Fast, GPU-based Diffuse Global Illumination for Point Models

Fourth Progress Report

Submitted in partial fulfillment of the requirements
for the degree of

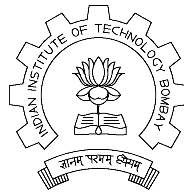
Ph.D.

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 26, 2008

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 Prekshu Ajmera who supported me all through my work. I would also like to thank Prof. Srinivas Aluru, Iowa State University for his useful suggestions on the construction of octrees on the GPU.

This work was funded by an Infosys Ph.D. fellowship grant. I would also like to thank NVIDIA Pune for providing the graphics hardware and support whenever required. Also, I would like to thank the Stanford 3D Scanning Repository as well as Cyberware for freely providing geometric point models to the research community.
Last but not the least, I would like to thank all the friends and members of ViGiL for their valuable support during the work.

Rhushabh Goradia

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. 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. 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. FMM, by itself, exhibits high amount of parallelism to be exploited for achieving multi-fold speed-ups.*

Graphics Processing Units (GPUs), traditionally designed for performing graphics specific computations, now have fostered considerable interest in doing computations that go beyond computer graphics; general purpose computation on GPUs, or "GPGPU". GPUs may be viewed as data parallel compute co-processors that can provide significant improvements in computational performance especially for algorithms which exhibit sufficiently high amount of parallelism. One such algorithm is the Fast Multipole Method (FMM). This report describes in detail the strategies for parallelization of all phases of the FMM and discusses several techniques to optimize its computational performance on GPUs.

The heart of FMM lies in its clever use of its underlying data structure, the Octree. We present two novel algorithms for constructing octrees in parallel on GPUs and discuss their performance based on memory efficiency and running time. These algorithms can eventually be combined with the GPU-based parallel FMM framework.

Correct global illumination results for point models require knowledge of mutual point-pair visibility. ***Visibility Maps (V-maps)*** *have been designed for the same. Parallel implementation of V-map on GPU offer considerable performance improvements and has been detailed in this 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. 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.

Contents

1 Introduction **1**

1.1 Point Based Modelling and Rendering 2

1.2 Global Illumination 3

1.2.1 Diffuse and Specular Inter-reflections 5

1.3 Fast computation with Fast Multipole Method 7

1.4 Parallel computations using the GPU 8

1.5 Octrees and FMM 10

1.5.1 Octrees 10

1.5.2 Spatial Locality Based Domain Decomposition 11

1.5.3 Visibility between Point Pairs 12

1.6 Problem Definition and Contributions 13

1.7 Overview of the Report 13

2 General Purpose Computations on GPU (GPGPU) **15**

2.1 Programming a GPU for General Purpose Computations 15

2.2 NVIDIA CUDA Programming Model 16

2.3 GPU Program Optimization Techniques 18

3 Parallel FMM on the GPU **21**

3.1 Fast computation with Fast Multipole Method 21

3.2 Prior Work 23

3.2.1 Direct N-Body Simulations on the GPU . . . 23
3.3 Parallel FMM computations on GPU . . . 25
3.4 Implementation Details . . . 25
3.4.1 Upward Pass . . . 26
3.4.2 Downward Pass . . . 29
3.4.3 Quality Comparisons . . . 32
3.4.4 Timing Comparisons . . . 33

4 Space Filling Curves **35**

5 Octrees **39**
5.1 Octrees: Introduction . . . 39
5.1.1 Non-Adaptive and Adaptive Octrees . . . 39
5.2 Prior Work: Octree Construction on the GPU . . . 40
5.2.1 Problems . . . 43
5.3 Octree on the GPU . . . 43
5.3.1 Implementation 1: Non-Adaptive Octree using Direct Indexing [AGCA08] . . . 43
5.3.2 Implementation 2:
Parallel Memory Efficient Bottom-Up Adaptive Octree . . . 48
5.3.3 Implementation details . . . 52
5.3.4 Implementation 3:
Parallel Memory Efficient Top-Down Adaptive Octree . . . 61
5.3.5 Comparison between Implementation 1 and Implementations 2/3 . . . 65
5.3.6 GPU Optimizations . . . 65
5.4 Results . . . 66

6 View Independent Visibility using V-map on GPU **67**
6.1 Prior Work: CPU-based V-map Construction [Gor07] . . . 67
6.1.1 Visibility Maps . . . 68
6.1.2 Point–Pair Visibility Algorithm . . . 69
6.1.3 Octree Depth Considerations . . . 70
6.1.4 Construction of Visibility Maps . . . 71
6.2 GPU-based V-map Construction . . . 73
6.3 The Visibility Map . . . 74

6.4 V-map Computations on GPU . . . 76
6.4.1 Multiple Threads Per Node Strategy . . . 76
6.4.2 One Thread per Node Strategy . . . 77
6.4.3 Multiple Threads per Node-Pair . . . 77
6.5 Leaf-Pair Visibility . . . 78
6.5.1 Prior Algorithm . . . 79
6.5.2 Computing Potential Occluders . . . 79
6.6 GPU Optimizations . . . 80
6.7 Results . . . 81
6.7.1 Visibility Validation . . . 81
6.7.2 Quantitative Results . . . 82

7 Discussion: Specular Inter-reflections and Caustics in Point based Models 85

7.1 Introduction . . . 85
7.2 Photon Mapping . . . 86
7.2.1 Photon Tracing (First Pass) . . . 86
7.2.2 Preparing the Photon Map for Rendering . . . 88
7.2.3 Rendering (Second Pass) . . . 89
7.2.4 Radiance Estimate . . . 91
7.2.5 Limitations of Photon Mapping . . . 91
7.3 Our Approach . . . 92
7.3.1 Splat-Based Ray Tracing . . . 93
7.3.2 Ray Tracing . . . 96
7.3.3 Optimizing Photon Generation and Sampling . . . 98
7.3.4 Optimized Photon Traversal and Intersection tests . . . 99
7.3.5 Fast Photon Retrieval using Optimized kNN Query Algorithm . . . 99

8 Conclusion and Future Work 103

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 Point Model Representation. Explicit structure of points for bunny is visible. Figure on extreme right shows the same bunny with continuous surface constructed 2

1.3 Example of Point Models 3

1.4 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. 4

1.5 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? . . . 4

1.6 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". 5

1.7 Specular (Regular) and Diffuse Reflections 6

1.8 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 . 6

1.9 GPUs are fast and getting faster [OLG$^+$07] 9

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

1.11 Example showing importance of visibility calculations between points [GKCD07] 12

2.1 Hardware model of Nvidia's G80/G92 GPU 16

2.2 Each kernel is executed as a batch of threads organized as a grid of thread blocks 17

3.1 A grid of thread blocks that calculates all N^2 forces. Here there are four thread blocks with four threads each [NHP07]. 24

3.2 Top Left: A Cornell Room with the Ganesha's point model on CPU. Top Right: Corresponding GPU result. Bottom Left: A Cornell Room with the Bunny's point model on CPU. Bottom Right: Corresponding GPU result. Both the results assume 50 points per leaf. 32

3.3 Point models rendered with diffuse global illumination effects of color bleeding and soft shadows. Pair-wise visibility information is essential in such cases. Note that the Cornell room as well as the models in it are input as point models. 33

3.4 FMM Upward Pass : Bunny with 124531 points 33

3.5 FMM Upward Pass : Ganpati with 165646 points 34

3.6 Downward Pass (a) Bunny with 124531 points (b) Ganpati with 165646 points 34

4.1 Left: The 2-D z-SFC curve for $k = 3$, Right: 10 points in a 2-D space. The points are sequentially labeled in the z-SFC order. 35

4.2 Octree can be viewed as multiple SFCs at various resolutions 37

4.3 Bit interleaving scheme for a hierarchy of cells 37

5.1 A quadtree built on a set of 10 points in 2-D. 40

5.2 Storage in texture memory. The data pool encodes the tree. Data grids are drawn with different colors. The grey cells contain data. 41

5.3 Lookup for point M in the octree 41

5.4 At each step the value stored within the current node's data grid is retrieved. If this value encodes an index, the lookup continues to the next depth. Otherwise, the value is returned. . . 42

5.5 Hierarchy of cells in two dimensions. Gray cells indicate data. 44

5.6 One pass of the algorithm. Threads are at level 1 and nodes at level 2 44

5.7 The constructed octree 45

5.8 Table defining total number of nodes for trees of different height 47

5.9 Calculation of Postorder number of a node 47

5.10 Parallel Bottom Up Adaptive Octree Implementation 48

5.11 A 2D particle domain and corresponding quadtree and compressed quadtree 50

5.12 Two cases of the compressed octree construction 52
5.13 Computing Parent and Child 57
5.14 Compressed Octree to Octree 59
5.15 Spatial Clustering of Points 61
5.16 Spatial Clustering of Points 62
5.17 Partition Array of a Node 64
5.18 Top-Down Octree Construction (Bunny 124531 points) (sec. 5.3.4) 66
5.19 Top-Down Octree Construction (Ganpati 165646 points) (sec. 5.3.4) 66

6.1 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. 68
6.2 Leaf nodes (or cells, or voxels) are at level three. 69
6.3 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 $\overline{pq}$. 70
6.4 Point-Point visibility is obtained by performing a number of tests. Now its extended to Leaf-Leaf visibility 71
6.5 Dragon viewed from the floor (cyan dot). The quality is unacceptable for octrees of heights of 7 (left) or less. The figure on the right is for an octree of height 9. 74
6.6 Visibility links for Node N 75
6.7 The visibility map is constructed recursively by a variation of depth first search. In general, it is advantageous to have links at high level in the tree so that we can reason efficiently using coherency about the visibility of a group of points. 75
6.8 Parallelism at a node level 76
6.9 Parallelism across nodes at the same level 77

6.10 The visibility map on the GPU uses thousands of threads concurrently by working at the large number of leaves (a) and stores the result in a table. The links at other levels are set based on a lookup computation. 78
6.11 Visibility between points p and q 78
6.12 (a) Constructing parent sphere from child, (b) Line Segment-Sphere Intersection test 80
6.13 Visibility tests where purple color indicates portions visible to the candidate eye (in cyan) . . . 82
6.14 V-map construction times (CPU & GPU) for models with differing octree heights 83
6.15 Dragon viewed from the floor (cyan dot). The quality is unacceptable for octrees of heights of 7 (left) or less. The figure on the right is for an octree of height 9. 83
6.16 Point models rendered with diffuse global illumination effects of color bleeding and soft shadows. Pair-wise visibility information is essential in such cases. Note that the Cornell room as well as the models in it are input as point models. 84

7.1 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. 88
7.2 Building (a) the caustics photon map and (b) the global photon map. 89
7.3 Example output of Photon Mapping Algorithm [Jen96] showing reflection, refractions and caustics 92
7.4 (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 δ_ϵ 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 $\mathbf{n}_i$. 94
7.5 (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. 97

7.6 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. 101

Chapter 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).

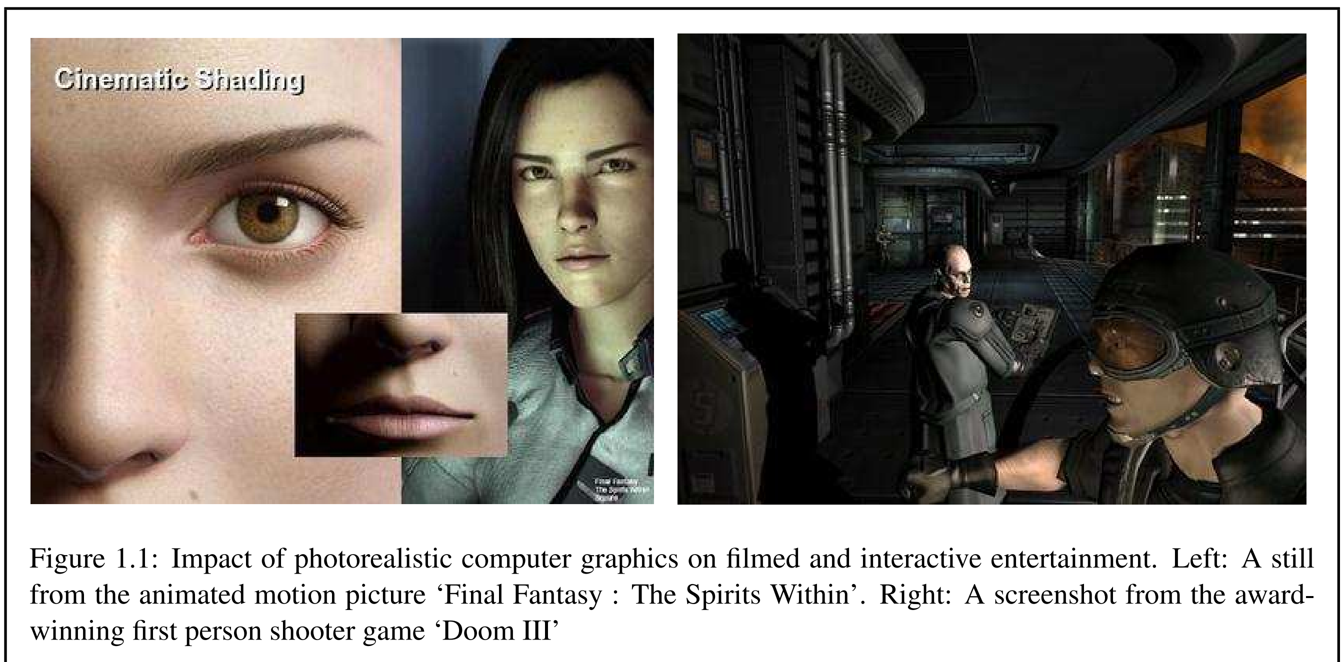


Figure 1.1: Impact of photorealistic computer graphics on filmed and interactive entertainment. Left: A still from the animated motion picture ‘Final Fantasy : The Spirits Within’. Right: A screenshot from the award-winning first person shooter game ‘Doom III’

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. Graphics Processing Units (GPUs) have been used for increased speed-ups.
Before moving further, let us be familiar with the terms point models and global illumination.

1.1 Point Based Modelling and Rendering

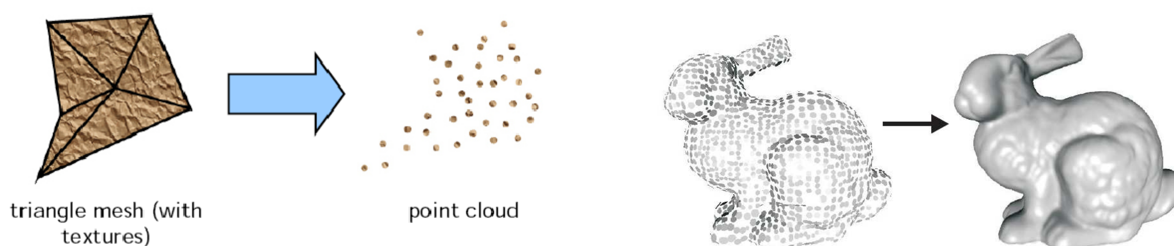


Figure 1.2: Point Model Representation. Explicit structure of points for bunny is visible. Figure on extreme right shows the same bunny with continuous surface constructed

Point models are nothing but a discrete representation of a continous surface i.e. we model each point as a surface sample representation (Fig 1.2). There is no connectivity information between points. Each point has certain attributes, for example co-ordinates, normal, reflectance, emmisivity values.

Figure 1.3: 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 3.3 shows some example point based models.

1.2 Global Illumination

Local illumination refers to the process of a light source illuminating a surface through direct interaction. However, the illuminated surface now itself acts as a light source and propagates light to other surfaces in the environment. Multiple bounces of light originating from light sources and subsequently reflected throughout the scene lead to many visible effects such as soft shadows, glossy reflections, caustics and color bleeding. The whole process of light propagating in an environment is called Global Illumination and to simulate this process to create photorealistic images of virtual scenes has been one of the enduring goals of computer graphics. More formally,

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

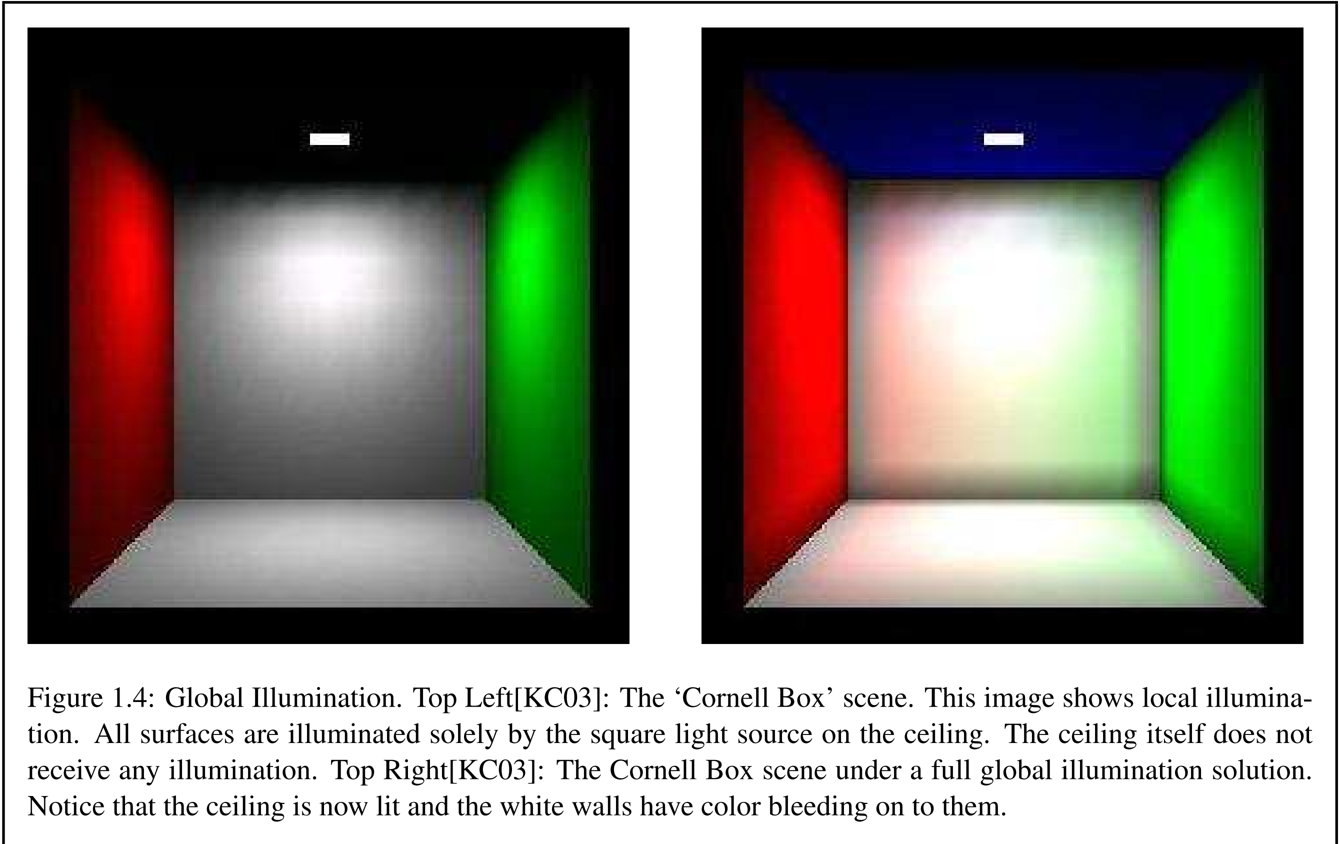


Figure 1.4: Global Illumination. Top Left[KC03]: The 'Cornell Box' scene. This image shows local illumination. All surfaces are illuminated solely by the square light source on the ceiling. The ceiling itself does not receive any illumination. Top Right[KC03]: The Cornell Box scene under a full global illumination solution. Notice that the ceiling is now lit and the white walls have color bleeding on to them.

light which has undergone reflection from other surfaces in the world (indirect illumination).

Figures 1.4 and 1.6 gives you some examples images showing the effects of *Global illumination*. It is a simulation of the physical process of light transport.

Figure 1.5: 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?

Three-dimensional scanned *point models* of cultural heritage structures (Figure 1.5) 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 entities.

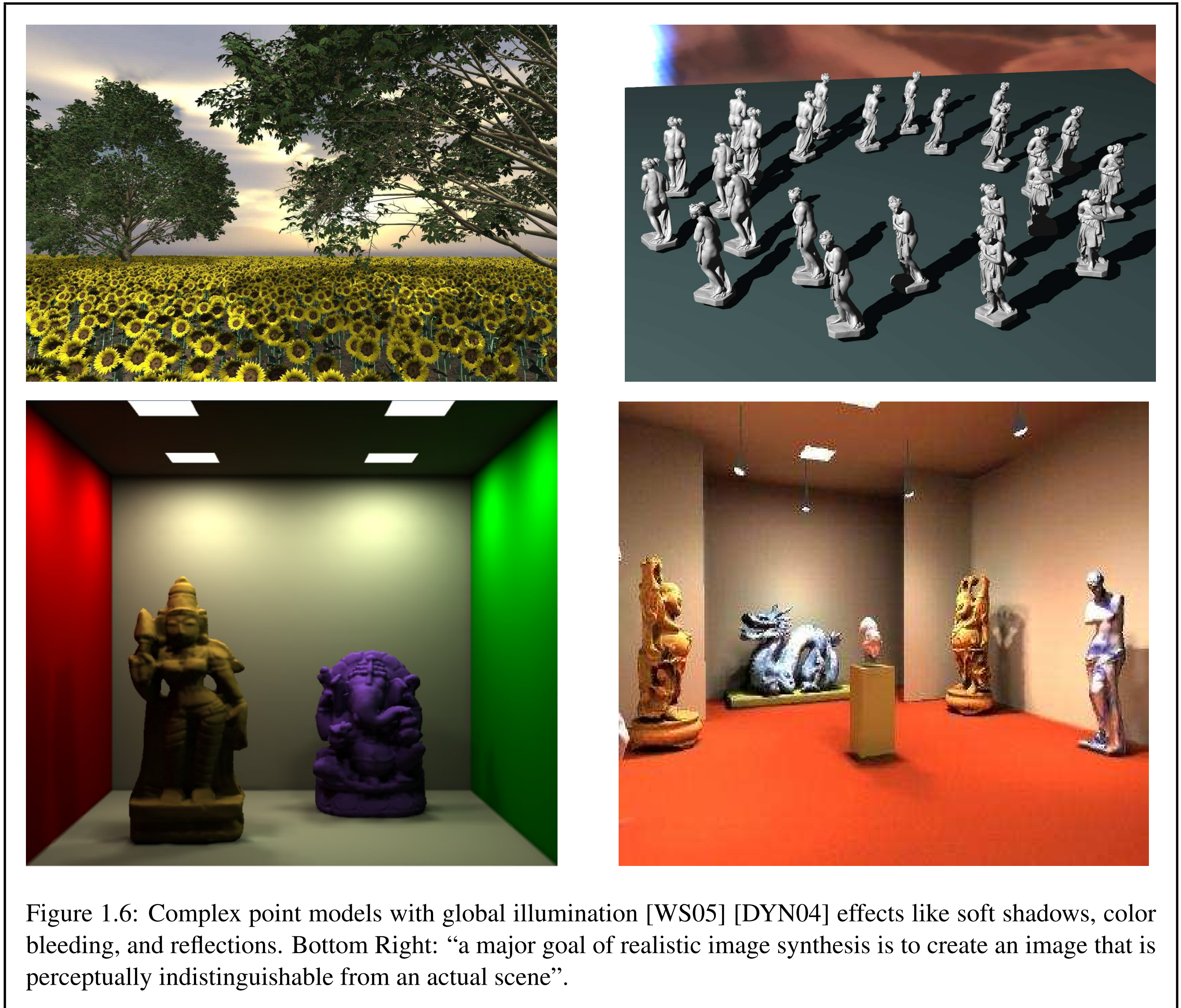


Figure 1.6: Complex point models with global illumination [WS05] [DYN04] effects like soft shadows, color bleeding, and reflections. Bottom Right: "a major goal of realistic image synthesis is to create an image that is perceptually indistinguishable from an actual scene".

Global Illumination effects are the results of two types of light reflections and refractions, namely *Diffuse and Specular*.

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 (2π steradians) i.e. they reflect light equally in all directions.

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

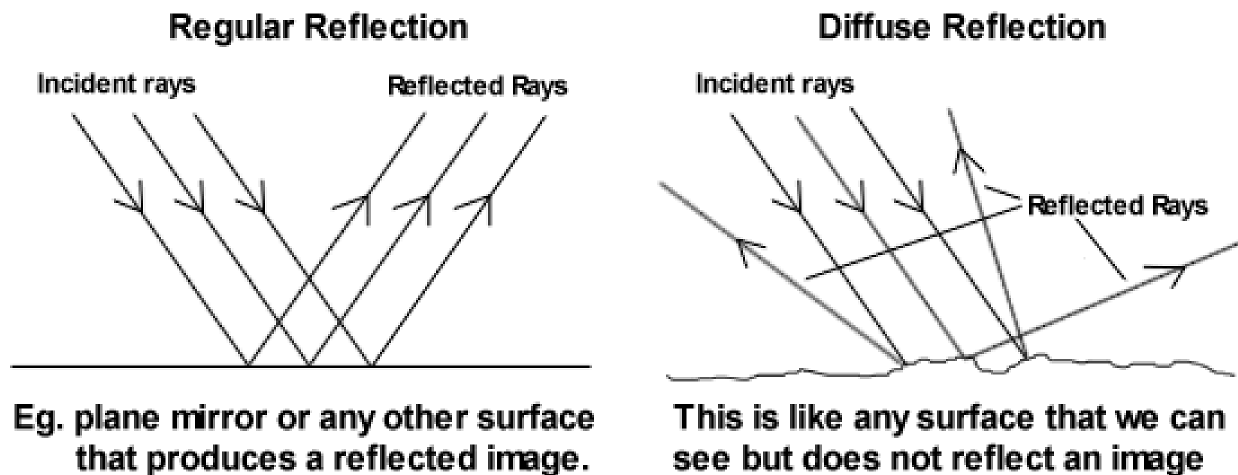


Figure 1.7: Specular (Regular) and Diffuse Reflections

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 $\theta_i = \theta_r$.

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.

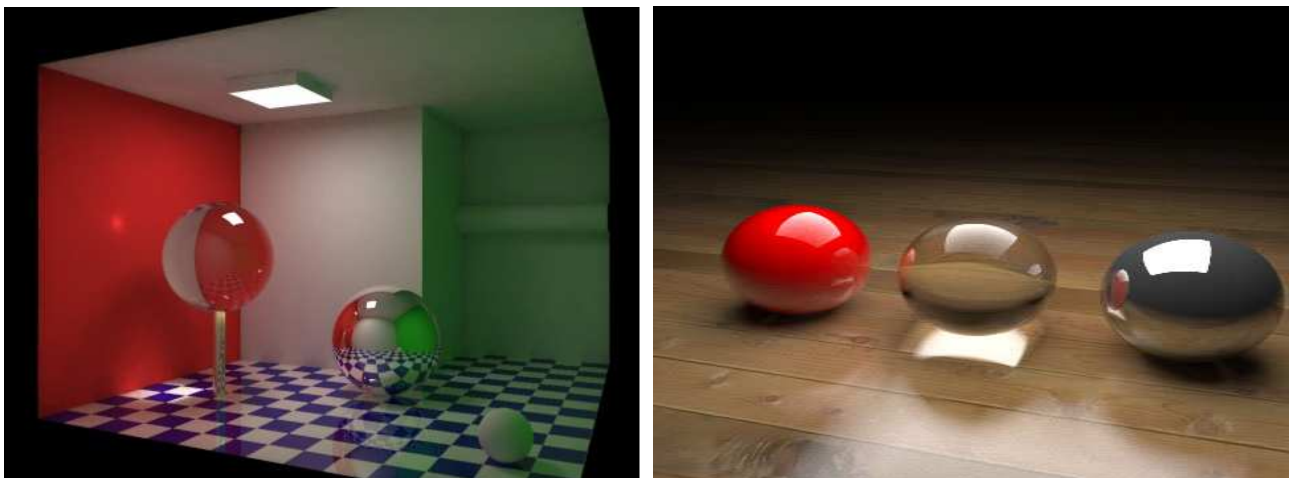


Figure 1.8: Left: Colors transfer (or "bleed") from one surface to another, an effect of diffuse inter-reflection. Also notable is the caustic projected on the red wall as light passes through the glass sphere. Right: Reflections and refractions due to the specular objects are clearly evident

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 1.8 shows color bleeding and specular inter-reflections including caustics. *Radiosity* and *Ray-Tracing* are two basic global illumination algorithms used for diffuse and specular effects generation (respectively).

There have been two main approaches to solve the global illumination problem - Finite Element and Monte Carlo. In the finite element approach to solve the Global Illumination problem the whole scene is decomposed into primitive elements such as triangles and light transport is simulated between these elements. The Monte Carlo approach is equivalent to tracing light rays emanating from the source and their subsequent reflections/refractions before they reach the eye. Subset of the global illumination problem in which all surfaces are diffuse assumes great importance in applications such as architectural walkthroughs since the illumination has to be computed exactly once for any view.

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 1.5) 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. We [GKCD07] provided an efficient solution to the above mentioned problem on the CPU. We used a FMM-based radiosity kernel to provide a global illumination solution to any input scene given in terms of points.

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, to a fixed prescribed accuracy ϵ when direct multiplication requires $O(N^2)$ operations. Global illumination problem requires the computation of pairwise interactions among each of the surface elements (points or triangles) in the given data (usually of order $> 10^6$) and thus naturally fits in the FMM framework.

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. We exploit the inherent parallelism of this method to implement it on the data parallel architecture of the GPU to achieve multifold speedups. Further, the same parallel implementation on the GPU, designed for point models, can also be used for triangular models.

1.4 Parallel computations using the GPU

The graphics processor (GPU) on today's video cards has evolved into an extremely powerful and flexible processor. The latest GPUs have undergone 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. NVIDIA's CUDA [CUDa] programming environment offers the familiar C-like syntax which makes programs simpler and easier to build and debug. CUDA's programming model allows its users to take full advantage of the GPU's powerful hardware but also permits an increasingly high-level programming model

that enables productive authoring of complex applications. The result is a processor with enormous arithmetic capability [a single NVIDIA GeForce 8800 GTX can sustain over 330 giga-floating-point operations per second (Gflops)] and streaming memory bandwidth (80+ GB/s), both substantially greater than a high-end CPU.

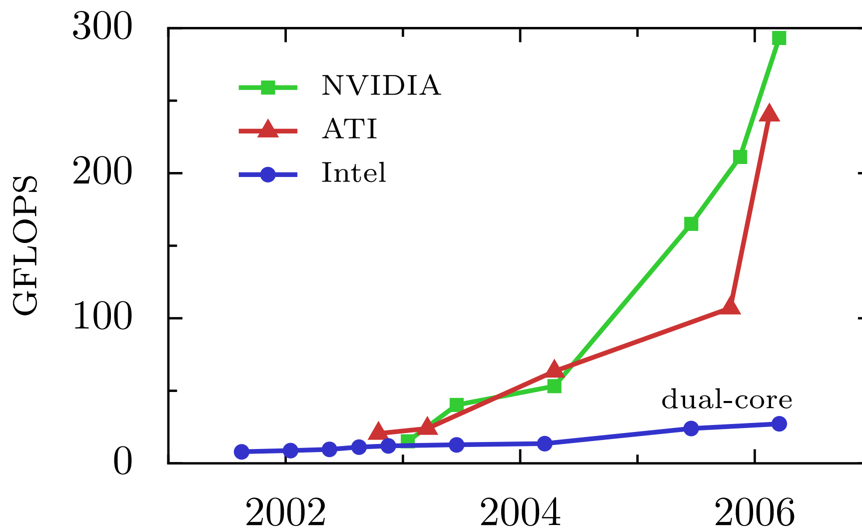


Figure 1.9: GPUs are fast and getting faster [OLG+07]

Architecturally, GPUs are highly parallel streaming processors optimized for vector operations. The programmable units of the GPU follow a single instruction multiple-data (SIMD) programming model. For efficiency, the GPU processes many elements in parallel using the same program (kernel). Each element is independent from the other elements, and in the base programming model, elements cannot communicate with each other. All GPU programs must be structured in this way: many parallel elements, each processed in parallel by a single program. Each element can operate on 32-bit integer or floating-point data with a reasonably complete general-purpose instruction set. Elements can read data from a shared global memory (a “gather” operation) and, with the newest GPUs, also write back to arbitrary locations in shared global memory (“scatter”).

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 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, finance etc.

The GPU is designed for a particular class of applications which give more importance to throughput than latency and have large computational requirements and offer substantial parallelism. Many specific algorithms like bitonic sorting, parallel prefix, 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 compute capabilities of the GPUs is parallel Fast Multipole Method. FMM, if divided at a high level, consists of five sequential passes:

1. Octree Construction
2. Interaction List Construction
3. Upward pass on the Octree
4. Downward pass on the Octree
5. Final Summation of Energy

Upward Pass, Downward pass and Final Summation stages are the ones which take more than 97% of the run time. Hence we implemented these 3 stages on the GPU while the Octree Construction and Interaction List Construction stages were performed on the CPU. These will eventually be implemented on GPU as well. We have used the latest Nvidia's G80/G92 architechture GPUs with CUDA as the programming environment.

1.5 Octrees and FMM

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, to a fixed prescribed accuracy ϵ when direct multiplication requires $O(N^2)$ operations. This is mainly possible because of the underlying hierarchical data structure, the *octree*.

1.5.1 Octrees

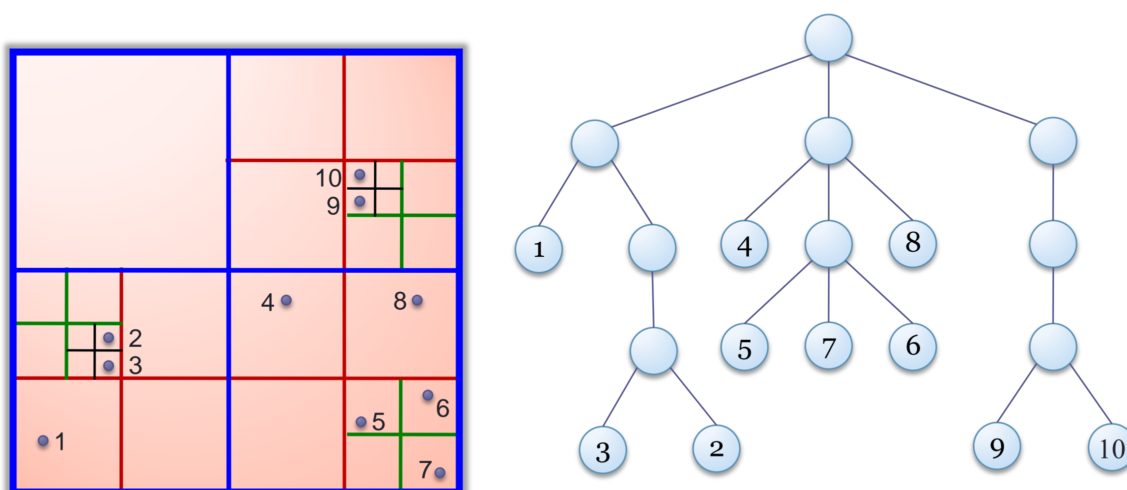


Figure 1.10: A quadtree built on a set of 10 points in 2-D.

Octrees are hierarchical tree data structures that organize multidimensional data using a recursive decomposition of the space containing them. Such a tree is called a *quadtree* in two dimensions, *octree* in three dimensions and *hyperoctree* in higher dimensions. Octrees can be differentiated on the basis of the type of data they are used to represent, the principle guiding the decomposition and the resolution which can be fixed or variable. In practice, the recursive subdivision is stopped when a predetermined resolution level is reached, or when the number of points in a subregion falls below a pre-established constant. This results in the formation of an *adaptive octree*. An example is shown in figure 1.10. In this report we present two novel algorithms for constructing octrees in parallel on GPUs, one based on using parallel Space Filling Curve (SFC) for a bottom-up octree construction and other on spatial clustering of points for a top down construction (ch. 5).

These parallel octree construction algorithms could potentially be combined with the parallel FMM implementation on the GPU.

1.5.2 Spatial Locality Based 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. Octrees can be thought of as one of the domain decomposition methods.

Domain decomposition is the first step in many scientific computing applications. Computations within any subregion often require information from other, mostly adjacent, subregions of the domain. Whenever information from neighboring elements is not locally available, processors need to communicate to access information. As communication is significantly slower than computation, domain decomposition methods attempt to minimize inter-processor communications and, in fact, try to overlap computation and communication for even better performance.

Another important goal is to achieve load balance. The load on a processor refers to the amount of computation that it is responsible for. Achieving load balance while simultaneously minimizing communication is often non-trivial. This stems from the fact that the input data need not necessarily be uniformly distributed in the underlying domain of the problem. Moreover, the type of data accesses required may also vary from application to application. Thus, designing a domain decomposition method that simultaneously optimizes these can be challenging.

Spatial locality based domain decomposition methods make particular sense for particle-based methods. In these methods, particles interact with other particles, often based on spatial locality, providing a direct justication for parallel domain decomposition methods. The Fast Multipole Method is one such example. Such description of interactions using geometric constraints particularly suits spatial locality based parallel domain

decomposition methods.

As far as the runtime of the domain decomposition algorithm is concerned, spatial locality based domain decomposition methods have a particular advantage because of the simplicity of the underlying model of partitioning multidimensional point data. It is also easy to parallelize the decomposition algorithm itself (eg. octrees), which is useful in reducing the run-time overhead along with the remaining application (eg. FMM for global illumination) run-time, when scaling to larger and larger systems. Thus, an intellegent, parallel octree construction algorithm, satisfying above constraints, and its application to parallel FMM on GPUs is interesting.

1.5.3 Visibility between Point Pairs

Figure 1.11: Example showing importance of visibility calculations between points [GKCD07]

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. For example, in Fig. 1.11, shadows wouldn't have been possible if there wasn't any visibility information. 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**. We provided a view-

independent visibility solution for global illumination for point models in [GKCD07][Gor07] using Visibility Map (V-map). However, this CPU-based sequential implementation of V-map takes considerable amount of time and hence not very useful for practical applications. We exploit the inherent parallelism in the V-map construction algorithm and attempt to make it work faster with multi-fold speed-ups. Parallel implementation of V-map on GPU [GAC08] offer considerable performance improvements (in terms of *speed*) and has been detailed in this report.

1.6 Problem Definition and Contributions

After getting a brief overview of the topics, let us now define the problem we pose and what all we have contributed till now.

Problem Definition: *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 **(Done)**.
- Inter-reflection effects include both diffuse **(Done)** and specular effects like reflections, refractions, and caustics. Capturing specular reflections is a part of work to be done in the coming year, which essentially, when combined with the diffuse inter-reflection implementation, will give a complete global illumination package for point models.
- We compute diffuse inter-reflections using the **Fast Multipole Method**(FMM) **(Done)**.
- Parallel implementation of visibility and FMM algorithms on Graphics Processing Units(GPUs) so as to achieve speedups for generating the global illumination solution **(Done)**.
- Have a parallel octree construction algorithm which could be potentially combined with a parallel FMM algorithm on future GPUs **(Done)**.

1.7 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. Chapter 2 gives an overview of modern day GPUs and presents several techniques to optimize the performance of the computations that run on the GPU. Chapter 3 presents an introduction to the FMM algorithm for Radiosity Kernel and our parallel implementation of the same on the

GPU. We provide a step by step overview of different kernel functions for each phase of the FMM algorithm along with efficient speed results. In chapter 4 we then focus on a spatial locality based parallel domain decomposition method: Space Filling Curves and their usefulness for constructing parallel octrees. We also discuss how SFC linearization across multiple levels of an octree can give us its postorder traversal. We then move on to different parallel octree implementations on GPU in chapter 5. These implementations can be combined with the parallel, GPU-based FMM algorithm. Chapter 6 discusses our GPU-based, parallel V-map construction algorithm and reports multi-fold speed-ups. Chapter 7 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. Finally, chapter 8 summarizes the work done in the course of this year and outlines possible avenues for future research along these lines.

Chapter 2

General Purpose Computations on GPU (GPGPU)

We begin by describing the way modern GPU applications are written for general purpose computations. After this we give a brief overview of the NVIDIA's Compute Unified Device Architecture or CUDA [CUDa] programming model which we use in our GPU-based implementations. At the end we discuss various programming techniques to optimize the computational performance on GPU and get a better run time.

2.1 Programming a GPU for General Purpose Computations

One of the historical difficulties in programming GPGPU applications has been that despite their general-purpose tasks having nothing to do with graphics, the applications still had to be programmed using graphics APIs. In addition, the program had to be structured in terms of the graphics pipeline, with the programmable units only accessible as an intermediate step in that pipeline, when the programmer would almost certainly prefer to access the programmable units directly. This difficulty is overcome with the evolution of programming environments like NVIDIAs CUDA [CUDa], which provide a more natural, direct, nongraphics interface to the hardware and, specifically, the programmable units. Today, GPU computing applications are structured in the following way.

1. The programmer directly defines the computation domain of interest as a structured grid of threads.
2. Each thread runs a SIMD general-purpose program and computes the value.
3. The value for each thread is computed by a combination of math operations and both read accesses from and write accesses to global memory. The same buffer can be used for both reading and writing, allowing more flexible algorithms (for example, in-place algorithms that use less memory).
4. The resulting buffer in global memory can then be used as an input in future computation.

This programming model is a powerful one for several reasons. First, it allows the hardware to fully exploit the applications data parallelism by explicitly specifying that parallelism in the program. Next, it strikes a careful balance between generality (a fully programmable routine at each element) and restrictions to ensure good performance (the SIMD model, the restrictions on branching for efficiency, restrictions on data communication between elements and between kernels/passes, and so on). Finally, its direct access to the programmable units eliminates much of the complexity faced by previous GPGPU programmers in coopting the graphics interface for general-purpose programming. NVIDIA's CUDA [CUDa] programming environment offers the familiar C-like syntax which makes programs simpler and easier to build and debug. CUDA's programming model allows its users to take full advantage of the GPUs powerful hardware but also permits an increasingly high-level programming model that enables productive authoring of complex applications.

2.2 NVIDIA CUDA Programming Model

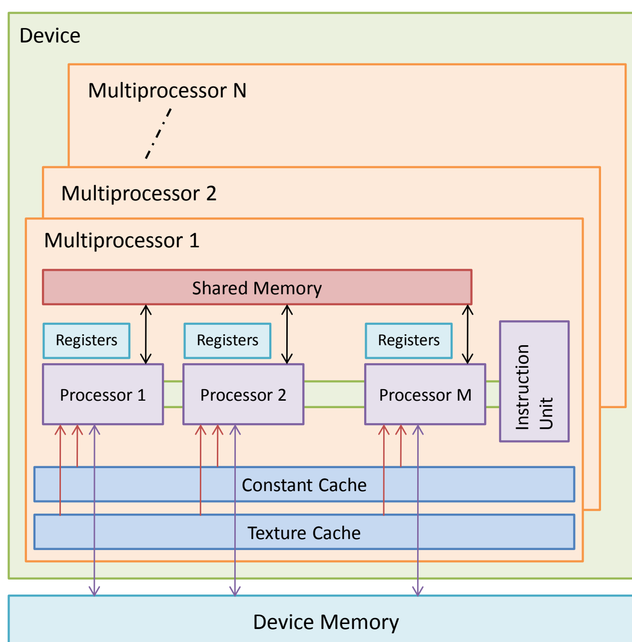


Figure 2.1: Hardware model of Nvidia's G80/G92 GPU

The NVIDIA G80 GPU, is the current generation of NVIDIA GPU, and has also been released as the Tesla compute coprocessor. It consists of a set of multiprocessors (16 on our GeForce 8800GT), each composed of 8 processors. All multiprocessors talk to a global device memory, which in the case of our GPU is 512 MB, but can be as large as 1.5 GB for more recently released GPUs/coprocessors. The 8 processors in each multiprocessor share 16 kB local read-write "shared" memory, a local set of 8192 registers, and a constant memory of 64 kB over all multiprocessors, of which 8 kB can be cached locally at one multiprocessor.

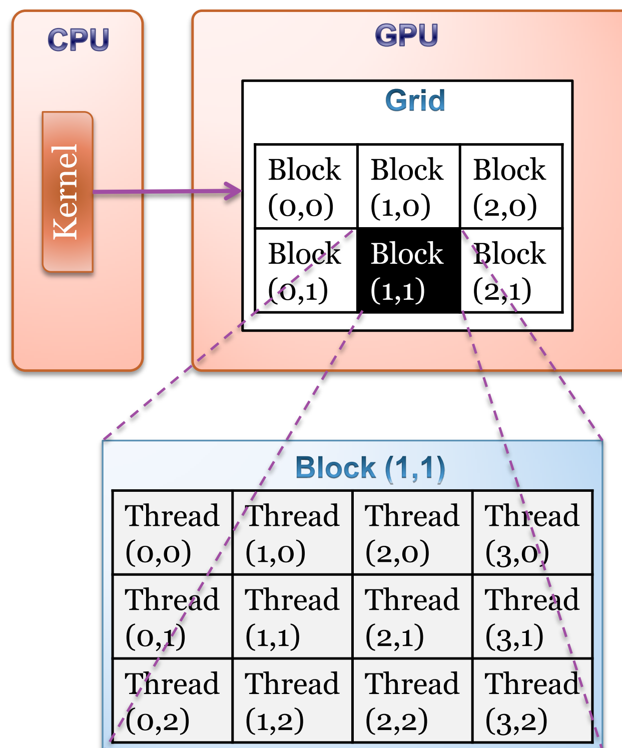


Figure 2.2: Each kernel is executed as a batch of threads organized as a grid of thread blocks

A programming model (CUDA) and a C compiler (with language extensions) that compiles code to run on the GPU are provided by NVIDIA. This model is supposed to be extended over the next few generations of processors, making investment of effort on programming it worthwhile. Under CUDA the GPU is a compute device that is a highly multithreaded coprocessor. Thus, in an application any computation that is done independently on different data many times, can be isolated into a function called *kernel* that is executed on the GPU as many different threads. The batch of threads that executes a kernel is organized as a grid of thread blocks (see Figure 2.2). Each thread block is itself a grid of threads that executes on a multiprocessor that have access to its local memory. They can cooperate together by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses. They perform their computations and become idle when they reach a synchronization point, waiting for other threads in the block to reach the synchronization point. Each thread is identified by its thread ID (one, two or three indices). The choice of 1,2 or 3D index layout is used to map the different pieces of data to the thread. The programmer writes data parallel code, which executes the same instructions on different data, though some customization of each thread is possible based on different behaviors depending on the value of the thread indices. A GPU may run all the blocks of a grid sequentially if it has very few parallel capabilities, or in parallel if it has a lot of parallel capabilities.

To achieve efficiency on the GPU, algorithm designers must account for the substantially higher cost (two

orders of magnitude higher) to access fresh data from the GPU main memory. This penalty is paid for the first data access, though additional contiguous data in the main memory can be accessed cheaply after this first penalty is paid. An application that achieves such efficient reads and writes to contiguous memory is said to be *coalesced*. Thus programming on the nonuniform memory architecture of the GPU requires that each of the operations be defined in a way that ensures that main memory access (reads and writes) are minimized, and coalesced as far as possible when they occur. For further reference look at NVIDIA CUDA Programming Guide [CUDa].

2.3 GPU Program Optimization Techniques

In this section we look at several techniques to optimize the performance of a program running on the GPU. Though all of them may not be easy to apply in a particular problem, they are all worth experimenting with. It is always beneficial to first understand them properly and then design a GPU program to the problem under consideration.

1. **Expose As Much Parallelism As Possible (GPU Thread Parallelism)**: Structure the algorithm to maximize independent parallelism. If threads of same block need to communicate, use shared memory and synchronize the threads using __syncthreads() function of CUDA. If threads of different blocks need to communicate, use global memory and split computation into multiple kernels. Note that there is no synchronization mechanism between blocks. High parallelism is especially important to hide memory latency by overlapping memory accesses. Once the parallelism of the algorithm has been exposed it needs to be mapped to the hardware as efficiently by carefully choosing the execution configuration of each kernel invocation (refer to the NVIDIA CUDA Programming Guide [CUDa] for more details).

2. **Expose As Much Parallelism As Possible (CPU/GPU Parallelism)**: Take advantage of asynchronous kernel launches by overlapping CPU computations with kernel execution. One can also take advantage of the asynchronous CPU-GPU memory transfers that overlap with kernel execution.

3. **Optimize Memory Usage For Maximum Bandwidth**: Processing data is cheaper than moving it around, especially for GPUs as they devote many more transistors to ALUs than memory. Thus, the less memory bound a kernel is, the better it will scale with future GPUs. Therefore, we should try to maximize the use of low-latency, high-bandwidth memory. We should optimize memory access patterns to maximize bandwidth. Leverage parallelism to hide memory latency by overlapping memory accesses with computation as much as possible. This means that the kernels should posses high arithmetic intensity (ratio of math to memory transactions). Sometimes, recomputing data can prove to be better than simply caching it.

4. **Minimize CPU-GPU Data Transfers**: CPU-GPU memory bandwidth is much lower than GPU memory bandwidth. Use page-locked host memory for maximum CPU-GPU bandwidth (3.2 GB/s common on PCI-e x16, 4 GB/s measured on nForce 680i motherboards, 8GB/s for PCI-e 2.0). However, one has to be cautious since allocating too much page-locked memory can reduce overall system performance. Minimize CPU-GPU data transfers by moving more code from CPU to GPU even if that means running kernels with low parallelism computations. Intermediate data structures can be allocated in device memory, operated on, and deallocated without ever copying them to CPU memory. Transfering data in group is also useful. Because of the overhead associated with each transfer, one large transfer is much better than many small ones.

5. **Optimize Memory Access Patterns**: Effective bandwidth can vary by an order of magnitude depending on the type of access pattern. Optimize access patterns to get coalesced global memory accesses, shared memory accesses with no or few bank conflicts, cache-efficient texture memory accesses and same-address constant memory accesses.

   Global memory is not cached on NVIDIA G80 cards. Therefore, when accessing global memory, there are, in addition, 400 to 600 clock cycles of memory latency which is highest for any type of memory accesses. Moreover, the device memory (global memory) has much lower bandwidth than on-chip memory. Thus, it is likely to be a performance bottleneck. Global memory bandwidth is used most efficiently when the simultaneous memory accesses by threads in a half-warp can be coalesced into a single memory transaction of 64 bytes, or 128 bytes. Using float4 (instead of float3) data allows coalesced memory access to the arrays of data in device memory, resulting in efficient memory requests and transfers. If register space is an issue, then store the three-dimensional vectors as float3 variables.

   The constant memory is cached so a read from constant memory costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the constant cache. For all threads of a warp, reading from the constant cache is as fast as reading from a register as long as all threads read the same address. The cost scales linearly with the number of different addresses read by all threads.

   The shared memory is hundreds of times faster than the local and global memory since it is present on the chip itself. Thus, the use of shared memory within kernels should be maximized. In fact, for all threads of a warp, accessing the shared memory is as fast as accessing a register as long as there are no bank conflicts between the threads. Shared memory is divided into equally sized n banks. Thus, an effective bandwidth of n times the bandwidth of a single bank can be achieved if any memory read or write request made of n addresses fall in n distinct memory banks. In case of bank conflict the access has to be serialized. The G80 GPU architecture also supports concurrent reads from multiple threads to

a single shared memory address, so that there are no shared-memory bank conflicts.

A common way of scheduling some computation on the device is to block it up to take advantage of shared memory. First partition the data set into data subsets that fit into shared memory and then handle each data subset with one thread block. Load the subset from global memory to shared memory and synchronize the threads so that each thread can safely read shared memory locations that were written by different threads. Now, perform the computation on the subset from shared memory because each thread can efficiently multi-pass over any data element. Again synchronize the threads if required to make sure that shared memory has been updated with the results and copy the results back to the global memory.

The texture memory is also cached so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance.

6. **Optimize Instruction Usage**: The use of arithmetic instructions with low throughput should be minimized to optimize instruction usage. This includes trading precision for speed when it does not affect the end result, such as using intrinsic (for e.g. __sinx()) instead of regular functions (for e.g. sinx()) or single-precision instead of double-precision. Particular attention must be paid to control flow instructions due to the SIMD nature of the GPU. Any flow control instruction (if, switch, do, for, while) can significantly impact the effective instruction throughput by causing threads of the same warp to diverge, that is, to follow different execution paths. If this happens, the different executions paths have to be serialized, increasing the total number of instructions executed for the warp.

7. **More Optimizations Through Loop Unrolling**: Significant performance improvements can be achieved through a simple technique like loop unrolling. The branch that happens in a **for** loop is a waste of time. So, the ratio of useful processing per branch increases with unrolling. However, note that unrolling will not always improve performance. It could possibly result in a much larger register usage and adversely decrease the occupancy as a result. To counteract the register pressure induced by the loop unrolling optimizations, the GPU shared memory can be used for storage of intermediate results. As with everything in CUDA, it is highly dependent on the exact algorithm and there is no substitute for experimentation. It has been found that especially the loops with global memory accesses in them benefit a lot from unrolling. In loops with only shared memory operations, the performance difference is not very large.

Chapter 3

Parallel FMM on the GPU

3.1 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

$$\mathbb{X} = \{x_1, x_2, \ldots, x_N\}, \quad x_i \in \mathbb{R}^3, \quad i = 1, \ldots, N, \tag{3.1}$$

$$\mathbb{Y} = \{y_1, y_2, \ldots, x_M\}, \quad y_j \in \mathbb{R}^3, \quad j = 1, \ldots, M \tag{3.2}$$

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 ϕ 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 ϕ 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.

With rapid improvements in performance and programmability, GPUs have fostered considerable interest in doing computations that go beyond computer graphics and are being used for general purpose computations. GPUs may be viewed as data parallel compute co-processors that can provide significant improvements in computational performance especially for algorithms which exhibit sufficiently high amount of parallelism. FMM is one such algorithm. Our interest lies in design of a parallel FMM algorithm suited to modern day NVIDIA's $G80$/$G92$ GPU architechture using CUDA. We discuss such an algorithm in this chapter. It uses only a static data decomposition and does not require any explicit dynamic load balancing, either within an iteration or across iterations.

3.2 Prior Work

Recently, several researchers have reported the use of GPUs, either in isolation or in a cluster to speed up the N-Body problem using direct algorithms (not FMM), in which the interaction of every pair of particles is considered [NHP07]. While impressive speedups are reported, these algorithms require $O(N^2)$ memory to utilize $O(N^2)$ available parallelism and is limited by memory bandwidth. We look at an interesting implementation of the All Pairs N-Body simulation on the GPU [NHP07] in which some of the computations are serialized to achieve the data reuse needed to reach peak performance of the arithmetic units of the GPU and to reduce the required memory bandwidth.

3.2.1 Direct N-Body Simulations on the GPU

Given N bodies with an initial position $\mathbf{x}_i$ and velocity $\mathbf{v}_i$ ($1 \leq i \leq N$), the force vector $\mathbf{f}_{ij}$ on body i caused by its gravitational attraction to body j is given by the equation:

$$\mathbf{f}_{ij} = G\frac{m_i m_j}{\|\mathbf{r}_{ij}\|^2} \cdot \frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|}$$

, where m_i and m_j are the masses of bodies i and j, respectively; $\mathbf{r}_{ij} = \mathbf{x}_j - \mathbf{x}_i$ is the distance vector from body i to body j; and G is the gravitational constant.

The total force $\mathbf{F}_i$ on body i, due to its interactions with the other $N-1$ bodies, is obtained by summing all interactions:

$$\mathbf{F}_i = \sum_{\substack{1 \leq j \leq N \\ j \neq i}} \mathbf{f}_{ij} = Gm_i \cdot \sum_{\substack{1 \leq j \leq N \\ j \neq i}} \frac{m_j \mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^3}$$

An $N \times N$ grid of all pair forces can be used to store each force $\mathbf{f}_{ij}$. Thus, the total force $\mathbf{F}_i$ on body i is obtained from the sum of all entries in row i. Each $\mathbf{f}_{ij}$ can be computed independently, so there is $O(N^2)$ available parallelism. **This approach requires $O(N^2)$ memory and is substantially limited by memory bandwidth**. However, if we serialize some of the computations we can achieve data reuse required to reach peak performance of the arithmetic units and to reduce the memory bandwidth required.

Consider a tile (2D grid) of $p \times p$ forces. Computations for a tile are arranged so that the interactions in each row (interaction of a body with p other bodies) are evaluated sequentially, updating the force vector, while the rows are evaluated in parallel. The result of the tile calculation is p updated forces.

Now we define a block of p threads that executes some number of tiles in sequence. The number of rows defines the degree of parallelism and more the number of columns the more is the amount of data reuse. Before a tile is executed, each thread fetches one body into shared memory, after which the threads synchronize. Consequently, each tile starts with p successive bodies in the shared memory. In a thread block, there are N/p tiles, with p threads computing the forces on p bodies (one thread per body). Each thread computes all N

interactions for one body. To get the results for all the N bodies we also define a $1D$ grid of N/p thread blocks. Evaluation of a full grid is shown in figure 3.1.

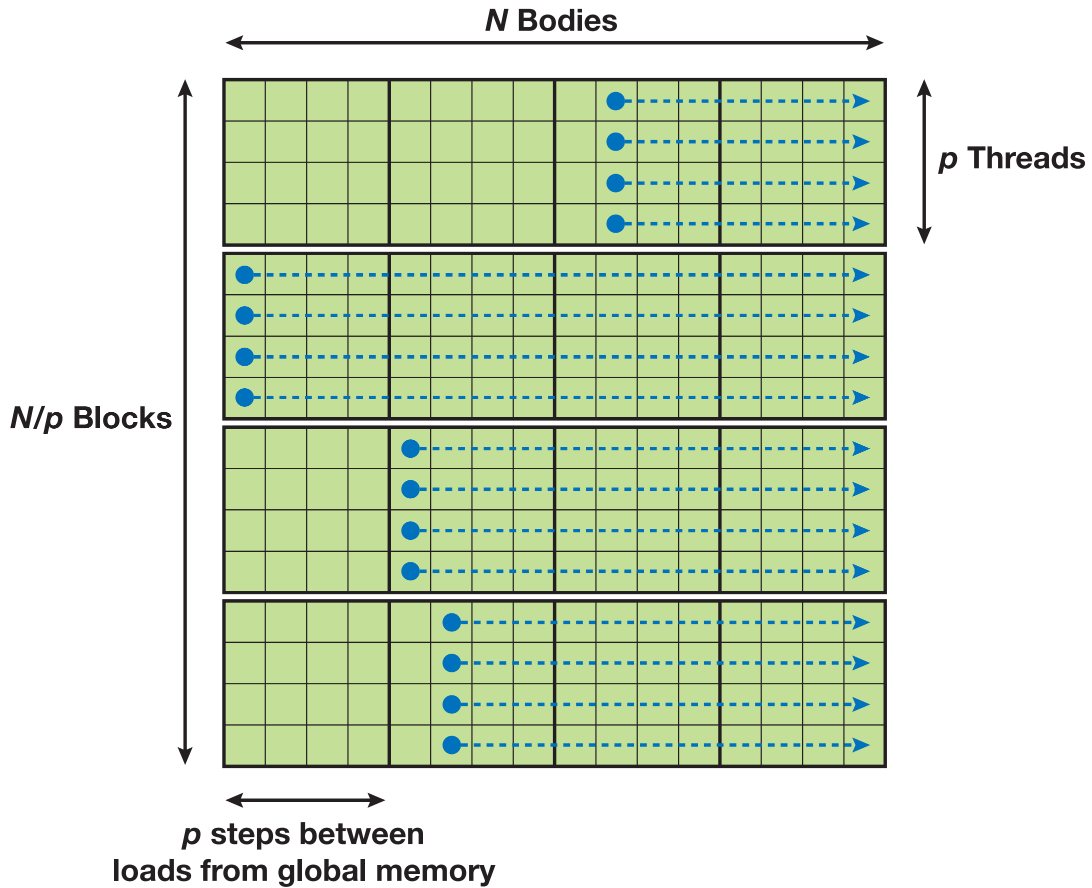


Figure 3.1: A grid of thread blocks that calculates all N^2 forces. Here there are four thread blocks with four threads each [NHP07].

The all-pairs approach just mentioned requires substantial time and memory bandwidth to compute. Thus, it is not generally used on its own in the simulation of large systems. Instead, the all-pairs approach is typically used as a kernel to determine the forces in close-range interactions. It is combined with a faster method based on a far-field approximation of longer-range forces, which is valid only between parts of the system that are well separated. Fast Multiple Method is a well known algorithm of this form. Recently, Gumerov Nail and Duraiswami Ramani [GD07] demonstrated the possibility of implementing the FMM on GPUs. They implemented the FMM on GPUs for *only* the Laplace kernel and obtained a performance acceleration in the range 30-50.

3.3 Parallel FMM computations on GPU

The FMM algorithm for Radiosity kernel [KC03] that we use is consistent with the single precision floating point arithmetic on the GPU. While there are softwares to emulate double precision on the GPU, their use is reported to show a decrease of the computational speed by up to 10 times (see the e.g., in [GST05]). And while we will still see an acceleration relative to the CPU, this will not be as dramatic. GPU manufacturers envision in the closest future the release of GPUs with double precision hardware (both ATI and NVIDIA have announced that this feature will be released in mid 2008). In this case, single precision algorithms can be modified accordingly and the fastest methods for high precision computations can be implemented and tested, without writing artificial libraries.

Thus, with the currently existing GPUs, computations with 3, 4, or 5 digit accuracy are appropriate, which cover a broad class of practical needs. FMM achieves the user-defined accuracy using a truncation number p, which essentially signifies number of terms to be considered from the infinitely long series expansion of the kernel function required for separating the source and receiver terms. Computations with a truncation number $p = 8$ and higher using 4 byte floats can produce a heavy loss of accuracy, overflows/underflows (due to summation of numbers of very different magnitude) and cannot be used for large scale problems. On the other hand, computations with relatively small truncation numbers, like p = 3, 4, and 5 are stable, and can produce the required 3, 4 or 5 digits of accuracy for problems with number of particles of the order $\approx 10^6$, and we focus on this range of truncation numbers in our implementation.

The different parallelization strategies used in our implementation are quite similar to those used by [GD07]. One important thing to note here is that our FMM kernel for radiosity is far more complex which makes the FMM implementation highly difficult. As such the number of terms p in the truncated series expansion of the radiosity kernel is chosen to be 3. It produces results with sufficiently good amount of accuracy (error less that 10^{-4}) acceptable for showing good Global Illumination effects.

3.4 Implementation Details

The Fast Multipole Method consists of the following five phases:

- Octree Construction
- Generating visible interaction lists
- Upward Pass
- Downward Pass
- Final Summation

Our parallel FMM algorithm specifically solves the last three phases (Upward pass, Downward pass and Final summation stage) on the GPU. We assume, as a part of pre-processing step, that we have been given an octree constructed for the input 3D model along with the interactions lists for each of the octree nodes (containing only visible nodes). The octree can be constructed on the CPU or on the GPU (using algorithms discussed in chapter 5), while the visible interaction lists construction happens on the CPU. These 2 phases will eventually be implemented on GPU and combined with the rest of the algorithm.

INPUT: A 3D model with its defined octree and visible interaction lists.
OUTPUT: A Global Illumination solution for the given model.

Our input octree is a long one dimensional array with each level of octree stored one after the other(starting from the *root*). The parent-child relationship is established using the array indices.

We also define four one dimensional arrays, each corresponding to one of the interaction list's type (far, near, multipole, local). The size of each of these arrays is the sum total of the number of nodes in the interaction lists of every node (for e.g. size(far cell list) = $\sum_i$ size(far cell list of each node i)). The relationship between each node and each of its interaction lists is defined by storing in it the start and the end indices of each of its interaction list in the four global interaction list arrays.

A 3D input point model is stored as a single point array with its necessary attributes (co-ordinates, normal, diffuse surface color, emmissivity, gaussian weights). Incase of triangular models, they are converted to points using gaussian quadrature weights theory [KC03].

In the next section we present the different kernel functions implemented for upward, downward and direct summation passes of the FMM algorithm. Each kernel is executed, for different levels of the octree, in CUDA on a one dimensional grid of one dimensional blocks of threads. By default each block contains 128 threads, which was obtained via an empirical study of optimal thread-block size. This study also showed that for good performance the thread block grid should contain not less than 64 blocks for a 16 multiprocessor configuration. If the number of nodes at a level is not a divisor of the block size, only the remaining number of threads is employed for computations of the last block.

3.4.1 Upward Pass

3.4.1.1 Step 1: Generating Multipole Expansion Co-efficients for the Leaves

We need to calculate, in parallel, for each leaf in the octree, the multipole, or S expansion of all particles (sources) contained in the node about the center of the node. The expansions from all particles (sources) in the node are consolidated in a single expansion by summing the coefficients corresponding to each particle

(source).

One solution for parallelization here is to assign each thread to handle one source expansion. A drawback of this method is that after generation of expansions they need to be consolidated, which will necessitate data transfer to GPU global memory, unless they form a block of threads handled by one processor. The block size for execution of any subroutine in GPU can be defined by the user, but it is fixed during execution. In the FMM each node may have different number of particels (sources). Thus if a node is handled by a block of threads then threads could be idle, which, of course, reduces the utilization efficiency. GPU speedups compared to the serial CPU code in this case are in range 2-5, which appear to be rather low when compared with the performance of other steps.

The efficiency of this step of calculating multipole expansions substantially increases when we adopt a different parallelization model of having *one thread per node*. In this case one thread performs expansion for each of the sources in the leaf and consolidates these expansions. So one thread produces full multipole expansion for the entire leaf. The advantage of this approach is that the work of each thread is completely independent and so there is no need for shared memory. This perfectly fits the situation when each leaf may have different number of sources, as the thread that finishes work for a given leaf simply takes care of another leaf, without waiting or need for synchronization with other threads. The disadvantage of this approach is that to realize the full GPU load the number of boxes should be sufficiently large. Indeed, if an optimal thread block size is 128 and there are 16 multiprocessors (so we need at least 64 blocks of threads to realize an optimal GPU load), then the number of nodes should be at least 8192 for a good performance. Note that at maximum leaf level = 4 we have at most $8^4 = 4096$ leaves, and for maximum level = 5 this number becomes $8^5 = 32768$ leaves. So the method can work efficiently only for large enough problems which is the case with us.

1. For every level of octree, starting from the last level of octree, upto the root do
   (a) Allocate threads equal to number of nodes at the current level
   (b) For every thread, *in parallel*, Do
   (c) If current node is a leaf Then
      i. Calculate the multipole expansion of all particles (sources) contained in the current leaf about the center of that leaf.
      ii. Consolidate each of these expansions in a single expansion at the current leaf's center.

The time spent for this step usually does not exceed a couple of percent of the overall FMM run time and also the FMM on the GPU is efficient only for relatively large problems.

3.4.1.2 Step 2: Generating Multipole Expansion Coefficients for the Internal Nodes

We need to calculate, in parallel, for each level $l = l_{max} - 1, ...2$, for each node b at that level, the multipole, or S expansion coefficients $\mathbf{M}(b)$ due to all particles in that node by translating and aggregating the multipole expansion coefficients of all its children.

We have not yet come upon an optimal strategy for this subroutine. The current version is based on the fact that the resulting multipole, or S expansions for the parent nodes can be generated independently. So, each thread can be assigned one parent node. However, the work load of the GPU in this case becomes very small for low l_{max} and more or less reasonable speedups can be achieved only if several threads are allocated to process a parent node. Since each parent node in the octree has at most 8 children and for each child the multipole-to-multipole, or $S|S$ translation can be performed independently, we used a two dimensional 64×8 blocks of threads and one dimensional grid of blocks. In this setting each parent node was served by 8 threads, with the thread id in y varying from 0 to 7 for identification of the child nodes.

1. For every level of octree, starting from the second last level, upto the root do
   (a) Allocate a 2D grid of threads equal to number of nodes at the current level times 8 (idx is parent and $idy = 0 - 7$ are children nodes. For empty children threads remain idle)
   (b) For every thread, *in parallel*, Do
      i. If current node is a non-leaf node Then
      ii. Translate one S coefficient corresponding to the child idy to the center of current node and write to the shared memory based on idx, idy
   (c) Synchronize the threads
   (d) For every thread with $idy = 0$, *in parallel*, Do
      i. Sum up all the coefficients with the same idx and store in variable sum
      ii. Write the result back to the global memory corresponding to the current (parent) node.

We do not expect to achieve much speedup in this step. The complexity of depends only on the number of children for each parent node. For non-adaptive structure this number is equal to 8^l for level l. When $l_{max} = 3$ which has only 512 children and 64 parent nodes at most, the efficiency of *translation/per thread* parallelization is low. We have already mentioned earlier that for the current GPU architecture sizes involving 8192 parallel processes or more can be run at full efficiency. Even for $l_{max} = 4$, the full load is not achieved. Thus, for $l_{max} \geqslant 5$ we expect GPU to gain speedups over the CPU code, which includes computations not only for l_{max}, but for all levels from l_{max} to 3.

Note that the upward pass is a very cheap step of the FMM and normally takes not more than 1% of the total time. This also diminishes the value of putting substantial resources and effort in achieving high speedups for this step.

3.4.2 Downward Pass

We repeat the following steps for each level of the octree, starting from level 2 to the maximum level l_{max}. Downward pass and the final summation phases are combined into a single phase.

3.4.2.1 Step 1: Multipole to Local Translations

For each node, in parallel, translate and aggregate the multipole, or S expansion coefficients of every node in the far cell interaction list of the current node into local, or R expansion coefficients about the current node's center.

1. Allocate threads equal to number of nodes at the current level
2. For every thread, *in parallel*, Do
   (a) For each node A in the far cell list of current node Do
      i. Translate the multipole expansion coefficients of A into local, or R expansion coefficients about the center of current node.
      ii. Aggregate each of these expansions in a single expansion at the current node's center.

3.4.2.2 Step 2: Local List Translations

For every node, in parallel, in addition to converting the multipole expansion coefficients of all nodes in the interaction list into local expansion coefficients at the node's center, the local expansion coefficients obtained from the individual particles contained in the local interaction list are also aggregated.

1. Allocate threads equal to number of nodes at the current level
2. For every thread, *in parallel*, Do
   (a) For each node n in the local list of current node Do
      i. Obtain the local expansion coefficients obtained from the individual particles contained in n about the center of current node.
      ii. Aggregate each of these expansions in a single expansion at the current node's center and add up to its existing local expansion coefficients.

3.4.2.3 Step 3: Local to Local Translations

In addition to multipole-to-local and local-list translations, we further need to calculate, in parallel, for each node b at current level, the local, or R expansion coefficients about its center by translating and aggregating the local expansion coefficients from its parent.

1. Allocate threads equal to number of nodes at the current level
2. For every thread, *in parallel*, Do
   (a) Obtain the local expansion coefficients from its parent node about the center of current node.
   (b) Add up to the existing local expansion coefficients about current node's center.

This step is very similar to the step 2 of upward pass. For parallelization of this step, the *one thread per node* strategy is used.

3.4.2.4 Step 4: Evaluate Local Expansion at Points

Evaluate, in parallel, the local expansions at individual points in each of the leaves, from the corresponding leaf's center.

1. Allocate threads equal to number of nodes at the current level
2. For every thread, *in parallel*, Do
   (a) If current node is a leaf Then
      i. Obtain the radiosity values at individual points in the current leaf, from the local expansion coefficients at current leaf's center.

This step is very similar to the multipole expansion generator discussed above in step 1 of the upward pass. For parallelization of this step, the *one thread per node* strategy is used. The performance of this step is approximately the same as of the multipole expansion generator.

3.4.2.5 Step 5: Near Cell List Translations

For every node in parallel, evaluate the near neighbor interactions (if current node is a leaf) between the points in the current node and every point in each of the nodes in its near cell interaction list. This, and the remaining steps, are a part of the final summation phase.

1. Allocate threads equal to number of nodes at the current level

2. For every thread, *in parallel*, Do

   (a) If current node is a leaf Then

      i. For each node n in the near cell list of current node Do

         A. For all points in current leaf

         B. For all points in n

         C. Evaluate the radiosity interaction directly

         D. Add up the evaluated value to the exisiting radiosity values of the points

3.4.2.6 Step 6: Multipole List Translations

For every node, in parallel, in addition to evaluating the near neighbors and local expansion coefficients at each particle, we also evaluate the multipole expansion coefficients of all nodes in the multipole interaction list.

1. Allocate threads equal to number of nodes at the current level

2. For every thread, *in parallel*, Do

   (a) If current node is a leaf Then

      i. For each node n in the multipole list of current node Do

         A. Translate the multipole expansion coefficients of n from its center to individual points of current leaf

         B. Add up the evaluated value to the exisiting radiosity values of the points

This scheme for downward pass is efficient on CPU, but may not be the best for GPU, where the cost of one random access to global memory is equal up to 150 float operations, and instead of reading precomputed data GPU may rather compute them at higher rate. Moreover, for low l_{max} the GPU kernel may even run slower than the serial CPU!. However, even in this case it is not recommended to switch between the CPU and GPU, since such a switch involves the slowest memory copying process (CPU-GPU), and if possible all data should stay on the GPU global memory. *Performance improves a lot* as the size of the problem and, respectively, the maximum level of the octree increases and for $l_{max} = 8$ the time ratio reached 20 or so.

In the above steps we explained the kernel functions implemented for the upward, downward and final summation passes of the FMM algorithm. Each point has 3 primary colors associated with it viz. Red, Green and Blue. Hence, we run all the above steps thrice, corresponding to each color. Further, we converge to the final Global Illumination solution by iterating all the above steps (for all colors) three times (Empirical evidences prove that the solution converges to a good extent in 3 iterations).

3.4.3 Quality Comparisons

First we compare the results obtained on the GPU for quality with the corresponding implementation on the CPU. To compare CPU and GPU implementations we use 3-d point models of bunny and Ganesha in a Cornell room each having four light sources on the ceiling. As we can see in Fig. 3.2 the *CPU-GPU results look identical*. Global illumination effects like color bleeding and soft shadows are also clearly visible. Note that for FMM the visual quality of result does not depend on the kind of GPU used (NVIDIA 8800 GTS or Quadro FX 3700). The GPU should just support CUDA and have enough memory ($>$ 256Mbs).

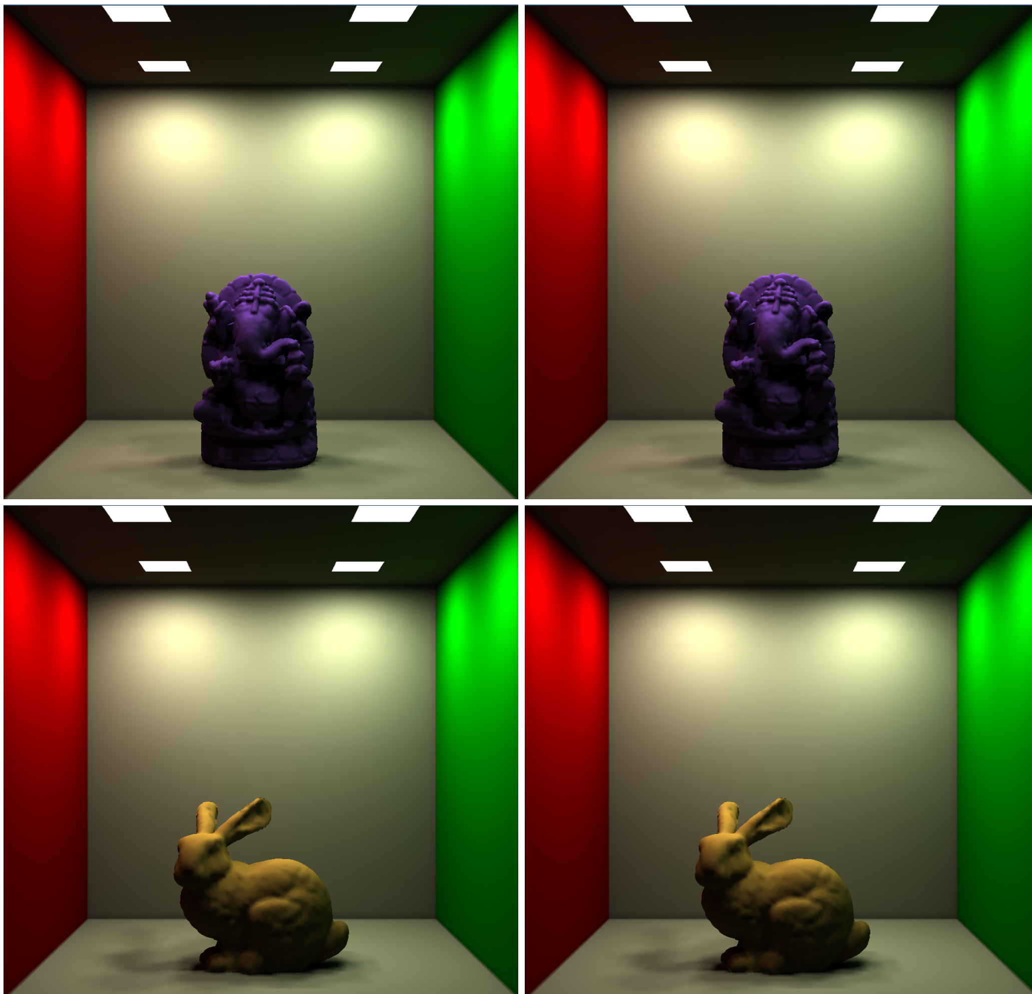


Figure 3.2: Top Left: A Cornell Room with the Ganesha's point model on CPU. Top Right: Corresponding GPU result. Bottom Left: A Cornell Room with the Bunny's point model on CPU. Bottom Right: Corresponding GPU result. Both the results assume 50 points per leaf.

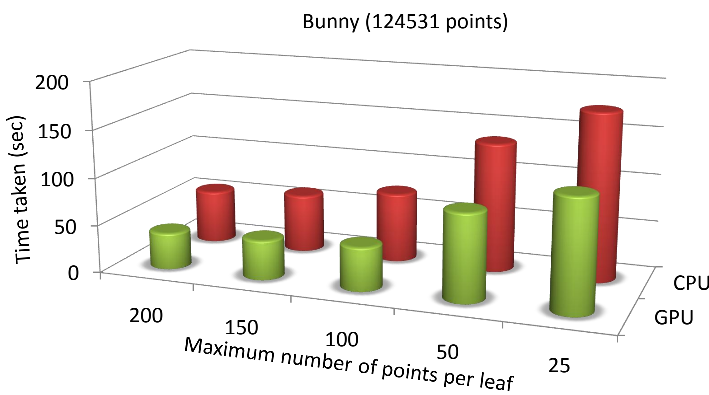
Note that we converge to the final Global Illumination solution shown in Fig. 3.2 by performing the FMM algorithm (Upward and then Downward passes) for all colors (RGB) three times. We can see that the solution converges to a good extent in 3 iterations.

Figure 3.3: Point models rendered with diffuse global illumination effects of color bleeding and soft shadows. Pair-wise visibility information is essential in such cases. Note that the Cornell room as well as the models in it are input as point models.

3.4.4 Timing Comparisons

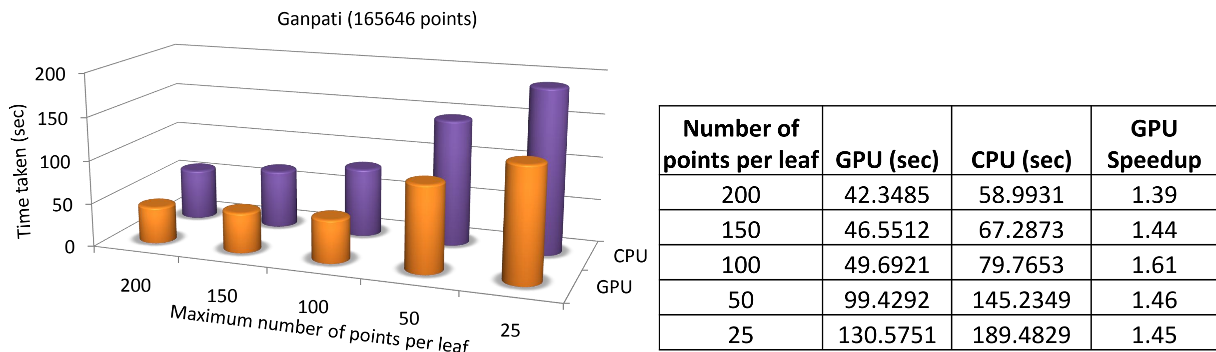
The timing calculations are done on a machine having a dual core AMD Opteron 2210 processor with 2 Gbs of RAM, NVIDIA GeForce 8800 GTS with 320 Mbs of memory and Fedora Core 7 (x86_64) installed on it. The total time taken by the upward and downward passes of the FMM algorithm for all 3 iterations and all 3 colors RGB is shown in the results below. The time taken by each iteration is approximately same.

3.4.4.1 Upward Pass (for all 3 iterations, 3 colors and p=3)



Number of points per leaf	GPU (sec)	CPU (sec)	GPU Speedup
200	38.3485	55.9931	1.46
150	41.5512	61.2873	1.47
100	45.6921	72.7653	1.59
50	91.4292	135.2349	1.47
25	117.5751	180.4829	1.53

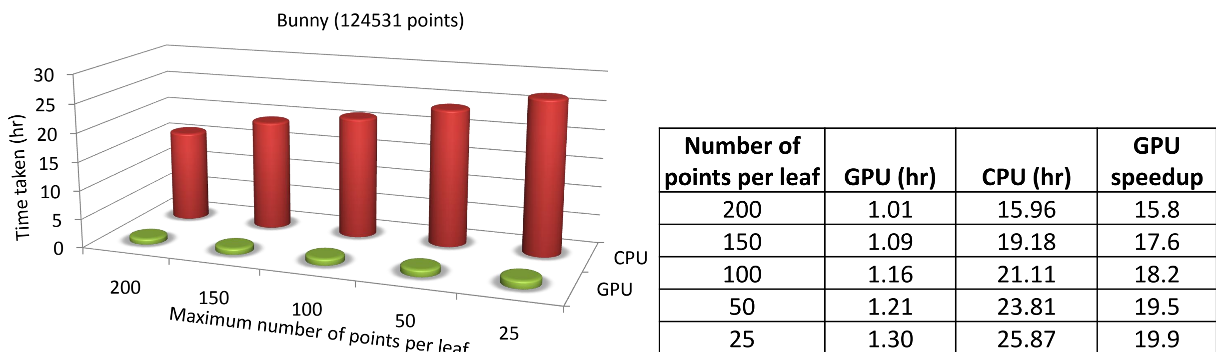
Figure 3.4: FMM Upward Pass : Bunny with 124531 points



Number of points per leaf	GPU (sec)	CPU (sec)	GPU Speedup
200	42.3485	58.9931	1.39
150	46.5512	67.2873	1.44
100	49.6921	79.7653	1.61
50	99.4292	145.2349	1.46
25	130.5751	189.4829	1.45

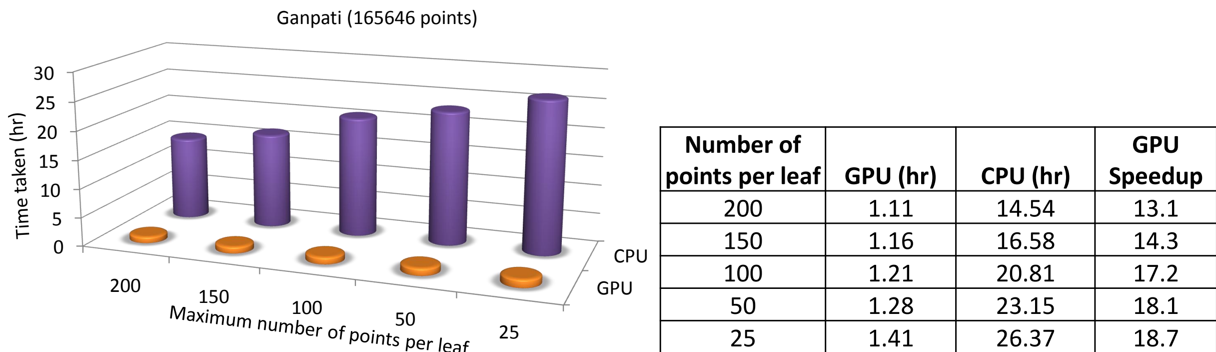
Figure 3.5: FMM Upward Pass : Ganpati with 165646 points

3.4.4.2 Downward Pass (for all 3 iterations, 3 colors and p=3)



Number of points per leaf	GPU (hr)	CPU (hr)	GPU speedup
200	1.01	15.96	15.8
150	1.09	19.18	17.6
100	1.16	21.11	18.2
50	1.21	23.81	19.5
25	1.30	25.87	19.9

(a)



Number of points per leaf	GPU (hr)	CPU (hr)	GPU Speedup
200	1.11	14.54	13.1
150	1.16	16.58	14.3
100	1.21	20.81	17.2
50	1.28	23.15	18.1
25	1.41	26.37	18.7

(b)

Figure 3.6: Downward Pass (a) Bunny with 124531 points (b) Ganpati with 165646 points

Thus, we see that the GPU outperforms the CPU by factors of 13-20 in the downward pass of the FMM algorithm. We also see that the upward pass of the FMM algorithm consumes less than 1% of the time taken by the downward pass. Thus, the speedup achieved in the upward pass does not play an important role in the overall FMM speedup. *The overall speedup achieved is the speedup achieved in the downward pass.*

Chapter 4

Space Filling Curves

While performing a domain decomposition one would like to be able to decompose the domain in a manner that is not only easy to implement in practice, but also possesses a robust mathematical representation that enables fast computation of data ownership. It should also be easy to parallelize the domain decomposition algorithm itself, which is useful in reducing the runtime overhead. Moreover, it should provide good quality load balancing. In this section we discuss a space filling curve based domain decomposition that meets the above requirements. We will finally use it to implement parallel octree on the GPU.

Let us consider a 3 dimensional cube. On bisecting it recursively k (resolution of decomposition) times along each dimension, we get a 3 dimensional matrix having $2^k \times 2^k \times 2^k$ non-overlapping cells of equal size. A Space Filling Curve (SFC) maps the 3 dimensional location of these cells to a 1 dimensional linear ordering (see Figure 4.1 for SFC ordering in 2 dimensions). This process is often referred as *linearization*.

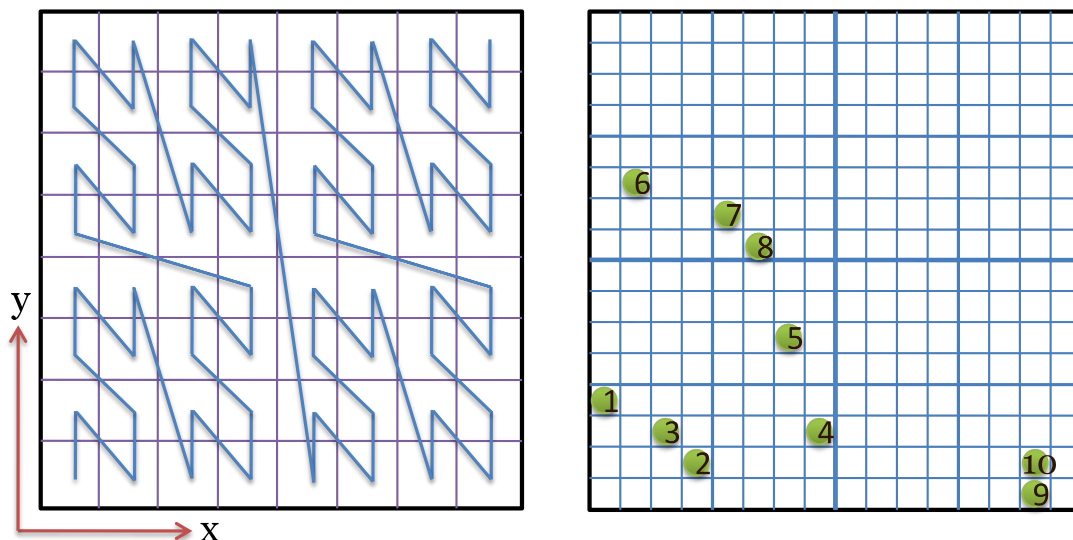


Figure 4.1: Left: The 2-D z-SFC curve for $k = 3$, Right: 10 points in a 2-D space. The points are sequentially labeled in the z-SFC order.

Partitioning the SFC ordering of the particles equally to processors ensures load balancing. If the computational load per particle is not identical but known, the SFC linearization can be partitioned such that the total load per processor is about the same. If the load is not known, the first iteration can be run with an equally partitioned SFC linearization, and load on each particle estimated. This can be used to readjust the load balance, by simply moving the boundaries of the processors on this one dimensional linear ordering. However, the runtime to order 2^{3k} cells, $\Theta(2^{3k})$, is expensive because typically $n \ll 2^{3k}$. Thus, a method to directly order the cells containing the particles is required.

Let us consider a 2 dimensional particle space of side length D and let its bottom left corner be at the origin. To begin with we first find the integer coordinates of the cells containing each of the input particles. For resolution k, the integer coordinates of a cell having a particle $P(P_x, P_y)$ will be

$$\left(\left\lfloor \frac{2^k P_x}{D} \right\rfloor, \left\lfloor \frac{2^k P_y}{D} \right\rfloor\right)$$

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 $2k$ bit integer. This interleaving scheme is the characteristic defining function of z-SFC. For example, the index of a cell with coordinates $(3, 1) = (11, 01)$ is given by $1011 = 11$. Once the indices corresponding to all the points are generated, SFC linearization is achieved by a parallel integer sort [Chr].

We can see that SFC decomposition is very similar, though not identical to an octree decomposition. It is hardly surprising that they can be related. When drawing an octree, we can draw the children on a node in the order in which z-SFC visits the subcells represented by the children. Thus, octrees can be viewed as multiple SFCs at various resolutions (see figure 4.2).

Observe that removing the last 2 bits (3 in 3-D) from the index of a cell gives the index of its parent (see figure 4.3). Thus, the process of assigning indices can be viewed hierarchically. However, ambiguity arises when cells at different levels have the same indices (For e.g. 00 in level 1 is same as 0000 in level 2). A simple way to overcome this ambiguity is to prepend the bit representation of an index with a '1' bit. Thus, root cell becomes 1 and the cells with Z-SFC indices 00 and 0000 are now 100 and 10000.

The advantage of such bit representation is that it allows primitive operations on cells using fast bit operations:

1. *check if a cell C_1 is contained in cell C_2*: If C_2 is a prefix of C_1, then C_1 is contained in C_2; otherwise, not.

2. *find the smallest cell containing C_1 and C_2*: Find the longest common prefix of C_1 and C_2 that is a multiple of dimension d.

Figure 4.2: Octree can be viewed as multiple SFCs at various resolutions

3. *find the immediate subcell of C_1 that contains a given cell C_2*: For dimension d and level l, if $dl + 1$ is the number of bits representing C_1, the required immediate subcell is given by the first $d(l + 1) + 1$ bits of C_2.

Figure 4.3: Bit interleaving scheme for a hierarchy of cells

In this way we can obtain a unique index for each cell of the octree. It is easy to see that index of each cell can be computed independently in parallel either by using multiple passes, calculating the indices in the parent

level using the child level or just by direct calculation based on the level of the cell and its 2-D id in that level.

However, linearization of cells that cuts across multiple levels of the octree can also be beneficial. Given any two cells, they are either disjoint or have a subcell-supercell relationship. Thus, to establish a total order on the cells of an octree, 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 is the resulting linearization of all cells in an octree (or compressed octree [chapter 5.3.2.1]) is identical to its postorder traversal.

Chapter 5

Octrees

Octree is one of the numerous hierarchical data structures, based on recursive domain decomposition, that are used for representing spatial data. Its development has been motivated to a large extent by a desire to save storage by aggregating data having identical or similar values. However, the savings in execution time that arise from this aggregation are often of equal or greater importance. Using octree as the base hierarchical structure for FMM implementation is one of the important reasons for its success with respect to pulling down the run-time complexity of the algorithm.

We discuss, in this chapter, three different parallel octree implementations on the GPU which can eventually be combined to parallel FMM implementations on GPU. The first constructs a Non-adaptive octree using Direct Indexing. The other two construct an adaptive octree; one is a bottom-up construction approach using Space Filling Curves (SFC) (as discussed in chap. 4) and Compressed octrees (discussed in Sec. 5.3.2.1), while other implementation is a top-down approach using spatial clustering of points. Note that the octree construction phase does not consume more than 1% of the overall FMM run-time and hence it is insignificant if the octree is implemented on the CPU or the GPU.

5.1 Octrees: Introduction

Octrees can be differentiated on the basis of the type of data they are used to represent, the principle guiding the decomposition and the resolution which can be xed or variable.

5.1.1 Non-Adaptive and Adaptive Octrees

Let us look at a top-down method to construct octrees.

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. The resulting tree is called a region octree to reflect the fact that each node of the tree corresponds to a non-empty subdomain. An example is shown in Fig. 5.1. In practice, the recursive subdivision is stopped when a predetermined resolution level is reached, or when the number of points in a subregion falls below a pre-established constant. This forms an *adaptive octree*. A *non-adaptive octree* is formed when the maximum resolution is fixed.

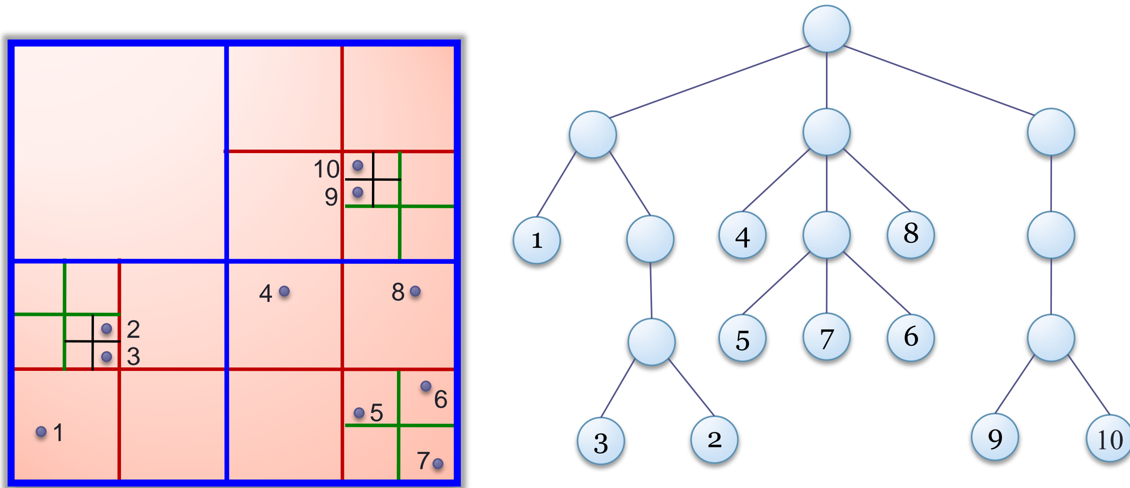


Figure 5.1: A quadtree built on a set of 10 points in 2-D.

5.2 Prior Work: Octree Construction on the GPU

In this section we survey the already existing implementation of the Octree on the GPU [LHN05] which uses indices stored within a texture node to link the tree nodes just like the way a CPU implementation of octree uses pointers.

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 5.2 shows the octree storage.

Let us first define the following terminology:

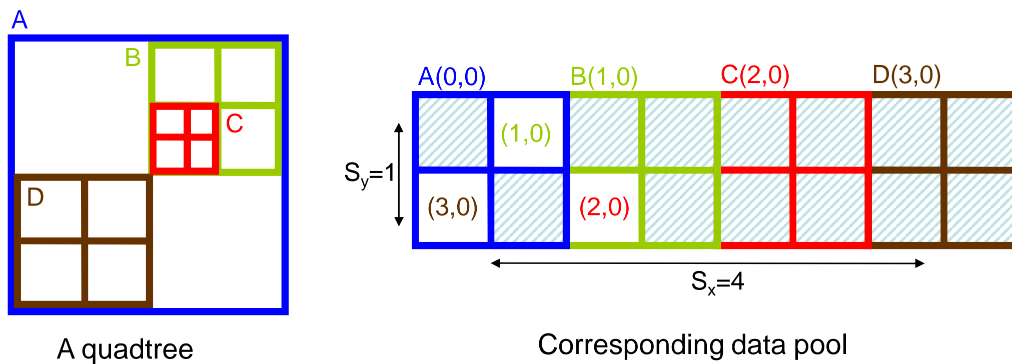


Figure 5.2: Storage in texture memory. The data pool encodes the tree. Data grids are drawn with different colors. The grey cells contain data.

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

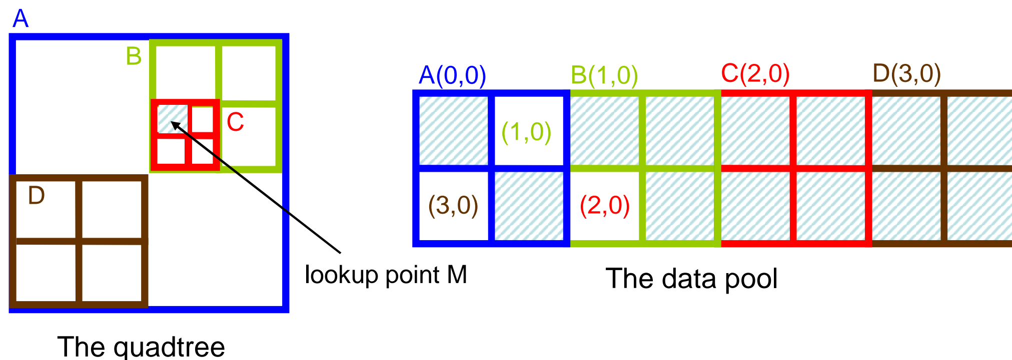


Figure 5.3: Lookup for point M in the octree

Now the tree is stored in the texture memory and we want to retrieve the value stored in the tree at a point $M \in [0,1] \times [0,1]$ (see figure 5.3). Let $I_D = (I_{D_x}, I_{D_y})$ be the index of the data 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 data grid I_D the value stored at the location corresponding to M which in turn requires the computation of the

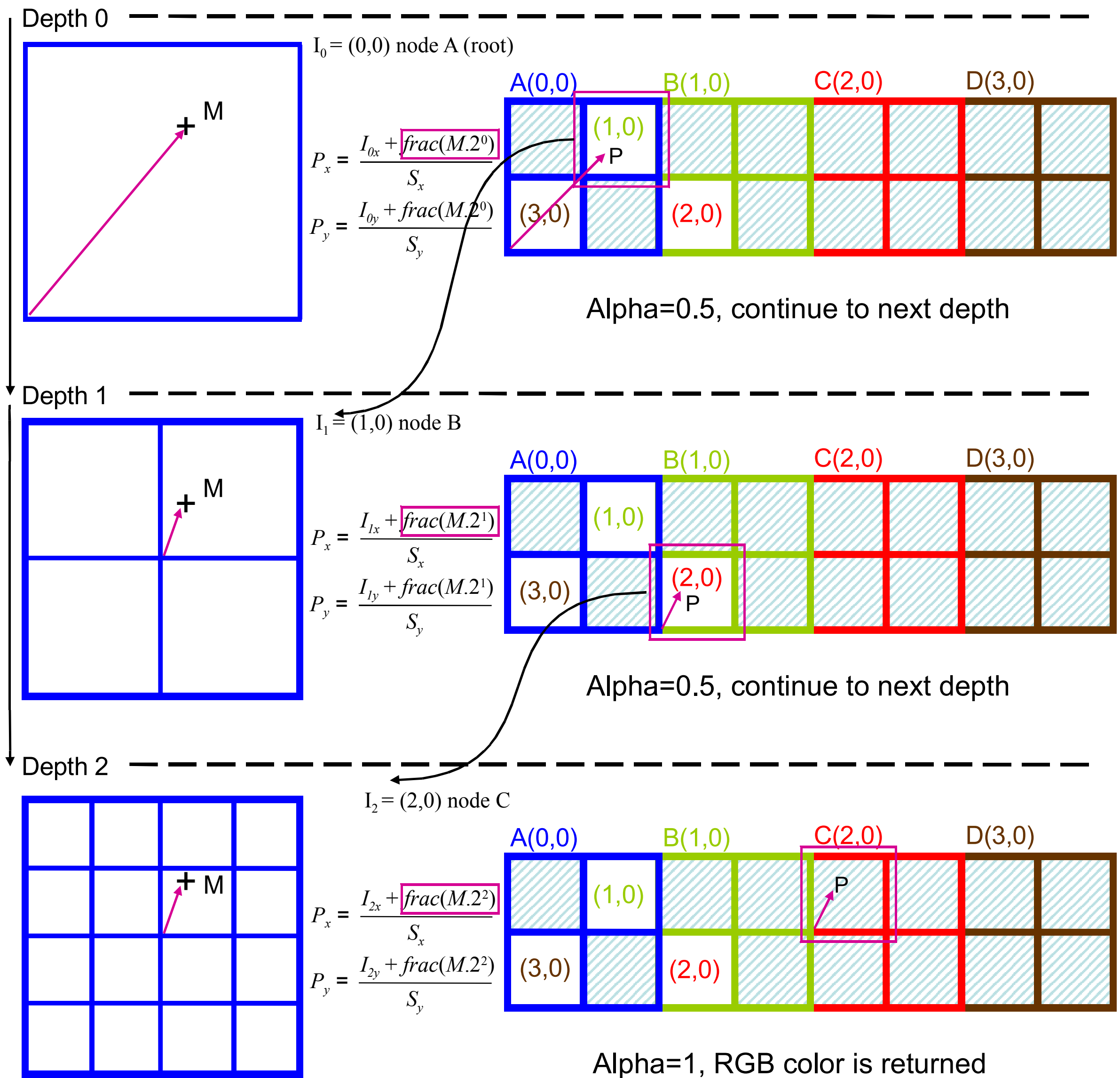


Figure 5.4: At each step the value stored within the current node's data grid is retrieved. If this value encodes an index, the lookup continues to the next depth. Otherwise, the value is returned.

coordinates of M *within* the node.

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 data grid I_D. Thus, the lookup coordinates within the data 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 data grids along each *row* and *column* of the data pool respectively. The RGBA value stored at P in the data 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 data grid (I_{D+1}) and continue to the next tree depth. Fig. 5.4 summarizes this entire process.

5.2.1 Problems

A potential problem with this implementation is that it is very difficult to create such a data representation in parallel not only on the GPU but also on the CPU. For eg. if we look at Fig. 5.2 we see that the children of the root node A are B and D. Now in the storage, B and D are not adjacent and C which is a child of B comes in between. Thus, we do not know a priori the position where a particular node in the octree is going to land in the texture.

Another important drawback is that same amount of memory is allocated for both leaves and internal nodes. We know that in an octree all the data is stored only in the leaves. The internal nodes do not contain much information. So allocating same memory space for a internal node and a leaf is not a clever idea.

5.3 Octree on the GPU

We looked at the currently existing implementation of the octrees on GPUs in previous section and we also discussed some of the drawbacks of that implementation. In this section we present three possible parallel octree implementations on the GPU which overcome those drawbacks. We also discuss the superiority of one algorithm over the other while comparing them on the grounds of memory efficiency and running time.

5.3.1 Implementation 1: Non-Adaptive Octree using Direct Indexing [AGCA08]

INPUT: n points belonging to some 3-d domain, maximum resolution L of the octree to be constructed
OUTPUT: Octree represented using L arrays one for each level. The parent-child relationships are established using direct indexing due to non-adaptive nature of memory allocated for each level.

Let the root in an octree denote level zero and let us consider quadtrees for understanding. The maximum number of nodes at level k in an octree are 2^{kd} where d is the dimension (d=2 for quadtree). Also assume that we know the maximum level L upto which the particle space has been divided to generate the octree. Therefore, we allocate L 2-dimensional arrays and each having size 2^{kd} depending on its level to get the hierarchy for the octree (see Figure 5.5). Thus, as far as the memory is concerned we allocate as much memory required for a non-adaptive octree since we can't do dynamic memory allocation on the GPU and we do not have any prior knowledge of the number of nodes in a particular level. We'll later see how to maintain the adaptive structure

information in the tree.

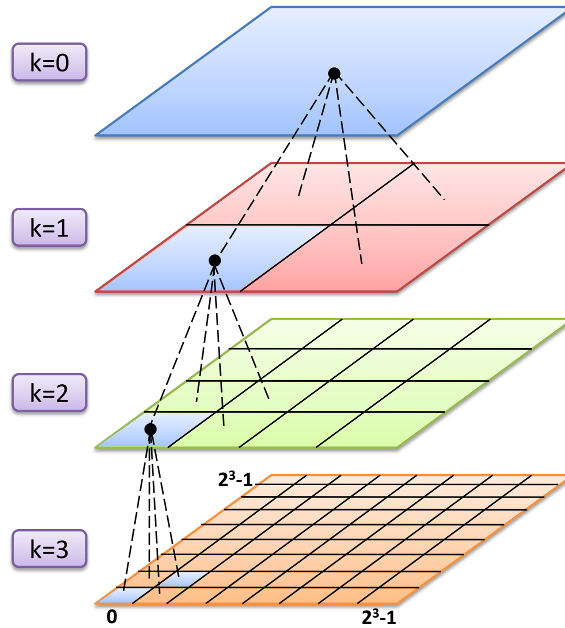


Figure 5.5: Hierarchy of cells in two dimensions. Gray cells indicate data.

The 2-D position of the parent of a node in the upper layer can directly be calculated from the 2-D position of the child node. Along with the actual node data, within each node we also store the following fields: *nodeType* {-1 for empty node, -2 for filled node i.e. node containing some data, -3 for filled internal node}, *numEmptyNodes* {number of empty nodes in the subtree of the node (including itself)} and *dataLocation* {(level, 2-D position in that level)}.

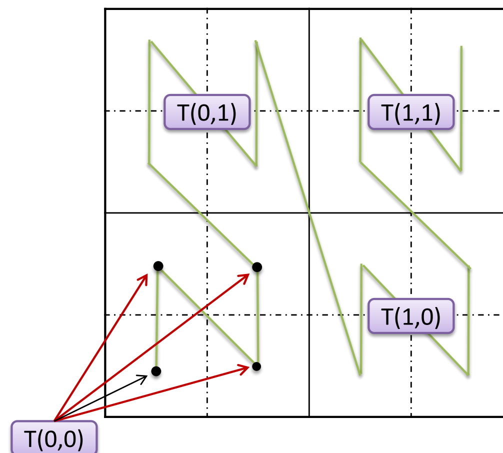


Figure 5.6: One pass of the algorithm. Threads are at level 1 and nodes at level 2

Please refer to the section 2.2 on CUDA to get an idea about the thread allocation on the GPU. We run our algorithm in multiple passes considering two levels L_i and L_{i-1}; $i = k_{max}, ..., 1$ in each pass. In the first

pass we first allocate $\frac{2^{kd}}{4}$ threads on the GPU for the last level so that each thread can handle four nodes (see Figure 5.6). These four nodes come one after the other in the SFC linerization of that level. Now each thread checks the number of empty nodes among those four nodes.

- If all the nodes are empty then it sets the nodeType field of its parent to -1 and numEmptyNodes field to the number of empty nodes in the subtree plus 1. The dataLocation field of the parent still remains null.
- If three nodes are empty, then it sets the nodeType of the non-empty node to -1 and in its parent it sets numEmptyNodes to the number of empty nodes in the subtree plus 1, nodeType to the nodeType of non-empty node and dataLocation to be the dataLocation in the non-empty node.
- In other cases, it just set the nodeType field of the parent to -3 and numEmptyNodes to the number of empty nodes in the subtree.

The same procedure is repeated for the remaining levels to generate the complete octree. It is important to note that the implementation is highly data parallel with zero communication between the GPU threads. The pseudocode for this implementation is shown in algorithm 1.

Once the tree is constructed (see figure 5.7) we find the postorder traversal of the tree in parallel. Even though our octree is adaptive but memory is allocated non-adaptively. This observation can be exploited to directly calculate the postorder number of a node in $O((2^d - 1)\log_{2^d}(2^{dk}))$ time for p threads, dimension d and maximum level k. For a quadtree this turns out to be $O(k)$.

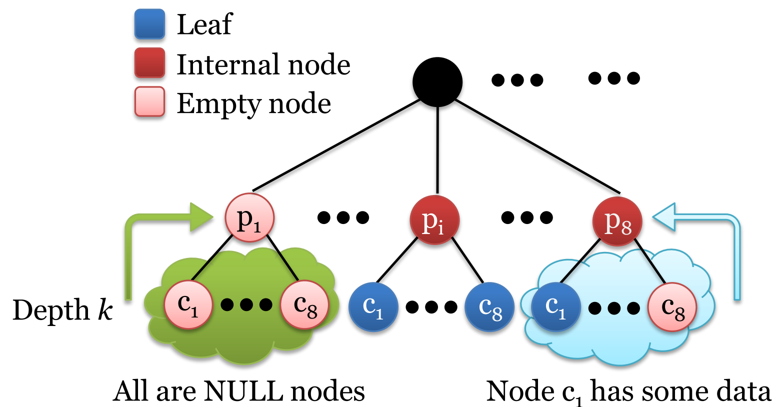


Figure 5.7: The constructed octree

To find out the postorder traversal, we first allocate threads on the GPU in such a way so that each node gets exactly one thread. We define a kernel program which calculates the postorder number of a node in the adaptive octree. Now each thread runs one kernel program on the node for which it is allocated. Lets say that we are looking at a node N in level L and with 2-D id in that level converted to a 1-D id I. While calculating the postorder number $PONA$ (Post Order Non Adaptive) of a node in the non-adaptive tree, kernel also calculates

Algorithm 1 Bottom-up octree construction

```
1: d = 2 // for 2-D case
2: Initialize nodeType field of filled nodes in last level to -2 and -1 for empty nodes
3: Initialize numEmptyNodes field of filled nodes in last level to 0 and 1 for empty nodes
4: Initialize dataLocation field of filled nodes in last level to [2-D id, level] and NULL for empty nodes
5: for i = k_max to 1 do
6:    Allocate a grid of 2^((i-1)d) threads
7:    for all threads T(x, y) in parallel do
8:       numEmpty = findNumEmpty(L_i(2x, 2y), L_i(2x + 1, 2y), L_i(2x, 2y + 1), L_i(2x + 1, 2y + 1))
9:       if numEmpty == 4 then
10:         L_{i-1}(x,y).nodeType = −1
11:         L_{i-1}(x,y).numEmptyNodes = emptyNode_1.numEmptyNodes + emptyNode_2.numEmptyNodes +
            emptyNode_3.numEmptyNodes + emptyNode_4.numEmptyNodes + 1;
12:      else if numEmpty == 3 then
13:         L_{i-1}(x,y).nodeType = nonEmptyNode.nodeType
14:         nonEmptyNode.nodeType = −1
15:         nonEmptyNode.numEmptyNodes += 1
16:         L_{i-1}(x,y).numEmptyNodes = emptyNode_1.numEmptyNodes + emptyNode_2.numEmptyNodes +
            emptyNode_3.numEmptyNodes + nonEmptyNode.numEmptyNodes
17:         L_{i-1}(x,y).dataLocation = nonEmptyNode.dataLocation
18:      else if numEmpty == 2 then
19:         L_{i-1}(x,y).nodeType = −3
20:         L_{i-1}(x,y).numEmptyNodes = emptyNode_1.numEmptyNodes + emptyNode_2.numEmptyNodes +
            nonEmptyNode_1.numEmptyNodes + nonEmptyNode_2.numEmptyNodes
21:      else
22:         L_{i-1}(x,y).nodeType = −3
23:         L_{i-1}(x,y).numEmptyNodes = emptyNode.numEmptyNodes + nonEmptyNode_1.numEmptyNodes
            + nonEmptyNode_2.numEmptyNodes + nonEmptyNode_3.numEmptyNodes
24:      end if
25:   end for
26: end for
```

the number of empty nodes NE (Number of Empty) before the current node in the postorder numbering of non-adaptive tree. Thus, the final postorder number of the node will be $PONA - NE$.

To calculate PONA we make use of a table structure (figures 5.8, 5.9) which for a node at level i in the quadtree defines the number of elements in its subtree including itself. Use of such a table eliminates the repeated calculatations of the number of elements in the subtree of a node, within each kernel. Now each thread runs a loop that iterates from root to the level of the node in consideration. In the first iteration it determines the number of children of root (and hence the number of nodes in their subtrees using table structure) before the child in which current node is present. It also determines the number of empty nodes in the children of root before the child in which current node is present. It then sets the child of root in which the node is present to be the new root and repeats the same procedure. At the end of the loop it subtract NE from $PONA$ to get the final postorder number of the node. The pseudocode for postorder calculation is given in algorithm 2.

Maximum level	Total number of nodes in tree
0	1
1	5
2	21
3	85
4	341
5	1365
6	5461
...	...

Figure 5.8: Table defining total number of nodes for trees of different height

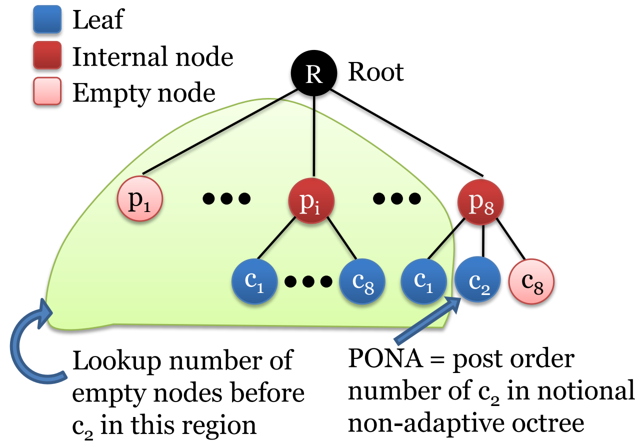


Figure 5.9: Calculation of Postorder number of a node

Algorithm 2 Postorder traversal

1: **for** all threads in **parallel do**
2:   **if** node is not empty **then**
3:     $PONA = 0, k = L_{max}$
4:     $CL = 1, NE = 0, start = 0$
5:     **while** $L > 0$ **do**
6:       $NST = I/2^{d(L-1)}$
7:       $PONA$ += $NST * Table[k-1]$
8:       **for** $i = 0; i < NST; i$++ **do**
9:         NE += $Tree[CL][i + start].numEmptyNodes$
10:       **end for**
11:       $I = I \mod 2^{d(L-1)}$
12:       $L = L - 1, k = k - 1$
13:       $CL = CL + 1, start = (start + NST) * 2^d$
14:     **end while**
15:     $PONA$ += $Table[k] - 1$
16:     NE += $Tree[CL-1][I_{orig}].numEmptyNodes$
17:     **return** $PONA - NE$
18:   **end if**
19: **end for**

5.3.2 Implementation 2: Parallel Memory Efficient Bottom-Up Adaptive Octree

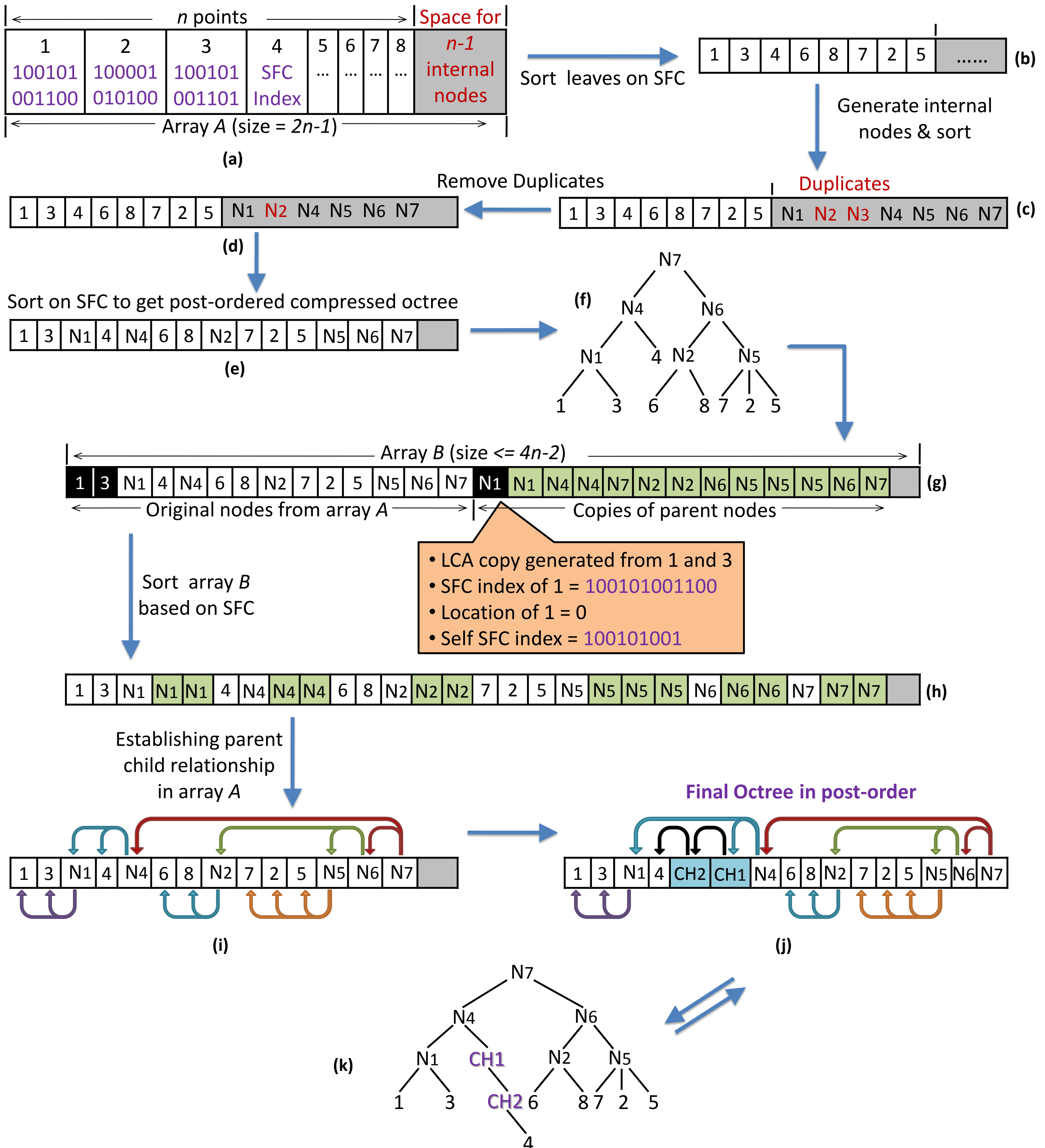


Figure 5.10: Parallel Bottom Up Adaptive Octree Implementation

A major drawback in implementation one is that it works only for non adaptive octrees. This uniform

nature is responsible for the direct evaluation of the parent child relationship among nodes. Even if we store an adaptive octree in this structure, we will end up wasting a large amount of memory corresponding to the empty nodes. Note, that the GPU does not support dynamic memory allocation. So, the maximum number of nodes (8^l) at each level $l \geqslant 0$ are defined at compile time only.

Now we pay our attention to the four steps of the new parallel octree construction algorithm on the GPU. We will first construct a compressed octree which will eventually be extended to octree. It performs data-distribution independent clustering and is useful for N-body simulations.

5.3.2.1 Compressed Octrees

In an octree each node corresponds to a cell which encloses at least one particle. Now in the cell hierarchy it may happen that a cell contains all its particles in a very small volume so that, its recursive subdivision may result in just one cell containing the particles for a large number of steps. In terms of the octree, long chains may form without any branching. Though the nodes along such a chain represent different volumes of the underlying space, they do not contain any extra information. As such no information is lost if the chains are compressed into a single node resulting in a compressed octree. Thus, *a compressed octree is an octree without chains*.

However, the appeal of compressed octrees might appear largely theoritical since in practice chains in octrees do not pose serious performance degradation in the run time or storage. But the properties of compressed octrees enable design of elegant algorithms which can be extended to octrees, if necessary. Furthermore, since the size of compressed octrees are distribution independent, they can be used to prove rigorous run-time bounds which in turn can explain why octree based methods perform well in practice.

5.3.2.2 Constructing Compressed Octrees

To encapsulate the spatial information otherwise lost in the compression, two cells are stored in each node v of a compressed octree, large cell $L(v)$ of v and small cell $S(v)$ of v. The large cell is defined as the largest cell that encloses all and only the particles the node represents. Similarly, the small cell is the smallest cell that encloses all and only the particles that the node represents. Note:

1. If a node (not leaf) 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. For a leaf, the small cell is the leaf itself and the large cell may or may not be same as the small cell.

2. The large cell of a node is an immediate subcell of the small cell of its parent.

3. The small cell of a leaf containing a single particle is defined to be the hypothetical cell with zero length containing the particle.

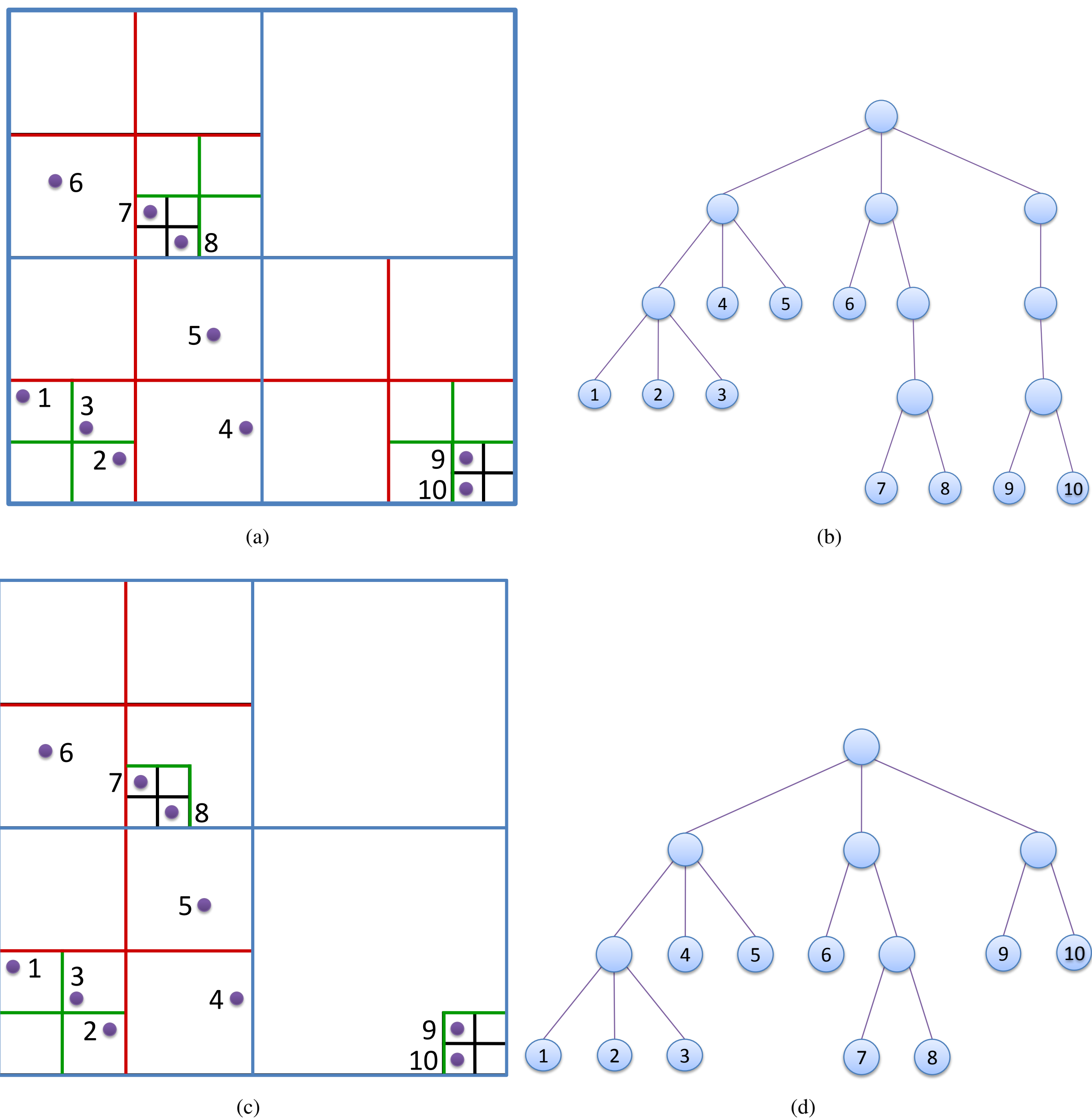


Figure 5.11: A 2D particle domain and corresponding quadtree and compressed quadtree

4. If the maximum resolution is specified, the small cell of a particle is defined to be the cell at the highest resolution containing the particle.

Let us consider the octree division of space upto level k. We construct the compressed octree bottom-up from leaves to the root. We first compute the root cell which is a cubical region (side length D) containing all the particles. The origin of this region is at the bottom left corner such that all the particles have positive coordinates in all the 3 dimensions. Thus, D is the maximum of all the coordinates in x, y and z directions taken simultaneously, for all the particles. Next, for each particle, we compute the index of the leaf cell containing it,

which is the cell at resolution k containing the particle. Thus a particle $P(P_x, P_y, P_z)$ will lie in the cell having integer coordinates $(\lfloor \frac{2^k P_x}{D} \rfloor, \lfloor \frac{2^k P_y}{D} \rfloor, \lfloor \frac{2^k P_z}{D} \rfloor)$ at level k. From the integer coordinates of the cell, the index can be generated as mentioned in chapter 4.

Now we sort the leaf cells based in their indices using any optimal parallel sorting algorithm. This creates the SFC-linearization of the leaf cells or the left to right order of leaves in the compressed octree. If multiple particles fall in a leaf cell then duplication can be eliminated during parallel sort and the particles falling within a leaf can be recorded. For convenience and without loss of generality, we assume that each point falls in a different leaf cell, giving rise to a tree with n leaves. The tree is then incrementally constructed using this sorted list of leaves starting from single node tree for the first leaf and the root cell. The pseudocode for tree construction is given in algorithm 3.

Algorithm 3 Bottom-up compressed octree construction

```
1:  Find the root cell and hence its side length D
2:  For each particle, compute the index of the leaf cell containing it
3:  Parallel sort the leaf indices to compute their SFC linearization
4:  Create a single node tree rooted a v for the first leaf and root cell
5:  while there are more leaves to insert do
6:     q = next leaf
7:     while q ⊈ L(v) do
8:        v = parent(v)
9:     end while
10:    if q ⊈ S(v) then
11:       Calculate the smallest subcell C of L(v) containing q and S(v)
12:       Create a new node u between v and its parent
13:       L(u) = L(v); S(u) = C
14:       Set L(v) and L(q) to the corresponding immediate subcells of C
15:       Make v and q children of u
16:    else if q ⊆ S(v) and v not a leaf then
17:       Set L(q) to the immediate subcell of S(v) containing q
18:       Insert q as a child of v
19:    end if
20:    v = q
21: end while
```

During the insertion process, we keep track of the most recently inserted leaf. Let q be the next leaf to be inserted. Starting from the most recently inserted leaf, traverse the path from leaf to the root until we find the first node v such that $q \subseteq L(v)$. Now two possibilities arise (see Figure 5.12):

1. q **not in** $S(v)$: In this case q lies in the region $L(v) - S(v)$, which was empty previously. Also, the smallest cell containing q and $S(v)$ is a subcell of $L(v)$ and contains q and $S(v)$ in different immediate subcells. Thus, we create a new node u between v and its parent and insert a new child of u with q as small cell.

2. **q in $S(v)$ and v not a leaf**: Consider the current children of v. None of them has q in its large cell because the node v is the first node on the path to the root which has q in its large cell. In other words, the compressed octree presently does not contain a node that corresponds to the immediate subcell of $S(v)$ that contains q. Thus, we insert q as a child for v corresponding to this subcell.

5.3.3 Implementation details

During the compressed octree construction we need to calculate the smallest subcell C of $L(v)$ containing q and $S(v)$ (line 11, algorithm 3). This is equivalent of finding the least common ancestor of two cells which may or may not be at the same level in the octree. For the internal nodes of the compressed octree we store only the small cell information, i.e. the coordinates of a corner and length of the cell. The large cell information can be obtained by subdividing the small cell of the parent node.

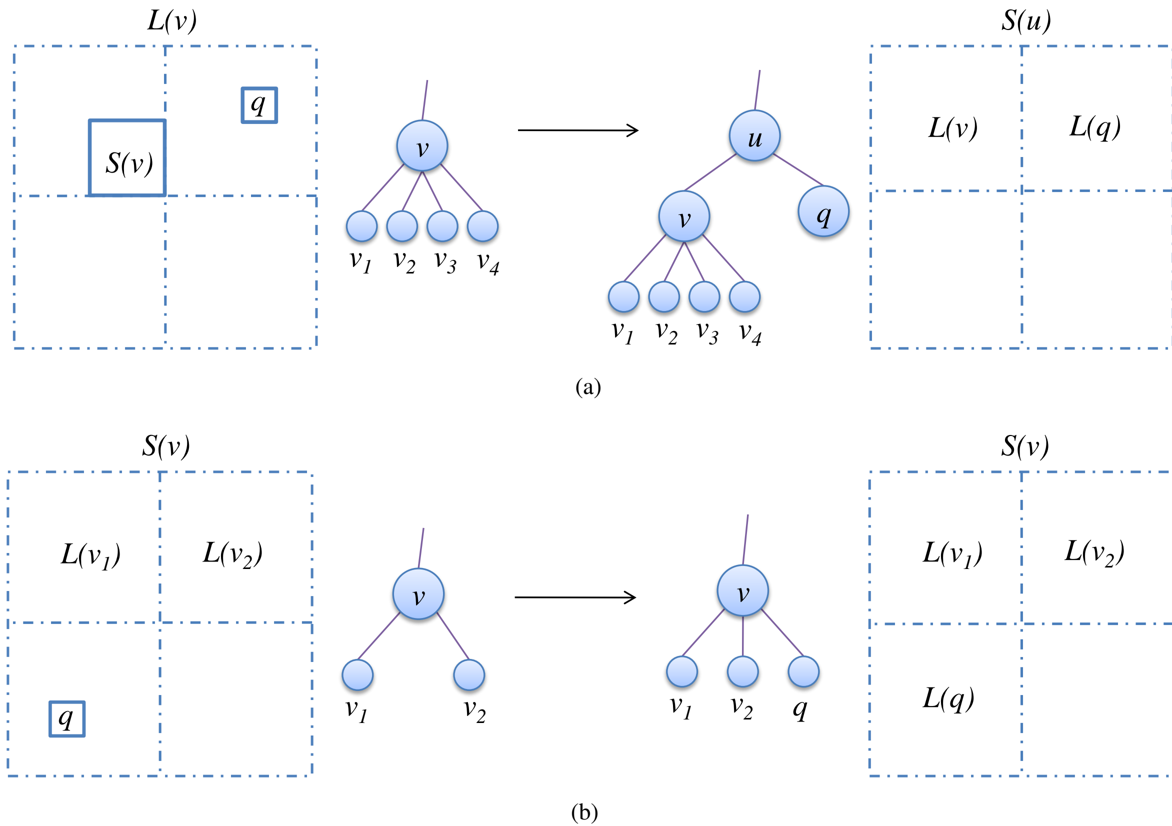


Figure 5.12: Two cases of the compressed octree construction

Let us denote the small cell and large cell of a node or leaf by [A(x,y), CellWidth], where A is the corner coordinate of the cell defined according to the axis scale at highest level k. The origin is at the bottom left corner of the 2-D particle region in consideration. The pseudocode for finding the least common ancestor is shown in algorithm 4. Note that the small cell of the leaf just added to the tree is the leaf itself.

The smallest subcell C of $L(v)$ containing q and $S(v)$ that we have found is actually the small cell $S(u)$ of

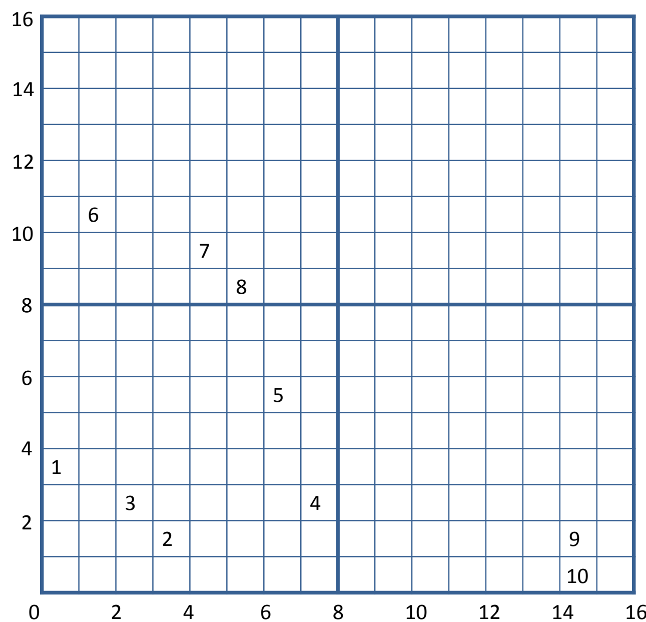
Algorithm 4 Least common ancestor

Input: Two cells A and B at levels k_A and k_B and widths w_A and w_B
Pre condition: $k_A \leqslant k_B$
Output: Least common ancestor of A and B

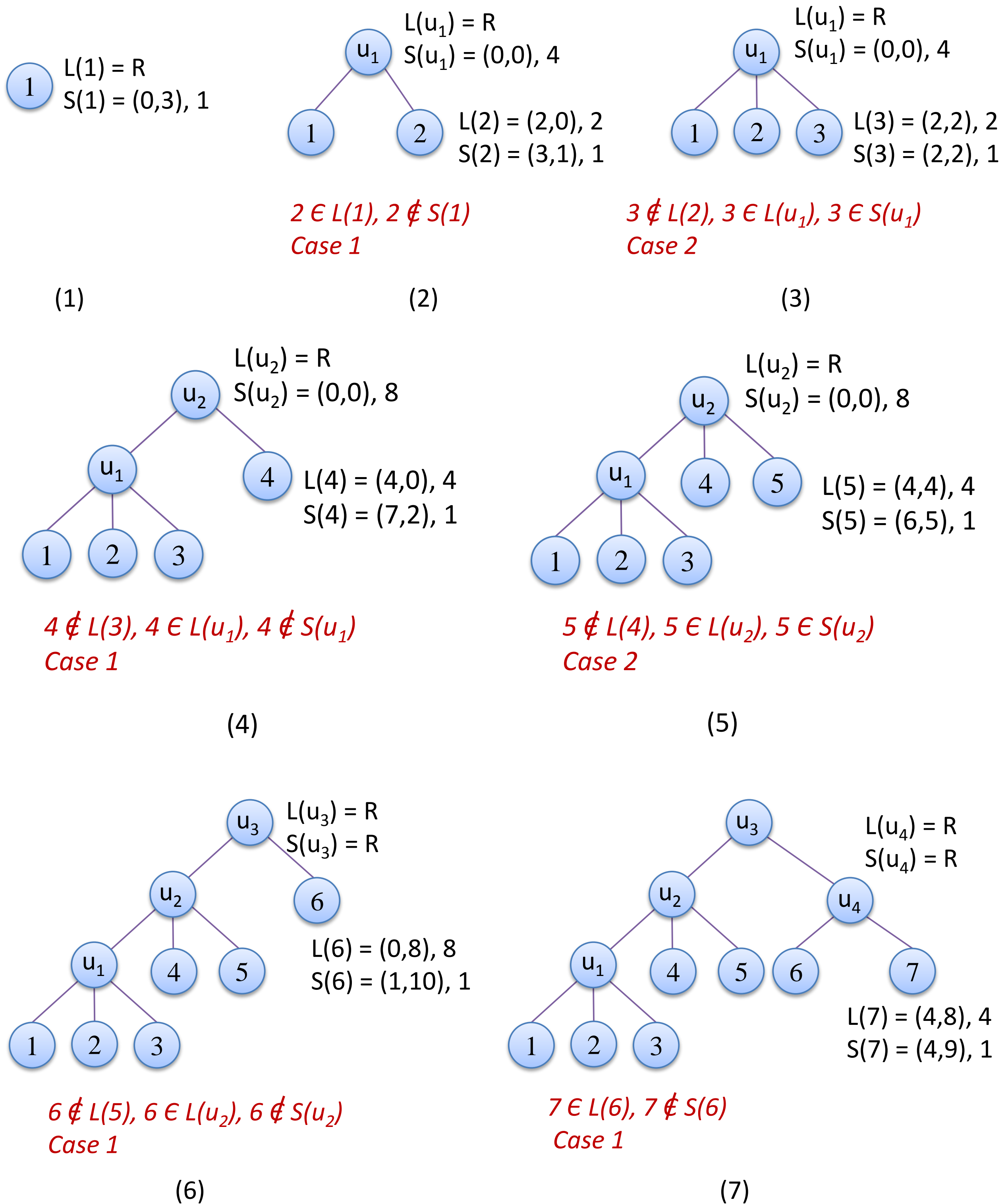
1: **while** $k_A < k_B$ **do**
2: $w_B = w_B * 2$
3: $B = [(\frac{B_x}{2}, \frac{B_y}{2}), w_B]$
4: $k_B = k_B - 1$
5: **end while**
6: **while** $A \neq B$ **do**
7: $w_A = w_A * 2$
8: $w_B = w_B * 2$
9: $A = [(\frac{A_x}{2}, \frac{A_y}{2}), w_A]$
10: $B = [(\frac{B_x}{2}, \frac{B_y}{2}), w_B]$
11: $k_B = k_B - 1$
12: **end while**
13: **return** $[A, w_A]$

the newly created node u between v and its parent (line 13, algorithm 3). The large cells $L(v)$ and $L(q)$ are set to the corresponding immediate subcells of C (line 14, algorithm 3).

A step-by-step construction of compressed octree from the octree shown in figure 5.11 is presented next. Note that a cell is represented by its bottom left coordinate and its width. R denotes the root cell. Nodes are inserted in the order 1, 2, 3, . . . , 9, 10.

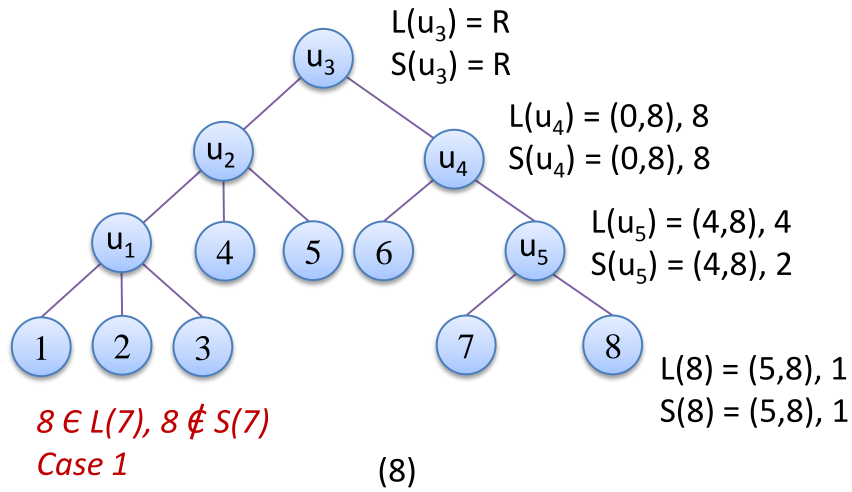


INPUT: n points belonging to some 3-d domain

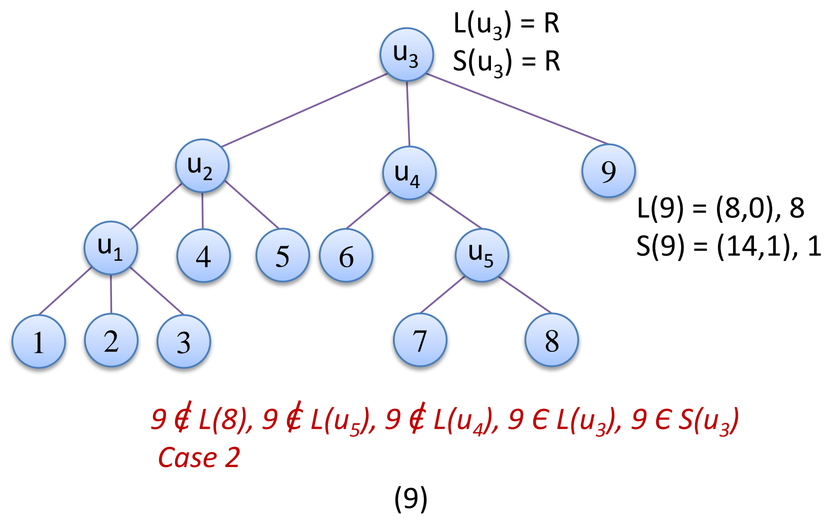


OUTPUT: Octree represented in post-order with parent-child relationships established.

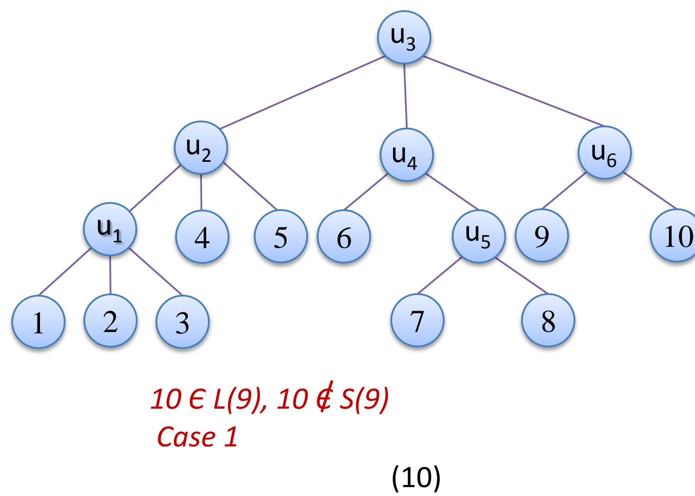
PROBLEM SETTING: For brevity we assume that the data of interest is available as points in a domain. For eg., these could be the points belonging to some 3-D point model of say, a Stanford bunny, or might represent centroids of triangular patches of some 3-D mesh. We make no assumption on the number of points in the



(8)



(9)



(10)

model. However, memory limitations of the GPU might possibly result in multiple points within a cell.

Before heading on, here are some of the intuitions behind the algorithm design.

1. BOTTOM-UP TRAVERSAL: Since every internal node in an octree has leaves in its subtree, given the leaves we can somehow decode this hierarchical inheritance information and generate the internal nodes.
2. PARALLEL STRATEGY: Each internal node can be considered as a Least Common Ancestor (LCA) of a particular leaf pair (in a compressed octree). Thus, given the leaves, generation of internal nodes can be parallelized since each of them can be generated independently from a leaf pair. Many leaf pairs might have the same LCA node resulting in duplicates which can be easily detected and removed.
3. Parent-Child relationship can be established and octrees can be generated from a given compressed octree using SFC indices across multiple levels.

The algorithm, with the help of Fig. 5.10, along with the implementation details is presented next.

1. CONSTRUCTING LEAVES
   (a) Read n points in the first n locations of an array A of size $2n - 1$. As shown in Fig. 5.10(a), we have 8 input points in this example.
   (b) Assuming a point per leaf, for every point, *in parallel*, do
      i. Generate the $3D$ co-ordinate of leaf cell to which it belongs (see ch. 4).
      ii. Generate SFC index (see ch. 4) for the leaf cell as shown in Fig. 5.10(a). For in-depth parallel GPU based SFC construction algorithm, please refer [AGCA08].
   (c) Sort [CUDb] the first n elements of array A, *in parallel*, based on SFC indices of leaves (Fig. 5.10(b)).
2. GENERATING INTERNAL NODES AND POST ORDER: *In Parallel*, for every adjacent leaves, find their LCA using the common bits (multiple of 3) in their SFC indices. For eg. say adjacent leaves L_1 and L_2 have their SFC indices as 100 101 1100 10 and 100 101 100 001 respectively, then the LCA is the internal node having SFC index 100 101
   (a) Allocate $n - 1$ GPU threads.
   (b) For every two adjacent leaves (say at locations i and $i + 1$) in array A, *in parallel*, generate the internal node and store it at location $n + i$ in array A (Fig. 5.10(c)).
   (c) Sort [CUDb], *in parallel*, the internal nodes generated, across levels based on their SFC indices. To do the same, we need to establish a total order on the cells across levels. If one is contained in the

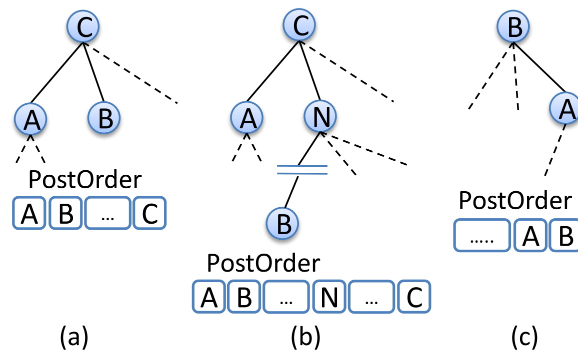


Figure 5.13: Computing Parent and Child

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. Fig. 5.10(c) shows sorted internal nodes with duplicates (N_2 and N_3) which might be generated.

(d) Allocate $n-2$ threads for a maximum of $n-2$ consecutive internal node pairs in the later half of array A to remove the duplicates.

(e) For every two adjacent internal nodes not having same SFC indices, *in parallel*, traverse back in the later half of array A starting from the current node to look for its duplicates and eliminate them (Remove node N_3 as shown in Fig. 5.10(d)).

(f) Sort array A, *in parallel*, based on SFC indices across levels to get the postorder traversal of a compressed octree (see ch. 4) (Fig. 5.10(e)).

Here we note that there might be some empty elements at the end of array A after sorting (the *gray* shaded area in Fig. 5.10(e)). We can not avoid this situation since CUDA does not support dynamic memory allocation and deallocation. So, an array of maximum required size $(2n-1)$ has to be declared at compile time only.

3. PARENT-CHILD RELATIONSHIP: The compressed octree represented by this post-ordered array A is shown in Fig. 5.10(f). The tree is shown only for the purpose of illustration as the parent-child relationships are still not established. To generate the parent-child relationship in the compressed octree, an intuition would be *since the tree is in the post order fashion*, a LCA of every two adjacent nodes would definitely be the parent of the first node in the pair considered. Three possible cases are shown in Fig. 5.13.

   Fig. 5.13(a) shows a case where both nodes A and B are siblings. Hence the LCA is the parent of A i.e. C. Fig. 5.13(b) is a case where B is the first node in the post-order fashion in the subtree of the node

N, adjacent to A. Again their LCA i.e. C is the parent of A. Third case shown in Fig. 5.13(c) is where, given two adjacent nodes in post-order fashion, node B is the parent of A. Hence their LCA is B.

Thus, considering every two adjacent nodes in post-ordered compressed octree, and generating their LCA gives us the SFC index of the parent of the first node in the pair, thereby establishing the parent-child relationship. Here are the implementation steps (Refer Fig. 5.10(g)) performed on GPU.

(a) Allocate an array B twice the size of the number of leaves and internal nodes (atmost $4n-2$). Copy the first half of array B with the current post-ordered array A of leaves and nodes.

(b) Allocate threads one less than the ($NumberOfLeaves + InternalNodes$).

(c) For every two adjacent nodes in the first half of array B, *in parallel*, do

    i. Generate the LCA from the SFC indices.

    ii. Copy the new node (copy of the parent of the first node in the pair considered) into the corresponding location in second half of the array B. (Generated copies of the nodes are shown in *green* in Fig. 5.10(g)).

    iii. Write in this new node, the SFC index of the first node of the node-pair which generated it, along with the location of that node in array A. This location information will eventually give the index of the child this parent node-copy was generated from. Fig. 5.10(g) shows an example of the same. We expand the copy-node N_1 (in *black*) generated by leaves 1 and 3 (both shaded in *black*) and show the information it stores (Information box shaded in *orange*).

(d) Sort array B, *in parallel* based on newly generated SFC indices. All the parents and their copies will come together (Fig. 5.10(h)).

(e) For every two adjacent nodes both having same SFC indices and atleast one of them not being a generated copy, *in parallel*, do

    i. Establish the parent-child relationship. Here we see that one of the nodes is the original node and another is its copy (generated in step 3(c)ii). The copy will give the location of the child in array A while we get the location of the parent from the original (Fig. 5.10(g) and Fig. 5.10(h)).

    ii. Scan ahead in array B and repeat step 3(e)i for all the copies of the original to establish the relationship between the parent and *all its children*. Step 3(e)i will be repeated atmost 7 times since in an octree, a parent can have atmost 8 children. Referring Fig. 5.10(h) and Fig. 5.10(i), step 3(e)i will be repeated twice for N_1 since we have two generated copies (and hence two children) of it. Similarly step 3(e)i will be repeated twice for N_4, N_2, N_6 and N_7 while thrice for N_5 since it has 3 children.

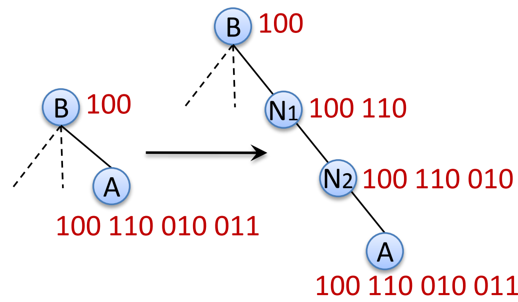


Figure 5.14: Compressed Octree to Octree

4. GENERATING OCTREE FROM COMPRESSED OCTREE

We now move on to the final step of our algorithm where we need to generate octree from compressed octree. Consider two adjacent nodes, say A and B with A being the child of B, and calculate the difference of octree depths between the two using their SFC indices, and finally add those many intermediate nodes in the chain between A and B. For eg. if A and B have SFC indices as $100\ 110\ 010\ 011$ and 100, then the level difference is 3 (B is at depth 1 while A at depth 4, assuming root at depth 0).

This difference indicates a chain of nodes between A and B which are missing in the compressed octree, as shown in Fig. 5.14. This chain can now easily be generated, thereby giving us the final octree. The implementation steps are summarized next.

(a) Allocate threads equal to size of current array A i.e. (no. of leaves + no. of internal nodes). Array is in post-order fashion.

(b) For every two adjacent nodes with *first node in the pair being a child of the second, in parallel*, write the level difference (level of first - level of second - 1) in the first node. This level difference gives us the count of memory needed between these two nodes.

(c) Do *parallel prefix* [CUDb] on level differences in A to get the total amount of memory needed to insert the internal nodes so as to make an octree (due to no support for dynamic memory allocation). While doing parallel prefix, keep track of number of nodes to be inserted before the current one, so that the index or the array location for the new node to be inserted can directly be identified.

(d) Allocate required memory for new nodes.

(e) Allocate threads equal to the size of current array A minus 1.

(f) *In parallel*, check for every node having a level difference greater than 1 with its parent, and generate new nodes to be inserted after the current node. Write them in the array location decided in step 4(c) above. As shown in Fig. 5.10(j), we add two *chain nodes* CH_1 and CH_2 between N_4 and 4 to get a complete octree as shown in Fig. 5.10(k).

DISCUSSION: Maximum memory required for implementation is just $4n - 2$ for storing array B. It is far less than occupied by octree implementation in [AGCA08] (Implementation 1). This implementation is slightly slower than one presented in [AGCA08] (Implementation 1). However, the advantage we gain due to high memory saving out-peforms the timing comparisions between them. Further, we easily win against the same algorithm implemented on the CPU. It is a nicely load-balanced algorithm as each thread does almost the same amount of computations through out the algorithm. Here are some example queries our octree supports and solution for the same.

- PARALLEL POST-ORDER TRAVERSAL: Since our output is in post-ordered form, this query is implicitly answered.
- PARENT-CHILD RELATIONSHIP: For dimension d and level l, if dl is the number of bits in the SFC index representing child C_1, then the parent can be *directly* given by its first $d(l-1)$ bits.
- GIVEN A POINT $(P_x,\ P_y,\ P_z)$, FIND WHICH NODE IT BELONGS TO: The co-ordinates of the desired node are $(\lfloor 2^k P_x/D \rfloor,\ \lfloor 2^k P_y/D \rfloor,\ \lfloor 2^k P_z/D \rfloor)$, where k is the number of times the space has been bi-sected and D is the sidelength of space enclosing all points in the model.
- IS NODE C_1 CONTAINED IN NODE C_2? C_1 is contained in C_2 if and only if the SFC value of C_2 is a prefix of the SFC value of C_1
- GIVEN C_2 AS A DESCENDANT OF C_1, RETURN CHILD OF C_1 CONTAINING C_2 For dimension d and level l , dl is the number of bits representing C_1.The required child is given by the first $d(l+1)$ bits of C_2.
- LEAST COMMON ANCESTOR OF NODES C_1 AND C_2: The longest common prefix of the SFC values of C_1 and C_2 which is a multiple of dimension d gives us the least common ancestor.

Many other such basic queries (like *Neighbor Finding*, *Leaves in a node's sub-tree* etc.) can be supported.

5.3.4 Implementation 3: Parallel Memory Efficient Top-Down Adaptive Octree

We now look at a new and quite a different way to generate an octree in parallel. The problem setting is same as that in § 5.3.2 but as opposed to algorithm of §5.3.2, this is a *top-down* parallel adaptive octree generation algorithm. The intuition behind this algorithm is to *iteratively cluster* the points belonging to the same node together, starting from the root till we construct the leaves. As each cluster generation is independent of the other, on each iteration, the cluster generation process can be parallelized. An example to explain the same is shown in Fig. 5.15.

In this section we present another octree construction algorithm on the GPU that we implemented using the latest NVIDIA GPUs featuring support for *atomic operations* like atomic add/substract, atomic increment/decrement, atomic max/min etc [CUDa]. An operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. These GPUs have G92 architecture as compared to the G80 architecture (no support for atomic operations) on which we have done other implementations. This algorithm is easy to understand and implement.

PROBLEM SETTING: We now look at a new and quite a different way to generate an octree in parallel. The problem setting is same as that in implementation 2 (see sec. 5.3.2) but as opposed to implementation 2, this is a top-down parallel adaptive octree generation algorithm. The intuition behind this algorithm is to *iteratively cluster* the points belonging to the same node together, starting from the root till we stop on constructing the leaves. As each cluster is independent of the other, on each iteration, the cluster generation process can be parallelized. An example to explain the same is shown in Fig. 5.15.

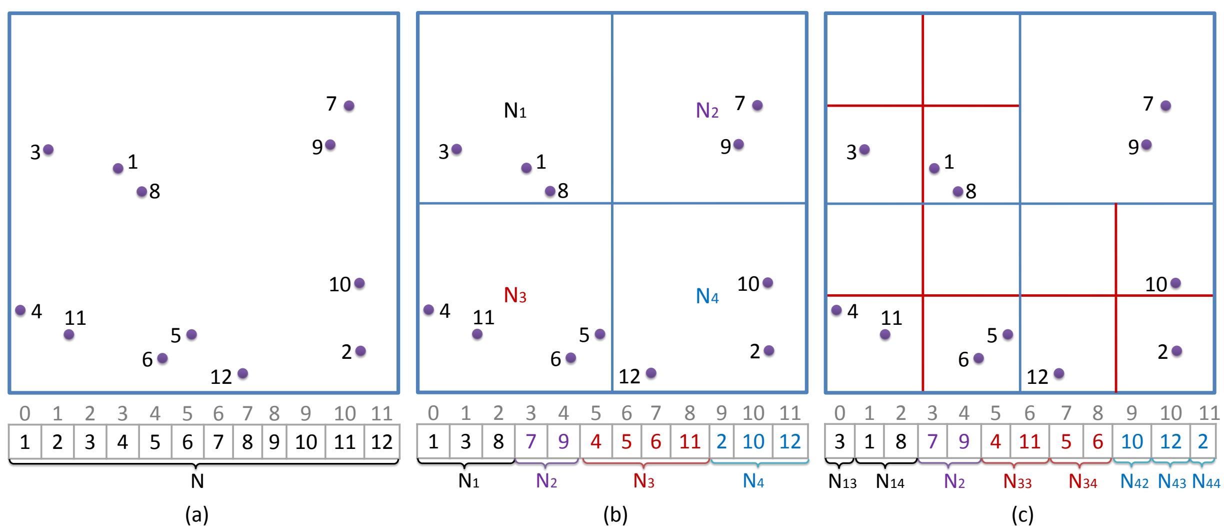


Figure 5.15: Spatial Clustering of Points

Here in Fig. 5.15(a), we see an array of points enclosed in some space. We now try to cluster these points based on their locations with respect to nodes of the octree. Assume the space enclosing the points to be the root of the octree. We now divide the root into its children as shown in Fig. 5.15(b). Here we see that points $1,\ 3,\ 8$ belong to child N_1 of root, points $7,\ 9$ belong to child N_2 and so on. Hence we swap these points accordingly in the array (Implementation fact: we swap the pointers, not the actual data) so that they cluster together as shown in Fig. 5.15(b). We iteratively repeat this process till we have less than some pre-defined points (2 as in Fig. 5.15) in a node and term it as a leaf. Fig. 5.15(c) shows this recursion and the final point array after all the swaps. The octree nodes generated now just need to store the *start and end bounds* defining their cluster of points in the point array. For eg., node N_1, as shown in Fig. 5.15, stores its *start bound* as array location 0 and end bound as 2, while node N_4 stores them as 9 and *end bound* as 11. Further, node N_{34}, child on N_3, stores bounds as 7 and 8 and so on for all the nodes.

This intuition on building the octree can easily be extended to a parallel algorithm. As we can see, we move down level by level. Thus, on every iteration, we swap the points and create new *partitions* in the point array. An important thing to note is that *all these partitions can be generated independently of one another and thus can be parallelized*. Hence, initially for the *root* we have a single thread generating 8 new partitions corresponding to 8 of its children. We then have *maximum* of 8 threads generating maximum of 64 new partitions corresponding to 64 grand-children on the *root* (*maximum* of 8 because some nodes might turn to leaves and won't be divided further and their thread stops); then *maximum* of 64 threads and so on. *The degree of parallelism increases as we move down to the greater depths of the octree generation process*.

Having got a brief overview, we now present the algorithm with some implementation details.

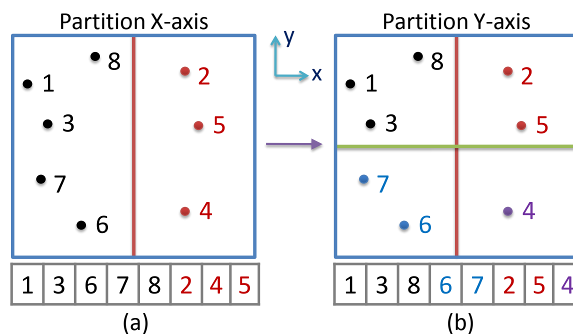


Figure 5.16: Spatial Clustering of Points

INPUT: n points belonging to some 3-d domain

OUTPUT: Octree with parent-child relationships established

1. Read points in an array P of size n.

2. Initialize the $root$ node of the octree as containing all points of P. Set the bounds defining cluster of points belonging to the root as 0 and $n-1$.

3. Now loop on current step

   (a) Allocate threads equal to the number of partitions. ($Num_Threads = 1$ initially for the root and then increases as we iterate)

   (b) For every thread, *in parallel,* do

      i. STOP the thread if the current partition is a leaf.

      ii. ELSE, create 8 new partitions and 8 new octree nodes. Record the respective partition bounds in the nodes created. To create 8 new partitions, we first divide the current partition along the longest axis (x, y or z) and swap the points belonging to one side of the partition with another as shown in Fig. 5.16(a). We then repeat the same process and divide the 2 new partitions along the second longest axis, as in Fig. 5.16(b), and finally along the third. For purpose of illustration, we have shown partitioning a quadtree instead of an octree.

   (c) STOP the loop when every thread running encounters a leaf and hence no new partitions are generated.

Here are some of the implementation details.

1. MEMORY ALLOCATION: Every iteration of the algorithm creates many new partitions and octree nodes. We need to allocate memory to store this newly generated information. The problem arises here because GPU doesn't allow for dynamic memory allocation. One way to get around this is to allocate maximum possible memory. But this eventually leads to storing the whole tree ($8^0 + 8^1 + 8^2 + ... + 8^l$) till level l, and there by wasting a huge amount of memory [AGCA08].

   A better solution is to *pre-compute*, in the current iteration, the number of nodes which will be generated at the next iteration. We can thus allocate *only the desired memory* before the next iteration starts.

   This can be achieved by setting a global *Num_Leaves* variable. This will be used to count the leaves which are formed in the current iteration and hence these won't be partitioned further. Every thread, after creating the partitions, checks whether any of the 8 partitions is a leaf or not. If YES (For eg. 2 of the 8 are leaves) it increments the global *Num_Leaves* variable by those many leaf-counts (For eg. *Num_Leaves*+=2). We use *atomic increments* available in latest G92 architechture of GPU so that every thread increments it by a desired amount and the final outcome is the total number of leaves at current

level. The new global memory allocated then would be (Nodes at current level - *Num_Leaves*)*8. 8 here refers to the 8 new partitions generated by each thread.

2. INDEXING MECHANISM: We know that the partitions generated by the iteration will be partitioned further in the next iteration, provided they don't represent a leaf. Thus, there might be threads in between which are stopped as they represent a leaf. Hence, a proper indexing and offset mechanism must be installed so that the threads know where to write the new partitions in the global array, as shown in Fig. 5.17.

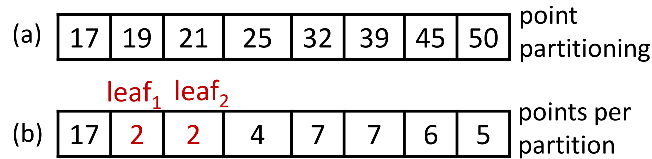


Figure 5.17: Partition Array of a Node

We have a $Node_n$ with say 50 points. Let $Node_1$, $Node_2$, $\ldots Node_8$ be the children of $Node_n$. As in Fig. 5.17, points $1-17$ belong to $Node_1$, $18-19$ to $Node_2$ and so on. Let us assume that a leaf is formed when the node has 3 or less points. Thus $Node_2$ and $Node_3$ are leaf nodes. Hence the memory allocated for next iteration is $(8-2)*8 \ = \ 48$ for 48 new partitions. So $thread_1$ will write its 8 partitions at locations $1-8$, $thread_2$ at $9-16$ and so on. But since $Node_2$ and $Node_3$ are leaves, $thread_4$ will now write the new partitions at locations where $thread_2$ was suppose to write i.e. $9-16$ and the remaining threads will follow the offset. So *every node must know how many leaves are present before itself in the array. One can find this using a simple parallel prefix sum [CUDb] on the array.*

Thus, the new location to write new partitions is, say for node *A* is *(original location to write - 8*Number of leaves before A)*. This gives a unique indexing for every thread and memory is allocated only as much as desired.

3. PARENT-CHILD: This relationship is established while partitioning itself as every child partition is generated from its parent, thereby giving us our desired octree.

DISCUSSION: Maximum memory required for implementation is just equal to storing non-empty octree nodes, very less compared to [AGCA08] (Implementation 1). However, it looses w.r.t time when compared to [AGCA08] but is very fast compared to the CPU implementation. As it performs data-dependent clustering, it generates a different octree compared to our first implementation. Example application areas include color quantization, collision detection, visibility determination. Thus they have different application areas and hence we dont compare them against each other. Parent-Child, containment, range, and neighbor-finding are some example queries which it can answer.

5.3.5 Comparison between Implementation 1 and Implementations 2/3

Implementation 1 requires much more memory as compared to implementations 2 and 3. If the tree is highly adaptive and have larger number of levels (say 12-15) then in implementation 1 we end up in allocating a large amount of memory (8^l for each level l) most of which remain unused as long as the octree is present on the GPU. Thus, a lot of device memory is wasted. Moreover, implementation 1 suffers from the problem of bank conflict since at a single instance many threads may try to read the same memory location in the tree. This, can hurt the overall performance. Thus, implementation 1 is not of much interest on current GPUs. however with future GPUs supporting huge memories, it can be a prospect since it provides the fastest of the answers.

5.3.6 GPU Optimizations

To improve the GPU kernel's performance in implementations 2 and 3, we utilize several optimization techniques enlisted below.

1. LOOP UNROLLING: Loop unrolling achieves a modest speedup compared to our initial implementation. We found that especially the loops with global memory accesses (as it is the case in our algorithm) in them benefit a lot from unrolling.

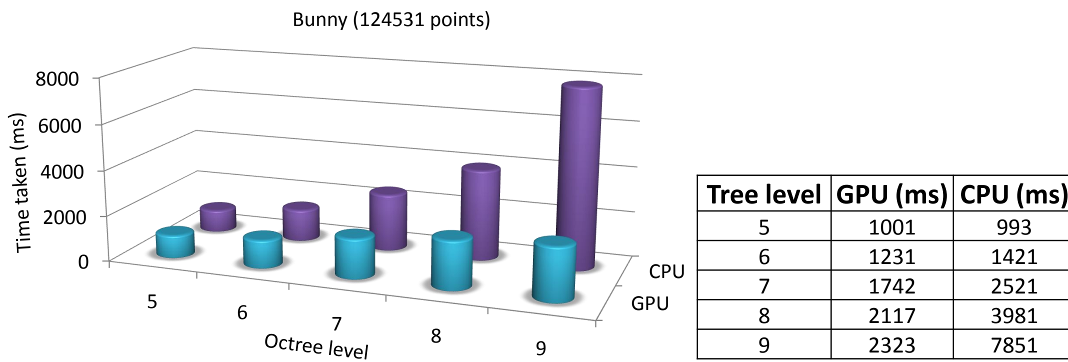
2. OPTIMAL THREAD AND BLOCK SIZE: Each thread block must contains $128 - 256$ threads and every thread block grid no less than 64 blocks for a 16 multiprocessor configuration for optimal performance, which was obtained via an empirical study. We made sure this was achieved. If the number of nodes at a level is not a divisor of the block size, only the remaining number of threads is employed for computations of the last block.

3. OPTIMAL OCTREE DEPTHS: The efficiency of both our kernels ($\mathbf{I}_2$ in implementation 2 and $\mathbf{I}_3$ in implementation 3) is substantial as, in the former, we have every thread working only on two adjacent nodes most of the times while in the latter every thread works on an independent partition. The advantage achieved is that the work of each thread is completely independent eliminating the need for any shared memory. This perfectly fits our situation where each thread (in any of the two implementations) on finishing its work or on making an early exit (say by encountering a leaf) simply moves on to next pair of adjacent nodes for $\mathbf{I}_2$ or a new partition for $\mathbf{I}_3$, without the need for synchronizing with other threads. Note that to realize the full GPU load the number of nodes to be considered should be sufficiently large. Indeed, if an optimal thread block size is 128 and there are 16 multiprocessors (so we need at least 64 blocks of threads to realize an optimal GPU load), then the number of nodes should be at least 8192 for a good performance. Thus, we realize a full GPU load for $\mathbf{I}_2$ at *even* a small enough point model (of size 8192, and assuming a point per leaf). On the other hand, for $\mathbf{I}_3$, if the octree is built till depth 4, we have

at most $8^4 = 4096$ leaves and for depth 5 this number becomes $8^5 = 32768$. Thus, GPU works to its full potential at a small enough octree depths and the efficiency increases as we move down to greater depths ($\geqslant 6$). GPU totally out-performs the CPU for depths $\geqslant 6$.

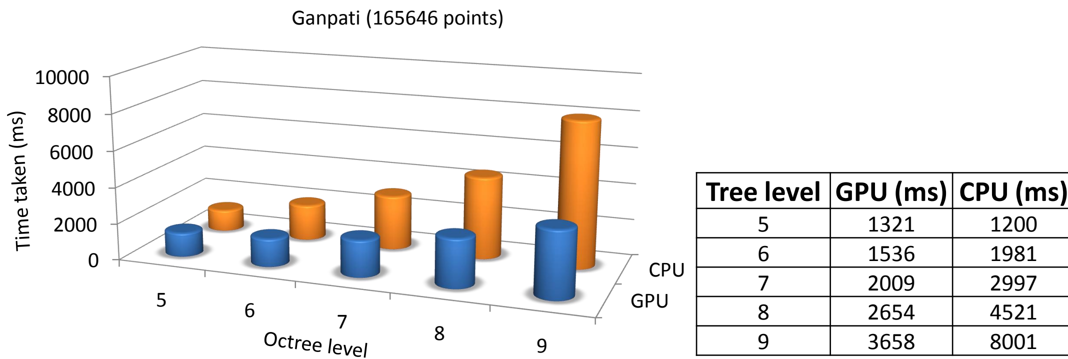
5.4 Results

In this section we compare our implementation of octree on the GPU with the corresponding implementation on the CPU based on running time. We use 3-d points models of bunny and Ganesha in a Cornell room as inputs to create the octree.



Tree level	GPU (ms)	CPU (ms)
5	1001	993
6	1231	1421
7	1742	2521
8	2117	3981
9	2323	7851

Figure 5.18: Top-Down Octree Construction (Bunny 124531 points) (sec. 5.3.4)



Tree level	GPU (ms)	CPU (ms)
5	1321	1200
6	1536	1981
7	2009	2997
8	2654	4521
9	3658	8001

Figure 5.19: Top-Down Octree Construction (Ganpati 165646 points) (sec. 5.3.4)

We see that the GPU outperforms the CPU at higher levels. We implemented the top-down GPU-based parallel octree construction algorithm using the latest NVIDIA GPUs featuring support for atomic operations like atomic increment/decrement etc. These GPUs have G92 architecture. The machine used has a Intel Core 2 Duo 1.86 GHz with 2 Gbs of RAM, NVIDIA Quadro FX 3700 with 512 Mbs of memory and Fedora Core 7 (x86_64) installed on it.

Chapter 6

View Independent Visibility using V-map on GPU

In point-based graphics [DTG00, KTB07, GD98, MGPG04], a scene represented as points is to be rendered from various viewpoints keeping global illumination in mind [DYN04]. That is, a point may be illuminated by other points, which in turn are illuminated by still other points, and so on. 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. A visibility preprocessing step for finding view independent mutual point-pair visibility is useful before the costly rendering equation is solved. The visibility map (V-map) data structure (see sec. 6.3) was introduced for this purpose. The basic idea here is to partition the points in the form of an octree. When large portions of a scene are mutually visible from other portions, a visibility link is set so that groups of points (instead of single points) may be considered in discovering the precise geometry-dependent illumination interaction. Ray shooting and visibility queries can also be answered in sub-linear time using this data structure.

6.1 Prior Work: CPU-based V-map Construction [Gor07]

We, in this section, present the CPU-based mutual point-pair visibility algorithm (Section 6.1.2) 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 for aggregated point clouds.

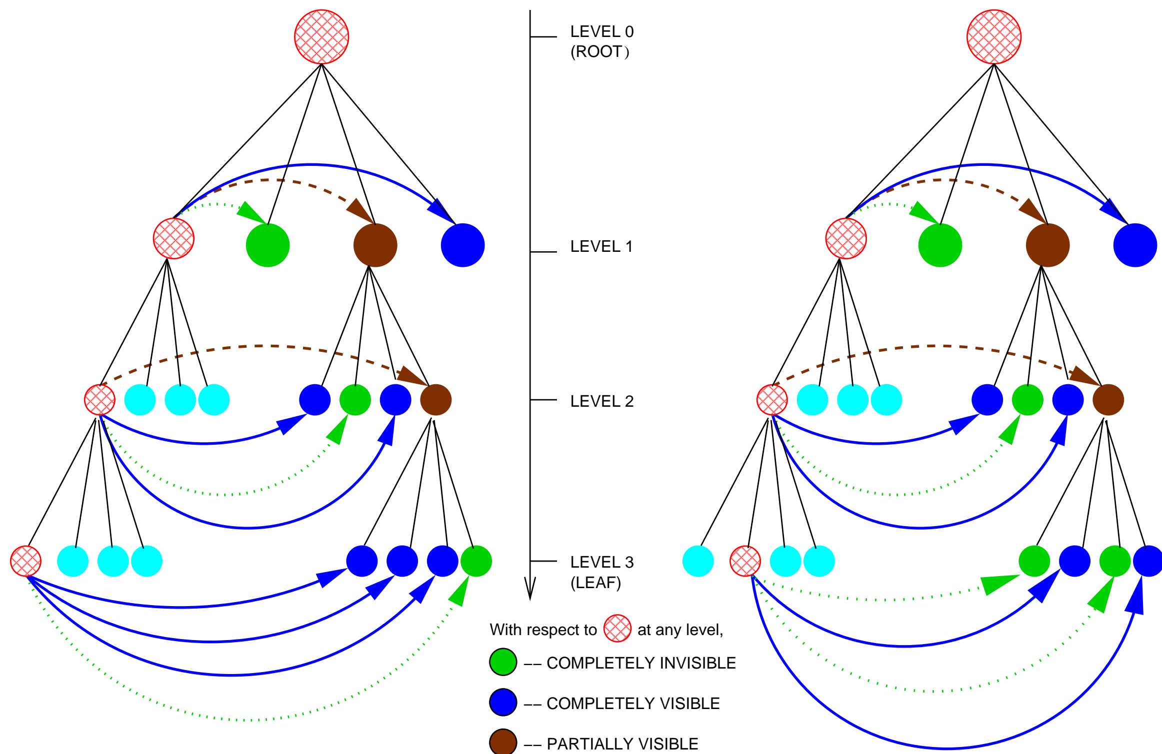


Figure 6.1: Views of the visibility map (with respect to the hatched node in red) is shown. Every point in the hatched node at the first level is completely visible from every point in only one node (the extreme one). At level 2, there are two such nodes. The Figure on the left shows that at the lowest 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.

6.1.1 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 6.2 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 6.1 provides different views of the *visibility map*. (The illustration shows directed links for clarity; in fact, the links are bidirectional.)

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.

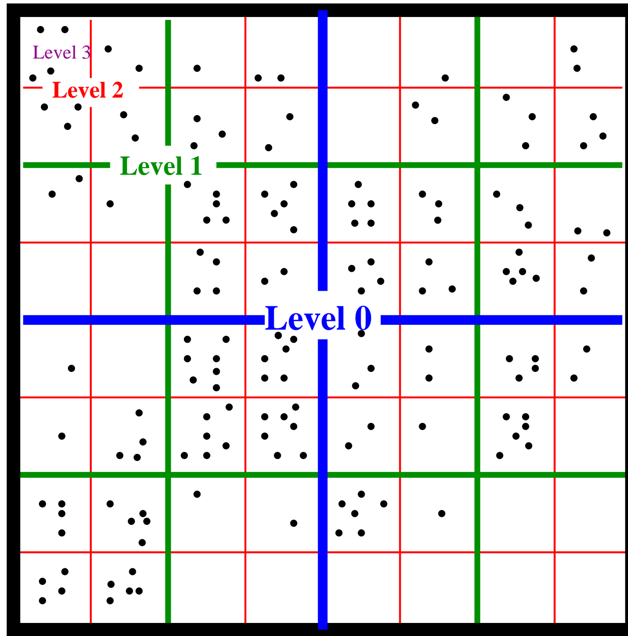


Figure 6.2: Leaf nodes (or cells, or voxels) are at level three.

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.

We first explain the modified *point–pair visibility algorithm* in subsection 6.1.2 and follow it up by extending it to construct the V-Maps in the *most efficient* manner in subsection 6.1.4.

6.1.2 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].

Consider two points *p* and *q* with normals n_p & n_q as in Figure 6.3. We run the following tests to efficiently produce $O(1)$ possible occluders.

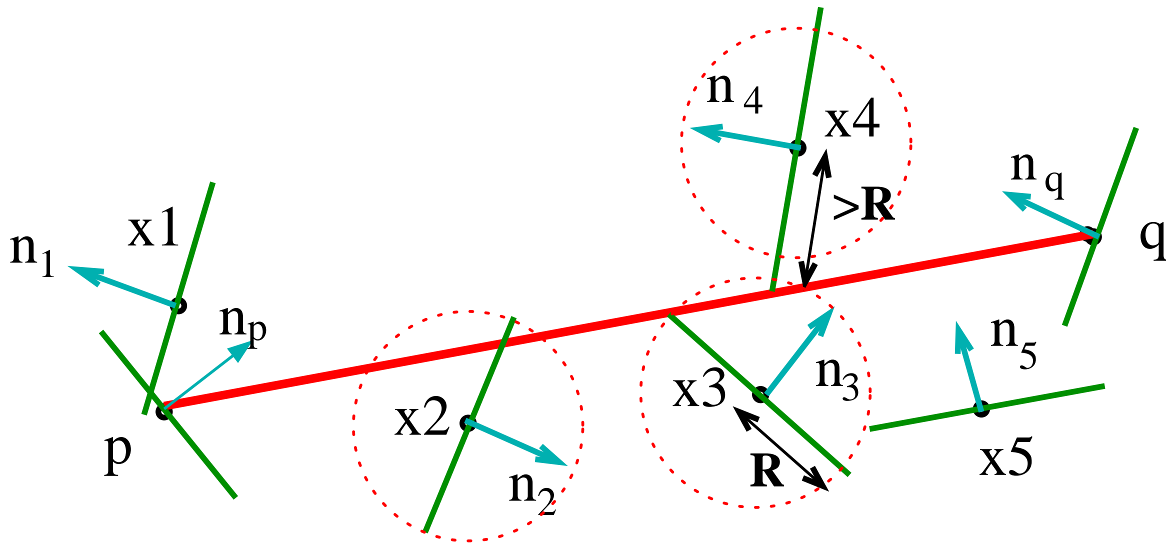


Figure 6.3: Only x_2 and x_3 will be considered as occluders. We reject x_1 as the intersection point of the tangent plane lies outside segment $\overline{pq}$, x_4 because it is more than a distance R away from $\overline{pq}$, and x_5 as its tangent plane is parallel to $\overline{pq}$.

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

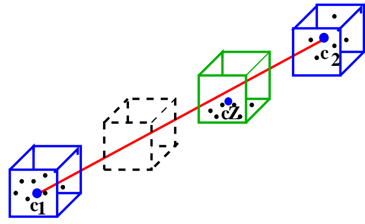
2. Determine the possible occluder set X, of points close to $\overline{pq}$ which can possibly affect their visibility. As an example, in Figure 6.3, points $(x_1, x_2, x_3, x_4, x_5) \in X$. An efficient way to obtain X is to start with the output of a 3D Bresenham line segment algorithm [E.62] between 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 6.3.

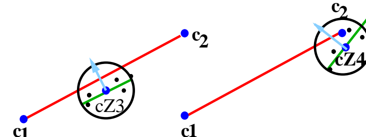
Any point from X which fails any of the tangent tests is considered an occluder to $\overline{pq}$. If we find K such occluders, q is considered invisible to p.
Elimination of thresholds as compared to previous point–pair visibility approach [Gor06] simplifies the tasks for the user and also helps in 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.

6.1.3 Octree Depth Considerations

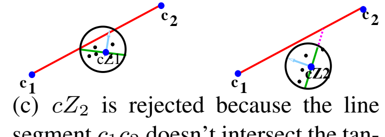
In a hierarchical setting, and for sake of efficiency, we may terminate the hierarchy to some level with several points in a leaf. A simple extension of our point–pair visibility algorithm to a *leaf–pair* would be to compute visibility between their centroids (p and q, Figure 6.3). Set X now comprises of centroids, each corresponding to a intersecting leaf (ILF). Our occlusion criteria is then:



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

- 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 6.4).

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 [Gor07].
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.

6.1.4 Construction of Visibility Maps

We now extend the leaf–pair algorithm (subsection 6.1.3) 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 [Gor06]. We also give a brief overview of how the constructed V-Map can be applied to compute a global illumination solution.

Algorithm 5 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 6 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 6.1.3)
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**

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 6.1.4 initially for the $root$ node, which sets up the relevant visibility links in New Interaction List(n-IL). This algorithm invokes Algorithm 6.1.4 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.

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.

6.2 GPU-based V-map Construction

A sequential implementation of the V-map §6.1 takes hours (for octrees with height 8 or more). Reducing the octree height (to say 7 or below) in the interests of time yields unacceptable results (Fig. 6.5).

Our work is concerned with computing the V-map data structure on the GPU. Specifically,

Figure 6.5: Dragon viewed from the floor (cyan dot). The quality is unacceptable for octrees of heights of 7 (left) or less. The figure on the right is for an octree of height 9.

1. If a black-box "kernel" is available to compute the relationships (e.g., gravitational interaction, point-pair visibility), we show how a hierarchical data structure can be built efficiently (see sec. 6.4) on the GPU using CUDA. For example, our V-map data structure shows 11 fold speedup (averaged over various models and octree heights 8 or more). While a point model of a dragon placed inside a point modelled Cornell room, sub-divided with octree height of 8, takes more than a couple hours, on the CPU, for visibility computation, GPU performs the same in some minutes.

2. The specific kernel of point-pair visibility (namely, is point p visible from point q) proposed in [GKCD07] is analyzed, and an alternate but related formulation is given which is more suitable for GPU implementations.

6.3 The Visibility Map

We assume that the data of interest is available as points; for example, these could be the points belonging to some 3-D point model of say, the Stanford bunny, or might represent centroids of triangular patches of some 3-D mesh. Given a 3D point model, we sub-divide the model space using an adaptive octree (leaves appear at various depths). The *root* represents the complete model space and the sub-divisions are represented by the descendants in the tree; each node represents a volume in model space.

The visibility map (or V-map) for a tree, as defined in [GKCD07], 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 at the same level; every point in any node in L is guaranteed to be visible from every point in p. Fig. 6.6 shows the link structure for some node N. Combining all such link structures defined for every node gives the complete V-map of the tree.

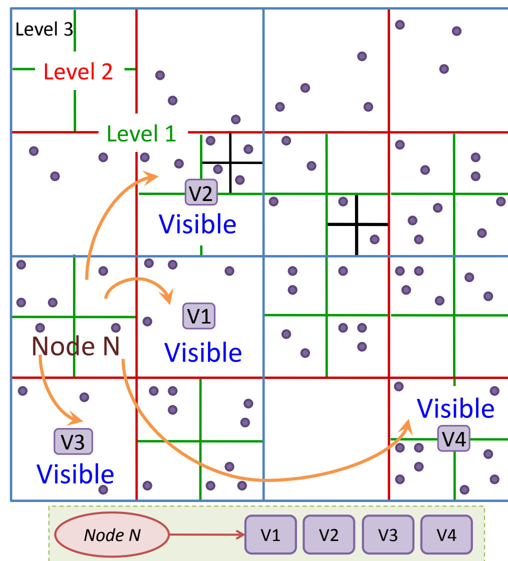


Figure 6.6: Visibility links for Node N

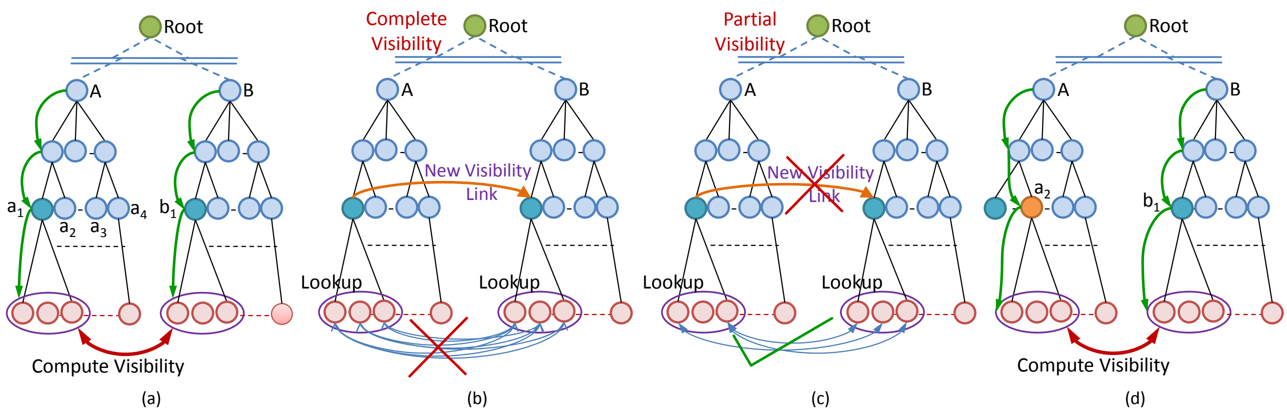


Figure 6.7: The visibility map is constructed recursively by a variation of depth first search. In general, it is advantageous to have links at high level in the tree so that we can reason efficiently using coherency about the visibility of a group of points.

CPU V-MAP CONSTRUCTION: Consider a node-pair AB in Fig. 6.7 for illustration to see if we should set the visibility link between the two. The same procedure is repeated for all node-pairs in the octree so as to define a complete V-map.

To establish the visibility link between A and B we need to check if all the points in the leaves defined by A are visible to all the points in B. In doing so, we might as well take advantage of this work, and set the visibility links of all descendants of A with respect to all descendants of B. In Fig. 6.7(a) we see (green arrows to the extreme left) how we recursively go down and compute the visibility between a_1 and b_1 with the help of their leaves. This information is now propagated upwards in the sub-tree depending on the type of relation established at the leaves. In our example, Fig. 6.7(b), if all leaves of a_1 are visible to all leaves of b_1, then the

visibility link between a_1 and b_1 is set (using *dynamic memory allocation*) with no links between their leaves However, if only some leaves of a_1 are visible to some leaves of b_1, then its a case of *partial* visibility and no new link is propagated upwards between a_1 and b_1 (Fig. 6.7(c)). The same process is repeated then for a_2 and b_1 (Fig. 6.7(d)) and then for rest of the descendants. It is easily observed that recursion (not available on the GPU) is a natural way of efficiently constructing the V-map.

With the setting given above (e.g., links only at the same level) we restrict the pairing of nodes to a subset of the all node pairs set. It is common in the scientific computing literature to define this subset as an *interaction list*. Every node has its own interaction list and its cardinality may vary from node to node.

As mentioned earlier, a sequential implementation of the V-map construction given in [GKCD07] takes hours (for octrees with height 8 or more). Employing octrees of greater height, it is therefore desirable to exploit the inherent parallelism in the V-map construction algorithm and get both quick and accurate results. Parallelism stems from the fact that the visibility between a node pair, say a_1 and b_1 in Fig. 6.7 is entirely independent of the visibility between another node pair, say a_2 and b_1.

6.4 V-map Computations on GPU

Various ways to parallelize the V-Map computations on the GPU are addressed below.

6.4.1 Multiple Threads Per Node Strategy

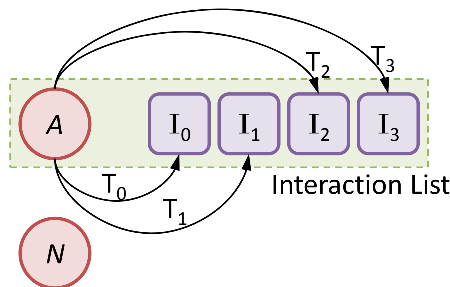


Figure 6.8: Parallelism at a node level

One of the intuitive ways to parallelize the algorithm is to make each thread compute the visibility between any node A with a node in its interaction list. For example, as shown in Fig. 6.8, thread T_0 computes visibility between A and I_0 (e.g.,. I_0 can be the node B of Fig. 6.7), thread T_1 computes visibility between A and I_1 and so on. Once visibility between A and all nodes in its interaction list is computed, we move to another node N and repeat the same. Thus we allocate *multiple threads per node but only one per node pair.*

The number of threads running concurrently is the size of the interaction list. The degree of parallelism here is limited by the size of the interaction list of a node which might be quite small (generally in tens or

hundreds). To unleash the power of GPU we need thousands of threads running concurrently, which is not the case here. However, the threads per node strategy can be combined with other strategies (see below, for example, sec. 6.4.2).

A more serious limitation is that each thread has to perform *recursion* as well as *dynamic memory allocation* for setting up links at the descendant levels. This is *not* possible on the current GPUs. Recursion can be implemented with a user stack; but the dynamic allocation problem persists.

6.4.2 One Thread per Node Strategy

Another intuitive way to compute visibility is to let each thread compute visibility between a node and all the nodes in its interaction list, going down the octree level by level, starting form the root. For example, as shown in Fig. 6.9 thread T_0 computes visibility between a node A and all the nodes in its list, thread T_1 computes visibility between a node N and all the nodes in its list and so on. Thus we allocate *only one thread to compute visibility between a node and its entire interaction list*.

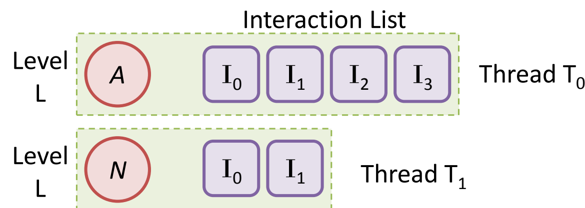


Figure 6.9: Parallelism across nodes at the same level

Note that all nodes at a particular level are considered concurrently before moving on to the next level. Thus the degree of parallelism is equal to the number of nodes at a particular level considered and changes with every level. The performance of the algorithm increases as we go down the octree, as the number of nodes per level tends to increase with greater octree depths (*root* being at depth 0).

One of the drawbacks of this parallel algorithm is the fact that it does not utilize the commutative nature of visibility. That is, if node N_1 is visible to node N_2, then N_2 is also visible to N_1. Although such cases can be detected, and threads made to stop execution, almost half of the threads would be wasted. Further, the same limitations (sec. 6.4.1) of dynamic memory allocation and recursion apply to this parallel algorithm, thereby making this case undesirable.

6.4.3 Multiple Threads per Node-Pair

Here we consider a node A and say node B belongs to its interaction list. We compute the visibility between all leaves of A with all leaves of B, *in parallel on the GPU* (and afterwords repeat the same for other node-pairs in the tree). The recursive part is computed on the CPU which uses traditional dynamic memory management.

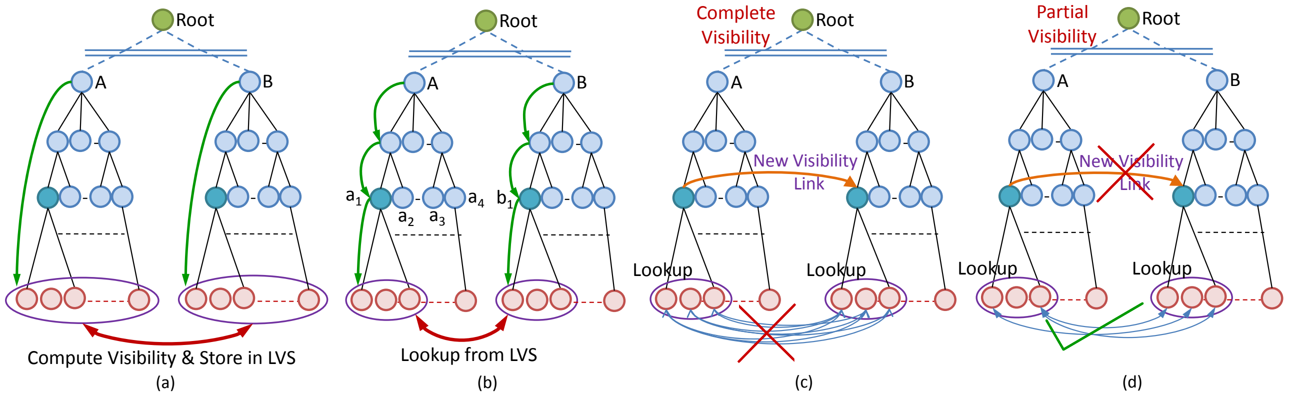


Figure 6.10: The visibility map on the GPU uses thousands of threads concurrently by working at the large number of leaves (a) and stores the result in a table. The links at other levels are set based on a lookup computation.

To achieve the same, we introduce a minor modification to the original CPU-based algorithm (compare Fig. 6.7(a) with Fig. 6.10(a)). In the CPU-based approach, we first recurse in the sub-trees of A and B and then having reached the leaf level, compute the leaf-leaf visibility. In contrast, in our GPU implementation, we first compute all leaf-pair visibility between two nodes and store it in a Boolean array *LVS* (Leaf Visibility Status). The CPU does the standard recursion, and having reached the leaf level, use a simple look-up to *LVS* to find the already computed answer.

For example, in Fig. 6.10, we first compute visibility of all leaves of A with respect to all leaves of B and store it in the *LVS* (Fig. 6.10(a)). We then recurse their sub-trees, as in the CPU implementation, and look up the visibility value from the *LVS* (Fig. 6.10(b)) to find whether the descendants (say a_1 and b_1) are completely visible (Fig. 6.10(c)) or partially visible (Fig. 6.10(d)). We repeat the same for all other descendant pairs (say (a_2, b_1)).

6.5 Leaf-Pair Visibility

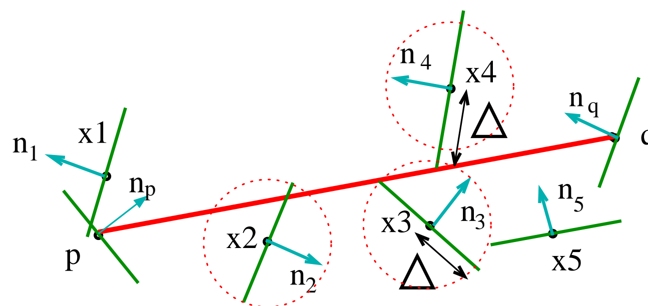


Figure 6.11: Visibility between points p and q

Having presented the strategy for constructing, *in parallel*, a global V-map for a given tree, we now discuss the leaf-leaf visibility algorithm performed by each thread. We build on the atomic point-pair visibility algorithm (Fig. 6.11) given in [GKCD07] and extend it for computing visibility between leaves.

6.5.1 Prior Algorithm

As presented in [GKCD07], to compute visibility between any two points p and q, we check if they face each other. If yes, we then determine a set of potential occluders using a 3D Bresenham's line algorithm. Bresenham's algorithm outputs a set *Y* of points which are collinear with and between p and q. Going down the octree recursively, all points from the leaves containing any point from *Y* is added to a set *X*. The Bresenham step-length is based on the sampling resolution of the original point dataset.

The potential occluders are pruned further based on the tangent plane intersection tests. In Fig. 6.11, the set *X* consists of *potential* occluders points (x_1, x_2, x_3, x_4, x_5). In fact, only points x_2 and x_3 are considered as *actual* occluders. Point x_1 is rejected as the intersection point of the tangent plane lies outside segment $\overline{pq}$, point x_4 because it is more than a distance Δ away from $\overline{pq}$, and point x_5 as its tangent plane is parallel to $\overline{pq}$. If K (a parameter) of actual occluders are found, q is considered invisible to p.

Instead of point pairs p and q, one may also use leaves, and apply the same idea. p and q (of Fig. 6.11) are now the leaf's centroids (not centers) and the potential occluders are the centroids of the leaves intersecting line segment $\overline{pq}$. Δ is the distance from the centroid of an intersecting leaf to its farthest point and is different for every leaf [GKCD07].

Applying this idea for the GPU model, we see that in this way of computing leaf-leaf visibility, we require each thread to recursively go down the octree for finding the potential occluders between the leaf pair considered. As octree height increases, the size of leaves becomes small, thereby reducing the step length of the 3D Bresenham's Line algorithm by a significant amount. This drastically increased the load and the computations as it requires more recursive traversals of the octree. In our GPU implementation, the Bresenham approach is therefore abandoned in computing potential occluders; however, the second stage of actual occluders is retained.

6.5.2 Computing Potential Occluders

At the time of construction of the octree, a sphere is initially, and implicitly constructed for every leaf node. The center of each sphere is the center of the respective leaf, and the radius is the distance from the center to the farthest point in the leaf. The spheres for internal nodes are constructed recursively using the maximum radius of their children (Fig. 6.12(a)). Spheres of siblings therefore might overlap, but this makes our visibility test a bit conservative without hampering the correctness of the results. Note that, only the radius of the sphere need to be stored, since the center is already present in the octree.

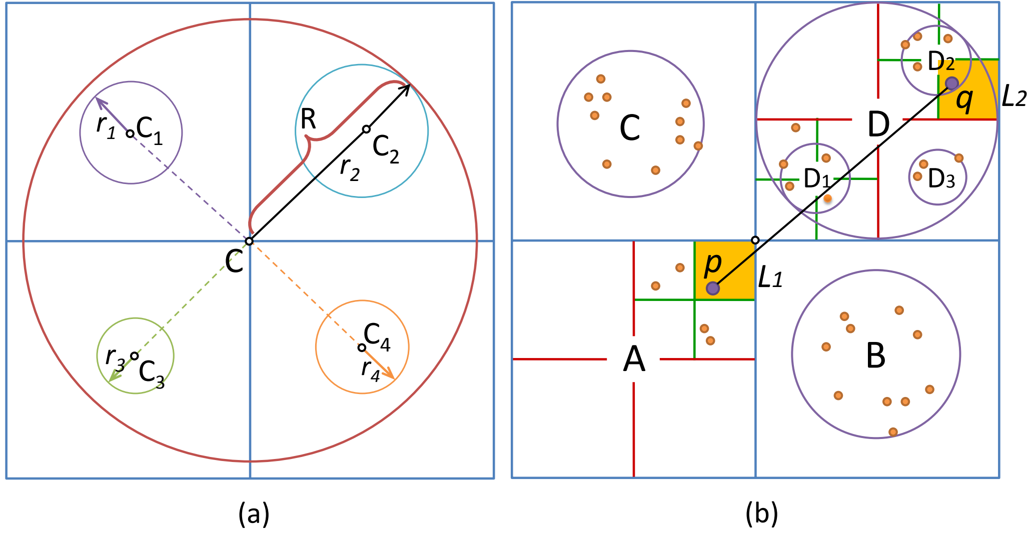


Figure 6.12: (a) Constructing parent sphere from child, (b) Line Segment-Sphere Intersection test

Consider the leaf pair L_1 and L_2. We now recursively compute the intersection status of the line segment $\overline{pq}$ (p being centroid of L_1 and q of L_2) and the spheres of nodes of the octree, starting from the root. It is significant to note that we are *not* interested in computing the actual point of intersection, only the Boolean decision of intersection between $\overline{pq}$ and the sphere under consideration. This is achieved by performing a very simple dot-product computation. If a node contains L_1 or L_2 or intersects line segment $\overline{pq}$, we recursively perform a similar test in the sub-tree of the intersecting node. Otherwise we discard the node.

For example, we discard the sub-tree of node C (Fig. 6.12(b)) as it does not intersect $\overline{pq}$ nor does it contain L_1 or L_2. On the other hand, we traverse the children of node D and recursively repeat the same for sub-trees of children D_1 and D_2. In this we reaching the (potential occluder) intersecting leaf nodes.

Each thread then performs the same tangent-surface intersection tests (as detailed in sec. 6.5.1) for their respective leaf-pairs. If mutually visible, each thread adds 1 to the corresponding location in the Boolean array *LVS* which will be eventually looked up.

6.6 GPU Optimizations

To improve the GPU kernel's performance, we utilize several optimization techniques enlisted below.

1. Asynchronous Computations: Asynchronous kernel launches were made by overlapping CPU computations with kernel execution. Thus, while the CPU is busy recursing the sub-trees of nodes to set visibility links at different levels of octree, our GPU kernel is busy performing leaf-pair visibility computations for the next node pair whose sub-trees will eventually be visited by CPU when the kernel finishes its part.

2. LOOP UNROLLING: Any flow control instruction (if, switch, do, for, while) can significantly impact the effective throughput by causing threads to diverge. Thus, major performance improvements can be achieved by unrolling the control flow loop. We found that especially the loops with global memory accesses (as is the case in our algorithm) can benefit from unrolling.

3. OPTIMAL THREAD AND BLOCK SIZE: Obtained via an empirical study, each thread block must contains $128 - 256$ threads and every thread block grid no less than 64 blocks for optimal performance on G80 GPU.

4. OPTIMAL OCTREE HEIGHTS: As every thread works on single leaf-pair, multiple threads are independent. Each leaf pair may have a different number of *potential occluders* to be considered. The thread that finishes work for a given leaf-pair simply takes care of another leaf-pair, without the need for any shared memory or synchronization with other threads. To effectively use the GPU, the number of leaf-pair should be sufficiently large. With 16 multi-processors, we need at least 64 thread-blocks, each having 256 threads to utilize the GPU. Thus the number of leaf pairs considered concurrently should be at least 16384 for good performance.

6.7 Results

The CUDA based parallel V-map construction algorithm, implemented on G80 NVIDIA GPU, was tested on several point models. In this section we provide qualitative visibility validation and quantitative results. Note that all input such as the models in the room, the light source, and the walls of the Cornell room are given as points.

6.7.1 Visibility Validation

We validate our proposed method here using an *adaptive octree structure*. We remark that the user divides the octree adaptively depending on the input scene density. Increasing or decreasing the levels of subdivision for a given scene is essentially a trade-off between quality of the visibility (user driven), and the computational time.

Fig. 6.13(a) shows a point model of an empty Cornell room with some artificial lighting to make the model visible. Note the default colors of the walls. We now introduce a bluish white Stanford bunny. In Fig. 6.13(b), the eye (w.r.t. which visibility is being computed) is on the floor, marked with a cyan colored dot. The violet (purple) color indicates those portions of the room that are visible to this eye. Notice the "shadow" of the bunny on the back wall. The same idea is repeated with the eye (marked in cyan) placed at different locations for various different point models (all bluish white in color) of the Buddha (Fig. 6.13(c)), and an Indian Goddess Satyavathi (Fig. 6.13(d)). We found that an octree of height 8 gave us accurate visibility results, however, it is

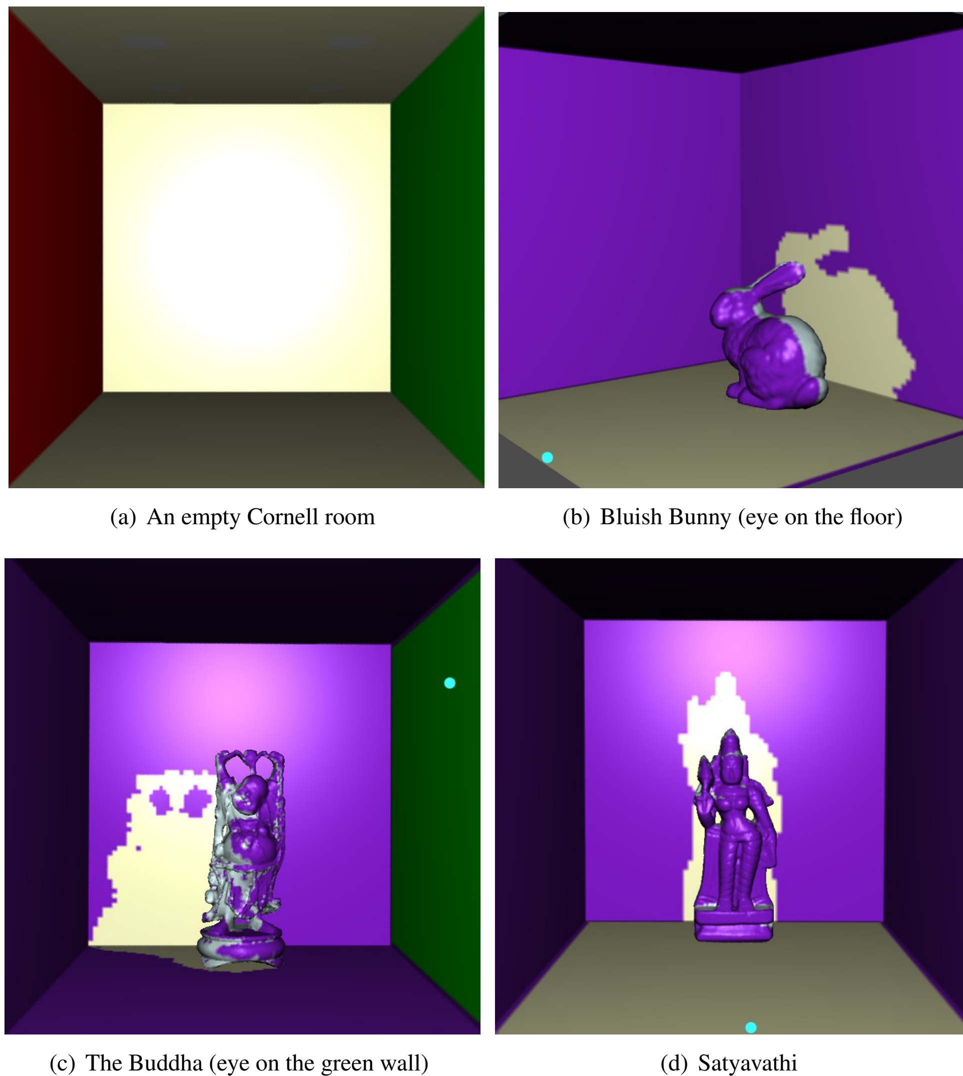


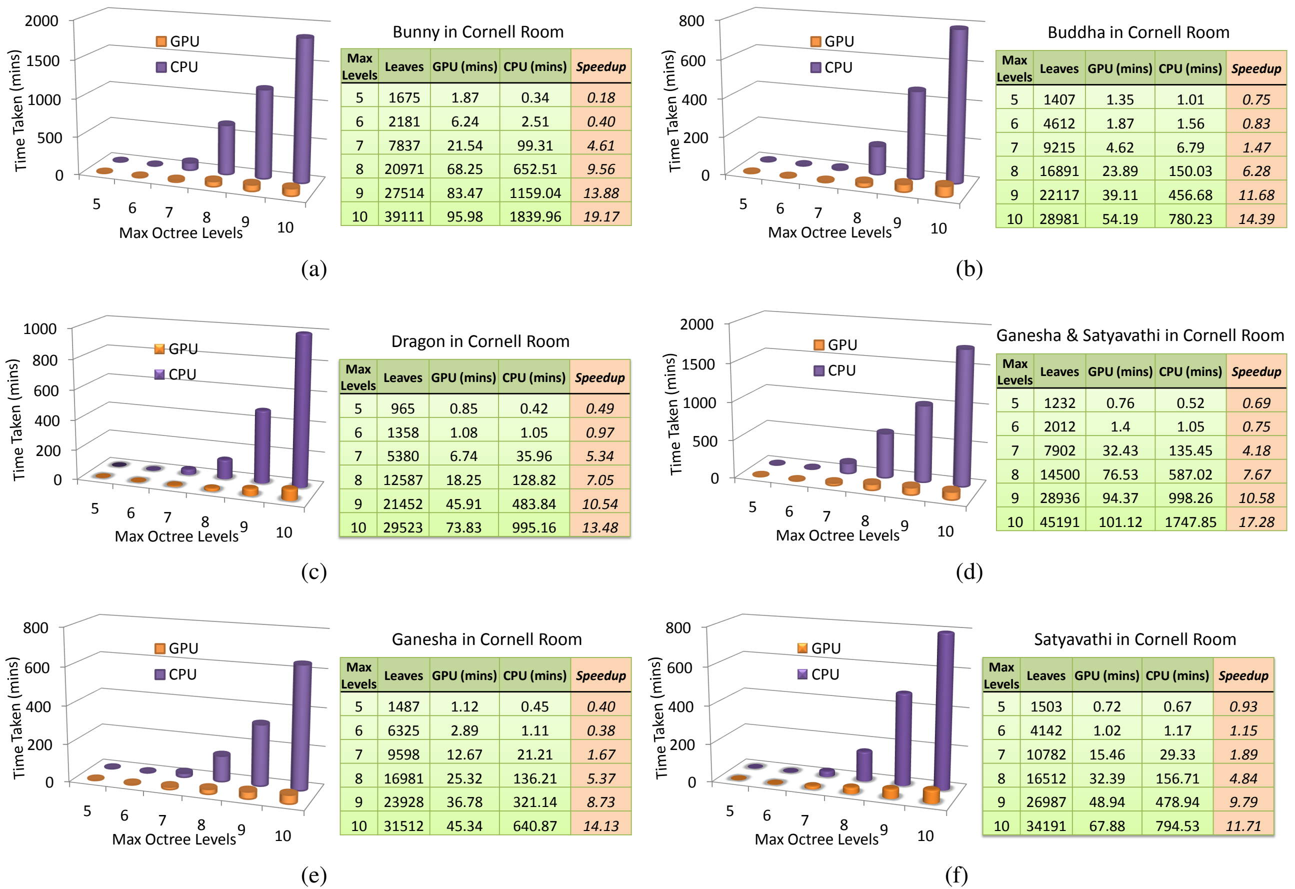
Figure 6.13: Visibility tests where purple color indicates portions visible to the candidate eye (in cyan)

more prudent to go to higher depths such as 9 or 10.

6.7.2 Quantitative Results

Fig. 6.14 shows the running time of our implementation. Each graph in Fig. 6.14 refers to a particular model, and shows the only-CPU, and CPU-GPU combo running time for various octree levels (5-10). For example., Fig. 6.14(a) shows results for the Stanford bunny in the Cornell room while Fig. 6.14(b) shows the same for Buddha. The table also shows the number of leaves (at various depths) in the various adaptive octrees. This is an important parameter on which the degree of parallelism indirectly depends. The running times tabulated, also depends on the number of threads per block. A block size of (16×16) gives the best results with 256 threads per block.

The CPU and GPU have almost identical run times if the model has octree height of 6 or below. For best



Bunny in Cornell Room

Max Levels	Leaves	GPU (mins)	CPU (mins)	*Speedup*
5	1675	1.87	0.34	*0.18*
6	2181	6.24	2.51	*0.40*
7	7837	21.54	99.31	*4.61*
8	20971	68.25	652.51	*9.56*
9	27514	83.47	1159.04	*13.88*
10	39111	95.98	1839.96	*19.17*

(a)

Buddha in Cornell Room

Max Levels	Leaves	GPU (mins)	CPU (mins)	*Speedup*
5	1407	1.35	1.01	*0.75*
6	4612	1.87	1.56	*0.83*
7	9215	4.62	6.79	*1.47*
8	16891	23.89	150.03	*6.28*
9	22117	39.11	456.68	*11.68*
10	28981	54.19	780.23	*14.39*

(b)

Dragon in Cornell Room

Max Levels	Leaves	GPU (mins)	CPU (mins)	*Speedup*
5	965	0.85	0.42	*0.49*
6	1358	1.08	1.05	*0.97*
7	5380	6.74	35.96	*5.34*
8	12587	18.25	128.82	*7.05*
9	21452	45.91	483.84	*10.54*
10	29523	73.83	995.16	*13.48*

(c)

Ganesha & Satyavathi in Cornell Room

Max Levels	Leaves	GPU (mins)	CPU (mins)	*Speedup*
5	1232	0.76	0.52	*0.69*
6	2012	1.4	1.05	*0.75*
7	7902	32.43	135.45	*4.18*
8	14500	76.53	587.02	*7.67*
9	28936	94.37	998.26	*10.58*
10	45191	101.12	1747.85	*17.28*

(d)

Ganesha in Cornell Room

Max Levels	Leaves	GPU (mins)	CPU (mins)	*Speedup*
5	1487	1.12	0.45	*0.40*
6	6325	2.89	1.11	*0.38*
7	9598	12.67	21.21	*1.67*
8	16981	25.32	136.21	*5.37*
9	23928	36.78	321.14	*8.73*
10	31512	45.34	640.87	*14.13*

(e)

Satyavathi in Cornell Room

Max Levels	Leaves	GPU (mins)	CPU (mins)	*Speedup*
5	1503	0.72	0.67	*0.93*
6	4142	1.02	1.17	*1.15*
7	10782	15.46	29.33	*1.89*
8	16512	32.39	156.71	*4.84*
9	26987	48.94	478.94	*9.79*
10	34191	67.88	794.53	*11.71*

(f)

Figure 6.14: V-map construction times (CPU & GPU) for models with differing octree heights

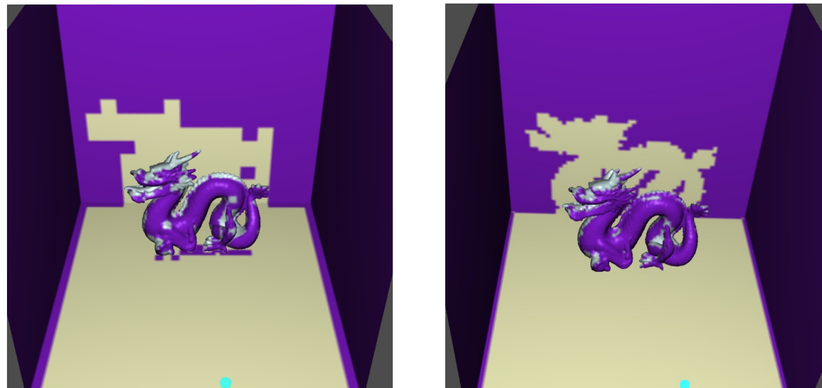


Figure 6.15: Dragon viewed from the floor (cyan dot). The quality is unacceptable for octrees of heights of 7 (left) or less. The figure on the right is for an octree of height 9.

throughput from the GPU we need at least 16384 leaf-pairs to be considered concurrently which is generally not the case for octrees built till level 6. The GPU starts out performing the CPU for octrees with greater heights (Fig. 6.14). However, the quality of the visibility solution is below par for octrees with heights 7 or below (Fig. 6.15). Thus, to get a good acceptable accuracy in results (Fig. 6.13) and throughput from GPU, we use octree

heights of at least 8. Speedup is also given in the tables. *We achieve an average speed-up (across all models) of* 15 *when the input models are divided with octree of height* 10. Thus, we see that the CUDA implementation of V-map construction algorithm is efficient and fast. Once constructed, they allow for an *interactive walkthrough* of the point model scene.

Figure 6.16: Point models rendered with diffuse global illumination effects of color bleeding and soft shadows. Pair-wise visibility information is essential in such cases. Note that the Cornell room as well as the models in it are input as point models.

As a proof of applicability, we now use the parallel constructed V-map in a global illumination algorithm, where the Fast Multipole Method is used to solve the radiosity kernel. Fig. 6.16 shows results with the color bleeding effects and the soft shadows clearly visible. The V-map also works well even in the case of aggregated input models (e.g., point models of both Ganesha and Satyavathi placed in a point model of a Cornell room). Note that the input is a single, large, *mixed* point data set consisting of Ganesha, Satyavathi, and the Cornell room. These models were not taken as separate entities nor were they segmented into different objects during the whole process.

Chapter 7

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

7.1 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 7.2 and conclude with some limitations of this technique. We then follow it up with our proposed method in Section 7.3 to get all the desired specular effects in a point model scene.

7.2 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.

7.2.1 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 7 Photon emission from a diffuse point light

---

procedure ***emitPhotons***

1: $n = 0$ // number of emitted photons
2: **while** not enough photons **do**
3: DO
4: // use simple rejection sampling
5: // to find diffuse photon direction
6: x = random number between -1 and 1
7: y = random number between -1 and 1
8: z = random number between -1 and 1
while ($x * x + y * y + z * z > 1$)
10: $d = < x, y, z >$
11: p = light source position
12: trace photon from p in direction d
13: $n = n + 1$
14: **end while**
15: 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 7.1.

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:

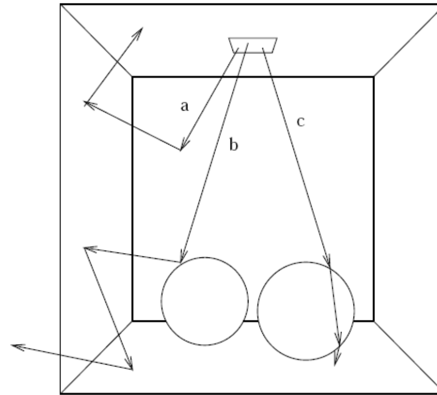


Figure 7.1: Photon paths in a scene (a Cornell box with a chrome sphere on left and a glass sphere on right): (a) two diffuse reflections followed by absorption, (b) a specular reflection followed by two diffuse reflections, (c) two specular transmissions followed by absorption.

- *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 7.2 for a simple test scene with a glass sphere and 2 diffuse walls. Figure 7.2(a) shows the construction of the caustics photon map with a dense distribution of photons,and Figure 7.2(b) shows the construction of the global photon map with a more coarse distribution of photons.

7.2.2 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*.

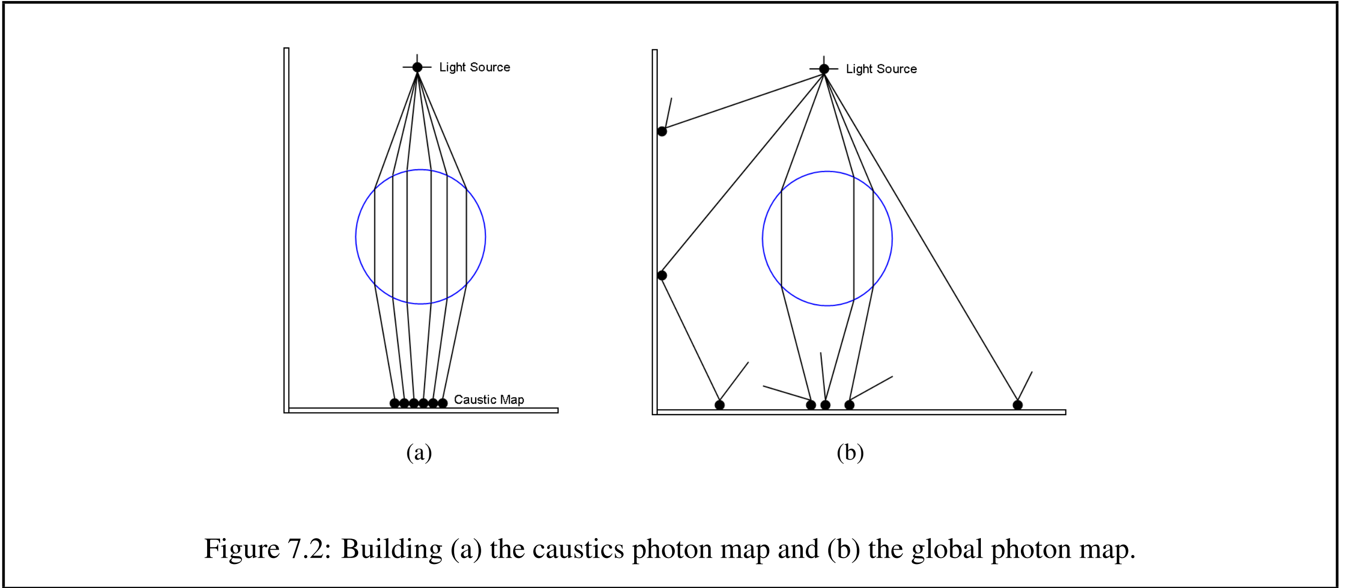


Figure 7.2: Building (a) the caustics photon map and (b) the global photon map.

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.

7.2.3 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:

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

There are 4 integrals in the above equation. 1st term computes *Direct Illumination*. 2nd terms computes *Specular and Glossy Reflection*. 3rd term computes *Caustics*. 4th 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.

7.2.4 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 7.3 shows an example output of Photon Mapping algorithm.

7.2.5 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 7.3: Example output of Photon Mapping Algorithm [Jen96] showing reflection, refractions and caustics

7.3 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 7.1). We will discuss here one of the very efficient techniques for doing the same, *Splat-based Ray Tracing* [LMR07], in the next section.

7.3.1 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 7.3.1.1.

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 7.3.2.

7.3.1.1 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 7.4(a) illustrates the optimal choice of $\mathbf{c}_j$.

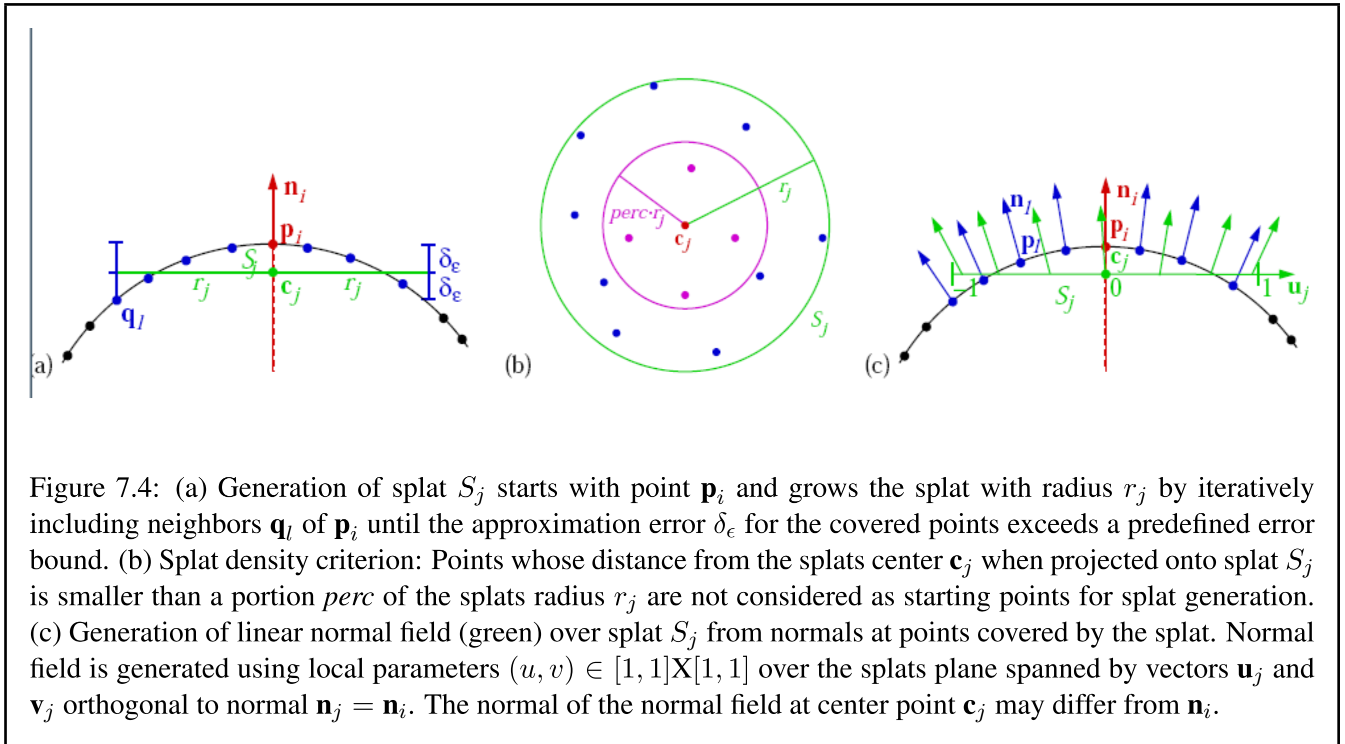


Figure 7.4: (a) Generation of splat S_j starts with point $\mathbf{p}_i$ and grows the splat with radius r_j by iteratively including neighbors $\mathbf{q}_l$ of $\mathbf{p}_i$ until the approximation error δ_ϵ for the covered points exceeds a predefined error bound. (b) Splat density criterion: Points 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 $(u, v) \in [1, 1]\text{X}[1, 1]$ 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 7.4(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 7.4(c) illustrates the idea.
To determine the tilting factors ν_j and ω_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 ν_j and ωj. Since the system is overdetermined, it can only be solved approximately.

7.3.2 Ray Tracing

7.3.2.1 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.

7.3.2.2 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 7.5(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 7.5(c). Otherwise, the splat is inserted into the leaf cell, see Figure 7.5(b).

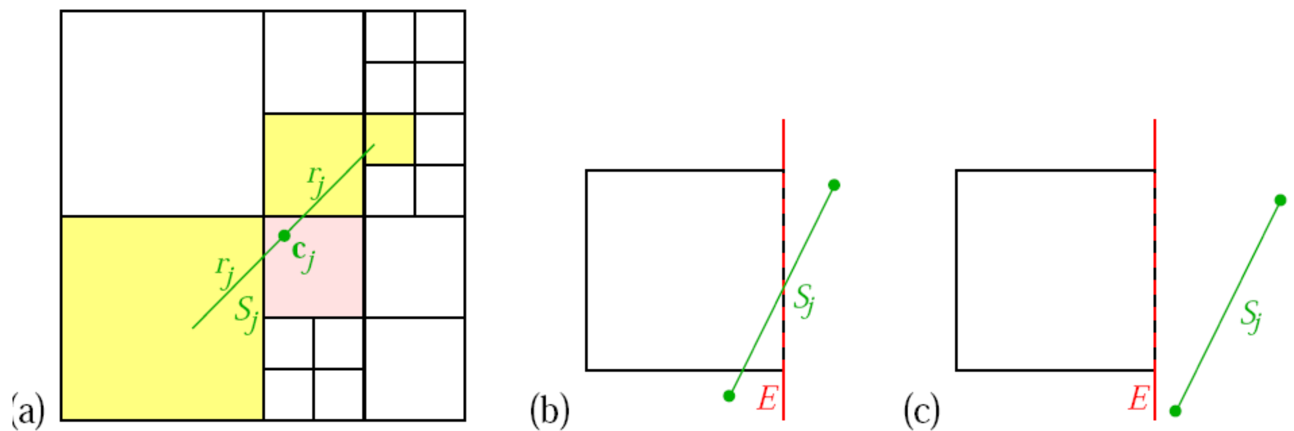


Figure 7.5: (a) Octree generation: In the first phase, the octree is generated while inserting splats S_j into the cells containing their centers c_j (red cell). In the second phase, splat S_j is inserted into all additional cells it intersects (yellow cells). (b)(c) The second test checks whether the 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.

7.3.2.3 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 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.

7.3.3 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.

7.3.3.1 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.

7.3.4 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.

7.3.5 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].

7.3.5.1 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:

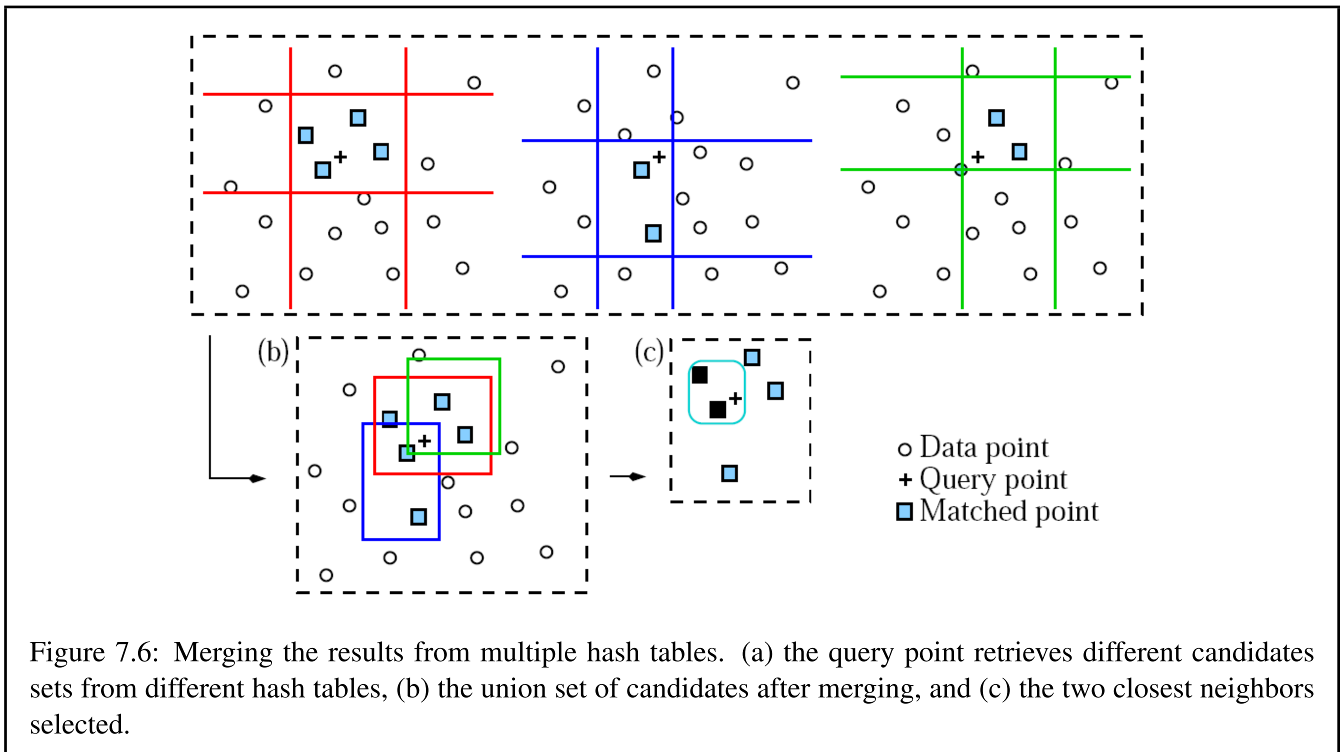


Figure 7.6: Merging the results from multiple hash tables. (a) the query point retrieves different candidates sets from different hash tables, (b) the union set of candidates after merging, and (c) the two closest neighbors selected.

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 7.6.) 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.

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 8

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. We extended, in this report, the novel, provably efficient, hierarchical, visibility determination scheme for point based models to the highly parallel structures of modern day GPUs. By viewing this visibility map as a 'preprocessing' step, photo-realistic global illumination rendering of complex point-based models have been shown. By extending the V-map construction algorithm on the GPU, efficient speed-ups have also been reported.

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 data decomposition and communication efficiency being the major challenges. In §3, we discussed one such algorithm which uses only a static data decomposition on octrees and offers communication efficiency. We exploited 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. Different implementations of parallel octree construction on GPU were also presented which are to be eventually merged with the GPU-based parallel FMM algorithm.

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. 7, 85

[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.

[AGCA08] Prekshu Ajmera, Rhushabh Goradia, Sharat Chandran, and Srinivas Aluru. Fast, parallel, gpu-based construction of space filling curves and octrees. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 1–1, New York, NY, USA, 2008. ACM. ii, 43, 56, 60, 63, 64

[Ama84] John Amanatides. Ray tracing with cones. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 129–135, 1984. 7

[AS99] S. Aluru and F. Sevilgen. Dynamic compressed hyper-octrees with applications to n-body problems. *Proceedings of Foundations of Software Technology and Theoritical Computer Science*, pages 21–33, 1999.

[AS07] S. Aluru and S. Seal. *Handbook of Parallel Computing: Models, Algorithms and Applications*, chapter Spatial Domain Decomposition Methods in Parallel Scientific Computing. Chapman and Hall, CRC Computer and Information Science Series, 2007.

[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. 8

[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. 8

[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`. 36

[CUDa] CUDA. NVIDIA CUDA Programming Guide. `http://developer.nvidia.com/cuda`. 8, 15, 16, 18, 61

[CUDb] CUDPP. CUDA Data Parallel Primitives Library. `http://www.gpgpu.org/developer/cudpp`. 56, 59, 64

[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. 98

[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. 8, 21

[DTG00] Philip Dutre, Parag Tole, and Donald P. Greenberg. Approximate visibility for illumination computation using point clouds. Technical report, Cornell University, 2000. 67, 69

[DYN04] Yoshinori Dobashi, Tsuyoshi Yamamoto, and Tomoyuki Nishita. Radiosity for point sampled geometry. In *Pacific Graphics*, 2004. iv, 5, 67

[E.62] Bresenham J. E. Bresenham's line drawing algorithm, 1962. 70

[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. 8

[GAC08] R. Goradia, P. Ajmera, and S. Chandran. Gpu-based hierarchical computation for view independent visibility. *Accepted at ICVGIP, Indian Conference on Vision, Graphics and Image Processing*, 2008. 13

[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. 67

[GD07] Nail A. Gumerov and Ramani Duraiswami. Fast multipole methods on graphics processors. *Astro GPU*, 2007. 24, 25

[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. 100

[GKCD07] R. Goradia, A. Kanakanti, S. Chandran, and A. Datta. Visibility map for global illumination in point clouds. *Proceedings of ACM SIGGRAPH GRAPHITE, 5th International Conference on Computer Graphics and Interactive Techniques*, 2007. v, 7, 12, 13, 74, 76, 79

[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. 8

[Gor06] Rhushabh Goradia. Fmm-based illumination maps for point models. *Second Progress Report, Ph.D.*, 2006. 22, 70, 71

[Gor07] Rhushabh Goradia. Global illumination for point models. *Third Progress Report, Ph.D.*, 2007. ii, 13, 67, 71

[GR87] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–348, 1987. 8, 21

[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. 7

[GST05] Dominik Göddeke, Robert Strzodka, and Stefan Turek. Accelerating double precision FEM simulations with GPUs. In *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*, Sep 2005. 25

[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). 98

[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. 86

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

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with cuda. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007.

[Jen96] H. W. Jensen. Global illumination using photon maps. *Eurographics Rendering Workshop 1996*, pages 21–30, June 1996. vii, 7, 86, 87, 92, 99, 104

[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. iv, 4, 22, 25, 26

[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.

[KTB07] Sagi Katz, Ayellet Tal, and Ronen Basri. Direct visibility of point sets. In *SIGGRAPH '07*, page 24. ACM, 2007. 67

[LHN05] S. Lefebvre, S. Hornus, and F. Neyret. *GPU Gems 2*, chapter Octree Textures on the GPU, pages 595–614. Addison Wesley, 2005. 40

[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. 85, 93

[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. 3

[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.

[MGPG04] N. J. Mitra, N. Gelfand, H. Pottmann, and L. Guibas. Registration of point cloud data from a geometric optimization perspective. In *Symposium on Geometry Processing*, pages 23–31, 2004. 67

[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. 99, 101

[Moo93] Andrew W. Moore. An introductory tutorial on kd-trees. *Carnegie Mellon University*, 1993. 89

[NHP07] L. Nyland, M. Harris, and J. Prins. *GPU Gems 3*, chapter Fast N-Body Simulation with CUDA, pages 677–696. Addison Wesley, 2007. v, 23, 24

[Nie92] H. Niederreiter. Random number generation and quasi-monte carlo methods. *Society for Industrial and Applied Mathematics*, 1992. 98

[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. 3, 7

[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.

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007. iv, 9

[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. 3

[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. 3

[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. 3, 7

[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. 99

[Sch06] Lars Schjoth. Diffusion based photon mapping. *International Conference on Computer Graphics Theory and Applications GRAPP*, 2006. 91

[SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Graphics Hardware 2007*, pages 97–106. ACM, August 2007.

[SJ00] G. Schaufler and H. Jensen. Ray tracing point sampled geometry. In *Eurographics Rendering Workshop Proceedings*, pages 319–328, 2000. 7, 85

[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. 7

[Wik] Wikipedia. The N-Body Problem. `http://en.wikipedia.org/wiki/N-body_problem`. (Last seen on 30th June, 2008).

[WS93] M. S. Warren and J. K. Salmon. A parallel hashed octree n-body algorithm. *Proceedings of Supercomputing*, pages 12–21, 1993.

[WS03] Michael Wand and Wolfgang Straer. Multi-resolution point-sample raytracing. *Graphics Interface*, pages 139–148, 2003. 3, 85

[WS05] Ingo Wald and Hans-Peter Seidel. Interactive Ray Tracing of Point Based Models. In *Proceedings of 2005 Symposium on Point Based Graphics*, 2005. iv, 5, 7, 85

[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.