMM computation consists of a tree construction phase followed by a force computation phase. The data decomposition required for efficient tree construction may conflict with the data decomposition required for force computation. Most of the parallelizations employ the *octree-based* FMM computation, and thus inherit the distribution-dependent nature of the algorithm. Considerable research efforts have thus been directed at developing parallel implementations of the adaptive FMM.

However, our interest lies in design of a parallel FMM algorithm that is distribution independent and rigorously analyzable. We discuss such an algorithm in an architecture independent fashion, using only well understood and basic communication operations such as parallel prefix, all-to-all communication and sorting. It uses only a static data decomposition and does not require any explicit dynamic load balancing, either within an iteration or across iterations. The algorithm can be efficiently implemented on any model of parallel computation that admits an efficient sorting algorithm.

### 3.1.2 Parallel computations on GPU

The graphics processor (GPU) on today's video cards has evolved into an extremely powerful and flexible processor. The latest GPUs have undergoing a major transition, from supporting a few fixed algorithms to being fully programmable. High level languages have emerged for graphics hardware, making this computational power accessible. Architecturally, GPUs are highly parallel streaming processors optimized for vector operations, with both MIMD (vertex) and SIMD (pixel) pipelines. With the rapid improvements in the performance and programmability of GPUs, the idea of harnessing the power of GPUs for general-purpose computing has emerged. Problems, requiring heavy computations, like those dealing with huge arrays, can be transformed and mapped onto a GPU to get fast and efficient solutions. This field of research, termed as *General-purpose GPU (GPGPU) computing* has found its way into fields as diverse as databases and data mining, scientific image processing, signal processing etc.

Many specific algorithms like bitonic sorting, parallel prefix sum, matrix multiplication and transpose, parallel Mersenne Twister (random number generation) etc. have been efficiently implemented using the GPGPU framework. *One such algorithm which can harness the capabilities of the GPUs is parallel adaptive fast multipole method.*

Before moving onto the core parallel FMM algorithm, let us get acquainted with some of the data structures and algorithms used to get the same viz. parallel domain decomposition methods like space-filling curves and parallel compressed octrees. These are detailed w.r.t their distributed multi-processor system architecture implementation [HAS02]. We, at the same time, provide vital hints and start up points on how to implement them on GPUs.

## 3.2 Spatial Locality Based Parallel Domain Decomposition

In the context of parallel scientific computing, the term *domain decomposition* is used to refer to the process of partitioning the underlying domain of the problem across processors in a manner that attempts to balance the work performed by each processor while minimizing the number and sizes of communications between them, the reason being communication is significantly slower than computation. In fact, it tries to overlap computation and communication for even better performance. Achieving load balance while simultaneously minimizing communication is quite challenging as the input data need not necessarily be uniformly distributed. Spatial locality based domain decomposition methods are best suited for particle-based methods (eg. Gravitational $N$-body problem) as particles interact with other particles based on their spatial locality.

In this section we present a brief overview of some of the most widely used spatial locality based parallel domain decomposition methods such as space filling curves (SFCs) and parallel octrees.

### 3.2.1 Space Filling Curves

Consider a $d$ dimensional hypercube. Say we bisect this hypercube $k$ times recursively along each dimension results in a $d$ dimensional matrix of $2^k \times 2^k \times \cdots \times 2^k = 2^{dk}$ non-overlapping hypercells of equal size. A Space Filling Curve (SFC) is a mapping of these hypercells, the location of each of which in the cell space is given by $d$ coordinates, to a one dimensional linear ordering. An example of linearization of two dimensional data is shown in Fig. 3.1(a) for $k$ = 1 and 2.

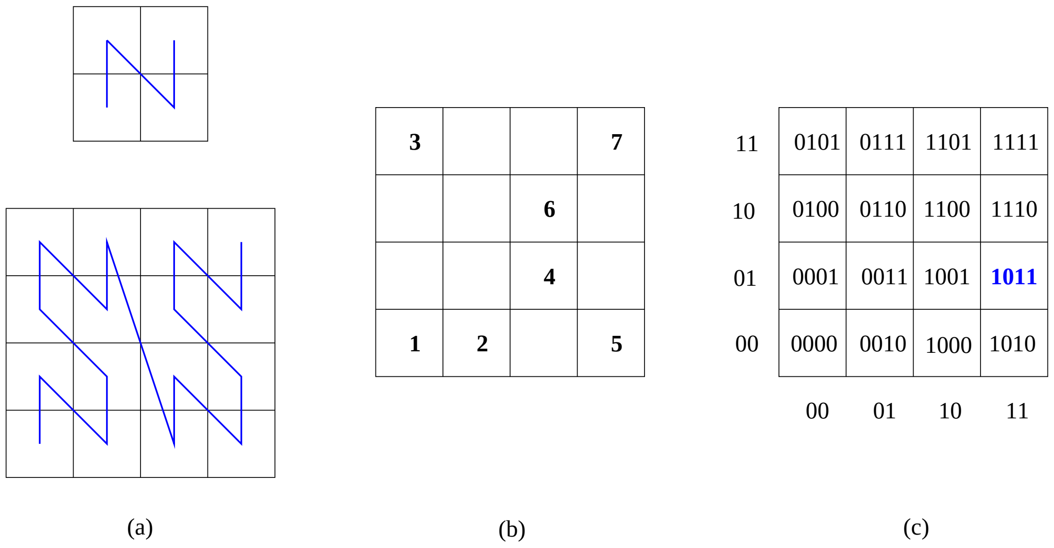


Figure 3.1: (a) The z-curve for $k$ = 1 and 2. (b) A $8 \times 8$ decomposition of two dimensional space containing 7 points. The points are labeled in the order in which the cells containing them are visited using a $Z$-SFC. (c) Bit interleaving scheme for generating index of a cell as per $Z$-SFC.

A sample ordering (of non-empty cells) is shown in Fig. 3.1(b). The resulting one dimensional ordering is divided into $p$ equal partitions which are assigned to processors. However, the runtime to order $2^{kd}$ cells, $\Theta(2^{kd})$, is expensive because typically $n \ll 2^{kd}$. Thus, a fast and efficient method to directly order the cells containing the points is required.

#### 3.2.1.1 SFC Construction

Let $D$ be the side length of a domain whose corner is at the origin of a $d$-dimensional coordinate system. The first step in SFC linearization is to find the coordinates of the cells containing each of the input points. Given a point in $d$ dimensional space with coordinate $x_i$ along dimension $i$, the integer coordinate along dimension $i$ of the cell containing the point is given by $\lfloor \frac{2^k x_i}{D} \rfloor$. The index of a cell in $Z$-SFC, also known as Morton Ordering is computed by representing the integer coordinates of the cell using $k$ bits and then interleaving the bits starting from the first dimension to form a $dk$ bit integer. In Fig. 3.1(c), the index of the cell with coordinates $(3, 1) = (11, 01)$ is given by $1011 = 11$. Thus, the index of the cell containing a given point can be computed in $O(kd)$ time in $d$ dimensions, or $O(k)$ time in three dimensions. Once the indices corresponding to all the points are generated, SFC decomposition is achieved by a parallel integer sort.

**Algorithm 10** SFC Linearization

```
procedure SFCLinearization
1: Choose a resolution $k$.
2: For each point, compute the index of the cell containing the point.
3: Parallel sort the resulting set of integer keys.
```

Partitioning the SFC linearization of the points equally to processors ensures load balancing. Because of its ease of repartitioning, SFCs have been widely used as a tool for domain decomposition in parallel scientific computing. In the first step SFC is used to preprocess the input data such that the computational domain is partitioned across multiple processors. In the second step, other data structures, usually tree based, are subsequently built locally on each processor on the part of the domain that it is mapped to. The numerical computations that drive the scientific application are then carried out using these latter data structures. The next section describes a more sophisticated approach of using a data structure called octree for both parallel domain decomposition and subsequent numerical computations.

### 3.2.2 Parallel Compressed Octrees

Octrees are hierarchical tree data structures that organize multidimensional points using a recursive decomposition of the space containing them. Such a tree is called a *quadtree* in two dimensions and *octree* in three dimensions. For purpose of illustrations, we will be using quadtrees in this report.

#### 3.2.2.1 Octree Construction

Consider a hypercube enclosing $n$ multidimensional points. The domain enclosing all the points forms the root of the octree. This is subdivided into $2^d$ subregions of equal size by bisecting along each dimension. Each of these regions that contain at least one point is represented as a child of the root node. The same procedure is recursively applied to each child of the root node terminating when a subregion contains at most one point (or some pre-defined number of points). An example is shown in Fig. 3.2.

#### 3.2.2.2 Compressed Octrees

In octrees, the manner in which any subregion is bisected is independent of the specific location of the points within it. Thus chains may form when many points lie within a small volume of space (eg. Fig. 3.2). These chains do not contain any extra information. As such, no information is lost if each of the chains are compressed into a single node resulting in a *compressed octree*. Note that each node in a compressed octree is either a leaf

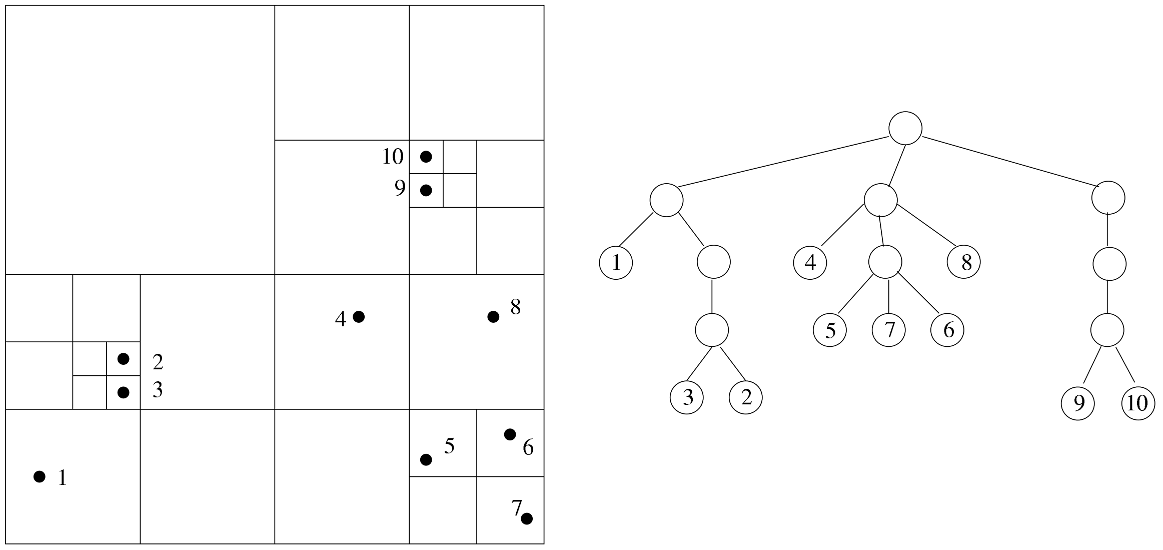


Figure 3.2: A quadtree built on a set of 10 points in 2-D.

or has at least two children. This ensures that the size of the resulting compressed octree is $O(n)$ and is independent of the spatial distribution of the points. The compressed octree corresponding to the octree in Fig. 3.2 is shown in Fig. 3.3.

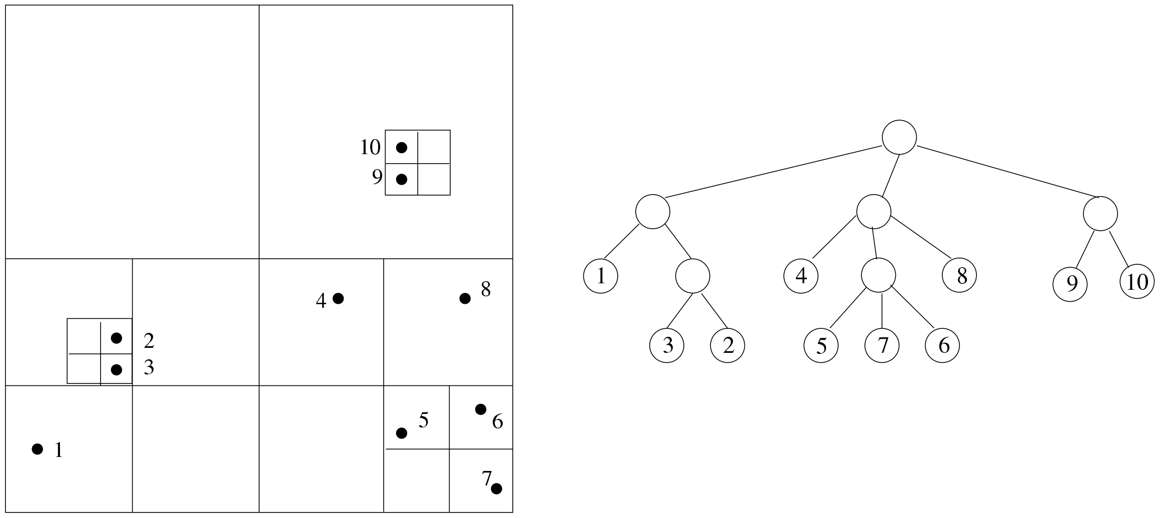


Figure 3.3: A compressed quadtree corresponding to the quadtree of Fig. 3.2

However, such compressed nodes should still encapsulate the fact that it represents multiple regions of space unlike the nodes that are not compressed. To achieve this, two cells are stored in each node $v$ of a compressed octree, *large cell of v* and *small cell of v*, denoted by $L(v)$ and $S(v)$, respectively. The large cell is defined as the largest cell that encloses all the points the node represents. Likewise, the small cell is the smallest cell that encloses all the points that the node represents. If a node is not a result of compression of a chain, then the large cell and the small cell of that node are the same; otherwise, they are different. Observe that the large cell of a node is an immediate subcell of the small cell of its parent. Since a leaf contains a single point, its small cell is defined to be the hypothetical cell with zero length containing the point. Or, when the maximum resolution is specified, the small cell of a point is defined to be the cell at the highest resolution containing the point.

#### 3.2.2.3 Octrees and SFCs

Octrees can be viewed as multiple SFCs at various resolutions. To define a linearization that cuts across multiple levels, we use the fact that given any two cells, they are either disjoint or one is contained in the other. Thus given two cells, if one is contained in the other, the subcell is taken to precede the supercell; if they are disjoint, they are ordered according to the order of the immediate subcells of the smallest supercell enclosing them. A nice property that follows from these rules is the resulting linearization of all cells in an octree (or compressed octree) is identical to *its postorder traversal*.

However, ambiguity may arise when distinguishing indices of cells at different levels of resolution. For example, in Fig. 3.4 it is not possible to distinguish between 00 (cell of length $D/2$ with coordinates (0,0)), and 0000 (cell of length $D/4$ with coordinates (00,00)), when both are stored in, say, standard 32-bit integer variables. A simple mechanism to overcome this is to prepend the bit representation of an index with a '1' bit. With this, the root cell is 1, the cells with $Z$-SFC indices 00 and 0000 are now 100 and 10000, respectively.

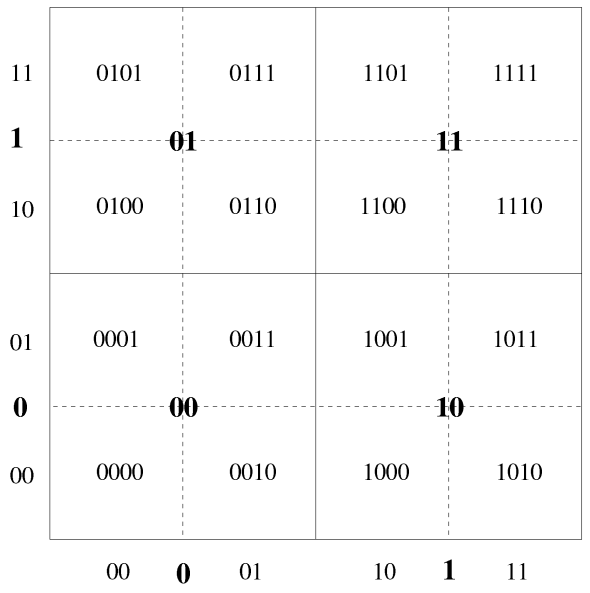


Figure 3.4: Bit interleaving scheme for a hierarchy of cells.

The process of assigning indices to cells can also be viewed hierarchically. A cell at resolution $i$ can be described using $i$-bit integer coordinates in each dimension. The first $i-1$ of these bits are the same as the coordinates of its immediate supercell. Thus, the index of a cell can be obtained by taking the least significant bit of each of its coordinates, concatenating them into a $d$-bit string, and appending this to the index of its immediate supercell.

#### 3.2.2.4 Parallel Compressed Octree Construction

The algorithm to construct parallel compressed octrees is given below.

**Algorithm 11** Parallel Compressed Octree

```
procedure ConstructParallelCompressedOctree()
1: For each point, compute the index of the leaf cell containing it.
2: Parallel sort the leaf indices to compute their SFC linearization.
3: Each processor obtains the leftmost leaf cell of the next processor.
4: On each processor, construct a local compressed octree for the leaf cells within it and the borrowed leaf
   cell.
5: Send the out of order nodes to appropriate processors.
6: Insert the received out of order nodes in the already existing sorted order of nodes.
```

After SFC linearization of leaf indices, we now generate parts of the compressed octree on each processor, such that each node and edge of the *global compressed tree* is generated on some processor. To do this, each processor first borrows the leftmost leaf cell from the next processor. This is because if we generate the lowest common ancestor of every consecutive pair of leaf nodes, we are guaranteed to generate every internal node in the compressed octree. It then runs a sequential algorithm to construct the compressed octree for its leaf cells together with the borrowed leaf cell as follows: Initially, the tree has a single node which represents the first leaf cell in SFC-order. The remaining leaf cells are inserted one at a time as per the SFC-order. If $C$ is the next leaf cell to be inserted, then starting from the most recently inserted leaf, walk up the path toward the root until the first node $v$ such that $C \subseteq L(v)$ is encountered. Now, two possibilities arise:

- **Case I:** If $C$ is not contained in $S(v)$, then $C$ is in the region $L(v)S(v)$, which was empty previously. The smallest cell containing $C$ and $S(v)$ is a subcell of $L(v)$ and contains $C$ and $S(v)$ in different immediate subcells. Create a new node $u$ between $v$ and its parent and insert a new child of $u$ with $C$ as small cell.

- **Case II:** If $C$ is contained in $S(v)$, $v$ is not a leaf node. The compressed octree presently does not contain a node that corresponds to the immediate subcell of $S(v)$ that contains $C$, i.e., this immediate subcell does not contain any of the points previously inserted. Therefore, it is enough to insert $C$ as a child of $v$ corresponding to this subcell.

The local tree is stored in its postorder traversal order in an array. For each node, indices of its parent and children in the array are stored.

Now we need to generate the postorder traversal order of the global compressed octree. To do so, nodes which should actually appear later in the postorder traversal of the global tree (out of order nodes), should be sent to the appropriate processors, which appear consecutively after the borrowed leaf in the postorder traversal of the local tree. The first leaf cell in each processor is gathered into an array of size $p$. Using a binary search in this array, the destination processor for each out of order node can be found. Nodes that should be routed to the same processor are collected together and sent using an all-to-all communication.
The received nodes are merged with the local postorder traversal array and their positions are communicated back to the sending processors. The net result is the postorder traversal of the global octree distributed across processors. Each node contains the position of its parent and each of its children in this array.

### 3.2.3 Spatial Domain Decomposition Methods on GPU

Here, we provide some hints about how to implement *spatial domain decomposition* methods on GPU. As we know, GPU's architecture can be considered similar to shared memory multi-processor system architecture, except for the fact that the GPU's memory is specifically designed to store color and texture information.

#### 3.2.3.1 Space Filling Curves on GPU

Algorithm 3.2.1.1 defines the procedure to construct SFC in parallel. No major changes will be required to Algorithm 3.2.1.1 to be implemented on GPUs, expect for the way the information will be stored in memory. The procedure to compute the indices will take as input the location information of points and the cells to which they belong. Cell locations can be stored as an indirection grid in texture memory whereas the points will be stored as an 1-D array. The *RGBA* value of the cell will index the location of the point (in texture memory) contained in it, as shown in Fig 3.5.
The procedure will return the index values computed. Further, to sort these values in parallel, we have an efficient implementation of *Bitonic Sort* algorithm on GPU.

#### 3.2.3.2 Bitonic Sort

A sorting network is a sorting algorithm, where the sequence of comparisons is not data-dependent. That makes them suitable for parallel implementations. Bitonic sort is one of the fastest sorting networks designed for parallel machines. A bitonic sequence is composed of two subsequences, one monotonically non-decreasing and the other monotonically non-increasing. Moreover, any rotation of a bitonic sequence is a bitonic sequence. Of course, a sorted sequence is itself a bitonic sequence: one of the sub-sequences is empty. Suppose we have a bitonic sequence of length $2n$, that is, elements in positions $[0, 2n)$. We can easily divide it into two halves,

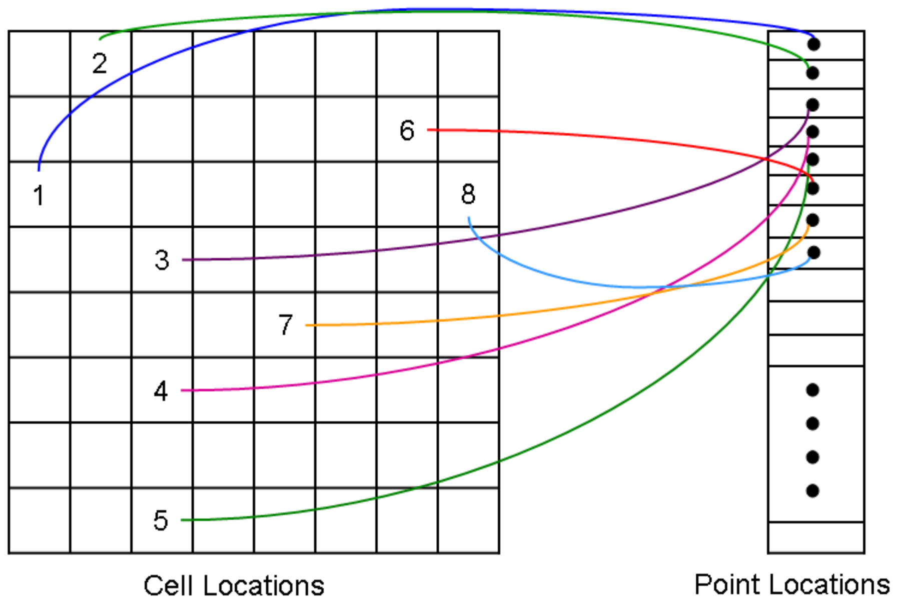


Figure 3.5: Cell locations stored as an indirection grid in texture memory and the points as an 1-D array. The *RGBA* value of the cell will index the location of the point (in texture memory) contained in it.

$[0, n)$ and $[n, 2n)$, such that each half is a bitonic sequence and every element in half $[0, n)$ is less than or equal to ( Or greater than or equal to) each element in $[n, 2n)$.

Simple comparison of elements in the corresponding positions in the two halves and exchanging them if they are out of order achieves this. This is called *bitonic merge*.

**Algorithm 12** Recursive Bitonic Sort

```
procedure BitonicSort
1: Sort only sequences a power of two in length, so a subsequence of more than one element can always be
   divided into two halves.
2: Sort the lower half into ascending order and the upper half into descending order to get a bitonic sequence.
3: Perform a bitonic merge on the sequence, which gives a bitonic sequence in each half and all the larger
   elements in the upper half.
4: Recursively bitonically merge each half until all the elements are sorted.
```

Let us first have a look at recursive bitonic sort shown in Algorithm 3.2.3.2. It uses methods sortup, sortdown, mergeup and mergedown to sort into ascending (descending) order and to recursively merge into ascending (descending) order.

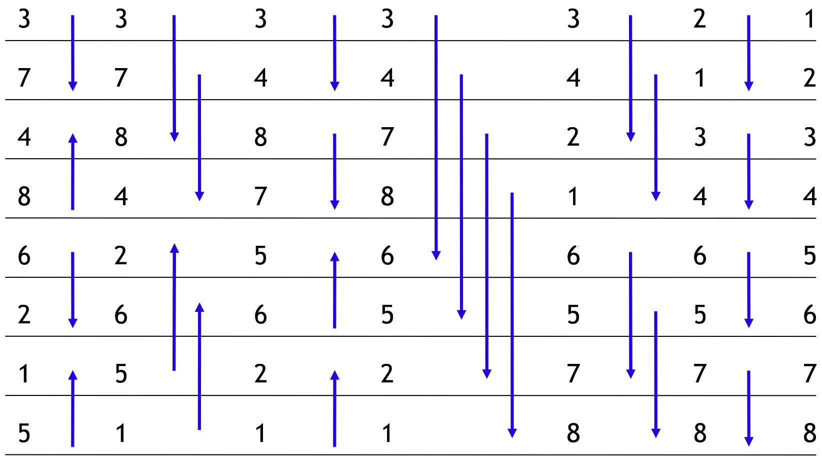


Figure 3.6: A simple parallel Bitonic Merge Sort of eight elements requires six passes. Elements at the head and tail of each arrow are compared, with larger elements moving to the head of the arrow. The nal sorted sequence is achieved in $O(\log_2 n)$ passes.

```
void sortup (int m, int n)          void sortdown (int m, int n)
{ //from m to m+n                   { //from m to m+n
  if (n == 1) return;                 if (n == 1) return;
  sortup (m, n/2);                    sortup (m, n/2);
  sortdown (m+n/2, n/2);              sortdown (m+n/2, n/2);
  mergeup (m, n/2);                   mergedown (m, n/2);
}                                   }


void mergeup (int m, int n)         void mergedown (int m, int n)
{                                   {
  if (n == 0) return;                 if (n == 0) return;
  for (int i=0; i<n; i++)             for (int i=0; i<n; i++)
  {                                   {
    if (x[m+i] > x[m+i+n])              if (x[m+i] < x[m+i+n])
      swap (m+i, m+i+n);                  swap (m+i, m+i+n);
  }                                   }
  mergeup (m, n/2);                   mergedown (m, n/2);
  mergeup (m+n, n/2);                 mergedown (m+n, n/2);
}                                   }
```

Method $sortup(\text{int}\, m, \text{int}\, n)$ sorts the $n$ elements in the range $[m, m + n)$ into ascending order. It uses method $mergeup(\text{int}\, m, \text{int}\, n)$ to merge the $n$ elements in the subsequence $[m, m + n)$ into ascending order. Methods mergeup and mergedown compare elements in the two halves, exchange them if they are out of order, and recursively merge the two halves. Similarly for $sortdown(\text{int}\, m, \text{int}\, n)$ and $mergedown(\text{int}\, m, \text{int}\, n)$.

The overall sort is performed by calling $sortup(0,\ N)$. Both sortup and sortdown recursively sort each half to produce an A-frame shape and then recursively merge that into an ascending or descending sequence.

If we look at the algorithm we see that recursive calls of merge can be done in parallel. The loops in the merges, comparing and conditionally exchanging elements $(m+i)$ and $(m+i+n)$ can also be run in parallel. It allows an array of $n$ processors to sort $n$ elements in $O(\log^2 n)$ steps[Chr].

### 3.2.4 Parallel Compressed Octrees on GPU

Algorithm 3.2.2.4 gives us an idea of implementing parallel compressed octrees on distributed multiprocessor architecture system.
The first two steps of the algorithm can be implemented *on GPU* in a fashion similar to the one discussed in subsection 3.2.3. Further, the process of assigning indices to the cell as well as locating cells from its supercells can be done using an efficient parallel implementation of *Prefix Sum* algorithm. Subsection 3.2.4.1 below gives an efficient implementation of the parallel Prefix Sum algorithm on GPU.

#### 3.2.4.1 Prefix Sum on GPU

The prefix-sum problem takes an array of numbers as input and outputs an array with partial sums. It is called prefix-sum because it computes sums over all prefixes of the array. For example, if one is to put into an array one's initial checkbook balance, followed by the amounts of the check one has written as negative numbers and deposits as positive numbers, then computing the partial sums produces all the intermediate and final balances.

#### 3.2.4.2 Parallel Algorithm

Prefix-sum problem is inherently sequential, but, given a number of processors, it can be parallelized efficiently to finish computation in $O(\log n)$ time. One such algorithm [Har] is presented below.

**Algorithm 13** Calculate Prefix sum of an array $x$ with $n$ numbers

```
1: for $d \leftarrow 1$ to $\log_2 n$ do
2:     for all $k$ in parallel do
3:         if $k \geq 2^d$ then
4:             $x[k] \leftarrow x[k - 2^{d-1}] + x[k]$
5:         else
6:             $x[k] \leftarrow x[k]$
7:         end if
8:     end for
9: end for
```

Fig. 3.7 shows the calculation of prefix sum on an example array of numbers, using this algorithm. The cells whose indices are not greater than $2^{d-1}$, where $d$ is the iteration number, are directly copied. The input array is stored in a two-dimensional texture. Each fragment uses its screen-space position to index into this texture. It sums the value at its position and $2^{d-1}$ position to the left. This is written to a separate output texture which is used as input to the next pass.

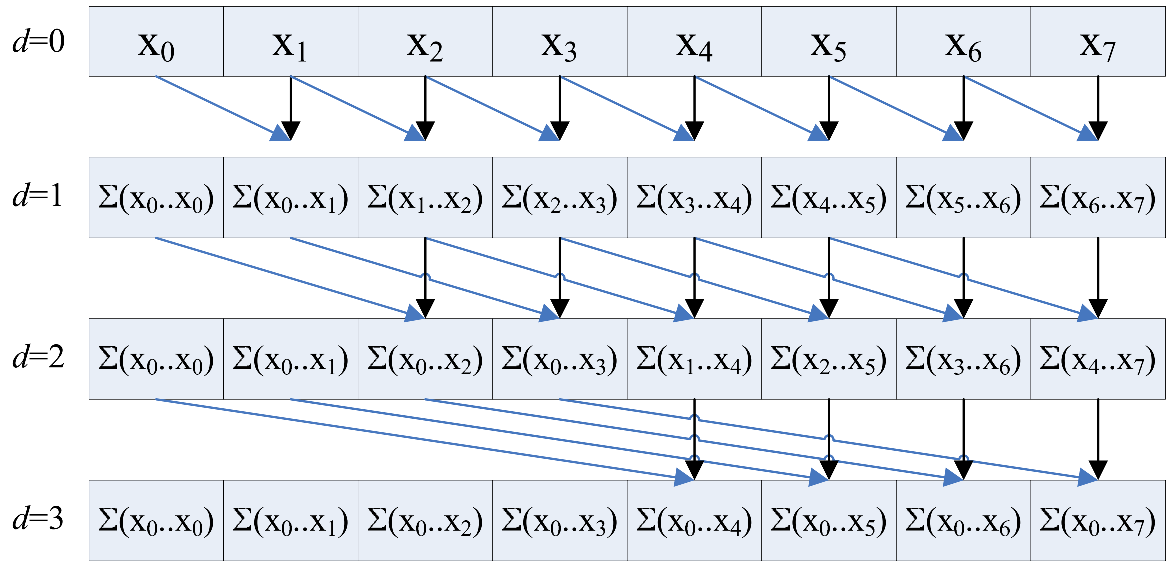


Figure 3.7: Computing the scan of an array of 8 elements

We notice that in any iteration $d$, only $n/2^d$ fragments are doing useful work. However, the Algorithm 13 does more computation in one pass, enabling it to finish the prefix-sum computation in $O(\log n)$ time, compared to $O(n)$ time taken by the sequential algorithm to compute the same result.

We also note that the $O(\log n)$ computation time is true only when there are $n$ or more processors which can compute in parallel in which case the execution time is dominated by step complexity rather than work complexity. However, the fragment pipeline can only execute a fixed maximum number of fragments in parallel. We call this limit the degree of parallelism. If we begin with an array which is greater than this limit, the fragment pipeline would have to break up the fragments into batches, which it executes sequentially.

Now to construct a local compressed octree for each processor, we first look at how simple octrees are constructed on GPU. The following subsection 3.2.4.3 gives one such implementation, where octrees are constructed as textures on GPU.

#### 3.2.4.3 Octree Textures on GPUs

A simple way to implement an octree on a CPU is to use pointers to link the tree nodes together. Each internal node contains an array of pointers to its children. A child can be another internal node or a leaf. A leaf only contains a data field. To implement a hierarchical tree on a GPU we need to define how to store the structure in texture memory and how to access the structure from a fragment program. In the GPU implementation pointers simply become indices within a texture. They are encoded as RGB values. The content of the leaves is directly stored as an RGB value within the parent node's array of pointers. Alpha channel is used to distinguish between a pointer to a child and the content of a leaf (alpha = 1 indicates data, alpha = 0.5 indicates index and alpha = 0 indicates empty cell). For simplicity, *quadtree* which is a 2D equivalent of an octree is discussed. Figure 3.8 shows the octree storage.

Let us first define the following terminology:

- **Indirection pool**: An 8-bit RGBA 3D texture in which the tree is stored.
- **Cell**: Each 'pixel' of the indirection pool.
- **Indirection grid**: The indirection pool is subdivided into indirection grids. An indirection grid has $2^d$ cells where $d$ is the dimension. Each node of the tree is represented by an indirection grid. It corresponds to the array of pointers of the CPU implementation described above. A cell of an indirection grid can be empty or contain either (a) data if the corresponding child in the tree is a leaf, or (b) the index of an indirection grid if the corresponding child is another internal node.

Now the tree is stored in the texture memory and we want to retrieve the value stored in the tree at a point $M \in [0,1] \times [0,1]$. Let $I_D = (I_{D_x}, I_{D_y})$ be the index of the indirection grid of the node visited at depth $D$. Let us also assign the root node $I_0$ to be $(0,0)$. The tree lookup starts from the root and successively visits the nodes containing the point $M$ until a leaf is reached. To do so, at level $D$ we need to read from the indirection grid $I_D$ the value stored at the location corresponding to $M$ which in turn requires the computation of the coordinates of $M$ *within* the node.

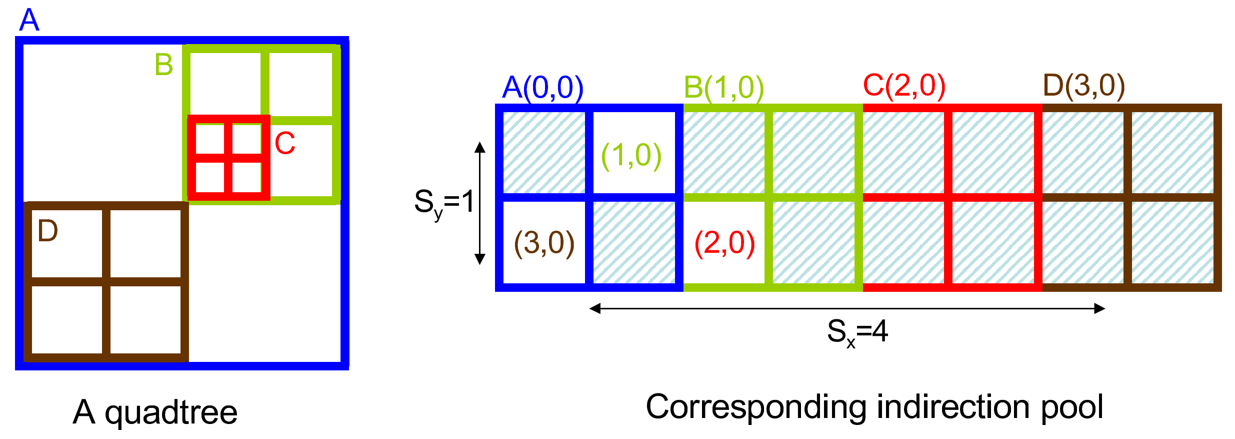
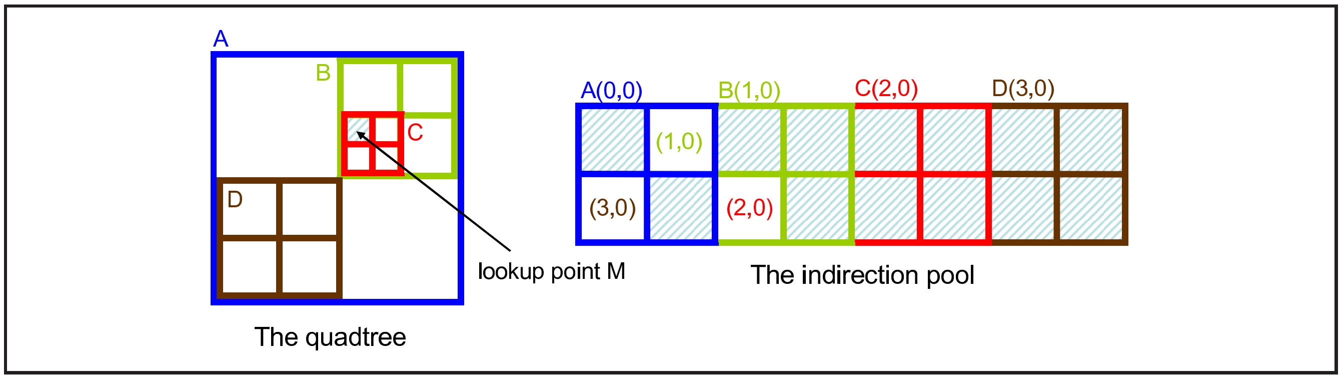


Figure 3.8: Storage in texture memory. The indirection pool encodes the tree. Indirection grids are drawn with different colors. The grey cells contain data.

At depth $D$ a complete tree produces a regular grid of resolution $2^D \times 2^D$ within the unit square (cube in 3D). Each node of the tree at depth $D$ corresponds to a cell of this grid. In particular $M$ is within the cell corresponding to the node visited at depth $D$. The coordinates of $M$ *within* this cell are given by $frac(M \cdot 2^D)$. These coordinates are used to read the value from the indirection grid $I_D$. Thus, the lookup coordinates within the indirection pool are thus computed as $P = (P_x, P_y)$ where

$$P_x = \frac{I_{D_x} + frac(M \cdot 2^D)}{S_x},\ P_y = \frac{I_{D_y} + frac(M \cdot 2^D)}{S_y}$$

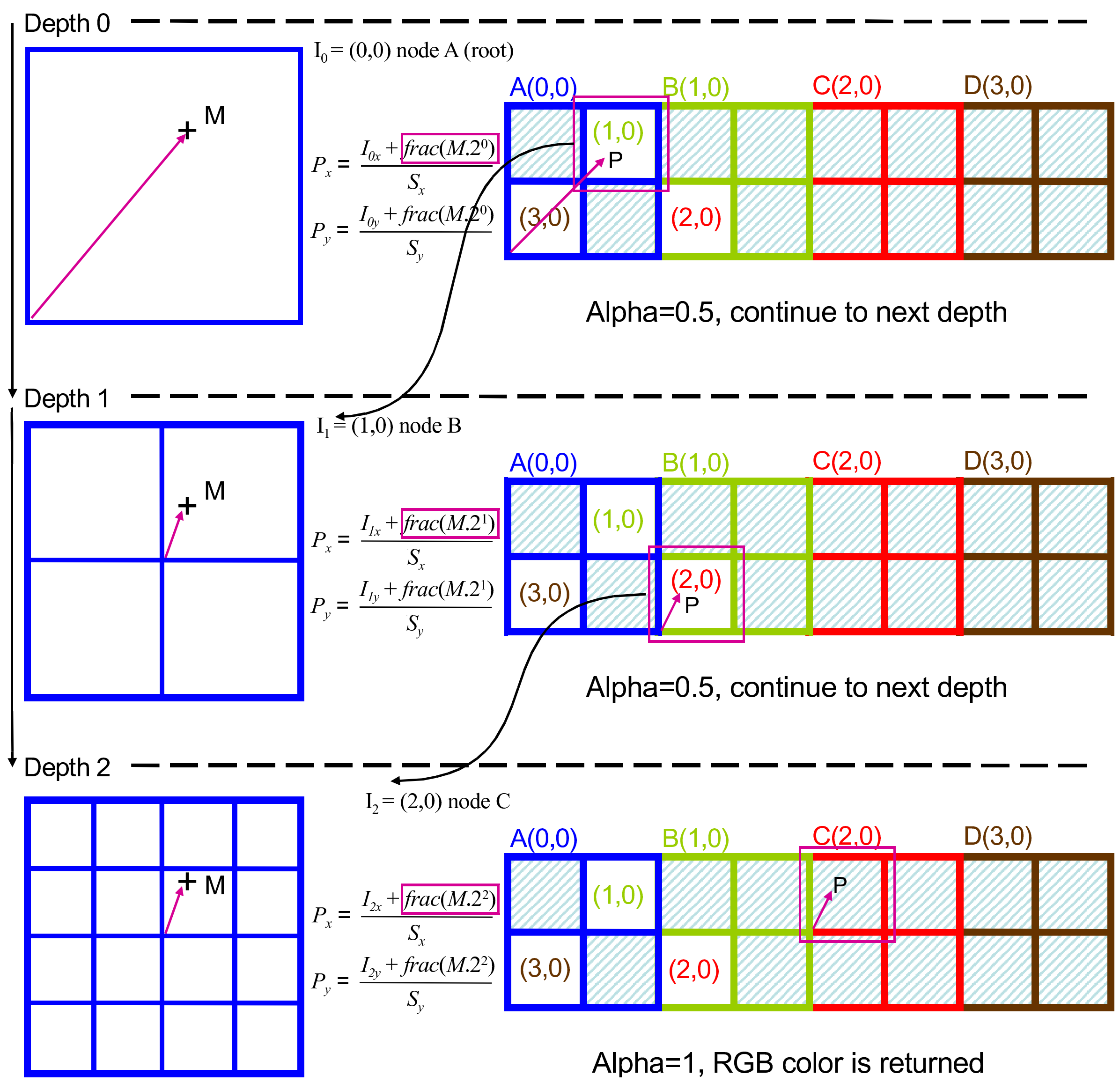
Here $S_x$ and $S_y$ denote the number of indirection grids along each *row* and *column* of the indirection pool respectively. The RGBA value stored at $P$ in the indirection pool is then retrieved. Depending on the alpha value, we will either return the RGB color if the child is a leaf, or we will interpret the RGB values as the index of the child's indirection grid ($I_{D+1}$) and continue to the next tree depth. Figure 3.9 summarizes this entire process.



Having seen how to implement octrees on GPU, we now need to answer the following queries for construction of compressed octrees on GPU.

- Apart from *point lookup* operation, what other operations need to be performed on compressed octrees ?
- How do we modify the octree implementation on GPU to store multiple data corresponding to each leaf-cell ?
- How do we modify the octree implementation on GPU (specifically the memory model) to store information about large-cell and small-cell of every node, along with the indices to its children ?
- How to access the last leaf node belonging to another processor, which is required for construction of local compressed octree ?
- Being a shared memory system, can we ignore the *all-to-all* communication step required previously ?
- How to identify and take care of *out-of-order* nodes ?
- How to handle multiple access by processors to same memory location simultaneously ?

Lookup at point M:



Depth 0
$I_0 = (0,0)$ node A (root)
M
$P_x = \frac{I_{0x} + \boxed{frac(M.2^0)}}{S_x}$
$P_y = \frac{I_{0y} + frac(M.2^0)}{S_y}$
A(0,0)
B(1,0)
C(2,0)
D(3,0)
(1,0)
P
(3,0)
(2,0)
Alpha=0.5, continue to next depth
Depth 1
$I_1 = (1,0)$ node B
M
$P_x = \frac{I_{1x} + \boxed{frac(M.2^1)}}{S_x}$
$P_y = \frac{I_{1y} + frac(M.2^1)}{S_y}$
A(0,0)
B(1,0)
C(2,0)
D(3,0)
(1,0)
(3,0)
(2,0)
P
Alpha=0.5, continue to next depth
Depth 2
$I_2 = (2,0)$ node C
M
$P_x = \frac{I_{2x} + \boxed{frac(M.2^2)}}{S_x}$
$P_y = \frac{I_{2y} + frac(M.2^2)}{S_y}$
A(0,0)
B(1,0)
C(2,0)
D(3,0)
P
(1,0)
(3,0)
(2,0)
Alpha=1, RGB color is returned

Figure 3.9: At each step the value stored within the current node's indirection grid is retrieved. If this value encodes an index, the lookup continues to the next depth. Otherwise, the value is returned.

Answer to these *hints/queries* forms the start point for a parallel compressed octree construction on GPU.

## 3.3 Parallel FMM Algorithm

Having looked at some basic *Spatial Domain Decomposition* methods and their algorithms and implementation overviews on both distributed multi-processor systems and GPUs, we now move further and review one particular implementation of FMM on parallel, distributed, multi-processor system [HAS02].

The FMM computation consists of the following phases: a) Building the compressed octree, b) computing multipole expansions using a bottom-up traversal, c) computing translations for each cell using its interaction list, d) computing the local expansions using a top-down traversal. All these phases afford substantial parallelism which is exploited within each phase.

### 3.3.1 Constructing Parallel Compressed Octree

A parallel compressed octree is built as described in Sec. 3.2.2.4. The tree construction requires an insignificant amount of the total run-time. However, this stage is very important since tree partitioning is key determinant of the load balancing and communication efficiency of the subsequent stages.

### 3.3.2 Near Field Computations

The computation of the nearfield for leaf cells associated with a processor requires information from the adjacent leaf cells. Therefore, each processor maintains an array of size $2p$ by gathering the integer keys of the first and last leaf nodes of every processor. Binary search in this array can be used to determine the processor responsible for a leaf cell. Now using an all-to-all communication each processor sends information of its leaf cells to processors that should contain leaf cells adjacent to it.

### 3.3.3 Building Interaction Lists

For a cell in a compressed octree, its parent box, corresponding to the cell of the parent node in the tree, need not necessarily be a cell of double the side length of that of the child cell and may reside on some other processor. To compute the parent of a cell we find the smallest cell containing it and its adjacent cell in the postorder traversal.

Thus, to compute the interaction list of a cell $b$ we find its parent and obtain the cells in the nearfield of the parent using bit arithmetic operations. Then we compute subcells of these cells such that size of each subcell is equal to the size of $b$. We discard those subcells that are in the nearfield of $b$. We finally partition these subcells into two arrays, one for subcells that are *local* to the processor and other for subcells that are *remote*. The latter array will thus have all the nodes whose information needs to be fetched at the *translation* phase of each iteration.

Following the building of interaction lists, multiple iterations of the remaining stages are run until convergence.

### 3.3.4 Computing Multipole Expansions

In this section, we describe how to compute the field radiated by each cell using multipole expansions. First, each processor scans its local array from left to right. When a leaf node is reached, its multipole expansion is directly computed from the particles within the leaf cell. If the node's multipole expansion is known, it is shifted to its parent and added to the parent's multipole expansion, provided the parent is local to the processor. As the tree is stored in postorder traversal order, if all the children of a node are present in the same processor, it is encountered only after all its children are. This ensures that the multipole expansion at a cell is known when the scan reaches it. This computation takes $O(\frac{n}{p} + k)$ time, where $k$ is the highest resolution. During the scan, some nodes are labeled *residual nodes* based on the following rules:

- If the multipole expansion due to a cell is known but its parent lies in a different processor, it is labeled a *residual leaf node*.
- If the multipole expansion at a node is not yet computed when it is visited, it is labeled a *residual internal node*.

Each processor copies its residual nodes into an array. It is easy to see that the residual nodes form a tree (termed the *residual tree*) and the tree is present in its postorder traversal order, distributed across processors.

Multipole expansion calculation has the associative property. Because of this, multipole expansions on the residual tree can be computed using an efficient parallel upward tree accumulation algorithm [SAF05]. The main advantage of using the residual tree is that its size is independent of the number of particles, and is rather small. Due to this reason the residual tree can be accumulated in $O(\log p + \log k)$ rounds as compared to $O(\log n)$ in case of global compressed octree. Thus, the worst-case number of communications are reduced from logarithm of the size of the tree to the logarithm of the height of the tree, which is much smaller.

### 3.3.5 Computing Multipole to Local Translations

First, for each node an all-to-all communication is used to request fields of nodes from the interaction lists that reside on remote processors. Another all-to-all communication is used to receive the fields at these nodes. Once all the information is available locally, the multipole to local translations are conducted within each processor as much as in the same way as in sequential FMM. Each processor performs $O(n/p)$ translations.

### 3.3.6 Computing Local Expansions

Similar to the multipole expansion calculation, the local expansion calculation is also associative. Thus, local expansions can be computed using a reverse of the algorithm for computing multipole expansions in parallel. First, local expansions for the residual tree are calculated. This requires $O(\log p + \log k)$ communication rounds. Then, local expansions for the local tree are computed using a right-to-left scan of the postorder traversal of the local tree. The exact number of communication rounds required is the same as in computing multipole expansions.

### 3.3.7 Parallel FMM on GPU

Implementing the above algorithm on GPU essentially means shifting to a shared memory system, thereby replacing the time consuming *all-to-all* communication steps by calls to local memory. *Hints* have been given to implement compressed octrees on GPU in subsection 3.2.4.3. Once the octrees are implemented, all it remains are traditional memory access operations of GPU to read and write the data at desired locations, and doing computations on those data on each GPU processing element in parallel. Shared memory makes it much more easier to build and use the interaction lists, store and compute on the residual trees and eliminate the time required for data communication (which was done using *all-to-all* primitive previously).

*Note that this gives us just a starting point for parallel implemention of FMM on GPU. Many intricate and vital details are still ignored and left unanswered. They will eventually be solved as the implementation work progresses.*

Chapter 4

# Discussion: Specular Inter-reflections and Caustics in Point based Models

## 4.1 Introduction

After having seen the algorithms and techniques for computing diffuse global illumination on point models, let us now focus on computing specular effects (reflections and refractions) including caustics for the point models. These, combined with already calculated diffuse illumination gives the user a *complete global illumination solution for point models*.

Attempts have been made to get these effects. Schaufler [SJ00] was the first to propose a ray-tracing technique for point clouds. Their idea is based on sending out rays with certain width which can geometrically be described as cylinders. The intersection detection is performed by determining the points of the point cloud that lie within such a cylinder followed by calculating the ray-surface intersection point as distance-weighted average of the locations of these points. The normal information at the intersection point is determined using the same weighted averaging. This approach *does not* handle varying point density within the point cloud. Moreover, the surface generation is view-dependent, which may lead to artifacts during animations. Wand [WS03] introduced a similar concept by replacing the cylinders with cones, but they started with triangular models as their input instead of point models. Adamson [AA03] proposed a method for ray-tracing point-set surfaces but was computationally too expensive, the running times being in several hours. Wald [WS05] then described a framework for interactive ray-tracing of point models based on a combination of an implicit surface representation, an efficient surface intersection algorithm and a specifically designed acceleration structure. However, implicit surface calculation was too expensive and hence they used ray-tracing *only* for shadow computations. Also, the actual shading was performed only by a local shading model. Thus, transparency and mirroring reflections were not modelled. Linsen [LMR07] recently introduced a method of Splat-Based Ray-Tracing for Point Models handling the shadow, reflections and refraction effects efficiently. However, they did not consider rendering caustics effects in their algorithm.

Our proposed method is a combination of many such methods discussed above, which combines the advantages each of them offer under one domain. We will successfully be able to get all the desired specular effects (reflections, refractions and caustics) along with producing a time and memory efficient algorithm for the same. We will also be able to fuse it with the diffuse illumination algorithm to give a complete global illumination solution.

Our proposed algorithm follow the Photon Mapping (for polygonal models) [Jen96] strategy closely. Therefore, we start by giving a brief overview of all the stages of photon mapping algorithm in Section 4.2 and conclude with some limitations of this technique. We then follow it up with our proposed method in Section 4.3 to get all the desired specular effects in a point model scene.

## 4.2 Photon Mapping

This section aims to give an overview of the photon mapping algorithm along with some of their limitations (for details refer [Jen96]).

The global illumination algorithm based on photon maps is a two-pass method. The first pass builds the photon map by emitting photons from the light sources into the scene and storing them in a photon map when they hit non-specular objects. The second pass, the rendering pass, uses statistical techniques on the photon map to extract information about incoming flux and reflected radiance at any point in the scene. The photon map is decoupled from the geometric representation of the scene. This is a key feature of the algorithm, making it capable of simulating global illumination in complex scenes containing millions of triangles, instanced geometry, and complex procedurally defined objects. We will look into the details related to the emission, tracing, storing of photons and rendering in the remainder of this section.

To help explain the algorithms presented in this section we adopt a notation for light transport introduced by Heckbert [Hec90]. In Heckbert′s notation a path traveled by light can be described by a regular expression of the interactions the light has been through. Possible interactions are: the light source (L), the eye (E), a diffuse reflection (D), a specular reflection (S). An example is the light path $LS^+DE$, which describes light coming from the light source, being specular reflected one or more times before being diffusely reflected in the direction of the eye. Incidentally, this is the path traveled by light when creating caustics.

### 4.2.1 Photon Tracing (First Pass)

The purpose of the photon tracing pass is to compute indirect illumination on diffuse surfaces. This is done by emitting photons from the light sources, tracing them through the scene, and storing them at diffuse surfaces.

**Photon Emission:** The photons emitted from a light source should have a distribution corresponding to the distribution of emissive power of the light source. If the power of the light is Plight and the number of emitted photons is $n_e$, the power of each emitted photon is

$$P_{photon} = P_{light}/n_e.$$

Pseudocode for a simple example of photon emission from a diffuse point light source is given below:

**Algorithm 14** Photon emission from a diffuse point light

```
procedure emitPhotons
$n = 0$ // number of emitted photons
while not enough photons do
    DO
    // use simple rejection sampling
    // to find diffuse photon direction
    $x$ = random number between $-1$ and $1$
    $y$ = random number between $-1$ and $1$
    $z$ = random number between $-1$ and $1$
    while ( $x * x + y * y + z * z > 1$ )
    $d = < x, y, z >$
    $p$ = light source position
    trace photon from p in direction d
    $n = n + 1$
end while
scale power of stored photons with 1/n
```

**Photon Tracing:** Once a photon has been emitted, it is traced through the scene using photon tracing. When a photon hits an object, it can either be reflected, transmitted, or absorbed (with some power loss), decided probabilistically based on the material parameters of the surface using Russian roulette [Jen96] Examples of photon paths are shown in Figure 4.1.

**Photon Storing:** Photons are only stored where they hit diffuse surfaces (or, more precisely, nonspecular surfaces). The reason is that storing photons on specular surfaces does not give any useful information: the probability of having a matching incoming photon from the specular direction is zero, so if we want to render accurate specular reflections the best way is to trace a ray in the mirror direction using standard ray tracing. For all other photon-surface interactions, data is stored in a global data structure, the *photon map*. Note that each emitted photon can be stored several times along its path. Also, information about a photon is stored at the surface where it is absorbed if that surface is diffuse. For each photon-surface interaction, the position, incoming photon power, and incident direction are stored.

**Three Photon Maps:** For efficiency reasons, it pays off to divide the stored photons into three photon maps:

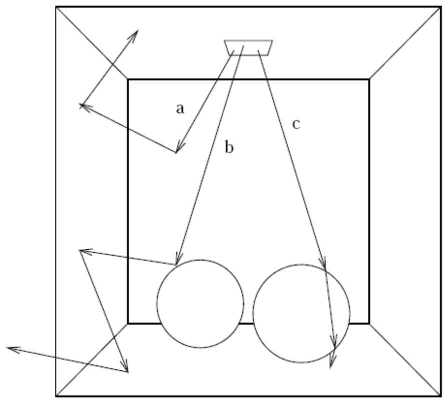


Figure 4.1: Photon paths in a scene (a Cornell box with a chrome sphere on left and a glass sphere on right): (a) two diffuse reflections followed by absorption, (b) a specular reflection followed by two diffuse reflections, (c) two specular transmissions followed by absorption.

- *Caustic Photon Map:* contains photons that have been through at least one specular reflection before hitting a diffuse surface: $LS^{+}D$.
- *Global Photon Map:* an approximate representation of the global illumination solution for the scene for all diffuse surfaces: $L\{S|D|V\}^{*}D$
- *Volume Photon Map:* indirect illumination of a participating medium: $L\{S|D|V\}^{+}V$.

A separate photon tracing pass is performed for the caustic photon map since it should be of high quality and therefore often needs more photons than the global photon map and the volume photon map. The construction of the photon maps is most easily achieved by using two separate photon tracing steps in order to build the caustics photon map and the global photon map (including the volume photon map). This is illustrated in Figure 4.2 for a simple test scene with a glass sphere and 2 diffuse walls. Figure 4.2(a) shows the construction of the caustics photon map with a dense distribution of photons,and Figure 4.2(b) shows the construction of the global photon map with a more coarse distribution of photons.

### 4.2.2 Preparing the Photon Map for Rendering

In the rendering pass, the photon map is a static data structure that is used to compute estimates of the incoming flux and the reflected radiance at many points in the scene. To do this it is necessary to locate the nearest photons in the photon map. This is an operation that is done extremely often, and it is therefore a good idea to optimize the representation of the photon map before the rendering pass such that finding the nearest photons is *as fast as possible*.

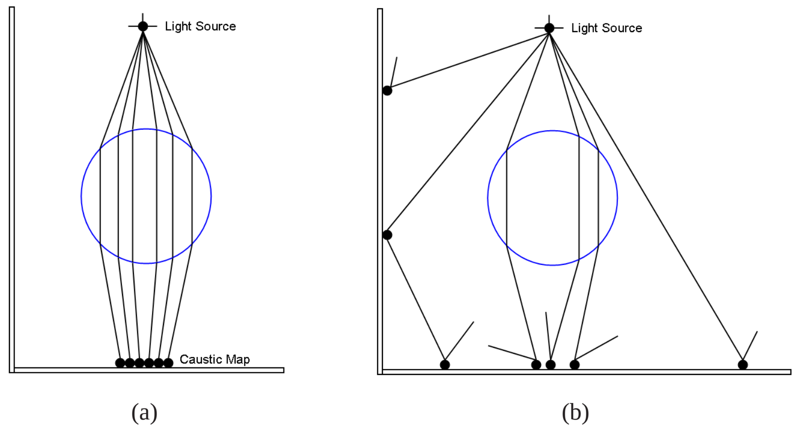


Figure 4.2: Building (a) the caustics photon map and (b) the global photon map.

The data structure should be compact and at the same time allow for fast nearest neighbor searching. It should also be able to handle highly non-uniform distributions  this is very often the case in the caustics photon map. A natural candidate that handles these requirements is a *balanced kd-tree*.

**The balanced kd-tree:** The time it takes to locate one photon in a balanced kd-tree has a worst time performance of O(logN) [Moo93], where N is the number of photons in the tree.

### 4.2.3 Rendering (Second Pass)

Given the photon map, we can proceed with the rendering pass. The photon map is view independent, and therefore a single photon map constructured for an environment can be utilized to render the scene from any desired view. The final image is rendered using distribution ray tracing in which the pixel radiance is computed by averaging a number of sample estimates. Each sample consists of tracing a ray from the eye through a pixel into the scene. The radiance returned by each ray equals the outgoing radiance in the direction of the ray leaving the point of intersection at the first surface intersected by the ray. The outgoing radiance, $L_o$, is the sum of the emitted, $L_e$, and the reflected radiance

$$L_o(x, \overrightarrow{w}) = L_e(x, \overrightarrow{w}) + L_r(x, \overrightarrow{w})$$

where the reflected radiance, $L_r$, is computed by integrating the contribution from the incoming radiance, $L_i$,

$$L_r(x, \overrightarrow{w}) = \int_{\sigma_x} f_r(x, \overrightarrow{w}', \overrightarrow{w}) L_i(x, \overrightarrow{w}') cos\theta_i dw_i'$$

where $f_r$ is the bidirectional reflectance distribution function (BRDF), and x is the set of incoming directions around x. The BRDF is separated into a sum of two components: A specular/glossy, $f_{r,s}$, and a diffuse, $f_{r,d}$

$$f_r(x, \overrightarrow{w}', \overrightarrow{w}) = f_{r,s}(x, \overrightarrow{w}', \overrightarrow{w}) + f_{r,d}(x, \overrightarrow{w}', \overrightarrow{w})$$

The incoming radiance is classified using 3 components:

- $L_{i,l}(x, \overrightarrow{w}')$ is direct illumination by light coming from the light sources.
- $L_{i,c}(x, \overrightarrow{w}')$ is caustics - indirect illumination from the light sources via specular reflection or transmission.
- $L_{i,d}(x, \overrightarrow{w}')$ is indirect illumination from the light sources which has been reflected diffusely at least once.

The incoming radiance is the sum of these three components:

$$L_i(x, \overrightarrow{w}') = L_{i,l}(x, \overrightarrow{w}') + L_{i,c}(x, \overrightarrow{w}') + L_{i,d}(x, \overrightarrow{w}')$$

By using the classifications of the BRDF and the incoming radiance we can split the expression for reflected radiance into a sum of four integrals:

$$\begin{aligned}
L_r(x, \overrightarrow{w}) &= \int_{\sigma_x} f_r(x, \overrightarrow{w}', \overrightarrow{w}) L_i(x, \overrightarrow{w}') cos\theta_i dw_i^{'} \\
&= \int_{\sigma_x} f_r(x, \overrightarrow{w}', \overrightarrow{w}) L_{i,l}(x, \overrightarrow{w}') cos\theta_i dw_i^{'} + \\
&\quad \int_{\sigma_x} f_{r,s}(x, \overrightarrow{w}', \overrightarrow{w}) (L_{i,c}(x, \overrightarrow{w}') + L_{i,d}(x, \overrightarrow{w}')) cos\theta_i dw_i^{'} + \\
&\quad \int_{\sigma_x} f_{r,d}(x, \overrightarrow{w}', \overrightarrow{w}) L_{i,c}(x, \overrightarrow{w}') cos\theta_i dw_i^{'} + \\
&\quad \int_{\sigma_x} f_{r,d}(x, \overrightarrow{w}', \overrightarrow{w}) L_{i,d}(x, \overrightarrow{w}') cos\theta_i dw_i^{'}
\end{aligned}$$

There are 4 integrals in the above equation. $1^{st}$ term computes *Direct Illumination*. $2^{nd}$ terms computes *Specular and Glossy Reflection*. $3^{rd}$ term computes *Caustics*. $4^{th}$ term computes *Multiple Diffuse Reflections*.

**Specular and Glossy Reflection:** Specular and glossy reflection is computed by evaluation of the term

$$\int_{\sigma_x} f_{r,s}(x, \overrightarrow{w}', \overrightarrow{w}) (L_{i,c}(x, \overrightarrow{w}') + L_{i,d}(x, \overrightarrow{w}')) cos\theta_i dw_i^{'}$$

The photon map is not used in the evaluation of this integral since it is strongly dominated by $f_{r,s}$ which has a narrow peak around the mirror direction. Using the photon map to optimize the integral would require a huge number of photons in order to make a useful classification of the different directions within the narrow peak of $f_{r,s}$. To save memory this strategy is not used and the integral is evaluated using standard Monte Carlo ray tracing optimized with importance sampling based on $f_{r,s}$.

**Caustics:** Caustics are represented by the integral

$$\int_{\sigma_x} f_{r,d}(x, \overrightarrow{w}', \overrightarrow{w}) L_{i,c}(x, \overrightarrow{w}') cos\theta_i dw_i^{'}$$

The evaluation of this term is dependent on whether an accurate or an approximate computation is required. In the accurate computation, the term is solved by using a radiance estimate from the caustics photon map. The number of photons in the caustics photon map is high and we can expect good quality of the estimate. The approximate evaluation of the integral is included in the radiance estimate from the global photon map.

### 4.2.4 Radiance Estimate

The reflected illumination is reconstruction from the photon map through a series of queries to the photon maps. Each query is used to estimate the reflected radiance at a surface point as the result of a local photon density estimate. A query to the photon map locates the *k* photons nearest the surface point for which the reflected radiance is to be estimated. In conjunction with the surface BRDF, the incoming direction, the surface point and the area encompassing the photons this information is used in a local density estimate that estimates the reflected radiance. This estimate is called the *radiance estimate* [Sch06].

The accuracy of the radiance estimate is controlled by two important factors; the resolution of the photon map and the number of photon used in each radiance estimate. If few photons are used in the radiance estimate, noise in the illumination becomes visible, if many photons are used edges and other sharp illumination features such as those caused by caustics are blurred. Unless an excessive number of photons are stored in the photon map, it is impossible to avoid either of these effects. This is the mentioned trade-off problem between variance versus bias as it manifests itself in photon mapping.

*Figure 4.3 shows an example output of Photon Mapping algorithm.*

### 4.2.5 Limitations of Photon Mapping

Although Photon Mapping is a well established technique for giving a complete global illumination solution, it too suffers from some limitations as itemized below:

- Works only for polygonal models. We need to modify the algorithm so that it works for point models as well.
- One obvious cost factor for photon mapping is the cost for performing $k$ nearest neighbor queries used for density estimation of photons. As we already have a pre-computed diffuse illumination, we are only interested in caustic maps. But still, although $kNN$ queries are commonly considered to be rather cheap, it is infact quite expensive when compared to a fast ray tracer for rendering (about $10$ times expensive) even for just caustic maps.
- Photon Generation and Tracing are quite slow as well. Needs to be optimized.

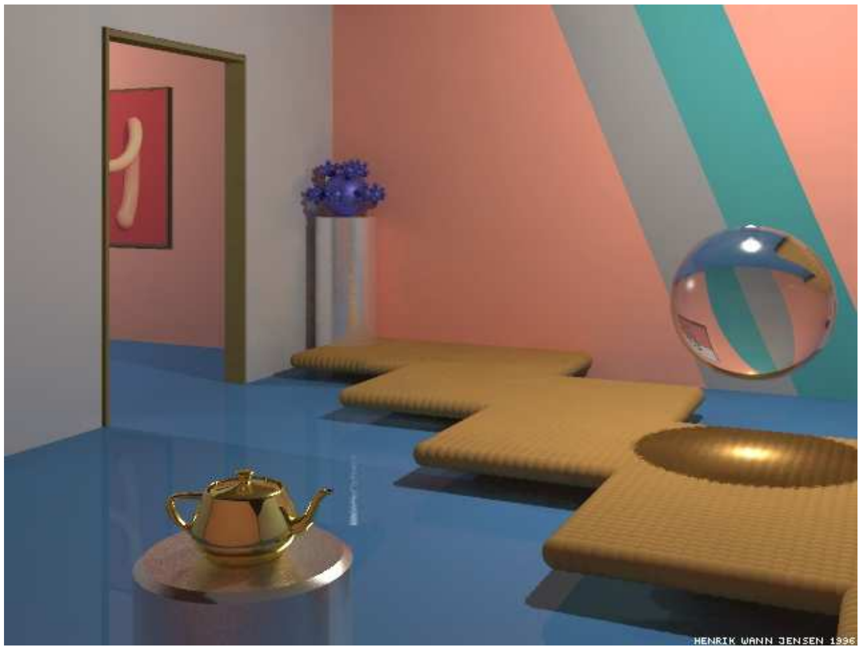


Figure 4.3: Example output of Photon Mapping Algorithm [Jen96] showing reflection, refractions and caustics

## 4.3 Our Approach

We now present our algorithm to generate specular effects for point models. We try to eliminate the restrictions of traditional Photon Mapping algorithm at the same time optimizing on the basic technique using a combination of several algorithms available in literature.

*Note that, our specular-effects generation algorithm takes as input a point model with diffuse global illumination solution already calculated for it. As the diffuse global illumination solution is view-independent, it provides us with an advantage of having an interactive walk-through of the input scene of point models. However, specular effects being view-dependent needs to be calculated for every new view-point in the ray-trace rendered frame. Thus, if specular effect generation takes a lot of time, we loose out of having an interactive walk-through of the scene. We desire not to loose this advantage, and try to optimize every algorithm required for specular effect generation.*

We saw traditional Photon Map works only for polygonal models, which have surface information. But point models do not have any kind of surface representations. We thereby make necessary modifications in this algorithm to apply it to point models. We can divide our goal in 2 major tasks:

- Modifying Path Tracing (First Pass)
- Modifying Ray Tracing (Second Pass)

All the other modules of the algorithm are independent of surface representations.

Fortunately, solution to both of the above tasks is the same. Proper analysis of the algorithm suggests that both Photon Tracing and the final rendering is done using Ray Tracing techniques. So, modifying the Ray Tracing technique to suit Point Models is sufficient. There has been some research efforts in the same direction (as discussed in Section 4.1). We will discuss here one of the very efficient techniques for doing the same, *Splat-based Ray Tracing* [LMR07], in the next section.

### 4.3.1 Splat-Based Ray Tracing

Surface splatting is established as one of the main rendering techniques for point clouds. This section presents a ray-tracing approach for objects whose surfaces are represented by point clouds. This approach is based on casting rays and intersecting them with *disks around points* or *splats* [LMR07].
Splats in their general form define a piece-wise constant surface. In particular, each splat has exactly one surface normal assigned to it. Assuming that the point cloud was obtained by scanning a smooth surface, the application of the rendering technique should result in the display of a smoothly varying surface. Since ray tracing is based on casting rays, whose directions depend on the surface normals, there's a need to define smoothly varying normals over the entire surface, i.e., also within each splat. To do so, estimated normals at each point of the point cloud are considered and splat radii are computed depending on local curvature properties. The generated splats should cover several points of the point cloud. The normals at the covered points of each splat are used to determine a smoothly varying normal field defined over a local parameter space of the splat. It can be beneficial to consider further surrounding points and their normals for the normal field computations. Details on the splat and normal field generation are described later in the Section 4.3.1.1.
The actual ray-tracing procedure is executed by sending out rays that intersect the splats, potentially being reflected or refracted. Surface normals are interpolated from the normal fields. Care has to be taken where splats overlap. The ray-splat intersection and the overall image generation is described in subsection 4.3.2.

#### 4.3.1.1 Splat Generation

Let $P$ be a point cloud consisting of $n$ points $\mathbf{p}_1, ..., \mathbf{p}_n \in \Re^3$. We generate $m$ splats $S_1, ..., S_m$ that cover the entire surface represented by point cloud $P$. For each of these splats we are computing its radius $r_i \in \Re, i = 1, ..., m$, and a normal field $\mathbf{n}_i(u, v), i = 1, ..., m$, where $(u, v) \in [-1, 1]$x$[-1, 1]$ with $u^2 + v^2 \leq 1$ describes a local parametrization of the splat.
**Splat Radius:** The radii of the m splats $S_1, ..., S_m$ should vary with respect to the curvature of the surface covered by the splat. In regions of high curvature, a piece-wise constant surface representation via splats requires us to use many splats with small radii to stay within a predefined error bound. In regions of low curvature, some few large splats may suffice to represent the surface well. For the definition of the error bound, the maximum distance of points of $P$ covered by the splat to their closest point on the splat is chosen.

Let $\mathbf{p}_i \in P$ be any of the points of point cloud $P$ and let $\mathbf{n}_i$ be the respective surface normal of the surface described by *P* at position $\mathbf{p}_i$. If the normal $\mathbf{n}_i$ is unknown, we determine the normal by computing the *k* nearest neighbors $\mathbf{q}_1, ..., \mathbf{q}_k \in P$ of $\mathbf{p}_i$, fit a plane through $\mathbf{p}_i$ and its neighbors in the least-squares sense, and set $\mathbf{n}_i$ to the normal of the fitting plane.

Let the neighbors of $\mathbf{p}_i$ be sorted in the order of increasing distance to $\mathbf{p}_i$. We initially define splat $S_j = (c_j, n_j, r_j)$ with center $c_j = \mathbf{p}_i$, normal $\mathbf{n}_j = \mathbf{n}_i$, and radius $r_j = 0$. Next, the splat is grown iteratively, until the error bound condition is violated.

At each iteration step, the radius is increased such that the splat covers $1$ additional neighbor of $\mathbf{p}_i$. The normal remains unchanged, but center $\mathbf{c}_j$ is moved along the surface normal $\mathbf{n}_i$ such that the splat position minimizes its maximal distance to all covered points of *P*. Figure 4.4(a) illustrates the optimal choice of $\mathbf{c}_j$.

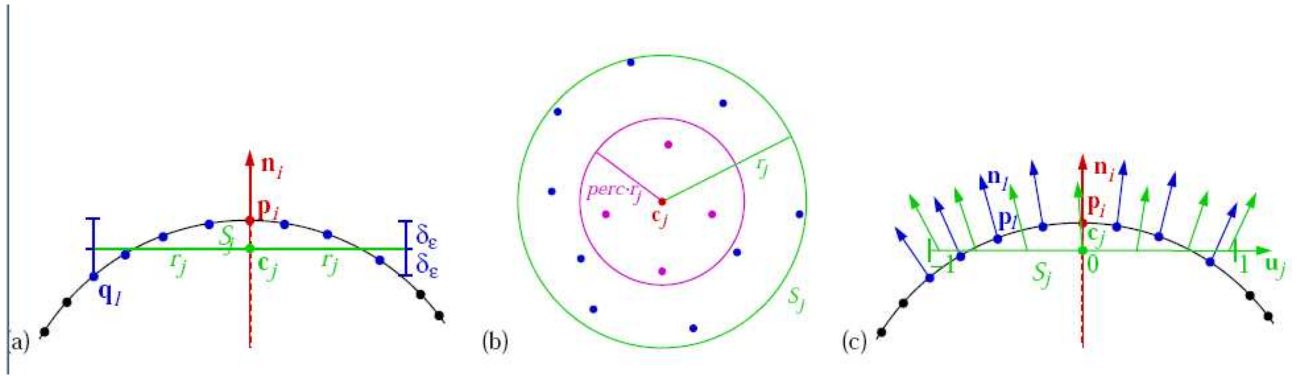


Figure 4.4: (a) Generation of splat $S_j$ starts with point $\mathbf{p}_i$ and grows the splat with radius $r_j$ by iteratively including neighbors $\mathbf{q}_l$ of $\mathbf{p}_i$ until the approximation error $\delta_\epsilon$ for the covered points exceeds a predefined error bound. (b) Splat density criterion: Points whose distance from the splats center $\mathbf{c}_j$ when projected onto splat $S_j$ is smaller than a portion *perc* of the splats radius $r_j$ are not considered as starting points for splat generation. (c) Generation of linear normal field (green) over splat $S_j$ from normals at points covered by the splat. Normal field is generated using local parameters $(u, v) \in [1, 1]$X$[1, 1]$ over the splats plane spanned by vectors $\mathbf{u}_j$ and $\mathbf{v}_j$ orthogonal to normal $\mathbf{n}_j = \mathbf{n}_i$. The normal of the normal field at center point $\mathbf{c}_j$ may differ from $\mathbf{n}_i$.

**Splat Density:** Let $S_j$ be the splat that covers the point $\mathbf{p}_i$ and its *k* nearest neighbors $\mathbf{q}_1, ..., \mathbf{q}_k$, again sorted by increasing distance to $\mathbf{p}_i$. To not generate holes in the surface, these *k* nearest neighbors should also include all natural neighbors of $\mathbf{p}_i$, when computing natural neighbors locally for points projected into a fitting plane. If the natural neighbors of one of the points $\mathbf{q}_l, l \in 1, ..., k$, are also among the *k* nearest neighbors of $\mathbf{p}_i$, no splat needs to be generated starting from $\mathbf{q}_l$ . Obviously, the smaller the distance of a neighbor $\mathbf{q}_l$ to point $\mathbf{p}_i$ is, the higher are the chances that the natural neighbors are already among the neighbors of $\mathbf{p}_i$.

This motivation led to the following criterion: If splat $S_j$ is generated starting from point $\mathbf{p}_i$, then no splats need to be generated starting from neighbored points within the projected distance *perc*.$r_j$ from the splats center $\mathbf{c}_j$

, where $perc \in [0,1]$ is a factor that defines the percentage of the splats radius used for the criterion, see Figure 4.4(b). The factor *perc* is defined globally for *P*, which is possible as it is multiplied with the locally varying radii $r_j$ . The optimal choice for *perc* is a value such that the generated splats cover the entire surface and have minimal overlap.

**Normal Field:** In order to generate a smooth-looking visualization of a surface with a piece-wise constant representation, there is a need to smoothly (e. g. linearly) interpolate the normals over the surface before locally applying the light and shading model. Since we do not have connectivity information for our splats, we cannot interpolate between the normals of neighbored splats. Instead, we need to generate a linearly changing normal field within each splat. The normal fields of adjacent points should approximately have the same interpolated normal where the splats meet or intersect.
Let $S_j = (c_j, n_j, r_j)$ be one of the splats generated as described above. In order to define a linearly changing normal field over the splat, we use a local parametrization on the splat. Let $\mathbf{u}_j$ be a vector orthogonal to the normal vector $\mathbf{n}_j$ and $\mathbf{v}_j$ be defined as $\mathbf{v}_j = \mathbf{n}_j \text{x} \mathbf{u}_j$ . Moreover, let $\|\mathbf{u}_j\| = \|\mathbf{v}_j\| = r_j$ . The orthogonal vectors $\mathbf{u}_j$ and $\mathbf{v}_j$ span the plane that contains splat $S_j$ . A local parametrization of the splat is given by

$$(u, v) \mapsto \mathbf{c}_j + u\mathbf{u}_j + v\mathbf{v}_j$$

with $(u, v) \in \Re^2$ and $u^2 + v^2 \leq 1$. The origin of the local 2D coordinate system is the center of the splat $S_j$.
Using this local parametrization, a linearly changing normal field $\mathbf{n}_j(u, v)$ for splat is defined $S_j$ by

$$\mathbf{n}_j(u, v) = \overrightarrow{\mathbf{n}}_j + u\nu_j\mathbf{u}_j + v\omega_j\mathbf{v}_j$$

The vector $\overrightarrow{\mathbf{n}}_j$ describes the normal direction in the splats center. It is tilted along the splat with respect to the yet to be determined factors $\nu_j, \omega_j \in \Re$. Figure 4.4(c) illustrates the idea.
To determine the tilting factors $\nu_j$ and $\omega_j$ is exploited the fact that the normal directions are known at the points of point cloud *P* that are covered by the splat. Let $\mathbf{p}_l$ be one of these points. $\mathbf{p}_l$ is projected onto the splat, local coordinates $(u_l, v_l)$ of $\mathbf{p}_l$ are determined, and the following equation is derived

$$\mathbf{n}_l = \overrightarrow{\mathbf{n}}_j + u_l\nu_j\mathbf{u}_j + v_l\omega_j\mathbf{v}_j$$

where $\mathbf{n}_l$ denotes the surface normal in $\mathbf{p}_l$. Proceeding analogously for all other points out of *P* covered by splat $S_j$, a system of linear equations is obtained with unknown variables $\nu_j$ and $\omega j$. Since the system is overdetermined, it can only be solved approximately.

## 4.3.2 Ray Tracing

### 4.3.2.1 Main Approach

The input of the ray-tracing procedure are the m splats $S_1, ..., S_m$ generated from point cloud *P*. Each splat $S_j$ is given by its center $\mathbf{c}_j$ , its radius $r_j$ , and its normal field $\mathbf{n}_j(u, v)$ using local parameters $(u, v)$ over the local coordinate system $(u_j, v_j)$.

The standard ray-tracing method that is applied sends out primary rays from the camera position through the center of each pixel of the resulting image onto the scene. The intersection of the primary rays with the objects of the scene using ray-splat intersections is computed. From the intersection points are sent out secondary rays, i.e., shadow rays towards all light sources, reflection rays in case of reflective surfaces, and refraction rays in case of transmissive surfaces. In the latter two cases, we enter the recursion until the ray-trace depth is met.

#### 4.3.2.2 Octree Generation

In order to process computations of ray-splat intersections efficiently, an octree for storing the splats is used. The generation of the octree and the insertion of the splats is done in two steps.

The first step is the dynamic phase, where the octree is generated. Starting with an empty octree represented by the root that describes the bounding box of the entire scene, each splat is iteratively inserted into that leaf cell that contains the center of the splat. As soon as one leaf cell would contain more than a given small number $c_s$ of splat entries, the leaf cell gets subdivided into eight equally-sized subcells. The splats that were stored in the former leaf cell get adequately distributed among its children, which are the new leaf cells. This first phase is as simple as generating an octree for points. The iteration stops once all splats have been inserted.

The second step is the static phase. Further splat insertions are made, but the structure of the octree does not change anymore, i.e., no further cell subdivisions are executed. The additional splat insertions are necessary, as splats have an expansion and may stretch over various cells. Thus, in this second phase, we want to insert the splats into all leaf cells they intersect, see Figure 4.5(a). Since such an exact cell-splat intersection is computationally rather expensive, the splats are inserted into leaf cells that potentially intersect the splat.

For each splat $S_j$, the tree is traversed top-down applying a nested test for each traversed cell. The first test checks for splat $S_j$ whether the axes-aligned box with center $c_j$ and side length $2.r_j$ intersects the cell. If the test fails, tree traversal for that branch stops. For all leaf cells, for which the first test was positive, a second test is performed. The second test uses the local parametrization of the splat. The local parameters (0,0), (0,1), (1,0), and (1,1) define a 2D square that bounds the splat. The position of these four points is checked against the leaf cell. If all four points lie on one side of one of the six planes that bound the leaf cell, the splat cannot intersect the leaf cell, see Figure 4.5(c). Otherwise, the splat is inserted into the leaf cell, see Figure 4.5(b).

#### 4.3.2.3 Ray-splat Intersection

The intersection of rays with splats is computed using the octree partitioning of the three-dimensional scene. For primary rays starting from the camera position (or eye point), the intersection of the ray with the bounding box of the octree is computed, i.e., with the cell represented by the octrees root. The leaf cell to which the intersection point belongs is determined, and then algorithm continues from there. From then on, primary and

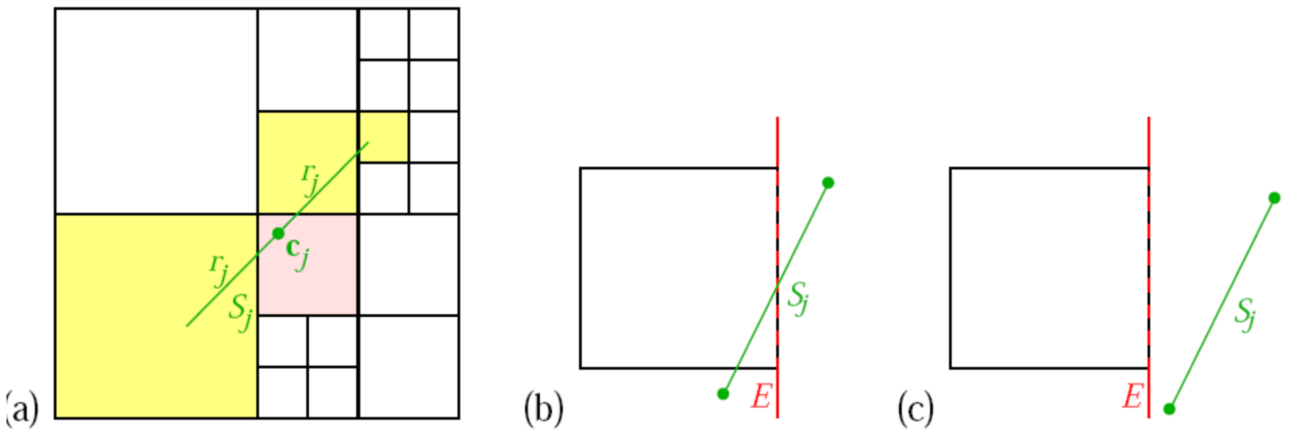


Figure 4.5: (a) Octree generation: In the first phase, the octree is generated while inserting splats $S_j$ into the cells containing their centers $c_j$ (red cell). In the second phase, splat $S_j$ is inserted into all additional cells it intersects (yellow cells). (b)(c) The second test checks whether the edges of the bounding square of splat $S_j$ intersect the planes *E* that bound the octree leaf cell. (b) $S_j$ is inserted into the cell. (c) $S_j$ is not inserted into the cell. This second test is only performed if the first test (bounding box test) was positive.

secondary rays can be treated equally.

If the rays hits a (leaf) cell of the octree, intersection of the ray with all splats stored within that cell is checked for. If the ray does not intersect any of the splats stored in that cell or if the cell is empty, the algorithm proceeds with the adjacent cell in the direction of the ray. If it ends up leaving the bounding box of the octree, the respective background color is reported back. If the ray intersects a splat stored in the current cell, it computes the precise intersection point and applies the shading, reflection, and refraction model possibly using recursive calls to compute the color, which is reported back. If the ray hits multiple splats stored in the current cell, the algorithm computes the intersection points and pick the most appropriate one.

After having a look at the *Splat-Based Ray Tracing* technique, we know how to incorporate Photon Mapping for point models (replacing the ray-tracer). But, Photon Mapping by itself still takes a lot of time to generate *visually pleasing* results. Hence, next, we try to optimize the traditional Photon Mapping algorithm to work faster (and possibly at interactive rates). Photon mapping, if divided, works in three stages:

- Photon Generation
- Photon Traversing and performing intersection tests
- Photon retrieval using $kNN$ queries while ray-trace rendering

We target each of the three stages of Photon Mapping one by one and try to optimize them as much as possible in the following sections.

### 4.3.3 Optimizing Photon Generation and Sampling

Recall that we generate only caustic photon maps as we already have a pre-computed diffuse global illumination solution. Thus, the obvious candidate for optimization is the time required for generating caustic photons. Looking at the Photon Mapping algorithm reveals that some of the cost factors for photon generation can not be improved on. For example, rays will be incoherent during photon generation, and each light path will require several surface interactions (for reflection and refraction) in order to generate a caustic photon. However, the number of paths that actually yield caustic photons can be influenced, and should be maximized.

#### 4.3.3.1 Sampling Caustics using Selective Photon Tracing

We use a method similar to Wald [GWS04] which uses, *Selective Photon Tracing (SPT)* [DBMS02]. Like [GWS04] we do not consider the temporal domain, but rather use *Selective Photon Tracing* for adaptively sampling path space: In a first step, a set of "pilot photons" is traced into the scene in order to detect paths that generate caustics. For those pilot paths, periodicity properties of the Halton sequence [Nie92] are exploited to generate similar photons.

By using *Selective Photon Tracing* the increase in the yield of caustic photons is roughly by a factor of four. Essentially, this means that the same number of caustic photons can be generated with only one fourth of all rays. As the improvement depends significantly on the (projected) size of the caustic generator, the results for smaller caustic generators are likely to be more significant than large ones.

This approach also handles indirect caustics , as the photons of one group also stay together after diffuse bounces. Most importantly, however, this method does not require any preprocessing and maintains the photon map's property of being independent of scene geometry and thus well-suited for both complex scenes and interactive setups.

More details of this algorithm can be found in [GWS04] and [DBMS02].

We thus, previously, had a ray-tracer (*Splat-Based Ray Tracing*) which is capable of generating specular effects (*sans caustics*) on point models. Combining this ray-tracer with the new faster caustic-map generation technique (using *Selective Photon tracing*) gives us quite a bit of speedup.

### 4.3.4 Optimized Photon Traversal and Intersection tests

The intersection tests performed for generating caustic photon map is similar to those performed while doing *Splat-Based Ray-Tracing* (ray-splat intersections), and thereby we need not worry about designing a new algorithm for the same. Further, *Splat-Based Ray-Tracing* uses *Octree data-structure* for traversal of the primary and secondary rays during ray-tracing. The use of Octree data structure provides us with quite a few advantages:

- We already have Octree data structure generated for input point model while doing diffuse illumination. Hence same structure can be *re-used*.
- The same traversal algorithm which *Splat-Based Ray Tracing* uses on Octrees can be used for photon traversal as well.
- Further more, we can go for an even more optimized algorithm for Octree Traversal using neighbor finding [Sam89]. Here we traverse the octree *horizontally* via neighbor finding instead of traversing vertically starting from the root to the desired node.

Thus, we already have a well-established data structure (Octree) and algorithm (ray-splat intersection) for performing optimal photon traversal and intersection tests of rays and splats around points.

### 4.3.5 Fast Photon Retrieval using Optimized $kNN$ Query Algorithm

We now have an optimized caustic photon generation code, an optimized photon traversal and ray-splat intersection code, a good ray-tracer capable of handling specular effects on point models. All it remains is to have an optimized $kNN$ query algorithm for fast photon retrieval while rendering.
Although, $kd$-trees provides for fast $kNN$ queries, they are still slow for interactive settings we desire. Also, its difficult to extend $kd$-trees to hardware, and would account for high latency or would require a large cache to avoid this latency on average.
The algorithm discussed here avoids the above mentioned issues of $kd$-trees and provides for low-latency and has sub-linear access time, there by providing for fast photon retrieval and optimized $kNN$ query algorithm. We just provide a brief overview. Details of this algorithm can be found in Ma [MM02].

#### 4.3.5.1 Low Latency Photon Retrieval Using Block Hashing

Jensen [Jen96] uses the kd-tree data structure to find these nearest photons. However, solving the kNN problem via kd-trees requires a search that traverses the tree. Even if the tree is stored as a heap, traversal still requires random-order memory access and memory to store a stack. More importantly, a search-path pruning algorithm, based on the data already examined, is required to avoid accessing all data in the tree. This introduces serial dependencies between one memory look up and the next, consequently slowing down the retrieval process.
We present here a hashing-based $AkNN$ (Approximate $kNN$) solution for fast retrieval of photons. This algorithm has bounded query time, bounded memory usage, and high potential for fine-scale parallelism. Moreover, the algorithm results in coherent, non-redundant accesses to block-oriented memory. The results of one memory look up do not affect subsequent memory lookups, so accesses can take place in parallel within a pipelined

memory system. The algorithm is based on array access, and is more compatible with current texture-mapping capabilities than tree-based algorithms.

A novel technique called *Block Hashing*(BH) is used to solve the approximate $kNN$ ($AkNN$) problem in photon mapping. The algorithm uses hash functions to categorise photons by their positions. Then, a $kNN$ query proceeds by deciding which hash bucket is matched to the query point and retrieving the photons contained inside the hash bucket for rendering purposes. One attraction of the hashing approach is that evaluation of hash functions takes constant time. In addition, once we have the hash value, accessing data we want in the hash table takes only a single access. These advantages permit us to avoid operations that are serially dependent on one another, such as those required by kd-trees, and hepls towards a low-latency implementation.

The technique is designed under two assumptions on the behavior of memory systems.

- Its assumed that memory is allocated in fixed-sized blocks.
- Its assumed that access to memory is via burst transfer of blocks that are then cached.

Thus if any part of a fixed-sized memory block is touched, access to the rest of this block will be virtually zero-cost. Therefore, in BH all memory used to store photon data is broken into fixed-sized blocks.

**Locality-Sensitive Hashing:** Since our goal is to solve the $kNN$ problem as efficiently as possible in a block-oriented cache-based context, our hashing technique requires hash functions that preserve spatial neighborhoods. These hash functions take points that are close to each other in the domain space and hash them close to each other in hash space. By using such hash functions, photons within the same hash bucket as a query point can be assumed to be close to the query point in the original domain space. Consequently, these photons are good candidates for the $kNN$ search. The algorithm uses the Locality-Sensitive Hashing (LSH) algorithm proposed by [GIM99] for the same.

The hash function in LSH groups one-dimensional real numbers in hash space by their spatial location. It does so by partitioning the domain space and assigning a unique hash value to each partition. To deal with $n$-dimensional points, each hash table will have one hash function per dimension. Each hash function generates one hash value per coordinate of the point and the final hash value is calculated by $\sum_{i=0}^{n-1} h_i P^i$ where $h_i$ are the hash values and P is the number of thresholds. Thus each photon gets mapped to three hash tables corresponding to its $x, y, z$ location co-ordinates. Details on how the thresholds for partitions are selected, how hash tables are created and what is an optimal bucket size can be referred from Ma [MM02].

Further, each of these photons occupies exactly six $32$-bit words in memory and are stored in fixed size memory blocks of $64$ $32$-bit words ($10$ photons per block).

**Block Hashing:** It, thus, contains a preprocessing phase and a query phase. The preprocessing phase consists of three steps after the photons have been traced in the scene.

- Organizing the photons into fixed-sized memory blocks
- Creation of a set of hash tables
- Inserting photon blocks into the hash tables.

Details of the pre-processing phase can be looked in Ma [MM02]. In the second phase, the hash tables will be queried for a set of candidate photons from which the $k$ nearest photons will be selected for each point in space to be shaded by the renderer.

**Querying:** A query into the BH data structure proceeds by delegating the query to each of the $L$ hash tables. These parallel accesses will yield as candidates all photon blocks represented by buckets that matched the query. The final approximate nearest neighbor set comes from scanning the unified candidate set for the nearest neighbors to the query point (see Figure 4.6.) Note that unlike $kNN$ algorithms based on hierarchical data structures, where candidates for the $kNN$ set trickle in as the traversal progresses, in BH all candidates are available once the parallel queries are completed. Therefore, BH can use algorithms like *selection* (instead of a *priority queue*) when selecting the $k$ nearest photons.

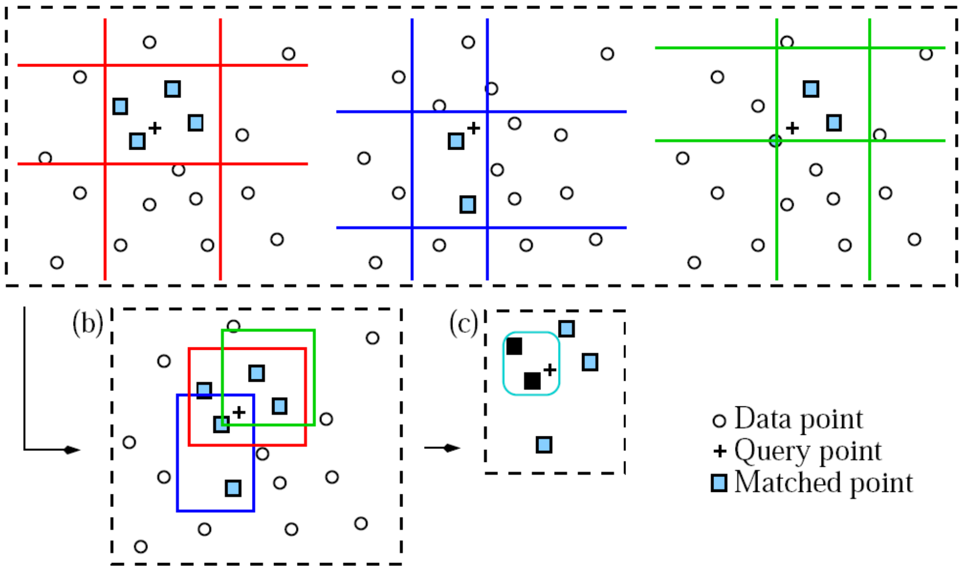


Figure 4.6: Merging the results from multiple hash tables. (a) the query point retrieves different candidates sets from different hash tables, (b) the union set of candidates after merging, and (c) the two closest neighbors selected.

*Thus, this completes the whole set-up of making Photon Mapping work for point models and optimizing every stage of the algorithm. However, issues like how to handle specular objects while computing purely diffuse global illumination using FMM is still a question. This is just a starting point and many issues need to be tackled while actual implementation.*

Chapter 5

# Conclusion and Future Work

Point-sampled geometry has gained significant interest due to their simplicity. The lack of connectivity touted as a plus, however, creates difficulties in many operations like generating global illumination effects. This becomes especially true when we have a complex scene consisting of several models, the data for which is available as hard to segment aggregated point-based models. Inter-reflections in such complex scenes requires knowledge of visibility between point pairs. Computing visibility for point models becomes all the more difficult, than for polygonal models, since we do not have any surface or object information.

Point-to-Point Visibility is arguably one of the most difficult problems in rendering since the interaction between two primitives depends on the rest of the scene. One way to reduce the difficulty is to consider clustering of regions such that their mutual visibility is resolved at a group level. Most scenes admit clustering, and the *Visibility Map* data structure we propose enables efficient answer to common rendering queries. In this report, we have given a novel, provably efficient, hierarchical, visibility determination scheme for point based models. By viewing this visibility map as a 'preprocessing' step, photo-realistic global illumination rendering of complex point-based models have been shown.

Further, we have used the *Fast Multipole Method (FMM)* as the light transport kernel for inter-reflections, in point models, to compute a description – *illumination maps* – of the diffuse illumination. Parallel implementation of FMM is a difficult task with load balancing, data decomposition and communication efficiency being the major challenges. In Sec. 3.3 we have discussed one such algorithm which uses only a static data decomposition using parallel compressed octrees, offers communication efficiency and guaranteed load balancing within a small constant factor. We now aim to exploit the parallel computing power of GPUs for implementation of the *Fast Multipole Method* based radiosity kernel as well as the point-pair visibility determination algorithm using *Visibility Maps* to provide an efficient, *fast* inter-visibility and global illumination solution for point models. Necessary *hints* were given for the same in the report.

A complete global illumination solution for point models should cover *both* diffuse and specular (reflections, refractions, and caustics) effects. Diffuse global illumination is handled by generating *illumination maps*. We, thus, further saw in the report how various algorithms from the literature were combined under a single domain to get us a *time-efficient* system designed to generate the desired specular effects for point models. We now aim to implement these algorithms, merge them together and get the specular effects solution for point models. We, thus, will have a *two–pass global illumination solver for point models*. The input to the system will be a scene consisting of both diffuse and specular point models. First pass will calculate the diffuse illumination maps, followed by the second pass for specular effects. Finally, the scene will be rendered using splat-based ray-tracing technique. However, a question remains that since we are parting the diffuse and specular effect calculations for the scene, how would we handle specular objects (and their effects on diffuse objects) while calculating *only* diffuse global illumination (This issue is very well handled in Photon Mapping [Jen96]) in the first pass of the global illumination solver. *This important issue needs to be investigated thoroughly*.

# References

[AA03] Anders Adamson and Marc Alexa. Ray tracing point set surfaces. In *SMI '03: Proceedings of the Shape Modeling International 2003*, page 272, Washington, DC, USA, 2003. IEEE Computer Society.

[ABCO$^+$03] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. Computing and rendering point set surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):3–15, 2003.

[Ama84] John Amanatides. Ray tracing with cones. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 129–135, 1984.

[BCL$^+$92] J. A. Board, J. W. Causey, J. F. Leathrum, A. Windemuth, and K. Schulten. Accelerated molecular dynamics simulation with the parallel fast multipole method. *Chemistry Physics Letters*, 198:89–94, 1992.

[BG] R. Beatson and L. Greengard. A Short Course on Fast Multipole Methods.

[Bit02] Jiri Bittner. *Hierarchical Techniques for Visibility Computations*. PhD thesis, Czech Technical University, 2002.

[CBC$^+$01] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3d objects with radial basis functions. *Proceedings of ACM SIGGRAPH*, pages 67–76, August 2001.

[CGR88] J. Carrier, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm for particle simulations. *SIAM Journal of Scientific and Statistical Computing*, 9:669–686, July 1988.

[CGR99] H. Cheng, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm in three dimensions. *Journal of Computational Physics*, 155:468–498, 1999.

[Chr] T. W. Christopher. Bitonic Sort Tutorial. `http://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm`.

[DBMS02] K. Dmitriev, S. Brabec, K. Myszkowski, and H. Seidel. Interactive global illumination using selective photon tracing. In *The 13th Eurographics Workshop on Rendering*, pages 21–34, 2002.

[DDP96] Frédo Durand, George Drettakis, and Claude Puech. The 3d visibility complex: A new approach to the problems of accurate visibility. In *Eurographics Rendering Workshop*, pages 245–256, 1996.

[DDP97] Frédo Durand, George Drettakis, and Claude Puech. The visibility skeleton: a powerful and efficient multi-purpose global visibility tool. *Computer Graphics*, 31:89–100, 1997.

[DS96] George Drettakis and François Sillion. Accurate visibility and meshing calculation for hierarchical radiosity. In *Rendering Techniques, 7th EG Workshop on Rendering*, pages 269–278, 1996.

[DS00] J. Dongarra and F. Sullivan. The top ten algorithms. *Computing in Science and Engineering*, 2:22–23, 2000.

[DTG00] Philip Dutre, Parag Tole, and Donald P. Greenberg. Approximate visibility for illumination computation using point clouds. Technical report, Cornell University, 2000.

[DYN04] Yoshinori Dobashi, Tsuyoshi Yamamoto, and Tomoyuki Nishita. Radiosity for point sampled geometry. In *Pacific Graphics*, 2004.

[E.62] Bresenham J. E. Bresenham's line drawing algorithm. 1962.

[EDD03] A. Elgammal, R. Duraiswami, and L. Davis. Efficient kernel density estimation using the fast gauss transform with applications to color modeling and tracking. *IEEE Transactions on PAMI*, 2003.

[GCCCed] Rhushabh Goradia, Anish Chandak, Biswarup Choudary, and Sharat Chandran. Fmm-based illumination maps for point models. In *Was submitted to Symposium on Point Based Graphics (pbg06)*, 2006 (Not Accepted).

[GD98] J. P. Grossman and William J. Dally. Point sample rendering. In *Rendering Techniques*, pages 181–192, 1998.

[GDB03] N. A. Gumerov, R. Duraiswami, and E. A. Borikov. Data structures, optimal choice of parameters, and complexity results for generalized multilevel fast multipole methods in $d$ dimensions. Technical report, Perceptual Interfaces and Reality Laboratory, Institute for Advanced Computer Studies, University of Maryland, College Park, 2003.

[GIM99] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[GKM96] L. Greengard, M. C. Kropinski, and A. Mayo. Integral equation methods for stokes flow and isotropic elasticity. *Journal of Computational Physics*, 125:403–414, 1996.

[Gor06] Rhushabh Goradia. Fmm-based illumination maps for point models. *Second Progress Report, Ph.D.*, 2006.

[GR87] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–348, 1987.

[Gre88] L. Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems*. MIT Press, Cambridge, Massachusetts, 1988.

[GSCH93] Steven J. Gortler, Peter Schröder, Michael F. Cohen, and Pat Hanrahan. Wavelet radiosity. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 221–230, New York, NY, USA, 1993. ACM Press.

[GTG84] C. M. Goral, K. E. Torrance, and D. P. Greenberg. Modeling the interaction of light between diffuse surfaces. *Computer Graphics*, 18(3):213–222, Jul 1984.

[GWS04] Johannes Günther, Ingo Wald, and Philipp Slusallek. Realtime caustics using distributed photon mapping. In *Rendering Techniques*, pages 111–121, jun 2004. (Proceedings of the 15th Eurographics Symposium on Rendering).

[HA05] B. Hariharan and S. Aluru. Efficient parallel algorithms and software for compressed octrees with applications to hierarchical methods. *Parallel Computing*, 31:311–331, 2005.

[Har] M. Harris. Parallel Prefix Sum (Scan) with CUDA. `http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.htm`.

[HAS02] B. Hariharan, S. Aluru, and B. Shanker. A Scalable Parallel Fast Multipole Method for Analysis of Scattering from Perfect Electrically Conducting Surfaces. *Proc. Supercomputing*, page 42, 2002.

[Hau97] A. Haunsner. Multipole expansion of the light vector. *IEEE Transactions on Visualization and Computer Graphics*, 3(1):12–22, Jan-Mar 1997.

[Hec90] P. S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics, ACM Siggraph Conference proceedings*, pages 145–154, 1990.

[HSA91] Pat Hanrahan, David Salzman, and Larry Aupperle. A rapid hierarchical radiosity algorithm. In *Computer Graphics*, volume 25, pages 197–206, 1991.

[Jen96] H. W. Jensen. Global illumination using photon maps. *Eurographics Rendering Workshop 1996*, pages 21–30, June 1996.

[Jen03] Henrik Wann Jensen. Monte carlo ray tracing. *Siggraph Course 4*, pages 15–30, 2003.

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150. ACM Press, 1986.

[KC03] A. Karapurkar and S. Chandran. Fmm-based global illumination for polygonal models. Master's thesis, Indian Institute of Technology, Bombay, 2003.

[KGC04] A. Karapurkar, N. Goel, and S. Chandran. Fmm-based global illumination for polygonal models. *Indian Conference on Computer Vision, Graphics, and Image Processing*, pages 119–125, 2004.

[LHN05] S. Lefebvre, S. Hornus, and F. Neyret. *GPU Gems 2*, chapter Octree Textures on the GPU, pages 595–614. Addison Wesley, 2005.

[LMR07] Lars Linsen, Karsten Muller, and Paul Rosenthal. Splat-based ray tracing of point clouds. *Journal of WSCG, (Proceedings of Fifteenth International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision - WSCG 2007), UNION Agency*, 2007.

[LPC$^+$00] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3D scanning of large statues. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 131–144. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[LTC06] Y. Landa, R. Tsai, and L.T. Cheng. Visibility of point clouds and mapping of unknown environments. In *ACIVS06*, pages 1014–1025, 2006.

[LW85] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical report, University of North Carolina at Chapel Hill, 1985.

[MM02] Vincent C. H. Ma and Michael D. McCool. Low latency photon mapping using block hashing. In *SIGGRAPH/Eurographics Graphics Hardware Workshop*, pages 89–98, 2002.

[Moo93] Andrew W. Moore. An introductory tutorial on kd-trees. *Carnegie Mellon University*, 1993.

[Nie92] H. Niederreiter. Random number generation and quasi-monte carlo methods. *Society for Industrial and Applied Mathematics*, 1992.

[OBA$^+$03] Yutaka Ohtake, Alexander Belyaev, Marc Alexa, Greg Turk, and Hans-Peter Seidel. Multi-level partition of unity implicits. *ACM Trans. Graph.*, 22(3):463–470, 2003.

[OCL96] Ming Ouhyoung, Yung-Yu Chuang, and Rung-Huei Liang. Reusable radiosity object. In *Computer Graphics Forum*, volume 15/3, pages 347–356. Eurographics / Blackwell Publishers, August 1996. ISBN 1067-7055.

[Pau03] Mark Pauly. *Point Primitives for Interactive Modeling and Processing of 3D Geometry*. PhD thesis, ETH Zurich, 2003.

[PD90] Harry Plantinga and Charles R. Dyer. Visibility, occlusion, and the aspect graph. *International Journal of Computer Vision*, 5(2):137–160, 1990.

[PGK02] Mark Pauly, Markus Gross, and Leif P. Kobbelt. Efficient simplification of point-sampled surfaces. In *VIS ’02: Proceedings of the conference on Visualization ’02*, pages 163–170, Washington, DC, USA, 2002. IEEE Computer Society.

[PKKG03] Mark Pauly, Richard Keiser, Leif P. Kobbelt, and Markus Gross. Shape modeling with point-sampled geometry. *ACM Trans. Graph.*, 22(3):641–650, 2003.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 335–342. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[RGed] Sharat Chandran Rhushabh Goradia, Anil Kanakanti. Visibility maps for point models for global illumination. In *Submitted to CGI/EuroGraphics/VRST*, 2007 (Not Accepted).

[RL00] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[Sac64] R. A. Sack. Addition theorems for functions of spherical harmonics. *Journal of Mathematical Physics*, 5(2):245–251, Feb 1964.

[SAF05] Fatih E. Sevilgen, Srinivas Aluru, and Natsuhiko Futamura. Research note: Parallel algorithms for tree accumulations. *J. Parallel Distrib. Comput.*, 65(1):85–93, 2005.

[Sag94] H. Sagan. Space Filling Curves. *Springer-Verlag*, 1994.

[Sam89] H. Samet. Implementing ray tracing with octrees and neighbor finding. *Computers and Graphics*, 13(4):445–460, 1989.

[Sch06] Lars Schjoth. Diffusion based photon mapping. *International Conference on Computer Graphics Theory and Applications GRAPP*, 2006.

[SJ00] G. Schaufler and H. Jensen. Ray tracing point sampled geometry. In *Eurographics Rendering Workshop Proceedings*, pages 319–328, 2000.

[SK98] A. James Stewart and Tasso Karkanis. Computing the approximate visibility map, with applications to form factors and discontinuity meshing. *Eurographics Workshop on Rendering*, pages 57–68, June 1998.

[SP94] F. Sillion and C. Puech. *Radiosity and Global Illumination*. Morgan Kaufmann Publishers, 1994.

[TH93] S. Teller and P. Hanrahan. Global visibility algorithms for illumination computations. In *Proc. of SIGGRAPH-93: Computer Graphics*, pages 239–246, 1993.

[TS91] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics*, 25(4):61–68, 1991.

[Wal05] Ingo Wald. High-Quality Global Illumination Walkthroughs using Discretized Incident Radiance Maps. *Technical Report, SCI Institute, University of Utah, No UUSCI-2005-010 (submitted for publication)*, 2005.

[WS03] Michael Wand and Wolfgang Straer. Multi-resolution point-sample raytracing. *Graphics Interface*, pages 139–148, 2003.

[WS05] Ingo Wald and Hans-Peter Seidel. Interactive Ray Tracing of Point Based Models. In *Proceedings of 2005 Symposium on Point Based Graphics*, 2005.

[ZPKG02] Matthias Zwicker, Mark Pauly, Oliver Knoll, and Markus Gross. Pointshop 3d: An interactive system for point-based surface editing, 2002.

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378, New York, NY, USA, 2001. ACM Press.