The Use of Geometric Structures in Graphics and Optimization

Norbert BUS

Directeur de Thése:	**Nabil H. MUSTAFA** et
	Venceslas BIRI
Co-Encadrant:	**Lilian BUZER**

École Doctorale Paris-Est
Mathématiques et Sciences et Technologies
de l'Information et de la Communication

Pour obtenir le grade de
DOCTEUR de l'UNIVERITÉ PARIS-EST

Soutenance prévue le 7 octobre 2015 devant le jury composé de:

Président	Frédéric	MAGNIEZ	CNRS Université Paris Diderot
Rapporteur	Tamy	BOUBEKEUR	Telecom ParisTech
Rapporteur	Bruno	LÉVY	INRIA Nancy
Examinateur	Frank	NIELSEN	École Polytechnique
Examinateur	Lilian	BUZER	Université Paris-Est
Directeur	Nabil H.	MUSTAFA	Université Paris-Est
Directeur	Venceslas	BIRI	Université Marne la Vallée

Abstract

Real-world data has a large geometric component, exhibiting significant geometric patterns. Therefore exploiting the geometric nature of data to design efficient methods has became a very important topic in several scientific fields, e.g., computational geometry, discrete geometry, computer graphics, computer vision. In this thesis we use geometric structures to design efficient algorithms for problems in two domains, computer graphics and combinatorial optimization.

Part I focuses on a geometric data structure called well-separated pair decomposition and its usage for one of the most challenging problems in computer graphics, namely efficient photo-realistic rendering. One solution is the family of many-lights methods that approximate global illumination by individually computing illumination from a large number of virtual point lights (VPLs) placed on surfaces. Considering each VPL individually results in a vast number of calculations. One successful strategy to reduce computations is to group the VPLs into a small number of clusters that are treated as individual lights with respect to each point to be shaded. We use the well-separated pair decomposition of points as a basis for a data structure for pre-computing and compactly storing a set of view independent candidate VPL clusterings showing that a suitable clustering of the VPLs can be efficiently extracted from this data structure. We show that instead of clustering points and/or VPLs independently, what is required is to cluster the product-space of the set of points to be shaded and the set of VPLs based on the induced pairwise illumination. Additionally we propose an adaptive sampling technique to reduce the number of visibility queries for each product-space cluster. Our method handles any light source that can be approximated with virtual point lights (VPLs), highly glossy materials and outperforms previous state-of-the-art methods.

Part II focuses on developing new approximation algorithms for a fundamental NP-complete problem in computational geometry. It focuses on the minimum hitting set problem, particularly on the case where given a set of points and a set of disks, we wish to compute the minimum-sized subset of the points that hits all disks. It turns out that efficient algorithms for geometric hitting set rely on a key geometric structure, called ϵ-net. We give an algorithm that uses only Delaunay triangulations to construct ϵ-nets of size $13.4/\epsilon$ and we provide a practical implementation of a technique to calculate hitting sets in near-linear time using small sized ϵ-nets. Our results yield a 13.4 approximation for the hitting set problem with an algorithm that runs efficiently even on large data sets. For smaller datasets, we present an implementation of the local search technique along with tight approximation bounds for its approximation factor, yielding an $(8+\epsilon)$-approximation algorithm with running time $\tilde{O}(n^{2.34})$. Our work related to fundamental computational geometry problems also includes a novel dynamic convex hull algorithm for simple polygonal chains handling insertion or deletion of a point in amortized constant time.

Resumé

Les données du monde réel ont une composante géométrique importante et représentent souvent des motifs géométriques. Les méthodes qui utilisent la nature géométrique des données sont activement développées dans plusieurs domaines scientifiques, comme, par exemple, la géométrie algorithmique, la géométrie discrète, la synthèse d'images, la vision par ordinateur. Dans le travail présent, nous utilisons les structures géométriques afin de modéliser des algorithmes efficaces pour deux domaines, celui de synthèse d'images et de l'optimisation combinatoire.

La première partie porter sur une structure de données géométriques, appelée *well-separated pair decomposition*, et son application pour un des problèmes les plus difficiles dans la synthèse d'images, le rendu photo réaliste efficace. Une solution consiste à appliquer toute une famille de méthodes de *many-lights* qui fait une approximation d'illumination globale par calcule individuelle d'illumination avec un grand nombre de VPLs (virtual point light) répartis sur les surfaces. L'application individuelle de chacun VPL résulte dans un grand nombre de calculs. Une des stratégies pour réduire les calculs est de faire les clusters qui sont considères comme un seul émetteur. Nous utilisons la well-separated pair decomposition de points comme le fondement de la structure des données susceptible de procéder à un calcul préliminaire et de conserver d'une façon compacte un grand nombre des clusterisations individuels potentiels ce qui montre que la clusterisation des VPL plus correspondante peut être extraite de cette structure de données d'une manière efficace. Nous montrons qu'au lieu de regrouper les points et/ou VPL indépendamment il vaut mieux produire les clusters sur l'espace de produit du nombre des points à nuancer et un groupe de VPL à la base de l'illumination des paires induite. En plus, nous proposons une technique adaptative afin d'échantillonner pour réduire le nombre des demandes de vérifications de visibilité pour chaque cluster de l'espace de produit. Notre méthode consiste à détenir chaque émetteur qui peut être rapproché par VPL, matériaux spéculaire et à effectuer les méthodes précédentes reconnus les meilleurs jusqu'au présent.

La deuxième partie est consacrée au développement de nouveaux algorithmes d'approximation pour un problème fondamental de NP complet dans la géométrie algorithmique, précisément le problème du *hitting set*, en s'intéressant au cas d'un groupe de points et d'un groupe de disques, nous souhaitons calculer le plus petit nombre de points qui touche tous les disques. Il arrive que les algorithmes efficaces à détecter le hitting set repose sur une structure géométrique clé, appelée ϵ-net. Nous donnons un algorithme utilisant uniquement la triangulation de Delaunay pour construire les ϵ-nets de taille $13.4/\epsilon$. Nous donnons une implémentation pratique de la technique à calculer les hitting sets dans le temps quasi-linéaire en utilisant des ϵ-nets de petites tailles. Nos résultats aboutissent à une approximation de 13.4 pour le problème de hitting set par un algorithme qui fonctionne même pour les grands ensembles de données. Pour les ensembles de taille plus petite, nous proposons une implémentation de la technique de recherche locale avec une approximation bornes supérieures, avec le résultat obtenu d'approximation de $(8 +$

ϵ) dans le temps $\tilde{O}(n^{2.34})$. Notre travail lié à des problèmes fondamentaux de la géométrie algorithmique comprend également un nouvel algorithme déterminant l'enveloppe convexe de manière dynamique pour les chaînes polygonales simples traitant l'insertion ou la suppression d'un point en temps amorti constant.

Acknowledgements

First, I would like to acknowledge my advisers and mentors from ESIEE and Université Paris-Est. I am very grateful to my supervisor Nabil Mustafa for his warm and open attitude, for believing in me and for his continuous support. He never let me feel intimidated by my limited expertise. I thank Venceslas Biri and Lilian Buzer for their support during these three years. I also thank Saurabh Ray for his valuable collaboration, Hugues Talbot for his help with the blade server and finally Gilles Bertrand and Michel Couprie for their support.

It is an honor to present this work to the members of my examination committee: Tamy Boubekeur, Bruno Lévy, Frank Nielsen and Frédéric Magniez. I am grateful for their precious time devoted to reviewing this thesis and participating in the defense.

My work would have been much more difficult, and certainly way less entertaining, if not for my friends who have stood next to me when I have tried to pursue my goals in France, far from my home country. I am very grateful for the support I received from my girlfriend: Maria Vakalopoulou and from my best friends: Mateusz Koziński and Ravi Kiran. Meeting you alone already made the endeavor of taking up the doctoral studies worth it. I also thank my parents, and my sister, for their unconditional support.

Finally, I would like to acknowledge the funding we got from LIGM and CNRS(UMR 8049). The work presented in this thesis was also partly funded by the grant ANR SAGA (JCJC-14-CE25-0016-01).

Contents

Publications of the Author xiii

A Detailed Overview: Problems, Techniques and Results 1

Part I Computer Graphics: Rendering Photo-Realistic Images 19

1 Introduction to Many-Lights Methods 21

2 Global Illumination Using Well-Separated Pair Decomposition 25

2.1 General Idea 25

2.2 Algorithm 27

2.3 Additional Structures for Illumination Computation 38

2.4 Results and Discussion 40

2.5 Limitations 49

3 IlluminationCut 55

3.1 General Idea 55

3.2 Algorithm 57

3.3 Results and Discussion 62

3.4 Limitations 72

4 Software 77

Part II Combinatorial Optimization 83

5 Overview of the Hitting Set Problem 85

6 Tighter Estimates for ϵ-nets for Disks 87

6.1 A Near Linear Time Algorithm for Computing ϵ-nets for Disks in the Plane . . 87

6.2 Implementation and Experimental Evaluation 94

7 Engineering the Agarwal-Pan Algorithm 97

7.1 General Idea 97

7.2 Algorithm 98
7.3 Implementation and Experimental Evaluation 102

8 Local Search 107
8.1 General Idea 107
8.2 Analysis of Quality for Local Search 108
8.3 An $\tilde{O}(n^{7/3})$-time Algorithm for Local Search 110

9 Dynamic Convex Hull for Simple Polygonal Chains 121
9.1 General Idea 121
9.2 Algorithm 122

10 Software 133

Conclusion 137

Bibliography 139

List of Figures

1 Clustering based on the light transport matrix 4
2 Iterative refinement of the clustering in tree based methods. 5
3 Well-separated pair decomposition 6
4 Product-space clustering 8
5 Comparison of the efficiency of clustering 9
6 Tight ϵ-nets 13
7 A subset of the hitting set for cities on Earth. 15
8 Dynamic convex hull for polygonal chains 16

2.1 The extended Embree framework 27
2.2 Well-separated clusters in the Museum scene 28
2.3 Well-separated cluster 30
2.4 Well-separated cluster error 30
2.5 Well-separated pair decomposition 33
2.6 A geometric condition. 33
2.7 WSPD size ratios for some graphical scenes. 34
2.8 The compressed octree and the WSPD for a set of points. 35
2.9 A close and a far *ws*-cluster 37
2.10 Museum scene rendered with the three different methods 42
2.11 Sibenik scene rendered with the three different methods 43
2.12 Conference scene rendered with the three different methods 44
2.13 San Miguel scene rendered with the three different methods 45
2.14 Varying number of VPLs for the *Museum* scene. 48
2.15 Relative error and render time for the Museum scene 49
2.16 Part of the *Museum* scene rendered with different values of samples per pixel. . 50
2.17 Evaluation of the refinement technique 50
2.18 Museum scene rendered with 4 samples per pixel 52
2.19 Conference scene rendered with 4 samples per pixel 53
2.20 Part of the Conference scene with highly glossy floor 54

3.1 Product-space clustering 56
3.2 Approximate images for various scenes 59

3.3 Banquet scene rendered with the 4 methods . . . 64
3.4 San Miguel scene rendered with the 4 methods . . . 65
3.5 Sponza scene rendered with the 4 methods . . . 66
3.6 Kitchen scene rendered with the 4 methods . . . 67
3.7 Typical errors of the rendering methods . . . 68
3.8 Decrease of clustering calculations . . . 69
3.9 Render times with varying number of VPLs . . . 71
3.10 Banquet scene rendered with the 9 samples per pixel . . . 73
3.11 San Miguel scene rendered with the 9 samples per pixel . . . 74
3.12 Sponza scene rendered with the 9 samples per pixel . . . 75
3.13 Kitchen scene rendered with the 9 samples per pixel . . . 76

4.1 Images rendered with the Embree framework [WWB+14]. . . . 77

6.1 Small ϵ-nets . . . 89
6.2 ϵ-net size for various datasets . . . 95

7.1 Computing D_1 and D_2. . . . 101
7.2 Different point set sizes for the *ForestFire* (left) and *China* (right) datasets. . . . 104
7.3 Different radii settings for the *KDDCU* (left) and *China* (right) datasets. . . . 105

8.1 Local search lower bound construction . . . 111
8.2 Case 2 in the proof of Lemma 8.8 . . . 113

9.1 Phases in the convex hull algorithm . . . 122
9.2 Convex hull data structures . . . 124
9.3 Regions. . . . 125
9.4 Deleting a point from the convex hull . . . 128
9.5 Deleting a point from the mixed case . . . 129

List of Tables

1 Summary of previous work on hitting sets 12

2.1 Octree depth 34
2.2 Maximum number of added clusters 38
2.3 Average number of subgroups per cluster. 39
2.4 Rendering statistics of the rendering methods 41
2.5 Memory requirements for the *Museum* scene (GB). 48
2.6 Rendering statistics and images for 4 samples per pixel. 51

3.1 Rendering statistics for 1600×1200 resolution images 63
3.2 Lightcuts with maximum cut size set to 2000. 69
3.3 Results for Banquet with indistinguishable quality 70
3.4 Peak memory requirements for Banquet (in GB). 71
3.5 Statistics for images of 1600×1200 resolution with 9 samples per pixel. . . . 72

6.1 The size of the ϵ-nets obtained by our method 96

7.1 Hitting set statistics 103
7.2 Memory usage in MB and range reporting statistics 104

Publications of the Author

Norbert Bus, Nabil H. Mustafa, and Venceslas Biri. “IlluminationCut.” In: *Computer Graphics Forum (Proceedings of Eurographics 2015)* 34 (2), 2015

Norbert Bus, Nabil H. Mustafa, and Venceslas Biri. “Global Illumination Using Well-Separated Pair Decomposition.” In: *Computer Graphics Forum* 34 (8), 2015

Norbert Bus, Nabil H. Mustafa, and Saurabh Ray. “Geometric Hitting Sets for Disks: Theory and Practice.” In: *23rd European Symposium on Algorithms (ESA)*. 2015

Norbert Bus, Shashwat Garg, Nabil H. Mustafa, and Saurabh Ray. “Improved Local Search for Geometric Hitting Set.” In: *32st International Symposium on Theoretical Aspects of Computer Science (STACS)*. 2015

Norbert Bus, Shashwat Garg, Nabil H. Mustafa, and Saurabh Ray. “Tighter Estimates for epsilon-nets for Disks.” In: *Computational Geometry: Theory and Applications* 53, 2016

Norbert Bus and Lilian Buzer. “Dynamic Convex Hull for Simple Polygonal Chains in Constant Amortized Time per Update.” In: *Proceedings of the 31th European Workshop on Computational Geometry (EUROCG)*. 2015

A Detailed Overview: Problems, Techniques and Results

This chapter aims to provide a compact introduction to the research conducted during the thesis. It describes:

- the problems studied during this thesis,
- the development of the relevant scientific areas,
- the proposed algorithms and data structures,
- and the obtained results.

Detailed technical descriptions of our results follow in the subsequent chapters of the thesis.

Designing efficient algorithms for solving various problems is one of the most important research directions in many scientific fields. The enormous amount of data that algorithms have to deal with requires both fast methods (in the sense of theoretical complexity) and efficient implementations of them. There is frequently a big difference between the best algorithms that achieve good theoretical performance and the best algorithms that run efficiently on real problem instances. One of the most famous examples might be the extremely successful simplex method proposed by Dantzig in 1947. Although its worst case behavior is exponential [KM72], still many linear program solvers use this instead of the provably polynomial methods (e.g., interior point methods [Kar84]) due to its simplicity and efficiency on practical problems. One of the main goals of this thesis is to provide methods that try to guarantee good theoretical results and at the same time their implementation is efficient and robust. Most of our algorithms have been implemented and published on-line, available at the author's website.

Developing efficient methods requires studying the structure and unique properties of the problems. Real-world data has a large geometric component, showing significant geometric patterns. Exploring the geometric nature of data to design efficient methods has became very important in several scientific fields, e.g., computational geometry, discrete geometry, computer graphics, computer vision. For example finite element methods are one of the fundamental tools for a wide range of engineering problems (ranging from design to analysis) and mesh generation lies in the core of these finite element methods [She98]. In this work we aim to investigate how geometric structures can be used to design efficient algorithms for problems in specifically two domains, computer graphics and combinatorial optimization. Although at first sight the two problems seem to be very far from each other, they share geometrical components and there are many problems in computer graphics that have a discrete formulation where combinatorial optimization algorithms could be potentially used. Ultimately, fast enough hitting set algorithms could be utilized to improve various computer graphics problems that rely on selecting a subset of points.

Computer Graphics: Rendering Photo-Realistic Images

One of the main problems in computer graphics is creating synthetic images of digital models, a challenging problem since the 1960's. It became a key component of many modern applications e.g., simulators, industrial design programs, virtual effect creation for the film industry and video games. The ever increasing need for creating the most realistic images of virtual scenes in the least amount of time has triggered many techniques to achieve this goal. The techniques used for rendering can be classified into two big categories based on the speed they require to produce a rendered image. Real-time rendering algorithms produce images that are less realistic and physically incorrect but apply techniques that appear visually pleasing while enabling fast calculations. Such algorithms are usually running on graphics cards that are specialized hardware to facilitate fast rendering of images. They achieve interactive frame rates therefore they are suitable for interactive applications such as games and virtual reality applications. This field is developing extremely fast, see e.g., the SIGGRAPH course notes [Tat09] on the topic held every year since 2006. Algorithms that do not achieve real-time performance, called off-line rendering algorithms, provide much higher fidelity to real photographs through more demanding algorithms. Most of them use ray tracing techniques that try to simulate the propagation of light (photons) as in real world scenes. These algorithms excel in solving the hard problem of calculating global illumination i.e., illumination not coming directly from a light source. These techniques lie at the heart of many CGI applications for movies. Current algorithms are able to render images of very good quality but with large computation times. The ultimate goal of the field is to create a real-time implementation of these techniques on commodity hardware. Our work targets speeding up high fidelity off-line rendering systems although we are not directly aiming for real-time performance.

The rendering equation. Photo-realistic rendering algorithms aim to solve the rendering equation, proposed by Kajiya in 1986 [Kaj86]. Here we provide a simple version of it not accounting for certain phenomena e.g., different wavelengths of light.

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_\Omega f_r(p,\omega_i,\omega_o) L_o(p,\omega_i)(\omega_i \cdot \mathbf{n})\, \mathrm{d}\omega_i \qquad (1)$$

This fundamental equation describes the radiance leaving a point in a stationary state. The equation states that the radiance $L_o(p, \omega_o)$ leaving a point $p \in \mathbb{R}^3$ located on a surface of the scene, in direction $\omega_o \in S^2$ is the sum of the radiance $L_e(p, \omega_o)$ emitted in this direction and the reflected light. The reflected light can be calculated by an integral over the hemisphere Ω aligned to the surface normal $\mathbf{n}$ at p. This integral accounts for all possible directions of received light, integrating the incoming radiance at the point modified by the bidirectional reflectance distribution function (BRDF) and the area of a surface patch perpendicular to the light direction projected over the surface at p. The latter is simply the dot product of ω_i and the surface normal $\mathbf{n}$. The BRDF is a function that represents the material properties of the surface, namely that given the direction of incoming radiance what portion of the it is reflected in a certain outgoing direction.

The most robust algorithms for solving the rendering equation are pure unbiased Monte Carlo algorithms such as path tracing [Kaj86], Metropolis light transport [VG97] or simple bidirectional path tracing [LW93]. These methods are considered to be gold standards for reference solutions. Unfortunately they produce noisy results that are slow to converge in the case of complex scenes. This is due to the stochastic integration methods applied to solve the rendering equation numerically. The noise is visually disturbing and the need to avoid it lead to other developments in the field. Biased algorithms like Photon Mapping [Jen01], point-based global illumination [Chr08] or the many-lights methods, such as Instant Radiosity [Kel97], are less robust solutions but they provide good performance in many practical applications without noise.

Many-lights methods. The last method and its improvements have proven to be very useful for approximating global illumination. By tracing light paths from light sources, they create virtual point lights (VPLs) at the intersections of the surface of the scene and these light paths. For details we refer the reader to [Kel97]. Global illumination is estimated by computing the direct illumination from all of the VPLs. The resulting equation is a discretization of the rendering equation:

$$L_o(x, \omega_o) \approx \sum_{s \in \mathcal{S}} f_r(x, \omega_s, \omega_o) L_s(p, \omega_s)(\omega_s \cdot \mathbf{n}) \tag{2}$$

where the summation is over the set of VPLs $\mathcal{S}$ and ω_s is the direction towards s. $L_s(p, \omega_s)$ denotes the radiance caused by s. Note that this equation isn't recursive anymore and has a discrete form which is beneficial for the application of combinatorial techniques to increase the efficiency of solutions. Since the work of Keller in 1997 [Kel97] there has been steady interest in improving and generalizing the method due to its simplicity and noise free images. Among many results it has been extended to handle highly glossy materials [HKW+09; DKH+10], specular light transfer with Rich-VPLs [SHD15] and participating media with virtual ray and beam lights [NND+12b; NND+12a]. In 2012 there has been a SIGGRAPH course [KHA+12] on the many-lights methods and next year a state-of-the-art report was published at EUROGRAPHICS [DKH+13]. Despite some limitations, they provide a unified and scalable approach to the problem of computing global illumination. However, as the number of VPLs needed for a good-fidelity approximation is very large, computing the illumination for each point by exhaustively summing up the contribution of each individual VPL can become prohibitively expensive. Overcoming this problem has lead to a series of work improving the efficiency of the many-lights methods.

Clustering VPLs. One of the earliest improvements of the method is the work of Paquette *et al.* from 1998 [PPD98], introducing the idea of clustering the VPLs and treating a cluster as a single VPL. The algorithm first builds a hierarchical clustering structure of the VPLs and then for each shaded point it extracts a clustering of the VPLs. Their method has significantly improved the running time of VPL algorithms but it was not robust. The criteria to extract the clustering was not able to adapt to different shaded points often resulting in inadequate clusterings. Since then there has been a long history of adaptive and efficient VPL clustering algorithms. The two most successful approaches for clustering the VPLs are matrix based and tree based methods. The former is based on the study of the light transport matrix [HPB07; HVAP+08; OP11; FBD15]. The rows of the light transport matrix correspond to pixels and columns correspond to VPLs, each cell encoding the contribution of a particular VPL to a pixel. These methods sample the rows of the light transport matrix and calculate the contribution of each VPL to these sample points. Based on this information they cluster the columns of the matrix (VPLs) by using a suitable metric to compare columns. The resulting clustering is used to shade all the points. See Figure 1. Clearly, one global clustering is not adapted to every

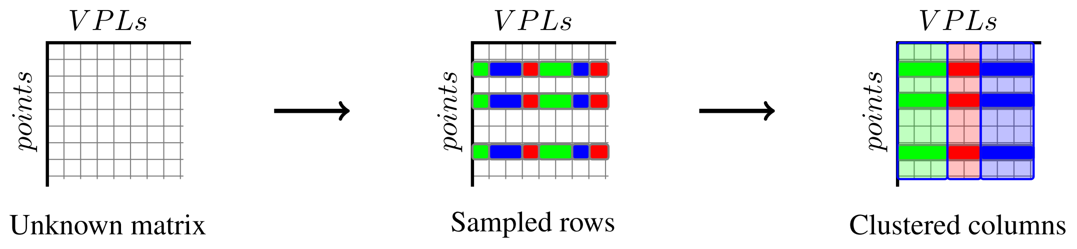


FIGURE 1: Clustering based on the light transport matrix

point. The state-of-the-art method in this family is LightSlice [OP11] that first creates clusters of points that are treated separately and a clustering is constructed for each of these clusters. The main drawback of these methods is that they lack a rigorous error bound for the clustering and due to the sampling they might miss important features in the matrix (especially high frequency details). The second line of work continues the tree based approach of [PPD98]. The most significant algorithm is Lightcuts [WFA+05] published in 2005 that solved the problem of not adapted clusterings by developing a clustering criteria that was able to adapt to complex lighting situations. The method guarantees low error at each shaded point. Lightcuts also builds

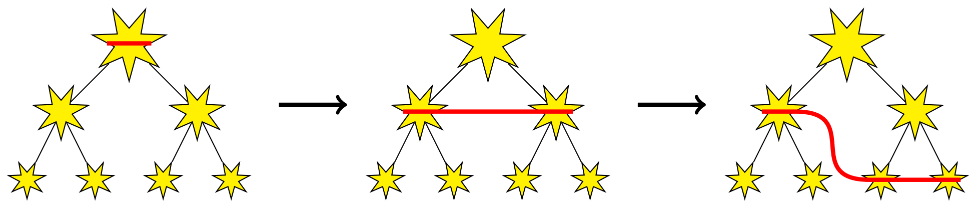


FIGURE 2: Iterative refinement of the clustering in tree based methods.

a hierarchical clustering structure for the VPLs called a lighttree. The leafs of the lighttree correspond to a single VPL and any clustering in the structure is represented by a *cut* in the tree. For each point to be shaded the algorithm progressively descends in the lighttree starting with the root in order to extract a clustering. This descend is carried out by replacing a cluster by its children, therefore refining the clustering. See Figure 2. This process is continued as long as the clustering does not have a low enough error. Ensuring that the error is small is achieved by the following technique. While descending in the tree the algorithm maintains an estimation of radiation caused by the current clustering. Each cluster is tested if the upper bound on the possible error caused by it is less than e.g., 1% of the estimation. If the criteria is not met the cluster with highest possible error is refined and the estimation is updated.

Lightcuts has triggered new interest in the many-light methods as the rendering time became tractable even for millions of VPLs while the error was controlled. Several extensions of the method have been published, some generalizing the effects that are possible to render e.g., Multidimensional Lightcuts [WAB+06] and Bidirectional Lightcuts [WKB12], while some of them aiming to further improve the efficiency of Lightcuts [BD08; WXW11]. Both of our proposed algorithms fall in to the latter category i.e., aiming to improve the efficiency of tree based algorithms. Since the Lightcuts technique is fundamental for the efficiency of many-lights methods, any improvement of it has the potential of improving all methods that are built on it. Let us now describe our contributions along with some motivation.

Global Illumination Using Well-Separated Pair Decomposition. Clustering the VPLs is a successful idea but invokes a problem: how to *efficiently* determine a clustering for each shaded point, since creating a clustering for each point from scratch is prohibitively costly. In the tree based methods this problem was solved by a hierarchical clustering structure since the structure is created only once prior to rendering and extracting a clustering from this structure for a point can be carried out fairly efficiently. But the question is whether this process can be further improved. Our proposed method makes the next logical step in the direction of improving the extraction of clusterings: rather than pre-computing a set of individual candidate clusters that form a hierarchy, from which clusterings are computed during rendering, we pre-compute a number of clusterings. Then, during the rendering phase, the clustering for a point can simply taken to be one of these pre-computed clusterings, together with some minor modification. Our data structure stores the clusterings compactly, it is view-independent and it is computed prior to rendering. This results in a very efficient rendering phase allowing for changing the camera position without the necessity to repeat the preprocessing steps.

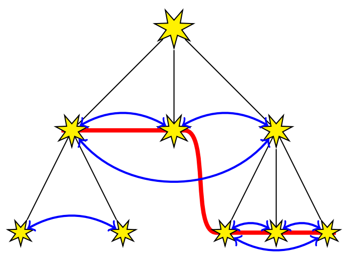


FIGURE 3: The well-separated pair decomposition where the sets correspond to nodes in an octree. Pairs shown in blue with a clustering shown in red for a shaded point whose closest VPL is the rightmost one.

We give a short informal description of the proposed method and data structures. Let us denote the set of VPLs by $\mathcal{S}$. Our approach has two main components: first we propose a method to pre-compute and compactly store several clusterings of the VPLs. Then during rendering, we show how to quickly extract a clustering for each point to be shaded using this pre-computed structure.

The naive algorithm, namely to explicitly store all the clusterings, would have prohibitively high memory requirements. We succeeded to overcome this limitation by encoding them compactly in a geometric structure. Our data structure is based on the well-separated pair decomposition, hereafter WSPD, that has been developed for the N-body simulation problems [CK95]. The WSPD is a set of pairs of subsets $\{\{R_1, Q_1\}, \{R_2, Q_2\}, \ldots\}$ where $R_i, Q_i \subseteq P$ such that

$i)$ for every pair of points $p, q \in P$, there is a unique index i such that $p \in R_i$ and $q \in Q_i$

$ii)$ for all pairs, the subsets R_i and Q_i are well-separated from each other.

Two subsets R_i, Q_i are well-separated if their distance from each other is much bigger than the radius of their enclosing spheres. Informally this means, that from the point of view of R_i the cluster Q_i can be considered as one point. Note that i) results in the fact that a WSPD implicitly stores for any point $p \in P$ a *well-separated* clustering of $P/\{p\}$. To extract such a clustering one only needs to list the pairs that contain p and list the subsets in these pairs that do not contain the point. The well-separated pair decomposition theorem [CK95] states that for any set of points $P \subset \mathbb{R}^3$ there exists a WSPD of size linear in $|P|$. These facts enable the storage of all clusterings without excessive memory requirements. One way to construct a WSPD is to build an octree for the point set and search for well-separated pairs consisting of subsets of points that correspond to nodes in the tree. Therefore, instead of storing a list of pairs, a WSPD is represented by links between nodes of the tree. This way, due to the octree structure, for any point $p \in P$ extracting a clustering becomes very simple: it suffices to list the pairs of the nodes that are on the path from p to the root. These properties make the WSPD very useful for our problem. Our algorithm builds a WSPD of the VPLs based on their position.

See Figure 3 for the octree based representation of a WSPD and a clustering for the rightmost VPL. It is clear that the pairs of the nodes on the path to the root form a clustering. We show that the well-separated condition under certain circumstances ensures low shading error i.e., if R_i was a set of points to be shaded then for any point in R_i the difference between calculating the radiance from each VPL in Q_i or treating them as a single cluster is low. This enables us to use the clusterings of the WSPD as VPL clusterings. Well-separated clusters approximate the geometric terms of the rendering equation, but ignore visibility and directional properties. To adjust for this, we further compute two additional structures in the pre-processing phase. First, we further group the lights in each cluster into a small number of subgroups by similar light normals. This additional grouping will be used to evaluate the illumination from the cluster more precisely. Second, we introduce representative lights that approximate local visibility for each cluster.

Given the WSPD built on the VPLs the rendering becomes very efficient. For an arbitrary point p to be shaded, find the closest point in $\mathcal{S}$ to p (an approximate nearest-neighbor is sufficient and will be used), and start with its (pre-computed) clusters as the clustering for p. Furthermore, refine each cluster by subdividing it into new clusters until they are well-separated from $\{p\}$. We show that refinement can only add a *constant* number of new clusters for any point p. This constant is provably *independent* of the number of lights in $\mathcal{S}$ or points to be shaded.

The method achieves good performance compared to Lightcuts. Depending on the complexity of the scene and implementation of the system it shows around 2-3 times speed-up of the rendering phase with equal quality, but with a rather long preprocessing time. The drawback of the method is that in the presence of glossy materials its performance degrades. Chapter 2 describes the technical details of this work and is based on the following paper.

> Norbert Bus, Nabil H. Mustafa, and Venceslas Biri. "Global Illumination Using Well-Separated Pair Decomposition." In: *Computer Graphics Forum* 34 (8), 2015

IlluminationCut. The drawbacks of our WSPD based method which relied mainly on geometric structures inspired us to further study the problem. Our second algorithm approaches the problem of speeding up clustering methods from a more combinatorial side. It handles highly glossy material and provides a tight error bound for the error resulting from the clustering. On the other hand, it is not view independent making it unsuitable for fast rendering with changing camera positions. The method is motivated by the study of the structure of all the clusterings that an algorithm produces during rendering an image. To give a clear explanation let us first present the structure of these clusterings for the state-of-the-art methods Lightcuts and Lightslice. We will illustrate them on the light transport matrix.

The main observation is that instead of clustering points or VPLs independently the key structure to study is the *clustering of their product-space*, namely the clustering of all point-VPL pairs.

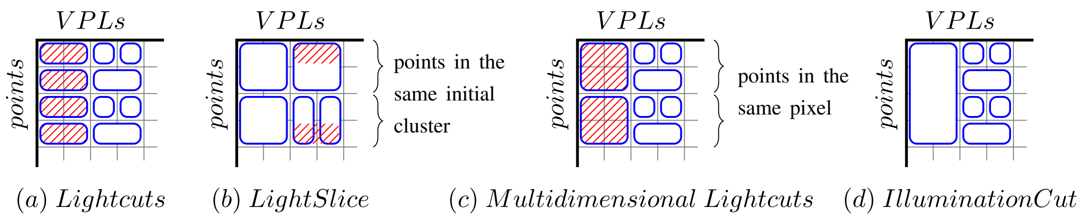


FIGURE 4: Partial light transport matrices, with rectangles denoting product-space clusters. Red stripes denote parts that could be improved. (a) Lightcuts creates clusters that could be merged; (b) LightSlice creates clusters that should be merged or refined; (c) Multidimensional Lightcuts only merges and refines clusters limited to points originating from the same pixel; (d) IlluminationCut merges and refines clusters for any set of points and VPLs.

Each cluster in this product-space consists of a subset of points (to be shaded) paired with a subset of VPLs.

In fact, both Lightcuts and LightSlice can be seen as constructing constrained product-space clusterings. Lightcuts recomputes a clustering for every point therefore in the product space each cluster created by Lightcuts consists of a single point paired with a set of VPLs. This constraint is wasteful as two points which are very similar could have been grouped together and therefore the expensive operation to extract this cluster could have been spared. See Figure 4 (a). LightSlice first groups all points to be shaded into a small number of roughly equal-sized clusters, called point-clusters, based on their geometric proximity then creates a clustering of the VPLs for each of these point-clusters. This means that it constructs a product-space clustering where the same set of points are grouped together in any cluster. For efficiency reasons each point-cluster is large, which severely limits how well the VPL-clusters paired to them can be adapted to each individual point in the point-cluster. See Figure 4 (b). The wasteful computation of Lightcuts has been long noticed and there have been a few attempts to resolve it [BD08; WXW11]. All these methods aim to create a common clustering for a group of points, similarly to LightSlice but starting from this common clustering they refine it for each individual point. These methods have to balance between how many points can share a clustering and how well adapted this clustering is to each point (or in other words how much additional refinement is needed).

Our proposed method, *IlluminationCut*, creates a product-space clustering without any a priori constraints on either the points or the VPLs that can appear in product-space clusters. These clusters capture similar point-VPL pairs such that shading every point in a cluster by using a single representative VPL instead of all VPLs in the cluster causes error that remains under a threshold. Treating cluster pairs enables us to amortize calculations that were previously carried out separately for each point in Lightcuts; and to construct non-uniform clusters with different subsets of points with different subsets of VPLs, which is more adaptive clustering than that of LightSlice. Moreover we don't require additional refinement since our clusters are already adapted to each individual point. See Figure 4 (d).

IlluminationCut builds on the Multidimensional Lightcuts approach [WAB+06], in that they both utilize two hierarchies (trees), one on points and one on VPLs to construct product-space clusters with bounded error by simultaneously descending on the trees. The difference is that the latter has to maintain a heap and repeatedly builds a tree only for points that originate from the same pixel (e.g., for use in spatial anti-aliasing). It does not exploit possible similarity among points originating from different pixels and so does not improve upon Lightcuts if there is only one point per pixel. See Figure 4 (c). The difficulty in extending the method over pixel boundaries is to maintain the tight upper bound on the error. We resolve this problem by introducing a two phase rendering algorithm. Phase I computes a coarse but fast approximation of the radiance at every shaded point. In Phase II, this approximate radiance is used to guide a top-down search of both trees simultaneously to construct the list of desired product-space clusters. The top-down search is almost identical to the one used in Lightcuts, only that now we start with the pair of roots of the two trees and we refine point and VPL nodes in an alternating order. This search continues as long as a product space cluster has a bigger maximum possible error than 1% of the minimum of the coarse approximate illumination for any point in the cluster. We use almost the same error bounds as in Lightcuts with only a very slight relaxation of the error bound, practically obtaining a same quality clustering. Finally, for all product-

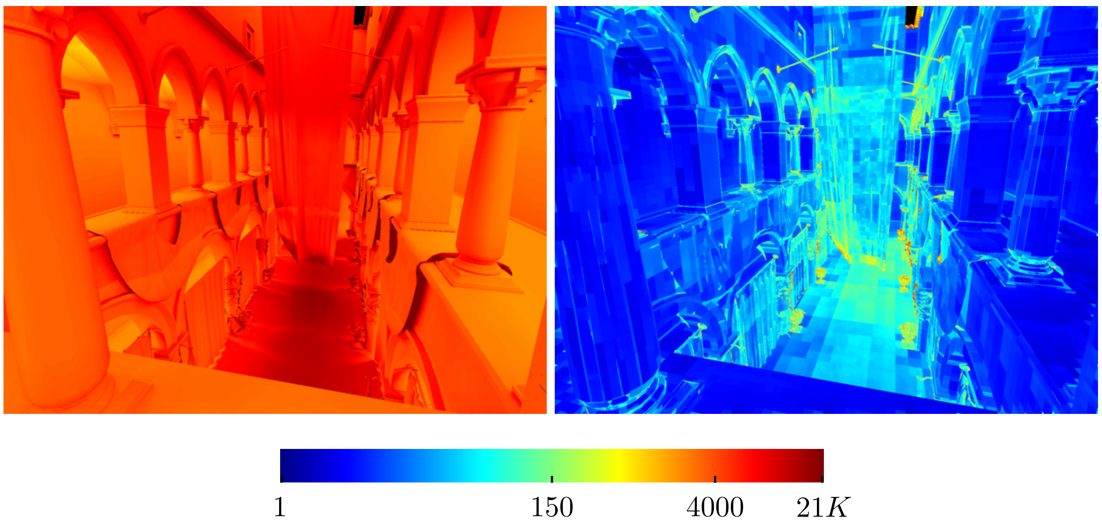


FIGURE 5: The logarithm of the number of clustering calculations per point for the well-known Sponza scene for Lightcuts (left) and IlluminationCut (right). IC amortizes the cost of clustering calculations very efficiently.

space clusters, illumination contribution from the representative is added to the radiance of each point in the cluster. To compare the efficiency of Lightcuts and IlluminationCut, see Figure 5, where we present the logarithm of clustering computations carried out per point. It is clear that our method is superior to Lightcuts in efficiency. Our method is further extended by adaptive visibility sampling within a product-space cluster i.e., we do not trace a ray to all the points in the cluster, only if a small sample of rays shows different visibility values. This reduces the number of rays traced for each product-space cluster without introducing high error since the our tight error bound already ensures that the error that can be caused by the cluster is small.

Our results improve on both the quality and the efficiency of previous methods. We achieve $3-6$ times speed-up over both Lightcuts and LightSlice by reducing the number of visibility queries, dramatically decreasing the computations needed to construct clusters, as well as eliminating the need for maintaining a heap during rendering in Lightcuts. The drawback of the method is that it exploits the similarity of shaded points and if the scene is very heterogeneous its performance degrades. Chapter 3 describes the technical details of this work and is based on the following paper.

> Norbert Bus, Nabil H. Mustafa, and Venceslas Biri. "IlluminationCut." In: *Computer Graphics Forum (Proceedings of Eurographics 2015)* 34 (2), 2015

The family of Lightcuts methods have attracted much attention since they have been published and we believe that future research will be able to provide several improvements. One of the most important directions is to optimize the error bounds used by them since it is a widely observed fact that in dark areas of the image and in the presence of highly glossy materials the current error bounds create unnecessarily refined clusterings.

Combinatorial Optimization

Combinatorial optimization is one of the fundamental fields of discrete mathematics with a very wide range of problems coming from both theoretical and applied research. It aims to find an optimal object from a finite set of objects. The difficulty in these problems is that an exhaustive search for the solution is not tractable due to the exponentially large set of possible solutions. Some of the most well-known problems in the field include the traveling salesman problem, integer programming or vehicle routing problems. Many problems in the field are NP-complete, therefore finding polynomial time algorithms that solve the problem exactly seem to be impossible. This has led to a vast amount of research on solving special cases of them or giving approximately good solutions. Our work aims to solve problems efficiently even for large datasets while obtaining a solution closest to the optimal.

The minimum hitting set problem, the focus of our work, is one of the most fundamental combinatorial optimization problems: given a range space $(P, \mathcal{D})$ consisting of a set P and a set $\mathcal{D}$ of subsets of P called the *ranges*, the task is to compute the smallest subset $Q \subseteq P$ that has a non-empty intersection with each of the ranges in $\mathcal{D}$. This problem is strongly NP-hard. If there are no restrictions on the set system $\mathcal{D}$, then it is known that it is NP-hard to approximate the minimum hitting set within a logarithmic factor of the optimal [RS97]. The problem is NP-complete even for the case where each range has exactly two points since this problem is equivalent to the vertex cover problem which is known to be NP-complete [Kar72; GJ79]. A natural occurrence of the hitting set problem occurs when the range space $\mathcal{D}$ is derived from geometry – e.g., given a set P of n points in $\mathbb{R}^2$, and a set $\mathcal{D}$ of m triangles containing points of P, compute the minimum-sized subset of P that hits all the triangles in $\mathcal{D}$. Unfortunately, for most natural geometric range spaces, computing the minimum-sized hitting set remains NP-hard. For example, even the (relatively) simple case where $\mathcal{D}$ is a set of unit disks in the plane is strongly NP-hard [HM87]. Therefore fast algorithms for computing provably good approximate hitting sets for geometric range spaces have been intensively studied for the past three decades (e.g., see the two recent PhD theses on this topic [Fra12; Gan11]). Although the problem seems to be very theoretical at first sight, it turns out that it can model a wide range of real world problems e.g., designing wireless networks [LN12], filtering data of bathymetric measurements [Fra12], image processing and VLSI design [HM85], model based diagnosis systems [LJ03] or classifier generation [VØ00]. These practical problems make efficient implementations of hitting set algorithms highly desired.

The special case studied in this work – hitting sets for disks in the plane – has been the subject of a long line of research. The case when all the disks have the same radius is easier, and has been studied in a series of works: Călinsecu *et al.* [CMW+04] proposed a 108-approximation algorithm, which was subsequently improved by Ambhul *et al.* [AEM+06] to 72. Carmi *et al.* [CKLT07] further improved that to a 38-approximation algorithm, though with the running time of $O(n^6)$. Claude *et al.* [CDD+10] were able to achieve a 22-approximation algorithm running in time $O(n^6)$. More recently Fraser *et al.* [DFLO+12] presented a 18-approximation algorithm in time $O(n^2)$ and Acharyya *et al.* [ABD13] presented an $(9+\epsilon)$-approximation algorithm in $O(n^{4+18/\epsilon}\log n)$ time. See Table 1 for the state-of-the-art and our contributions.

So far, besides ad-hoc approaches, there are two systematic lines along which all progress on the hitting-set problem for geometric ranges has relied on: rounding via ϵ-nets, and local-search. Both these approaches have to be evaluated on the questions of computational efficiency as well as approximation quality. In spite of all the progress, there remains a large gap between theory and practice – mainly due to the ugly trade-offs between running times and approximation factors. We have made progress on both approaches and we describe these methods in the following along with the necessary definitions.

Rounding via ϵ-nets. Given a range space $(P, \mathcal{D})$ and a parameter $\epsilon > 0$, an ϵ-net is a subset $S \subseteq P$ such that $D \cap S \neq \emptyset$ for all $D \in \mathcal{D}$ with $|D \cap P| \geq \epsilon n$. The famous "$\epsilon$-net theorem" of Haussler and Welzl [HW87] states that for range spaces with VC-dimension d, there exists an ϵ-net of size $O(d/\epsilon \log d/\epsilon)$ (this bound was later improved to $O(d/\epsilon \log 1/\epsilon)$, which was shown to be optimal in general [PA95; Mat02]). Sometimes, weighted versions of the problem are considered in which each $p \in P$ has some positive weight associated with it so that the total weight of all elements of P is 1. The weight of each range is the sum of the weights of the

CONGRUENT DISKS		
	Quality	**Time**
Călinsecu *et al.* [CMW+04]	108	$O(n^2)$
Ambhul *et al.* [AEM+06]	72	$O(n^2)$
Carmi *et al.* [CKLT07]	38	$O(n^6)$
Claude *et al.* [CDD+10]	22	$O(n^6)$
Fraser *et al.* [DFLO+12]	18	$O(n^2)$
Acharyya *et al.* [ABD13]	$(9+\epsilon)$	$O(n^{4+18/\epsilon}\log n)$
ARBITRARY DISKS		
	Quality	**Time**
Bronniman-Goodrich *et al.* [BG95]	$O(1)$	$O(n^3)$
Mustafa-Ray *et al.* [MR10]	$(1+\epsilon)$	$n^{O(1/\epsilon^2)}$
Agarwal *et al.* [AES12]	$O(\log n)$	$\tilde{O}(n)$
Agarwal-Pan [AP14]	$O(1)$	$\tilde{O}(n)$
OUR RESULTS FOR ARBITRARY DISKS		
Bus *et al.* [BMR15]	13.4	$\tilde{O}(n)$
Bus *et al.* [BGM+15]	$(8+\epsilon)$	$\tilde{O}(n^{7/3})$

TABLE 1: Summary of previous work.

elements in it. The aim is to hit all ranges with weight more than ϵ. The condition of having finite VC-dimension is satisfied by many geometric set systems: disks, half-spaces, k-sided polytopes, r-admissible set of regions etc. in $\mathbb{R}^d$. For certain range spaces, one can even show the existence of ϵ-nets of size $O(1/\epsilon)$ – an important case being for disks in $\mathbb{R}^2$ [PR08]. In 1994, Bronnimann and Goodrich [BG95] proved the following interesting connection between the hitting-set problem [1], and ϵ-nets: let $(P, \mathcal{D})$ be a range-space for which we want to compute a minimum hitting set. If one can compute an ϵ-net of size c/ϵ for the ϵ-net problem for $(P, \mathcal{D})$ in polynomial time, then one can compute a hitting set of size at most $c \cdot \textsc{Opt}$ for $(P, \mathcal{D})$, where $\textsc{Opt}$ is the size of the optimal (smallest) hitting set, in polynomial time. A shorter, simpler proof was given by Even *et al.* [ERS05]. Both these proofs construct an assignment of weights to points in P such that the total weight of each range $D \in \mathcal{D}$ (i.e., the sum of the weights of the points in D) is at least $(1/\textsc{Opt})$-th fraction of the total weight. Then a $(1/\textsc{Opt})$-net with these weights is a hitting set. Until very recently, the best such rounding algorithms had running times of $\Omega(n^2)$, and it had been a long-standing open problem to compute a $O(1)$-approximation to the hitting-set problem for disks in the plane in near-linear time. In a recent break-through, Agarwal-Pan [AP14] presented an algorithm that is able to do the required rounding efficiently for a broad set of geometric objects. In particular, they are able to get the first near-linear algorithm for computing $O(1)$-approximations for hitting sets for disks. The catch is that the approximation factor depends on the sizes of ϵ-nets for disks; despite over 7 different proofs of $O(1/\epsilon)$-sized ϵ-nets for disks, the precise bounds are not very encouraging. So far, the best constants for the ϵ-nets come from the proofs in [PR08] and [HKS+14]. The latter paper presents

[1]They actually proved a more general statement, but the following is more relevant for our purposes.

five proofs for the existence of linear size ϵ-nets for halfspaces in $\mathbb{R}^3$. The best constant for disks is obtained by using their first proof. A lifting of the problem of disks to $\mathbb{R}^3$ gives an ϵ-net problem with lower halfspaces in $\mathbb{R}^3$, for which [HKS+14] obtains a bound of $\frac{4}{\epsilon} f(\alpha)$ where $\alpha < \frac{1}{3}$ and $f(\alpha)$ is the best bound on the size of an α-net for lower halfspaces in $\mathbb{R}^3$. Using the lower bound of [PW90] for halfspaces in $\mathbb{R}^2$, $f(\alpha) \geq \lceil 2/\alpha \rceil - 1 \geq 6$, although we believe that it is at least 10 since even for $\epsilon = 1/2$, no ϵ-net construction of size less than 10 is known. Thus, the best constructions so far give a bound that is at least $24/\epsilon$ and most likely more than $40/\epsilon$. Furthermore, there is no implementation or software solution available that can even compute such ϵ-nets efficiently.

The breakthrough algorithm of Agarwal-Pan [AP14] has two problems: the constant in the approximation depends on the constant in the size of ϵ-nets, which are large and it uses sophisticated data-structures that have large constants in the running time. Our work makes an attempt to improve on both shortcomings. It turns out that Delaunay triangulations will be the key structures in our approach.

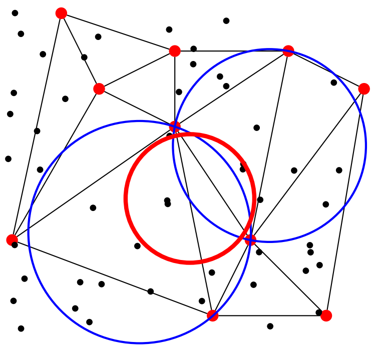


FIGURE 6: Delaunay triangulation of the sampled points (in red). Disks that are not hit (in red) are contained in two neighboring Delaunay disks (in blue).

The algorithm for constructing small sized ϵ-nets starts by taking a random sample of the points in P. The key observation is that any disk that is not hit by this random sample is contained in two neighboring Delaunay disks i.e., disks that correspond to neighboring Delaunay triangles. This enables the application of a divide and conquer approach i.e., create subproblems for each pair of neighboring Delaunay disks and solve them recursively. We show that such subproblems can be created efficiently i.e., in near linear time. Moreover, we give algorithms to solve certain subproblems without recursion by proposing a novel method to construct small sized ϵ-nets for large values of ϵ, i.e, $\epsilon > 1/2$. Through a careful analysis we show that by properly adjusting the probability with which we pick points into the random sample this randomized algorithm gives an ϵ-net of expected size $13.4/\epsilon$ in near linear time. We have implemented our algorithm showing that in practice it can output ϵ-nets of size $9/\epsilon$. Together with the result of Agarwal-Pan, this immediately implies: for any $\delta > 0$, one can compute a $(13.4 + \delta)$-approximation to the minimum hitting set for $(P, \mathcal{D})$ in time $\tilde{O}(n)$.

Technical description can be found in 6 and it is based on the following publication.

> Norbert Bus, Shashwat Garg, Nabil H. Mustafa, and Saurabh Ray. “Tighter Estimates for epsilon-nets for Disks.” In: *Computational Geometry: Theory and Applications* 53, 2016

To address the second shortcoming of the Agarwal-Pan algorithm we propose a modification of it that does not use any complicated data structures – just Delaunay triangulations, ϵ-nets. This comes with a price: although experimental results indicate a near-linear running time, we have been unable to theoretically prove that the algorithm runs in expected near-linear time. The idea for avoiding the use of range-reporting data-structure is to observe that the very fact that a disk D is not hit by $\mathcal{Q}$, where $\mathcal{Q}$ is an ϵ-net, makes it possible to use Q in a simple way to efficiently enumerate the points in D, since D lies in the union of two Delaunay disks in the Delaunay triangulation of $\mathcal{Q}$. The range counting data structure in the Agarwal-Pan algorithm is used to determine all the disks that contain a low number of points (in other words have a low weight). This structure is replaced by a second ϵ-net since disks that are not hit by it are guaranteed to have a low weight. Unfortunately this does not completely solve the problem since there might be disks that are hit but have low weight. We propose additional steps to ensure that all disks that have a low weight are found, but these steps prevent our algorithm to run in near linear time in worst case.

We have published an implementation of our algorithm and give detailed experimental results on both synthetic and real-world data sets, which indicates that the algorithm computes, on average, a 1.3-approximation in near-linear time. This surprisingly low approximation factor compared to the proven worst case bound is the result of fine tuning the parameters of the algorithm. The algorithm is capable of handling extremely large datasets. We have tested it on set of points consisting of the locations of cities on Earth (except for the USA) containing altogether 10 million points with disks of fixed radius around each point, see Figure 7 for a visualization based on a subset of the problem.

The complete technical description can be found in 7 and it is based on the following publication.

> Norbert Bus, Nabil H. Mustafa, and Saurabh Ray. “Geometric Hitting Sets for Disks: Theory and Practice.” In: *23rd European Symposium on Algorithms (ESA)*. 2015

Local search. There is a fundamental limitation of the rounding technique: it *cannot* give better than constant-factor approximations. The reason is that the technique reduces the problem of computing a minimum size hitting set to the problem of computing a minimum sized ϵ-net. And it is known that for some constant $c \geq 2$, there do not exist ϵ-nets of size smaller than c/ϵ. This limitation of the rounding technique – that it cannot give a PTAS – was the main barrier towards better quality algorithms until the usefulness of local search algorithms was introduced. Starting

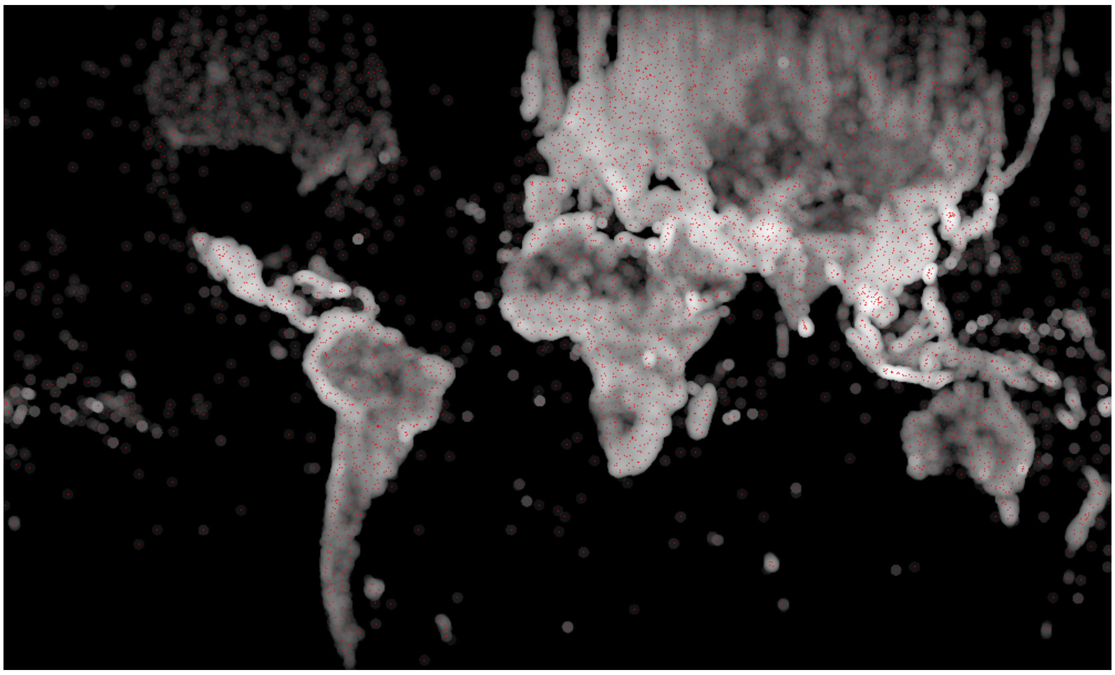


FIGURE 7: A subset of the hitting set for cities on Earth.

from the beautiful use of local search for clustering problems in Arya *et al.* [AGK+01], there has been recent progress in breaking the constant-approximation barriers for many geometric problems; e.g., see [KMN+02; AM06; CHP09]. For the hitting set problem on $(P, \mathcal{D})$, consider the following algorithm: start with any hitting set $S \subseteq P$, and repeatedly decrease the size of S, if possible, by replacing k points of S with $< k$ points of $P \setminus S$. Call such an algorithm a $(k, k-1)$-local search algorithm. Mustafa-Ray [MR10] showed that a $(k, k-1)$-local search algorithm for the hitting set problem gives a $(1 + c/\sqrt{k})$-approximation, for a fixed constant c, when the ranges are disks, or more generally, pseudo-disks in $\mathbb{R}^2$. The running time of their algorithm to compute a $(1 + \epsilon)$-approximation is $O(n^{O(1/\epsilon^2)})$.

The k-local search algorithm of Mustafa-Ray [MR10] can compute solutions arbitrarily close to the optimal, but it is extremely inefficient, even for reasonable approximation factors. For example, it takes time $O(n^{66})$[Fra12] to compute a 3-approximation. Furthermore, note that any attempts at progress on improving local search must take into account that as the hitting set problem is strongly NP-hard, it is unlikely that algorithms exist that do not have a dependency on $1/\epsilon$ in the exponent. Therefore in this work we undertake a closer study of $(k, k-1)$-local search for $k = 3$. We show that a $(3, 2)$-local search algorithm returns a 8-approximation to the minimum hitting set. We show this by creating a bipartite graph $G = (R, B, E)$ on the points of the optimal solution R and the points of the local search result B with edges E between nodes of R and B located in the same disk. The condition that no more steps of our local-search can be carried out implies that no three nodes of B have a neighborhood of less than three nodes which is crucial for showing that $|B| \leq 8|R|$. Furthermore, by giving a construction of a special graph, we show that this is tight i.e., there exist a set of points P and a set of disks

$\mathcal{D}$ where $(3,2)$ local search does not return hitting-sets of size less than 8 times the size of the optimal hitting set. A straightforward algorithm for $(3,2)$-local search proceeds as follows: each $(3,2)$ improvement step tries all $O(n^5)$ 5-tuples, and for each tuple it checks in time $O(n)$ if it is indeed an improvement. The total number of steps can be $O(n)$, giving a $O(n^7)$ naive running time. We show how to perform this search more efficiently, in expected time $O(n^{2.34})$. One of the key observations is that a pair of points considered by the local search algorithm might replace more than one single 3-tuple. We propose a randomized algorithm that efficiently find pairs that replace many 3-tuples of points hence at the same time decreases the number of required local search steps and improves the complexity of each individual step. Technical results are presented in Chapter 8 based on the following paper.

Norbert Bus, Shashwat Garg, Nabil H. Mustafa, and Saurabh Ray. "Improved Local Search for Geometric Hitting Set." In: *32st International Symposium on Theoretical Aspects of Computer Science (STACS)*. 2015

Both methods proposed for solving the approximate hitting set problem have improved upon existing algorithms, but it always remains an interesting question to see how far these algorithms can be pushed to achieve better running times or better approximation factors.

The last part of this work focuses on a different problem, namely efficient computation of convex hulls in the plane. The convex hull of a set of n points in a Euclidean space is the smallest convex set containing all the points. It has applications in, e.g., pattern recognition, image processing and micro-magnetic analysis [PS85; PGD+96]. We propose an on-line algorithm to construct the dynamic convex hull of a simple polygonal chain in the Euclidean plane supporting deletion of points from the back of the chain and insertion of points in the front of the chain. Both operations require amortized constant time. The key idea of our algorithm is to build two convex hulls, one handling insertions and the other handling deletions. The desired convex hull is simply the merge of these hulls. The two hulls are built as follows. Given a polygonal chain we create one empty convex hull that can be iteratively expanded when we are inserting points and we build another one for all the points currently in the chain. This hull will shrink as points are deleted and when it becomes empty we simply reset the data structures. See Figure 8.

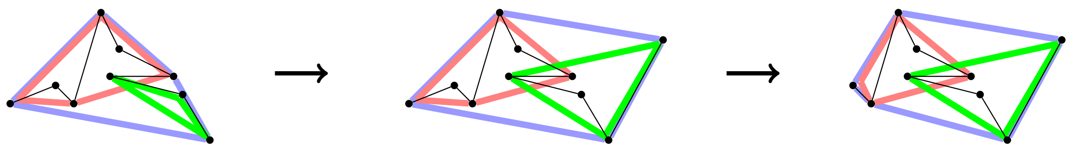


FIGURE 8: From left to right: initial setup, inserting a point, deleting a point. The red convex hull handles the deletion of points, the green convex hull handles the insertion of points and the blue convex hull is the merge of them.

The image on the left depicts an intermediate state after addition and deletion of points. The image on the middle shows adding a point. The convex hull that handles insertions (in green) is extended and the merge of the convex hulls (in blue) is updated. The image on the right depicts the deletion of a point. Here the previously built convex hull (in red) decreases and the merged hull is updated. Technical details can be found in Chapter 9. The presentation is the extended version of the following work.

Norbert Bus and Lilian Buzer. “Dynamic Convex Hull for Simple Polygonal Chains in Constant Amortized Time per Update.” In: *Proceedings of the 31th European Workshop on Computational Geometry (EUROCG)*. 2015

Part I

Computer Graphics: Rendering Photo-Realistic Images

Chapter 1

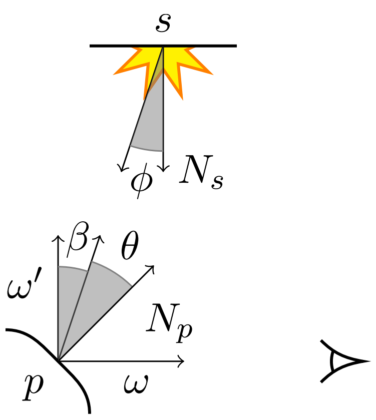
Introduction to Many-Lights Methods

Rendering photo-realistic images efficiently is a challenging task in computer graphics. As the complexity of scenes, materials and lighting increases, so does the need for fast and accurate rendering methods. Unbiased algorithms such as Metropolis light transport [VG97] or bidirectional path tracing [VG95] result in the best quality images and handle the widest range of illumination types but take long time to converge due to their stochastic nature. Several solutions have been proposed to speed up rendering and to alleviate noise quickly but most of them do not retain the unbiased property of pure path tracing algorithms. Such solutions include Photon Mapping [Jen01], point-based illumination [Chr08] and many-lights methods such as Instant Radiosity [Kel97].

This chapter aims to introduce the notations used in our many-lights algorithms and to present previous results on the problem. Many-lights methods have gained much attention recently since they produce high quality images in a fraction of the time taken by Monte Carlo methods without noise. They are based on the observation that one can generate virtual light sources prior to rendering that are responsible for approximating global illumination. The image is rendered by simply summing up the direct contribution of all virtual light sources. We omit the details of the procedure to generate virtual light sources (referred to as virtual point lights or VPLs hereafter). Details and a formal derivation of the method can be found in [Kel97]. Using VPLs we obtain a discretization of the rendering equation that makes handling the problem easier since it has no recursive part.

Denote by $\mathcal{S}$ the set of VPLs and by $\mathcal{P}$ the set of points to be shaded (i.e., points in the scene hit by the rays traced from the camera). We use similar notation as in Lightcuts [WFA+05]. The radiance for a point $p \in \mathcal{P}$ in direction ω caused by the direct contribution of the lights in $\mathcal{S}$ is denoted by $L(p, \omega)$. It is a function that sums up over all lights the product of the material, geometry, visibility and intensity terms, where each product represents the radiance caused by a single light:

$$L(p, \omega) = \sum_{s \in \mathcal{S}} M_s(p, \omega) \cdot G_s(p) \cdot V_s(p) \cdot I_s \tag{1.1}$$



I_s is the intensity of the light s and $V_s(p)$ denotes the visibility between s and p. $M_s(p, \omega)$ is the BRDF which depends on the material at p. We use Lambertian and Blinn micro-facet BRDFs (in Chapter 2 the latter is replaced by Phong BRDFs). They have the form $\frac{1}{\pi} k_{\text{diff}} \cos\theta$ and $\frac{1}{2\pi} k_{\text{spec}}(n+2) \cos(\beta)^n \cos\theta$ respectively ($\frac{1}{2\pi} k_{\text{spec}} \cos(\beta)^n \cos\theta$ in the case of the Phong BRDF), where each component of k_{diff} and k_{spec} has values between 0 and 1, and n is the specular coefficient. The angles β, ϕ and θ are denoted in the figure below where ω' is the view direction ω reflected with the surface normal N_p, and N_s is the normal of the light s. With these notations β is the angle between ω' and $s - p$, ϕ is the angle between N_s and $p - s$ while θ denotes the angle between N_p and $s - p$. The geometric term $G_s(p)$ captures the light attenuation $G_s(p) = \cos(\phi)/d(p, s)^2$, where $d(p, s)$ is the Euclidean distance between p and s.

For a cluster $C \subseteq \mathcal{S}$, let $rep(C)$ denote a representative light $s \in C$. Then the radiance at p from lights in C with representative $rep(C)$ can be approximated as:

$$L_C(p, \omega) = M_{rep(C)}(p, \omega) \cdot G_{rep(C)}(p) \cdot V_{rep(C)}(p) \cdot \sum_{s \in C} I_s \tag{1.2}$$

Let $\mathcal{C} = \{C_1, \ldots, C_k\}$ denote a clustering of $\mathcal{S}$ into k clusters. The radiance at P from all the lights in $\mathcal{S}$ can be approximated as:

$$L_{\mathcal{C}}(p, \omega) = \sum_{C \in \mathcal{C}} L_C(p, \omega) \tag{1.3}$$

For further details on the many-lights methods we refer the reader to the SIGGRAPH course notes on the many-lights problem [KHA+12] and the EG state-of-the-art report [DKH+13]. In what follows we give a brief introduction to various improvements and extensions of the many-lights methods that have been left out from the introduction.

Real-time techniques. The many-lights framework can be used for rendering images in real time with incremental updating of VPLs, though this is limited to few hundreds of VPLs [LSK+07]. Other methods achieving interactive frame rates include calculating level of details structures efficiently [HRE+11] and imperfect shadow maps [RGK+08]. Clustering VPLs into area lights for real-time GPU based rendering has been proposed in [PKD12]. [SIM+06] uses stochastic sampling of VPLs to achieve interactive frame rates. For a summary see [Rad08]. Our methods have different scope from all these approaches since they are designed to handle significantly larger number of VPLs.

Extensions and limitations. To render participating media, virtual ray and virtual beam lights have been introduced in [NND+12b; NND+12a], respectively. Limitations of VPL-based algorithms, like clamping [KK06] of VPL contributions due to the singularities or the limitation of only representing diffuse global illumination, can be solved using virtual spherical lights [HKW+09] or Rich-VPLs [SHD15] instead of virtual point lights. To handle more efficiently highly glossy material [KFB10], Davidovič et al. [DKH+10] use row-column sampling with an adaptive ray-casting strategy. Other techniques such as bidirectional lightcuts [WKB12] or specular gathering [DKL10] combine path tracing techniques and VPL global illumination. These methods greatly increase the rendering time compared to pure many-lights algorithms.

Good fidelity off-line rendering requires a large number of VPLs (typically hundreds of thousands of VPLs). There are two main approaches that avoid computing the radiance for all point-VPL pairs.

Sampling. To decrease the complexity of the problem, one can sample the VPLs according to their contribution to the image [GS10]. The method proposed in [GKP+12] calculates the exact illumination at a sparse set of locations and builds the probability distribution of the incoming light at each sample. These distributions are then used to importance sample the VPLs. [SIM+06] uses stochastic sampling of the VPLs to achieve real-time global illumination. Compared to light clustering implementations, sampling introduces unstructured noise. [SIP07] samples the VPLs using a modified Metropolis-Hastings algorithm [VG97].

Visibility estimation. To further speed up rendering [PGS+13] caches visibility queries. The authors in [BEL+13] propose a novel framework for stochastic evaluation of visibility. VisibilityCluster [WC13] first clusters the pixels and the VPLs separately and creates pairs of these clusters. Then it calculates approximate visibility for these pairs and uses this information to improve the importance sampling of VPLs.

Chapter 2

Global Illumination Using Well-Separated Pair Decomposition

This chapter describes a data structure that compactly stores all possible clusterings of the VPLs and enables efficient extraction of a clustering for any shaded point, resulting in a very fast rendering algorithm in the many-lights framework. This chapter is based on the following paper.

Norbert Bus, Nabil H. Mustafa, and Venceslas Biri. "Global Illumination Using Well-Separated Pair Decomposition." In: *Computer Graphics Forum* 34 (8), 2015

2.1 General Idea

One key idea for speeding up computations is to cluster the VPLs. In other words, for each point $p \in \mathcal{P}$, partition $\mathcal{S}$ into a small number of *clusters* (each partition of $\mathcal{S}$ into clusters is called a *clustering* of $\mathcal{S}$), and then consider each cluster as a single VPL when calculating its contribution to the shading at p.

The goal is to compute, for each $p \in \mathcal{P}$, a clustering that minimizes the shading error at p. This forces all algorithms to be *adaptive*, i.e., as one iterates over all the points to be shaded, the clustering has to be recomputed again with respect to the spatial and radiometric properties of each point (though various improvements are possible, at a loss of quality, by exploiting spatial coherence to re-use previous computation).

As the set $\mathcal{S}$ of VPLs is view-independent, an idea that improves efficiency is to compute the set of *all possible candidate* clusters of $\mathcal{S}$ *before* the rendering computation (and store in some hierarchical structure). Then during rendering, a clustering for each point is selected by choosing an appropriate subset of clusters from this pre-computed set. This is an expensive task which forces examining a large number of clusters (again, this can be somewhat ameliorated by exploiting the spatial coherence between neighboring pixels). This has been the basis of previous work; two well-known examples are Lightcuts [WFA+05] and LightSlice [OP11].

Our Contributions. We consider our main contributions to be two-fold. First, we make the next logical step in the many-lights clustering paradigm: rather than pre-computing a set of individual candidate clusters from which clusterings are computed during rendering, we pre-compute a number of clusterings of $\mathcal{S}$. Then, during the rendering phase, the clustering for a point p can simply taken to be one of these pre-computed clusterings, together with some minor modification. Our data structure stores the clusterings compactly, it is view-independent and it is computed prior to rendering. This results in a very efficient rendering phase. For a very natural criterion of clustering, we show that

- the total number of these pre-computed clusterings will be *independent* of the number of points in $\mathcal{P}$ to be shaded and will only depend on the size of $\mathcal{S}$.
- the modification required for each point will be provably small; in fact it will be independent of the size of $\mathcal{S}$ or $\mathcal{P}$.

Second, we develop the above framework into an accurate and efficient new many-lights clustering method. It computes a clustering relying on geometric and radiometric data for fast and accurate computation. We show that the complexity of our scheme is largely invariant on geometric scenes, and it is easily scalable with the number of VPLs.

We prove theoretical guarantees as well as experimentally validate the computational efficiency of our scheme, contrasting it with two of the most well-known earlier systems, Lightcuts [WFA+05] and LightSlice [OP11]. In particular, the advantages of our work include:

- Our pre-computation phase is view-independent, and so are the pre-computed clusterings. Unlike LightSlice and Lightcuts, this allows our algorithm to re-utilize computation with changing camera position.
- As all the clustering computations are moved to the pre-computation phase, the rendering phase takes constant time for each $p \in \mathcal{P}$, i.e., independent of the number of VPLs. On the other hand, Lightcuts has to maintain a heap and do clustering computations. Our method is able to produce similar output as Lightcuts with around 3 times average speedup.
- It outperforms LightSlice in speed and quality, achieving, e.g., 2 times speedup with consistently better quality. The errors of our algorithm are smooth, and visually difficult to detect, unlike for LightSlice which suffers from visible blocking effects.

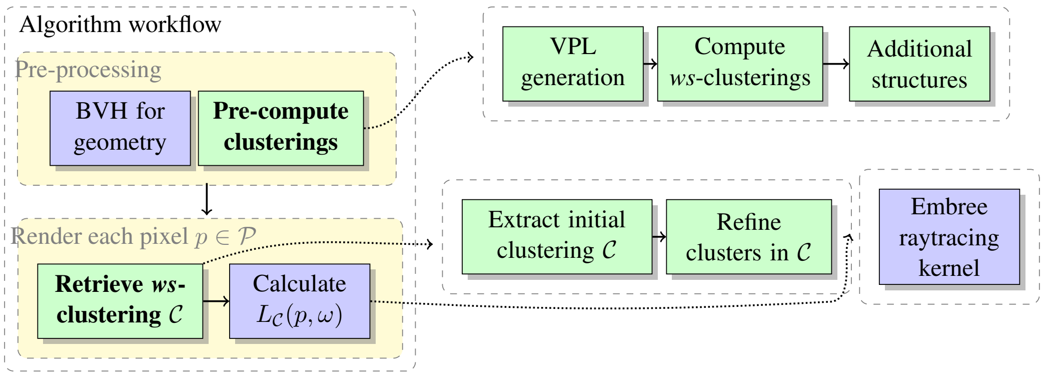

Algorithm workflow
Pre-processing
BVH for geometry
Pre-compute clusterings
Render each pixel $p \in \mathcal{P}$
Retrieve *ws*-clustering $\mathcal{C}$
Calculate $L_{\mathcal{C}}(p, \omega)$
VPL generation
Compute *ws*-clusterings
Additional structures
Extract initial clustering $\mathcal{C}$
Refine clusters in $\mathcal{C}$
Embree raytracing kernel

FIGURE 2.1: Overview of the system building blocks. The Embree framework (in purple) has been augmented by our algorithms (in green). Details of each block follow in Sections 2.2, 2.2 and 2.3.

- It also uses a significantly lower amount of memory than LightSlice which requires around 30 GB for scenes with around 0.5 million VPLs while our algorithm runs with 5 GB. This allows the usage of a considerably higher number of VPLs for rendering.

Broadly our work shows that the set of VPLs itself contain enough information such that with intensive preprocessing, a geometrically good clustering can be constructed for each point $p \in P$ with provably little effort.

Organization. We present the novel ideas involved in our approach in Section 2.2 along with theoretical details on our clustering representation and computation. Section 2.3 describes additional structures to further improve the clustering. Extensive experimental results and comparison with previous work are presented in Section 2.4. Finally, limitations are discussed in Section 2.5.

2.2 Algorithm

For a point $p \in \mathcal{P}$ to be shaded, consider a clustering of the set of VPLs $\mathcal{S}$ into k clusters $C_1, \ldots, C_k$ such that the radius of the smallest-ball containing each cluster is much smaller than the distance of that ball to p (this will be formulated precisely). Call such a cluster *well-separated*, or a *ws*-cluster for brevity, from p, and the clustering $\{C_1, \ldots, C_k\}$ a *well-separated clustering*, or a *ws*-clustering, for p. Intuitively, from the point of view of p, all the points in each cluster behave roughly like a single point. Figure 2.2 illustrates this for an arbitrary point p, shaded bright green, a few of the *ws*-clusters of $\mathcal{S}$ around it. Note that as the *ws*-cluster's distance to p decreases, the well-separated criterion automatically ensures that the radius of the cluster decreases as well. Our goal is to compute a *ws*-clustering of $\mathcal{S}$ for each point p to be shaded during the rendering phase.

Our approach has two main components: first we propose a method to pre-compute and compactly store several *ws*-clusterings of $\mathcal{S}$. Then during rendering, we show how to quickly extract a *ws*-clustering for each point to be shaded using this pre-computed structure. This pre-computed clustering will then be slightly modified to fit the spatial properties of the shaded point. We sketch the main components of the system in Figure 2.1, and outline them below.

Pre-Compute Clusterings The computation of *ws*-clustering for each point p is dependent on spatial properties of p, and requires individual computation during rendering for each point, an expensive task. Instead, we will do the following in a view-independent pre-rendering phase: compute a *ws*-clustering of $\mathcal{S}$ with respect to each light $s \in \mathcal{S}$. In other words, for each light s in $\mathcal{S}$, compute the partition of the remaining lights into clusters satisfying the *well-separatedness* criterion. The key to this construction will be the use of a partitioning data-structure, the *well-separated pair decomposition* [CK95] (henceforth denoted as WSPD). These *ws*-clusterings will be stored *implicitly* in a compact structure from which the *ws*-clustering for any light $s \in \mathcal{S}$ can be extracted quickly.

Well-separated clusters approximate the geometric terms of the rendering equation, but ignore visibility and material properties. To adjust for this, we will further compute two additional structures in this pre-processing phase. First, we further group the lights in each *ws*-cluster C into a small number of subgroups by similar light normals. This additional grouping will be used to evaluate the illumination from the cluster more precisely. Second, we introduce representative lights that approximate local visibility for each *ws*-cluster C as follows: sample a number of directions and compute the illumination of the lights in C reaching the boundary of the ball $b(C)$ in the sampled directions, where $b(C)$ is the smallest-ball containing C. This will be used to estimate the visibility of the lights in C to $b(C)$; the visibility test from $b(C)$ to p will be performed during the view-dependent rendering phase.

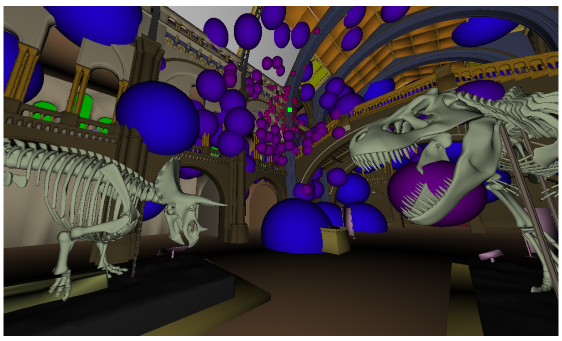


FIGURE 2.2: The *Museum* scene with the well-separated clusters (represented by their enclosing spheres) around a shaded point, shown as a bright green square in the middle of the image. For visualization purposes we only included a quarter of the clusters.

Retrieve ws-clustering During the rendering computation, for an arbitrary point p, find the closest point in $\mathcal{S}$ to p (an approximate nearest-neighbor is sufficient and will be used), and start with its (pre-computed) clusters as the clustering for p. Furthermore, refine each cluster by subdividing it into new clusters until they are well-separated from p. The number of clusters required to achieve this *ws*-clustering criteria could potentially be quite high, for two reasons: i) the clustering for the closest point in $\mathcal{S}$ to p could have many clusters, and ii) refinement could add many more new clusters to this initial clustering. It will be shown that refinement can only add a *constant* number of new clusters for any point p. This constant is provably *independent* of the number of lights in $\mathcal{S}$ or points in $\mathcal{P}$. Also, under some basic assumptions on the geometry of scenes, we will show that the average size of a *ws*-clustering for points of p will be logarithmic in the size of $\mathcal{S}$. Experimental evidence will confirm this behavior.

Calculate $L(p, \omega)$. Let $\mathcal{C}_p = \{C_1, \ldots, C_k\}$ be the final constructed *ws*-clustering for $p \in P$ during rendering. Furthermore let $\{C_i^1, \ldots, C_i^{k_i}\}$ be the k_i subgroups of each C_i by similar light normals. From a single subgroup C_i^j of $C_i \in \mathcal{C}_p$, we compute:

$$L_{C_i^j}(p,\omega) = M_{rep(C_i^j)}(p,\omega) \cdot G_{rep(C_i^j)}(p) \cdot \sum_{s \in C_i^j} I_s \tag{2.1}$$

Then we approximate the illumination at p from the cluster $C_i \in \mathcal{C}_p$ as:

$$L_{C_i}(p,\omega) = \left(\sum_{j=1}^{k_i} L_{C_i^j}(p,\omega) \right) \cdot V_p(\partial(b(C_i))) \cdot R(C_i, p) \tag{2.2}$$

where $V_p(\partial(b(C_i)))$ denotes the visibility from p to the boundary of the sphere $b(C_i)$ enclosing cluster C_i; this is computed by a shadow ray with the Embree raytracing kernel. $R(C_i, p)$ is the proportion of the summed intensity of the lights in C_i reaching the boundary of the ball $b(C_i)$ in the direction from the center of $b(C_i)$ to p; see Section 2.3 for its precise definition and computation.

Constructing *ws*-clusterings

For a cluster $C \subseteq \mathcal{S}$, define $b(C)$ to be the smallest-enclosing ball of the points in C. Let $r(C)$ be the radius of the ball $b(C)$. For any point p, $d(p, C)$ denotes the Euclidean distance of p to $b(C)$.

A well-separated cluster. We introduce a necessary condition that each cluster must satisfy when constructing a clustering of $\mathcal{S}$ w.r.t. a shaded point p – namely that it is *well-separated* from p, for a given parameter $0 \leq \epsilon \leq 1$ (ϵ will be called the *separation parameter*).

Definition 2.1. A cluster of lights C is well-separated from a point p if $r(C) < \epsilon \cdot d(p, C)$, where ϵ is the separation parameter.

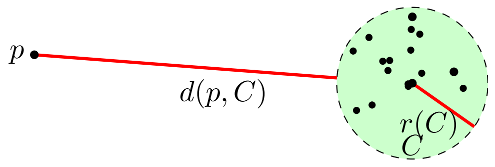


FIGURE 2.3: For $\epsilon = 0.5$, C is *ws* from p. In other words, we have $d(p, C) > 2 \cdot r(C)$.

The lights in a *ws*-cluster w.r.t. p are 'far enough' from p, and concentrated in a small ball (see Figure 2.3). This condition implies that from the point of view of p, all the lights in a *ws*-cluster are in a similar direction and the distances of p to the lights in C are approximately the same. Since the luminosity reaching p depends on the angle and the distance of lights in $\mathcal{S}$ from p (differences regarding visibility and light normals will be accounted for later in Section 2.3) it can be argued that treating all the lights in a *ws*-cluster as one point does not introduce significant error. This intuition is captured in the following theorem:

Theorem 2.2. *For a point p and a ws-cluster $C \subseteq \mathcal{S}$, assume that all lights in C face in the same direction and they have the same visibility from p. Then the error from representing C with any one light in C (which has the cumulative intensity summed over all the lights in C) is bounded by a function depending only on ϵ. In case the point to be shaded has Lambertian BRDF, it is*

$$\left|L(p, \omega) - L_C(p, \omega)\right| = O(\epsilon) \sum_{s \in C} \frac{I_s}{d(p, s)^2} \tag{2.3}$$

where $L(p, \omega)$ denotes the exact illumination from lights in C.

Proof. First we show ϵ-dependent bounds of the changes of angles and distances between p and the lights in a *ws*-cluster C. See Figure 2.4 for the interaction of p with two lights s, r in C where r denotes the light chosen as the representative of the cluster.

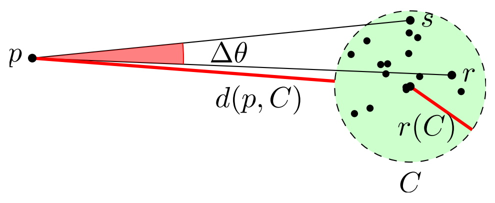


FIGURE 2.4: Changes in the angles and distances are bounded by a function of ϵ.

Estimating the change in the distances follows immediately from the *ws* property:

$$d(p, s) \le d(p, r) + 2r(C) \le d(p, r) + 2\epsilon d(p, s) \tag{2.4}$$

For bounding the change of angles $\Delta\theta$, it is clear that the angle between the two tangents of the circle from p is the upper bound. The center of the circle, p and the point on tangency form a right angle triangle:

$$\sin\frac{\Delta\theta}{2} = \frac{r(C)}{d(p,C)+r(C)} \leq \frac{\epsilon d(p,C)}{d(p,C)} \leq \epsilon \tag{2.5}$$

Hence using the approximation for the $sine$ function with Taylor series we have $\Delta\theta = O(\epsilon)$.

In the following we approximate the error for one cluster using the previous results. We assume that the visibility is always 1 (the case of 0 is trivial). Therefore the error is:

$$\begin{aligned} \big|L(p,\omega) - L_C(p,\omega)\big| &= \qquad (2.6)\\ &= |\sum_{s\in C} V_s(p) I_s M_s(p,\omega) G_s(p) \\ &\quad - V_r(p) M_r(p,\omega) G_r(p) \sum_{s\in C} I_s| \qquad (2.7)\\ &= \sum_{s\in C} I_s \big|M_s(p,\omega)G_s(p) - M_r(p,\omega)G_r(p)\big| \qquad (2.8)\end{aligned}$$

The above formula has the important property that the M_s and G_s functions are dependent on the cosines of angles and distances which are closely bounded because of the *ws* property. This result intuitively means that the error cannot be too big for a BRDF that relies on distances and angles. In the case of the diffuse BRDF, $M_s(p,\omega) = k_d(p)\cos\theta_s$ where $k_d(p)$ is the diffuse reflection coefficient and θ_s is the angle between the surface normal at p and $s - p$. Denote by ϕ_s the angle between the light normal and $p - s$ and for brevity denote $d(p,s)$ by r_s, then the formula becomes:

$$\sum_{s\in C} I_s \left| \frac{k_d(p)\cos\theta_s\cos\phi_s}{r_s^2} - \frac{k_d(p)\cos\theta_r\cos\phi_r}{r_r^2} \right| = \tag{2.9}$$

$$k_d(p)\sum_{s\in C} I_s \left| \frac{\cos\theta_s\cos\phi_s}{r_s^2} - \frac{\cos(\theta_s+\Delta\theta_s)\cos(\phi_s+\Delta\phi_s)}{(r_s+\Delta r_s)^2} \right| \tag{2.10}$$

We omit a complete analysis and just minimize the subtrahend assuming that $\Delta\theta, \Delta\phi, \Delta r \geq 0$ (the other cases are similar). Using the *ws* property and that $\cos(\theta+\Delta\theta) \geq \cos\theta - \Delta\theta$,

$$\leq k_d(p)\sum_{s\in C} I_s \left(\frac{\cos\theta_s\cos\phi_s}{r_s^2} - \frac{(\cos\theta_s-\Delta\theta_s)(\cos\phi_s-\Delta\phi_s)}{(1+2\epsilon)^2 r_s^2} \right) \tag{2.11}$$

Using that $\cos x \leq 1$ and the bound on $\Delta\theta$

$$\leq k_d(p)\sum_{s\in C} I_s \left(\frac{(4\epsilon+4\epsilon^2+\Delta\theta_s+\Delta\phi_s-\Delta\theta_s\Delta\phi_s)}{(1+2\epsilon)^2 r_s^2} \right) \tag{2.12}$$

$$= O(\epsilon)k_d(p)\sum_{s\in C}\frac{I_s}{r_s^2} \tag{2.13}$$

In other words, the error is proportional to the complete intensity received by a pixel. Note that this does not account for visibility difference of course, which we address with other methods.

□

Remark: In the case of Phong BRDF the $M_s(p,\omega)$ function becomes $k_s(p)cos^n\beta_s\cos\theta_s$ where β_s is the angle between ω and the direction of $s-p$ reflected on the surface normal (the subscript of k now refers to the word specular). In this case the above calculations are as follows:

$$\sum_{s\in C} I_s\left|\frac{k_s(p)\cos^n\beta_s\cos\theta_s\cos\phi_s}{r_s^2}-\frac{k_s(p)\cos^n\beta_r\cos\theta_r\cos\phi_r}{r_r^2}\right| \tag{2.14}$$

$$\begin{aligned}= k_s(p)\sum_{s\in C} I_s\Bigg|&\frac{\cos^n\beta_s\cos\theta_s\cos\phi_s}{r_s^2}\\ &-\frac{\cos^n(\beta_s+\Delta\beta_s)\cos(\theta_s+\Delta\theta_s)\cos(\phi_s+\Delta\phi_s)}{(r_s+\Delta r_s)^2}\Bigg|\end{aligned} \tag{2.15}$$

$$\begin{aligned}\le k_s(p)\sum_{s\in C} I_s\Bigg(&\frac{\cos^n\beta_s\cos\theta_s\cos\phi_s}{r_s^2}\\ &-\frac{(\cos\beta_s-\Delta\beta_s)^n(\cos\theta_s-\Delta\theta_s)(\cos\phi_s-\Delta\phi_s)}{(1+2\epsilon)^2r_s^2}\Bigg)\end{aligned} \tag{2.16}$$

One could upper bound this error but the subtrahend converges to 0 as $n\to\infty$ hence in this case one could only give a weak bound which depends on n as well. If we were to set ϵ according to such a bound one would have to build a too fine WSPD which would result in a prohibitively big runtime. We remark that this could be possibly overcome by refining the WSPD with a smaller epsilon during the rendering phase but only for those clusters that have a high value for $M_r(p,\omega)$. However we have not experimented with this approach.

To compute *ws*-clusterings efficiently, we will need to use a basic structure in the theory of geometric computing, the well-separated pair decomposition.

Well-separated pair decompositions. We first need to extend the notion of well-separatedness between a point and a cluster to that of between two clusters. Two point sets R and Q are *well-separated* from each other if, for a given separation parameter $\epsilon > 0$, the radius of both the balls $b(R)$ and $b(Q)$ is smaller than $\epsilon\cdot d(R,Q)$, i.e., $\max(r(R),r(Q)) < \epsilon\cdot d(R,Q)$ where $d(R,Q)$ is the distance between $b(R)$ and $b(Q)$.

Definition 2.3. A well-separated pair decomposition of $\mathcal{S}$ for a given separator parameter ϵ is a list of pairs of clusters $\{\{R_1,Q_1\},\ldots,\{R_s,Q_s\}\}$, where each $R_i,Q_i\subseteq\mathcal{S}$, and *i)* for every pair of points $p, q \in \mathcal{S}$, there is a unique index i such that $p \in R_i$ and $q \in Q_i$, and

ii) for all $i = 1 \dots s$, the clusters R_i and Q_i are well-separated from each other, with separation parameter ϵ.

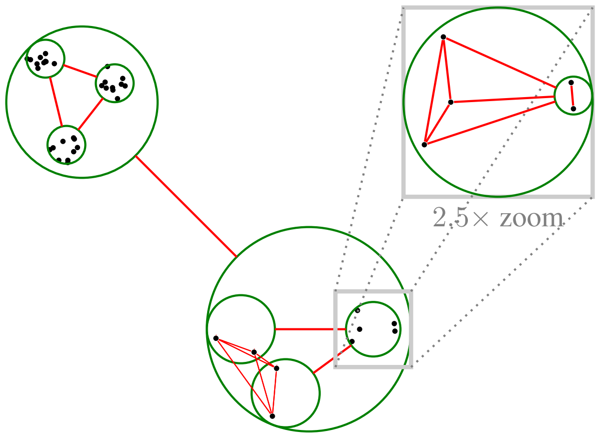


FIGURE 2.5: Each red edge represents a pair $\{R_i, Q_i\}$, where the sets R_i, Q_i are enclosed in green circles.

Here s is called the size of the WSPD. See Figure 2.5 for some example pairs $\{R_i, Q_i\}$ for a point set in two dimensions. A remarkable fact about WSPDs is that there always exist WSPDs of size *linear* in $|\mathcal{S}|$. In particular, for any $\epsilon > 0$, and any set of points $\mathcal{S} \subset \mathbb{R}^d$, there exists a WSPD of size $O(|\mathcal{S}|\epsilon^{-d})$ that can be computed in time $O(|\mathcal{S}| \log |\mathcal{S}| + |\mathcal{S}|\epsilon^{-d})$ [CK95] .

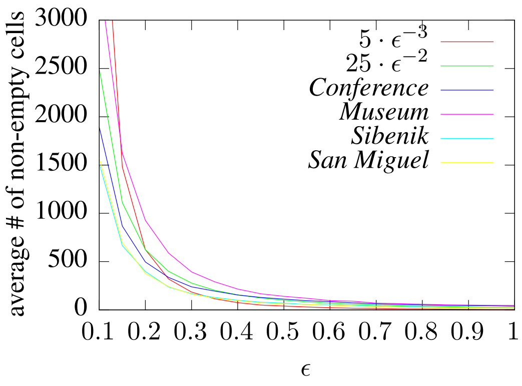


FIGURE 2.6: A geometric condition.

Recall that the set of VPLs $\mathcal{S}$ was placed on the surface of objects by tracing random light particles from the light sources. The geometry of graphics scenes is usually such that it satisfies a technical geometric condition. Namely, that if the three-dimensional space was partitioned into equal-sized cubes of size δ, the boundary of scene objects would intersect on average $O(\epsilon^{-2})$ cubes in a ball of radius δ/ϵ (note that there are $\Theta(\frac{(\delta/\epsilon)^3}{\delta^3}) = \Theta(\epsilon^{-3})$ cubes in a ball of radius δ/ϵ). Figure 2.6 shows that, for a variety of scenes, the average number of cells containing VPLs within a ball of radius δ/ϵ in our octree behaves more like $O(\epsilon^{-2})$ than $O(\epsilon^{-3})$. This

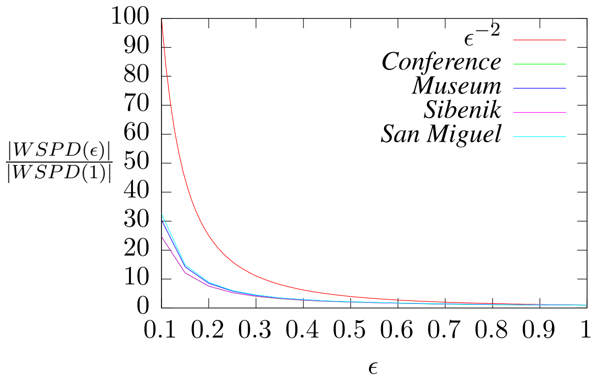


FIGURE 2.7: WSPD size ratios for some graphical scenes.

implies better than cubic dependency on $1/\epsilon$ of WSPD sizes in three dimensions, since the proof [CK95] uses a trivial cubic upper bound for non-empty cubes of size δ within a ball of radius δ/ϵ. Experimental results confirm this: Figure 2.7 plots, for several scenes, the ratio of the WSPD size for varying separation parameter ϵ to WSPD size for separation parameter 1. Observe that the behavior of the WSPD is relatively unchanged from one scene to the next.

Constructing well-separated pair decomposition of $\mathcal{S}$. We use a compressed octree of $\mathcal{S}$ as an underlying data structure to compute the initial *ws*-clusters for $\mathcal{S}$, to find approximate nearest neighbors, and for the local refinement for each point p during the rendering computation. The compressed octree is an octree where the non-branching paths are contracted into one edge. The compressed octree can be directly computed in linear time [Sam95]. Each node corresponds to an axis-aligned bounding box. We associate with each node the set of VPLs of $\mathcal{S}$ contained in its bounding box. Note that each leaf of the octree contains exactly one unique point of $\mathcal{S}$. For a node w, denote by R_w the corresponding set of VPLs. Note that the height of the octree is linear in the worst-case, though in practice it is logarithmic. In Table 2.1 we show the depth of the tree for 300K VPLs, which is logarithmic for a variety of scenes.

	Tree depth	
Scene	**Octree**	**Compressed**
Conference	19	14
Sibenik	20	14
Museum	18	15
San Miguel	22	14

TABLE 2.1: Octree depths with $320K$ VPLs, both with and without compression.

After constructing the compressed octree, Algorithm 1 computes the WSPD of $\mathcal{S}$ by utilizing a top down search on the tree for *ws*-pairs. Two nodes w, v of the octree will form a *ws*-pair if the corresponding sets R_w, R_v are well-separated. Note that instead of simply storing all the pairs of the WSPD as a list, we directly store the WSPD structure in the octree by storing, for

Algorithm 1: Create WSPD for the set of points $\mathcal{S}$

```
Function CreateWSPD($\mathcal{S}$)
    $root \leftarrow$ create compressed octree on $\mathcal{S}$
    $S$: stack of pairs, $S$.pushback($\{root, root\}$)
    while notEmpty($S$) do
        $\{w, v\}$ = $S$.pop()
        if isWellSeparated($R_w, R_v$) then
            Insert $v$ into $pairs(w)$
            Insert $w$ into $pairs(v)$
        else
            if $r(R_w) < r(R_v)$ then
                Swap($w, v$)
            foreach $i \in$ children($w$) do
                $S$.pushback($\{i, v\}$)
```

each node w of the octree, a list of pointers to all the other nodes with which w forms *ws*-pairs in the WSPD. We denote this list by $pairs(w)$. In Figure 2.8 we show a simple example for 2 dimensional data with the red links denoting the pairs of sets. The construction ensures a hierarchical structure on the clusters which enables us to easily refine a cluster if needed by simply descending in the octree.

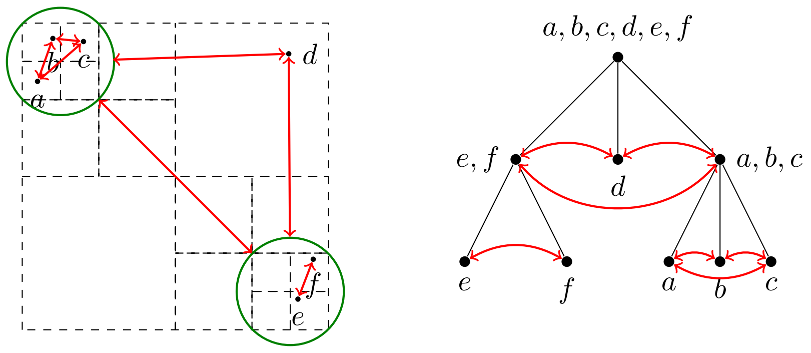


FIGURE 2.8: The compressed octree and the WSPD for a set of points.

Pre-computing *ws*-clusterings of $\mathcal{S}$. From the WSPD of $\mathcal{S}$ one can compute a set of clusterings $\{\mathcal{C}_s, s \in \mathcal{S}\}$, where $\mathcal{C}_s$ will be a *ws*-clustering of $\mathcal{S}$ for the point $s \in \mathcal{S}$. Namely, for each $s \in \mathcal{S}$, $\mathcal{C}_s$ is a partition of the lights in $\mathcal{S} \setminus \{s\}$ into a number of clusters, each of which is well-separated from s. Note that if the two sets $\{R, Q\}$ are well-separated, then for every point $p \in R$, Q is a *ws*-cluster with respect to p. The definition of WSPD ensures that for every pair of points p_1 and p_2, there is a unique $\{R, Q\}$ such that $p_1 \in R$ and $p_2 \in Q$. Therefore $\mathcal{C}_s = \{Q \mid \{R, Q\}$ is *ws*, $s \in R\}$ is a *ws*-clustering of $\mathcal{S}$ for s. If the WSPD has been computed, then a *ws*-clustering for each point $s \in \mathcal{S}$ can be extracted from it efficiently, as follows. Consider the leaf node of the compressed octree corresponding to s. Any node w of the octree on the path from this leaf to the root has $s \in R_w$, and so the *ws*-clustering of s is simply the union of

Algorithm 2: Constructing *ws*-clusters $\mathcal{C}_p$ for $p \in \mathcal{P}$.

```
Function ConstructWSPDClustering($p$)
    $\mathcal{C}_p \leftarrow \emptyset$
    $s \leftarrow$ closest point in $\mathcal{S}$ to $p$
    $\mathcal{C}_s \leftarrow$ ws-clustering of $s$
    foreach $Q \in \mathcal{C}_s$ do
        if $d(s,Q) \geq \frac{d(p,s)}{\epsilon}$ then
            add $Q$ to $\mathcal{C}_p$
        else
            refine $Q$ into subclusters that are ws from $p$
            add the resulting clusters to $\mathcal{C}_p$
    return $\mathcal{C}_p$
```

$pairs(w)$ for all such nodes w, and can be computed by traversing the octree from this leaf to the root.

Computing *ws*-clusterings of P during rendering. We now show how to use the clusterings $\mathcal{C}_s$, pre-computed for each $s \in \mathcal{S}$ before the rendering phase, to quickly compute a *ws*-clustering of $\mathcal{S}$ with respect to any point $p \in \mathcal{P}$.

Consider the case for an arbitrary point $p \in \mathcal{P}$. Compute the closest light, say $s \in \mathcal{S}$, to p. One could use a variety of known optimal algorithms, but for us an approximation will suffice. We find the smallest node of the compressed octree containing p and return an arbitrary light contained in it. A calculation shows that for randomly shifted octrees, the expected distance from the true nearest neighbor is bounded. Point location in a compressed octree takes $O(\log n)$ time with some additional data structures, but for us the naive implementation suffices as the tree has logarithmic depth (see Table 2.1).

Say s is at distance d from p. Take the *ws*-clustering $\mathcal{C}_s$ of s. These were pre-computed, and can be efficiently retrieved. The key idea now is to consider two types of clusters in $\mathcal{C}_s$ separately: *far* clusters in $\mathcal{C}_s$ are at distance further than d/ϵ from s and *close* clusters are those closer than d/ϵ from s. We show that each far cluster in $\mathcal{C}_s$ is an approximately *ws*-cluster from p. For the remaining close clusters in $\mathcal{C}_s$, we recursively subdivide them until they are *ws*-clusters from p. The subdivision uses the same octree that was used for the construction of the WSPD. See Algorithm 2. Note that the above algorithm is *adaptive* to the local geometry of the scene: for a point p closely surrounded by VPLs, it will refine at a smaller radius.

We now prove that the clusters far from s are approximately *ws*-clusters from p.

Lemma 2.4. *Let p be an arbitrary point and s be its nearest-neighbor with $d(p,s) = d$. Any ws-cluster $C_o \in \mathcal{C}_s$ disjoint with the ball of radius $\frac{d}{\epsilon}$ around s is approximately well-separated from p, namely it is well-separated with separation parameter $\epsilon' = \frac{\epsilon}{1-\epsilon}$.*

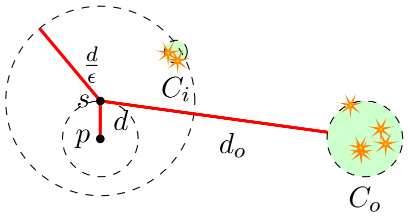


FIGURE 2.9: A close *ws*-cluster C_i and a far *ws*-cluster C_o from s.

Proof. See Figure 2.9. As C_o is *ws* from s, it follows that $\epsilon d_0 \geq r(C_o)$, where $r(C_o)$ is the radius of the cluster C_o. Also $d \leq \epsilon d_o$ since the cluster is disjoint. Triangle inequality implies $d(s, C_o) = d_o \leq d + d(p, C_o)$, and so $d(p, C_o) \geq d_o - d \geq d_o(1-\epsilon) \geq \frac{r(C_o)(1-\epsilon)}{\epsilon}$. Thus for the slightly larger value of the separation parameter $\epsilon' = \frac{\epsilon}{1-\epsilon}$, C_o is *ws* from p. □

We next prove that the additional number of clusters added in the refinement of a cluster close to s is low.

Theorem 2.5. *Let p be an arbitrary point and s be its nearest neighbor with $d(p,s) = d$. After refining the clusters in the set $\mathcal{C}^* \subseteq \mathcal{C}_s$ which intersect the ball with radius $\frac{d}{\epsilon}$ around s, there are at most $O(\frac{1}{\epsilon^6})$ new ws-clusters $\mathcal{C}'$ created. The resulting set $\mathcal{C}_s \backslash \mathcal{C}^* \cup \mathcal{C}'$ is a partition of $\mathcal{S}$ into ws-clusters around p, with separation parameter $\epsilon/(1-\epsilon)$.*

Proof. The second statement of the proof comes from Lemma 2.4. Take a cluster $C_i \in \mathcal{C}_s$ lying inside the ball of radius d/ϵ around s. If it is not *ws*-separated from p, partition the bounding-box of C_i into 8 equal-volume bounding boxes (the children of the node in the octree), and recursively check the *ws*-separated property of these new refined clusters with p. Eventually when a newly refined cluster is finally a *ws*-cluster from p, add it to $\mathcal{C}'$. To count the total number of new *ws*-clusters added to $\mathcal{C}'$, consider a cluster $C \in \mathcal{C}'$. It exists because its parent, say cluster D, was not a *ws*-cluster with p. i.e., $r(D) > \epsilon d_p$, where d_p is the distance of the ball of D to p. Because s is the nearest neighbor of p we know that no other point is in the small ball of radius d around p; in particular it cannot completely contain the ball of D and hence $r(D)/\epsilon \geq d_p \geq d - 2r(D)$, which implies that $r(D) \geq \frac{d}{2+\epsilon^{-1}}$. So each cluster added to $\mathcal{C}'$ has a parent with radius at least the above value. Grouping the parents by size (higher level parents are the same size but multiplied by some power of two) we can give an upper bound on their number by a simple packing argument since parents with the same size are disjoint (since octree nodes can either contain each other or be disjoint):

$$\sum_{i=0}^{\infty} \left(\frac{d/\epsilon}{2^i (\frac{d}{2+\epsilon^{-1}})} \right)^3 = O(\frac{1}{\epsilon^6}) \tag{2.17}$$

Since we have bounded the number of parents and each of them can have at most 8 children, this finishes the proof. □

While the above theorem may be surprising at first glance, we hope the following intuition sheds some light: consider the distance of the closest point $s \in \mathcal{S}$ to p. If this distance is small, then it is not hard to argue that the clusters for the closest light provide a good approximation of the clustering for p, and so little refinement is necessary. On the other hand, if this closest distance is large, then all points of $\mathcal{S}$ are 'far' from p, and so any *ws*-cluster from s is far from p and thus approximately well-separated from p; again little refinement is needed to approximate the separation (and thus illumination). This intuition is formalized in the proof of the theorem. For empirical validation, see Table 2.2 for the maximum number of new clusters added per point for a number of scenes with varying values of ϵ.

	Refined clusters for varying ϵ				
Scene	0.9	0.7	0.5	0.3	0.1
Conference	58	75	114	227	1401
Sibenik	57	74	109	219	1505
Museum	81	103	159	326	2350
San Miguel	80	108	167	370	3411

TABLE 2.2: Maximum number, over all $p \in P$, of added clusters during refinement with $320K$ VPLs.

Cluster sizes. We have proved that the clustering stored in the WSPD can be used to retrieve clustering for arbitrary points without increasing the number of clusters more than an additive constant. It remains to argue that the initial clustering for every light $s \in \mathcal{S}$ is compact. While one can construct examples of arbitrary points where the average number of *ws*-clusters for a point p is linear (as a function of $|\mathcal{S}|$), those are never realized in practice for the set of lights arising in geometric scenes. The spatial partitioning structures (octrees) turn out to be roughly balanced, and so the number of *ws*-clusters is logarithmic.

2.3 Additional Structures for Illumination Computation

We enhance the purely geometric WSPD based clustering with the following additional structures that improve the efficiency and accuracy in calculating illumination. Recall that given a *ws*-clustering $\mathcal{C}_p$ for p, for a *ws*-cluster $C_i \in \mathcal{C}_p$, the approximation of the radiance with a single representative has the form:

$$M_{rep(C_i)}(p, \omega) \cdot G_{rep(C_i)}(p) \cdot V_{rep(C_i)}(p) \cdot \sum_{s \in C_i} I_s. \tag{2.18}$$

Clustering refinement by direction. The WSPD data structure is able to efficiently bound angles and distances between points. However, the normals of the lights could vary widely in directions. To overcome this difficulty, the lights in each *ws*-cluster are grouped into a few subgroups with similar normals. For a *ws*-cluster C_i, construct the subgroups $C_i^1, \ldots, C_i^{k_i}$, where all the lights in each C_i^j, $j = 1 \ldots k_i$, will have similar normal directions. The approximation then becomes

$$L_{C_i}(p, \omega) = \left(\sum_{j=1}^{k_i} L_{C_i^j}(p, \omega) \right) \cdot V_{rep(C_i)}(p) \tag{2.19}$$

where $L_{C_i^j}(p, \omega)$ is defined in Equation 2.1. The subgroups are constructed by first picking a center $c^j \in C_i$ for each subgroup C_i^j, and then assigning each VPL $s \in C_j$ to the subgroups with most similar center. see Algorithm 3.

Algorithm 3: Computing subgroups of a given cluster C_i

```
1 Function ClusterNormals($C_i$)
2     $j \leftarrow 1$; $c^1 \leftarrow$ random light in $C_i$
3     $largestDistance \leftarrow \infty$
4     while $largestDistance \geq threshold$ do
5         For each $s \in C_i$, $d_s \leftarrow \min_{k \leq j} d(s, c^k)$
6         $c^{j+1} \leftarrow q$, where $q$ has largest $d_q$ value
7         $j \leftarrow j + 1$
8         $largestDistance \leftarrow d_q$
9     For each $l$, $C_i^l \leftarrow \{s \in C_i | l = \arg\min_{k<j} d(s, c^k)\}$
```

During the shading of a point $p \in \mathcal{P}$, as before, visibility test will still be performed once for each *ws*-cluster of p. However, the radiance $L_{C_i}(p, \omega)$ for a *ws*-cluster C_i will be calculated by summing up the radiance contributions separately for each subgroup C_i^j of C_i, using the normal of the center light for each subgroup. The distance $d(s, c^j)$ used in the algorithm is the Euclidean distance with $threshold$ set to 0.01, which did not result in too many subgroups on average (see Table 2.3). The number of subgroups is slightly higher for more complex scenes and decreases with ϵ since the clustering becomes more fine.

Scene	**Average number of subgroups**			
ϵ	0.9	0.7	0.5	0.3
Conference	8.3	8.1	6.7	5.3
Sibenik	9.5	9.3	8.0	5.9
Museum	23.3	21.2	16.8	11.4
San Miguel	44.72	39.6	31.2	18.5

TABLE 2.3: Average number of subgroups per cluster.

Visibility testing. Visibility differences within a cluster can cause errors with a fixed representative light. Consider for example a flat object with VPLs on both sides. Since the representative light is either on one side or the other, visibility queries falsely return occlusion if the pixel is on the other side of the object. To overcome this problem we propose to augment our *ws*-clusters with additional visibility information.

As stated in Theorem 2.2, without taking into account visibility differences, a *ws*-clustering with a single representative and clustered normals gives a good approximation to L (as a function of ϵ). The visibility computation for a *ws*-cluster C_i will be divided into two parts: a simple shadow test from p to the boundary of the ball of C_i for the outside visibility and then shadow testing from the boundary to each light for visibility inside the ball of C_i. Here again the geometric well-separated property of the *ws*-clustering comes in useful, as the angles from p to C_i are bounded (as a function of ϵ). We use a new approximation for $L(p, \omega)$, to better handle visibility:

$$L_{C_i}(p,\omega) = \left(\sum_{j=1}^{k_i} L_{C_i^j}(p,\omega) \right) \cdot V_p(\partial(b(C_i))) \cdot R(C_i, p) \tag{2.20}$$

$R(C_i, p)$ is the proportion of the summed intensity of the lights in C_i reaching the boundary of the ball $b(C_i)$ in the direction of p from the center of $b(C_i)$.

$$R(C_i, p) = \frac{\sum_{s \in C_i} V_p(s, \partial(b(C_i))) \cdot I_s \cdot \cos\phi_s}{\sum_{s \in C_i} I_s \cdot \cos\phi_s} \tag{2.21}$$

where $V_p(s, \partial(b(C_i)))$ denotes the visibility from s to the boundary of the sphere in the direction from the center of $b(C_i)$ to the shaded point p and ϕ_s is the angle between the same direction and the normal of the light. Note that as $\epsilon \to 0$, equation 2.20 converges to $L(p, \omega)$. Since computing $R(C_i, p)$ during rendering would be expensive, we do the following in the pre-processing phase. For each cluster C_i and for a small uniform set of directions on $\partial b(C_i)$, $R(C_i, p)$ is pre-computed and stored in a cubemap (with a resolution of 6×6 on each side). This enables quick lookup of $R(C_i, p)$ for p. During the rendering of a point p, nearest-neighbor interpolation on the cubemap yields $R(C_i, p)$. A shadow test to $\partial b(C_i)$ gives the V_p term.

High intensity clusters. To further minimize the error coming from a badly chosen representative for high intensity clusters we limit the radiance from each cluster (by further refining the cluster if necessary) to be less than 1% of the radiance received by a pixel. This refinement happens at the pre-processing phase only using approximate intensities between the cluster pairs.

2.4 Results and Discussion

In this section we present the experimental results on several scenes of varying complexity. Timings are for a workstation equipped with two Xeon X5570 processors each with 4 cores

		Museum	Sibenik	Conference	San Miguel
Scenes	Triangles	1.5M	0.07M	0.33M	10.5M
	Resolution	1024^2@1	1024^2@1	1024^2@1	1024^2@1
	VPLs	472K	540K	516K	400K
Lightcuts(1%) max cut: ∞	Preproc. time (s)	0.38	0.47	0.44	0.29
	Render time (s)	**515.06**	**216.29**	**219.43**	**1583.63**
	Avg # of rays	1872	1399	957	3226
	RMSE	**0.004674**	**0.003094**	**0.002220**	**0.005828**
	LMSE	0.000735	0.001124	0.003228	0.003134
	Rel. Error(%)	1.123480	1.094558	1.273182	2.617376
	Speedup	1.0	1.0	1.0	1.0
Lightcuts (2%) max cut: 2000	Preproc. time (s)	0.38	0.47	0.43	0.29
	Render time (s)	218.81	103.31	125.98	306.39
	Avg # of rays	884	724	556	714
	RMSE	0.006223	0.004401	0.003030	0.021054
	LMSE	0.001155	0.001599	0.002794	0.019569
	Rel. Error(%)	1.727831	1.860882	1.689145	8.528274
	Speedup	2.3	2.1	1.8	5.2
Lightslice	Preproc. time (s)	0.02	0.02	0.02	0.01
	Render time (s)	208.17	225.12	202.44	106.94
	Avg # of rays	665	795	750	463
	RMSE	0.009606	0.006820	0.005657	0.034085
	LMSE	0.009220	0.005171	0.009392	0.236342
	Rel. Error(%)	3.273127	3.342225	4.164747	13.855263
	Speedup	2.5	1.0	1.1	14.8
WSPD 0.7	Preproc. time (s)	52.00	50.01	40.52	35.10
	Render time (s)	62.93	31.10	35.24	86.91
	Avg # of rays	711	532	642	715
	RMSE	0.007049	0.004366	0.003515	0.014771
	LMSE	0.002025	0.002145	0.003384	0.028564
	Rel. Error(%)	2.416694	2.148956	2.592963	7.539747
	Speedup	8.2	7.0	6.2	18.2
WSPD 0.5	Preproc. time (s)	55.88	52.86	44.47	38.73
	Render time (s)	86.72	37.85	42.33	115.87
	Avg # of rays	901	635	665	915
	RMSE	0.005750	0.004254	0.002858	0.013161
	LMSE	0.001642	0.002147	0.003961	0.023084
	Rel. Error(%)	1.923812	1.826188	2.180026	6.760511
	Speedup	5.9	5.7	5.2	13.7
WSPD ϵ (equal RMSE compared to Lightcuts(1%))	Preproc. time (s)	76.86	64.31	60.41	232.91
	Render time (s)	**190.27**	**68.00**	**80.28**	**950.88**
	Avg # of rays	1913	1183	1318	6972
	RMSE	**0.004652**	**0.002833**	**0.002219**	**0.005864**
	LMSE	0.001040	0.001822	0.002624	0.004080
	Rel. Error(%)	1.439894	1.261353	1.597909	2.886980
	Speedup	2.7	3.2	2.7	1.6
	ϵ	0.25	0.25	0.25	0.09

TABLE 2.4: Rendering statistics for the three different methods, Lightcuts, LightSlice and the WSPD algorithm.

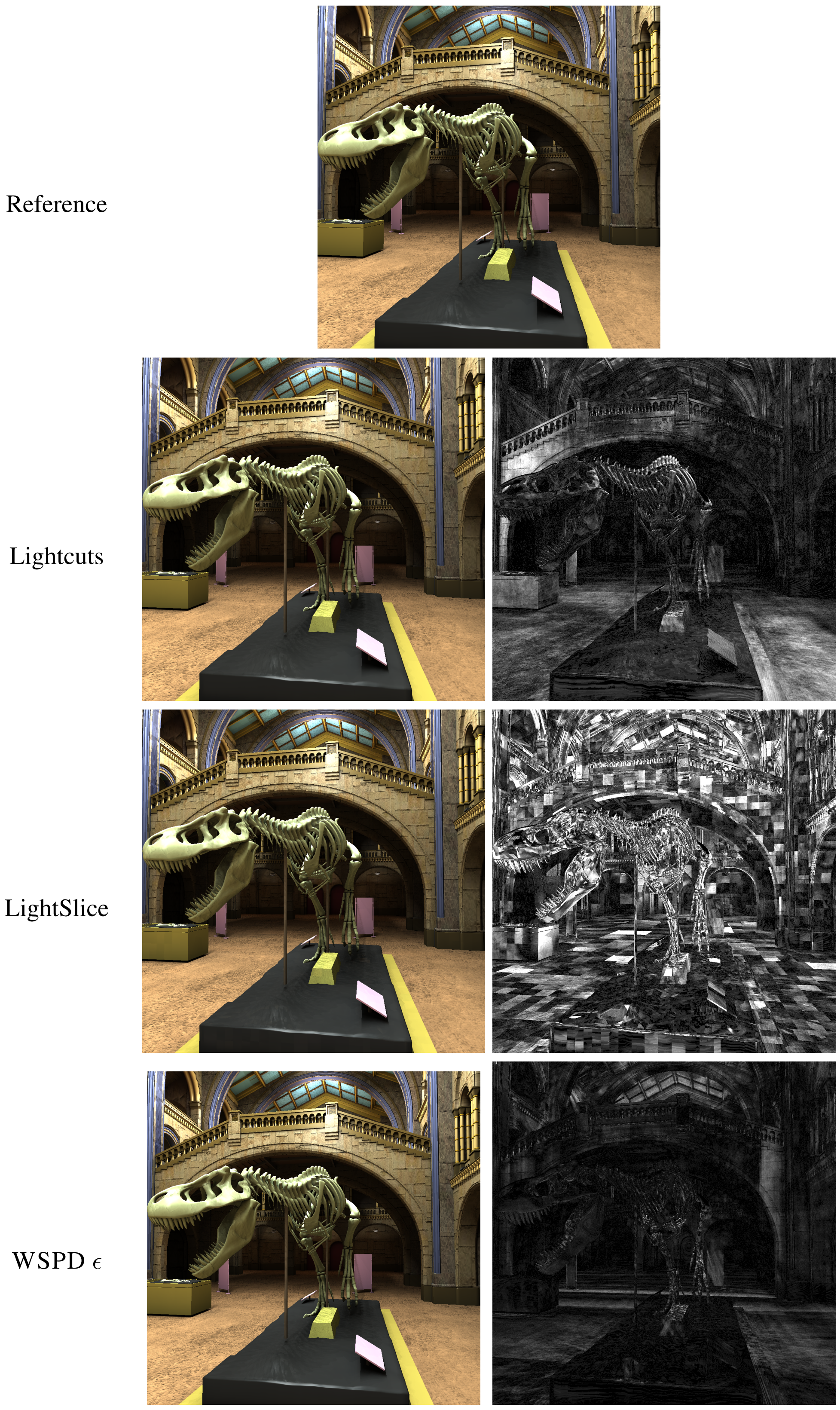


FIGURE 2.10: Museum scene rendered with the three different methods, Lightcuts, LightSlice and the WSPD algorithm together with the error images.

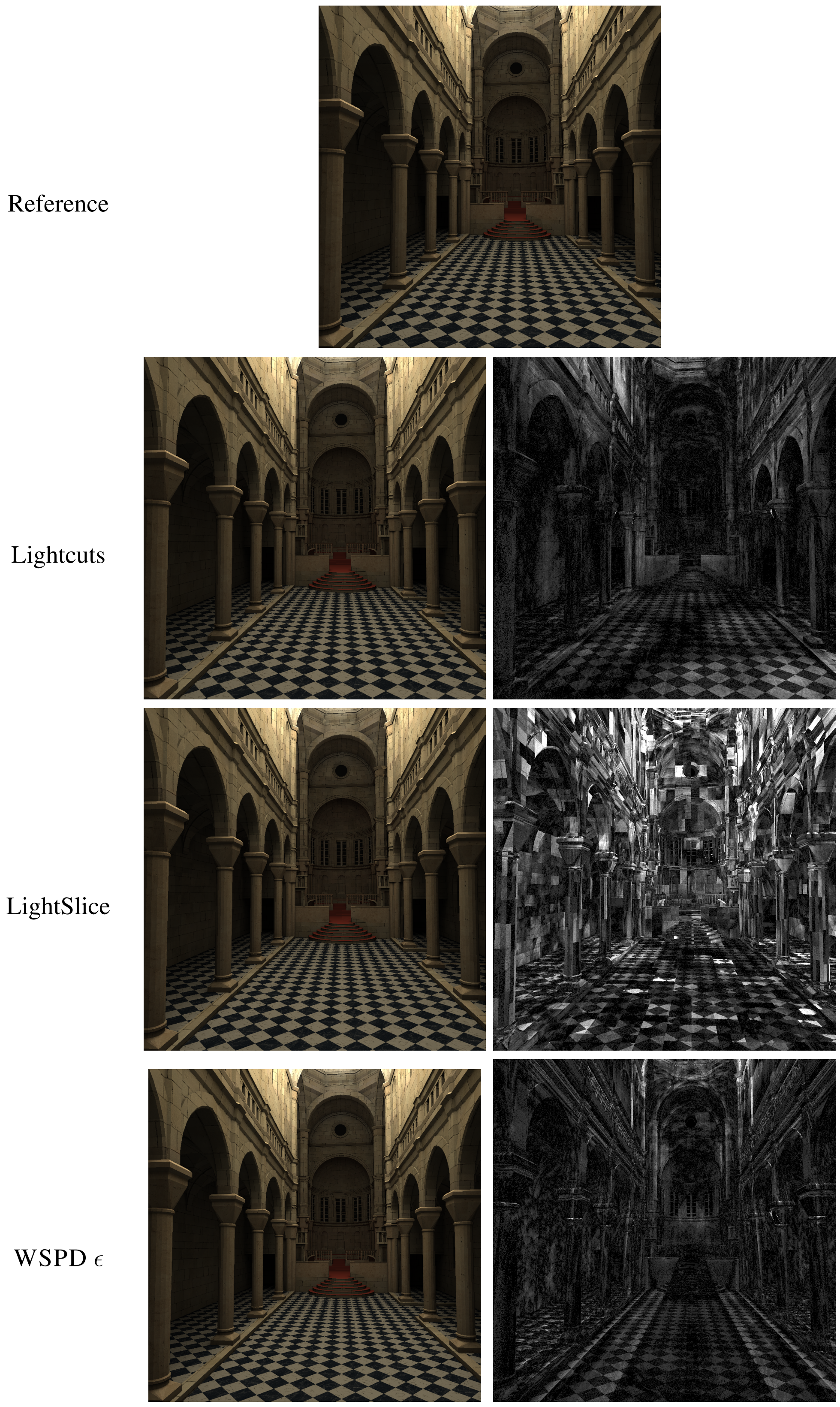


FIGURE 2.11: Sibenik scene rendered with the three different methods, Lightcuts, LightSlice and the WSPD algorithm together with the error images.

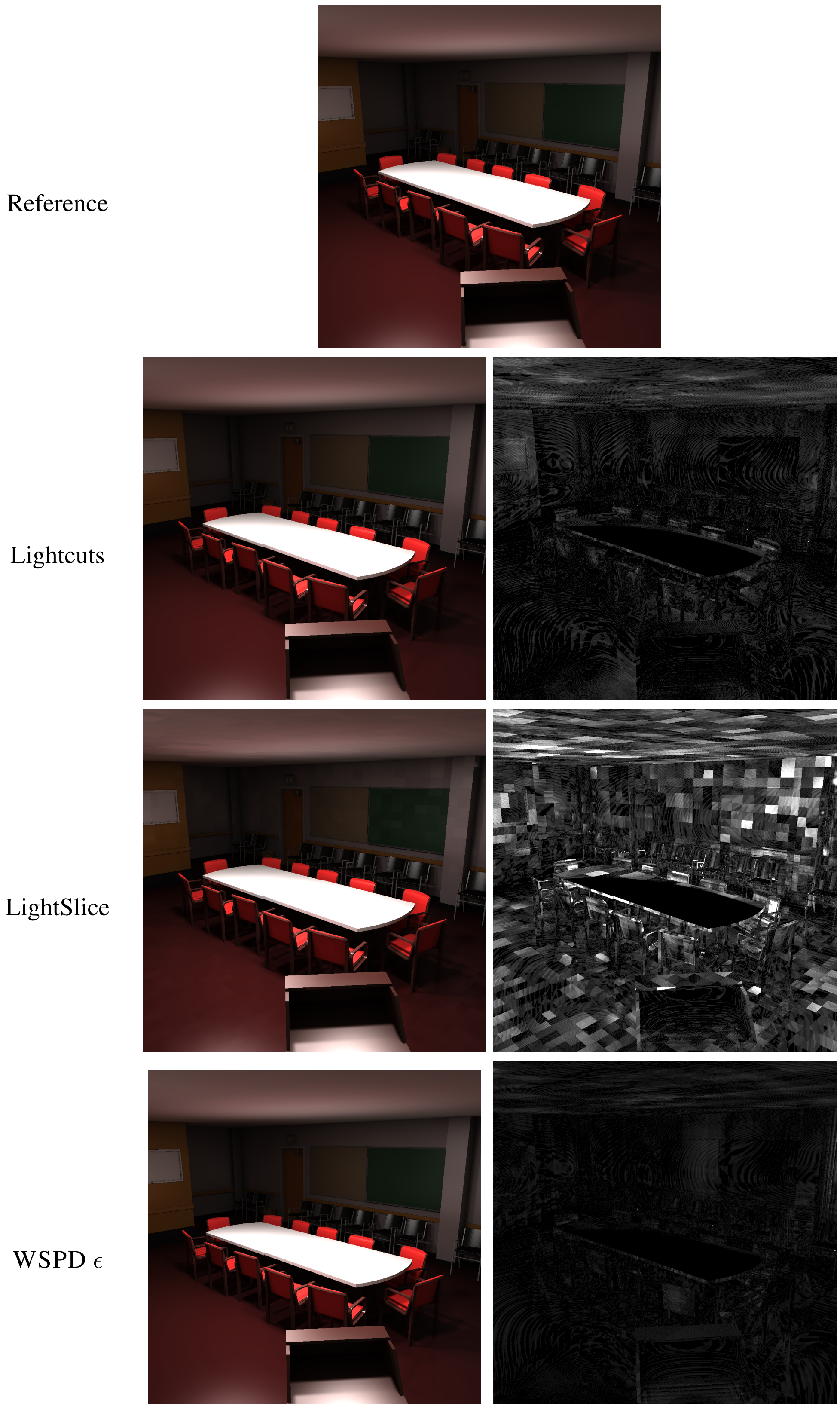


FIGURE 2.12: Conference scene rendered with the three different methods, Lightcuts, LightSlice and the WSPD algorithm together with the error images.

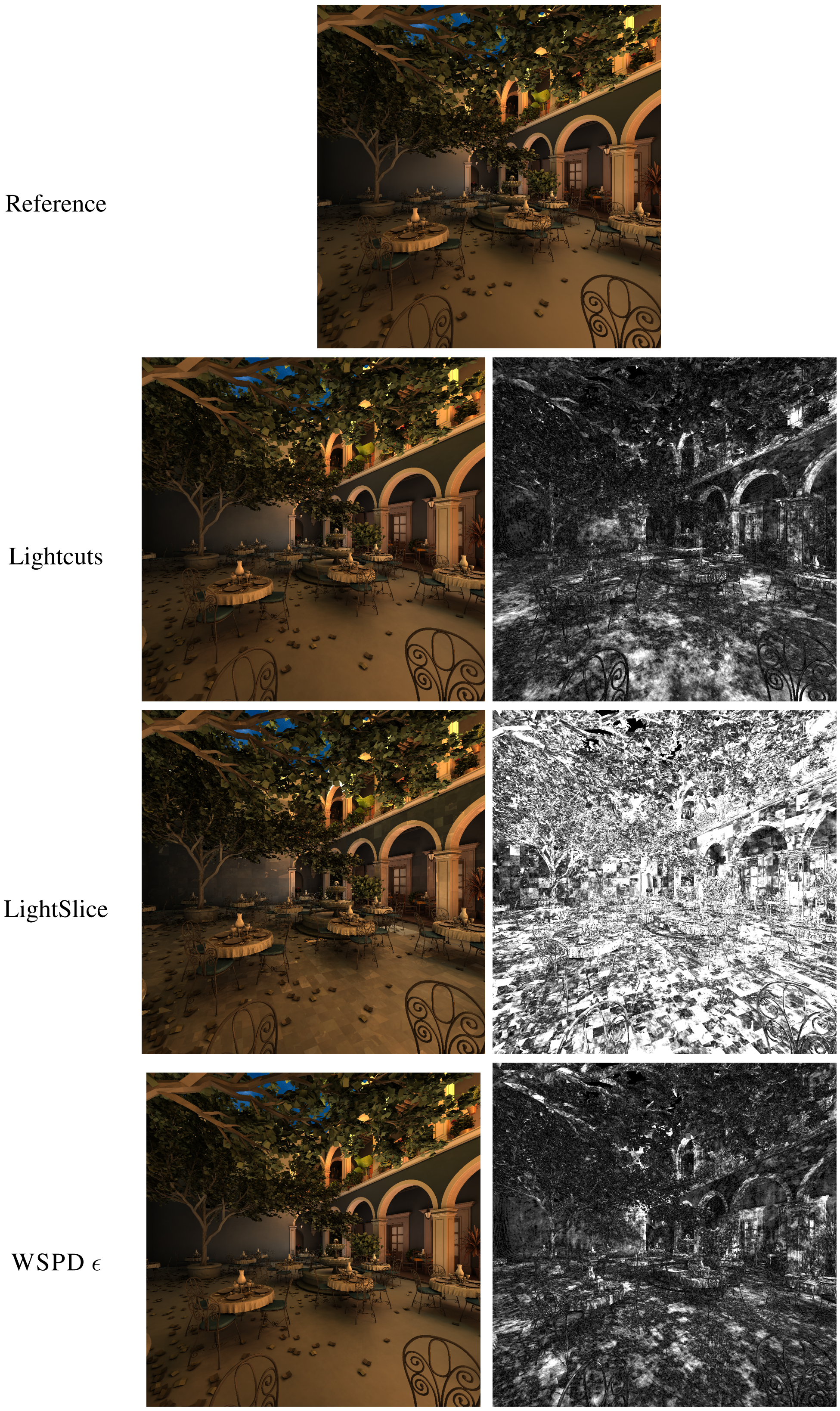


FIGURE 2.13: San Miguel scene rendered with the three different methods, Lightcuts, LightSlice and the WSPD algorithm together with the error images.

running at 2.93GHz and with 32 GB of memory. We compare our algorithm with two well-known methods: Lightcuts [WFA+05] and LightSlice [OP11]. The authors of LightSlice have made their code publicly available, which also includes an implementation of Lightcuts. In order to do fair comparisons between all three methods, we have ported their implementation of LightSlice (and Lightcuts) into the ray-tracing system INTEL EMBREE [WFW+13] without modifying the core of the algorithms. Our code is also written to use Embree as its ray-tracing engine.

Unless otherwise stated, we run LightSlice and Lightcuts with similar parameters as used in [OP11]: Lightcuts error bound is set to 1% and unbounded the maximum cut size. In order to compare our method to Lightcuts with the parameters set as in [WFA+05], we include results with 2% error and maximum cut size set to 2000. We use the version with 1% error threshold as the reference for equal quality comparisons. LightSlice is run with approximately 1400 slices and 400 columns (the number of slices determines the size of the reduced light transport matrix while the number of columns determines the number of clusters used per point). For our method, the user is free to set the separation parameter ϵ. This parameter closely tracks both theoretically and practically the quality of the resulting image. In general, setting ϵ to 0.5 gives a good compromise between quality and speed.

One inherent disadvantage of the many-lights technique is the presence of certain artifacts due to the overly-high contribution of some VPLs to their neighboring points. The standard way to avoid these problems is by *clamping*: limiting the contribution of any one VPL within some small Euclidean distance by applying a clamping threshold. The bias introduced by this technique requires the image rendered with all VPLs to be used as our reference image and not the path traced one. Our experiments use 1 and 4 samples per pixel to compare the quality of clustering obtained by the different methods.

Scenes. We test the algorithms on standard collection of scenes with only moderately glossy materials. The *Museum* has specular materials like the bones of the dinosaur and the canvas on its stand. Most of the primary lights are facing the ceiling to ensure that the scene is mainly lit by indirect illumination. In *Sibenik*, the light sources are facing upwards in the dome such that the scene is again lit by indirect illumination, and is made up of purely diffuse materials. The big uniformly colored surfaces in the *Conference* are challenging since the clustering methods have to be spatially consistent, with moderately shiny materials. The outdoor scene *San Miguel* is our largest scene consisting of 10M triangles lit by an environment map of sunset. This is our most challenging scene since the area under the tree and in the corridors is mostly lit by indirect illumination with lots of smooth shadows.

Performance. The images are rendered at a 1024×1024 resolution with 1 sample per pixel (*spp*) and with approximately 500K VPLs. We provide the running times for the pre-processing and the rendering phase along with the average number of shadow rays per pixel.

We provide three different error metrics. Denote by $F(x, y, c)$ the value of a color channel c in the image at coordinate (x, y) and by $\hat{F}(x, y, c)$ the same value in the reference image. The number of pixels multiplied by the number of color channels is $m = 3 \cdot 1024 \cdot 1024$ and each value of F and $\hat{F}$ is between 0 and 1.

- The normalized *RMSE* provides numerical difference against the VPL reference image:

  $$\text{RMSE} = \sqrt{\sum \frac{(F(x, y, c) - \hat{F}(x, y, c))^2}{m}}$$

  where the summation is over all pixels and color channels of the images.

- The *LMSE* represents the average squared difference of the gradients between the rendered image and the reference[SPA07]:

  $$\text{LMSE} = \frac{\sum(\nabla F(x, y, c) - \nabla\hat{F}(x, y, c))^2}{\sum \nabla F(x, y, c)}$$

  where $\nabla F(x, y, c) = F(x+1, y, c) + F(x-1, y, c) + F(x, y+1, c) + F(x, y-1, c) - 4F(x, y, c)$. A high *LMSE* error implies sharp discontinuities (e.g., sharp error edges), identifying more noticeable errors.

- Average relative error is given by:

  $$\text{Rel. Error} = \frac{100}{m} \cdot \sum \frac{|F(x, y, c) - \hat{F}(x, y, c)|}{\hat{F}(x, y, c)}.$$

The error images are calculated by taking the channel-wise Euclidean distance between the image, and the VPL reference image, and multiplying it by a factor of 32.

Table 2.4, Figure 2.10, 2.11, 2.12 and 2.13 show the results with all these statistics, rendered images and error images. In general, we find that with $\epsilon = 0.25$, the quality of our results is similar to Lightcuts with around $3\times$ speedup (the last row in Table 2.4). The average number of shadow rays for WSPD can be larger than that of Lightcuts; however, as proved earlier, almost no other computation except visibility testing is done by the WSPD algorithm. The WSPD method solely relies on the pre-computed pairs which results in a shorter rendering time since there is no additional work done. Lightcuts, on the other hand, has to descend the tree, maintaining an expensive heap data structure during this traversal. The cost of calculating upper bounds during rendering is also significant. Bounding the maximum cut size can result in significant loss of quality unless the ideal cut size is known a priori (e.g., as in *San Miguel*). Considering this run as a reference for equal quality comparison, our method with $\epsilon = 0.5$ still shows similar or even better quality with around 3 times speed-up (e.g., as in *Museum*).

LightSlice is able to explore the structure of VPLs and adapt to it more efficiently than Lightcuts, but dividing the image into slices results in visually disturbing blocking artifacts if the error is not low enough. This is captured by the high errors (especially the LMSE) of LightSlice compared with WSPD ϵ in all the scenes. The WSPD method locally adjusts the cluster radius based on the well-separated criteria. Thus the errors in the resulting image are smoothly distributed, with visually minimal artifacts. The value of the parameter ϵ closely tracks the quality; for scenes with complex shadows like *San Miguel*, ϵ has to be set lower (0.1) for comparable quality to Lightcuts. LightSlice relies also on a fixed parameter (number of columns, set to 400). For *San Miguel* it is unable to adapt to the complexity of the shadows with such a low number of columns, resulting in faster running times with high errors.

Scalability. See Table 2.5 for the total memory (GB) used by the three methods. Lightcuts is the most efficient on memory consumption, followed by WSPD. LightSlice, due to the light transport matrix storage, has prohibitively high memory consumption.

# VPLs:	75K	115K	200K	375K	776K
Lightcuts	0.59	0.6	0.63	0.74	0.84
LightSlice	4.53	6.31	9.85	18.81	36.58
WSPD 0.5	1.14	1.40	1.98	3.38	6.17

TABLE 2.5: Memory requirements for the *Museum* scene (GB).

In Figure 2.14 we plot the rendering times of the three methods with varying number of VPLs in the Museum scene. Note how our algorithm consistently outperforms Lightcuts even with 3M VPLs and both of them scale sub-linearly in the number of light sources. LightSlice is only usable as long as it does not allocate more memory than present in the system.

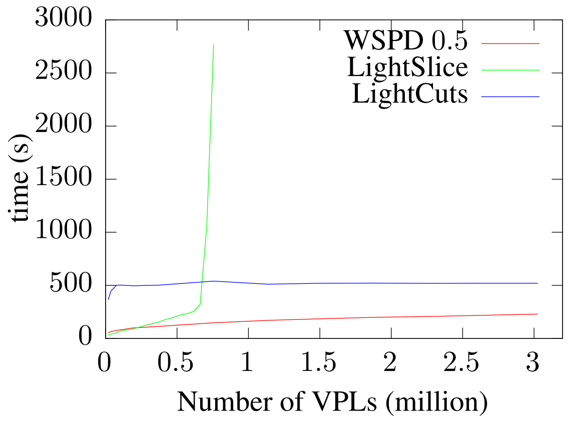


FIGURE 2.14: Varying number of VPLs for the *Museum* scene.

Trade-off: In Figure 2.15 we plot the relative error and the rendering time against ϵ in the *Museum* scene. The curve is not strictly monotonic since our algorithm is not deterministic (e.g., approximate nearest neighbor).

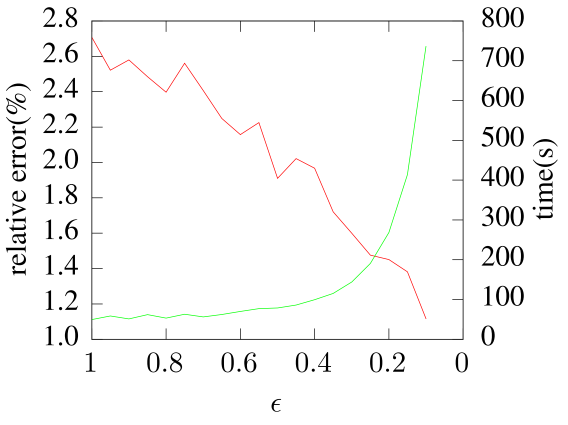


FIGURE 2.15: Relative error (red) and render time (green) with varying ϵ for the *Museum* scene.

Blocking artifacts. In contrast to LightSlice, our method and Lightcuts currently do not take advantage of using different representative lights for each sample per pixel. We have found that increasing the number of samples increases the quality of the images rendered with LightSlice. Table 2.6, Figure 2.18 and 2.19 show some results with 4 samples per pixel. Note that our algorithm still matches LightSlice in performance. See Figure 2.16 for the effect of multiple samples per pixel on the *Museum* scene. Our method has no visible artifacts (nor does Lightcuts) even with $\epsilon = 0.9$ and 1 sample per pixel. On the other hand LightSlice has visible blocks on the image due to the clustering of the shaded points. These errors can be reduced by increasing the number of columns for LightSlice but this results in a higher running time. If one increases the number of samples then LightSlice becomes competitive, although some artifacts remain even with 9 samples per pixel.

Refinement. This method ensures that the final clustering used for a point satisfies our theoretical criteria. However, with a high number of VPLs the approximate nearest neighbor search is very accurate therefore our refinement method has less importance. If the density of the VPLs is low in an area its usage becomes more important in order to avoid blocking artifacts. We demonstrate this in Figure 2.17, by rendering the scene with and without refinement.

2.5 Limitations

We have showed that the WSPD structure is suitable for compactly storing clustering information and for providing fast extraction of clusters during render time. We showed theoretical

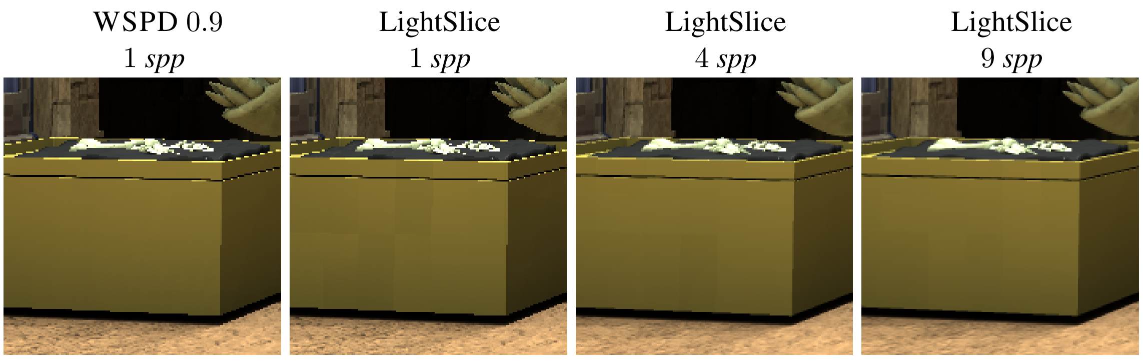


FIGURE 2.16: Part of the *Museum* scene rendered with different values of samples per pixel.

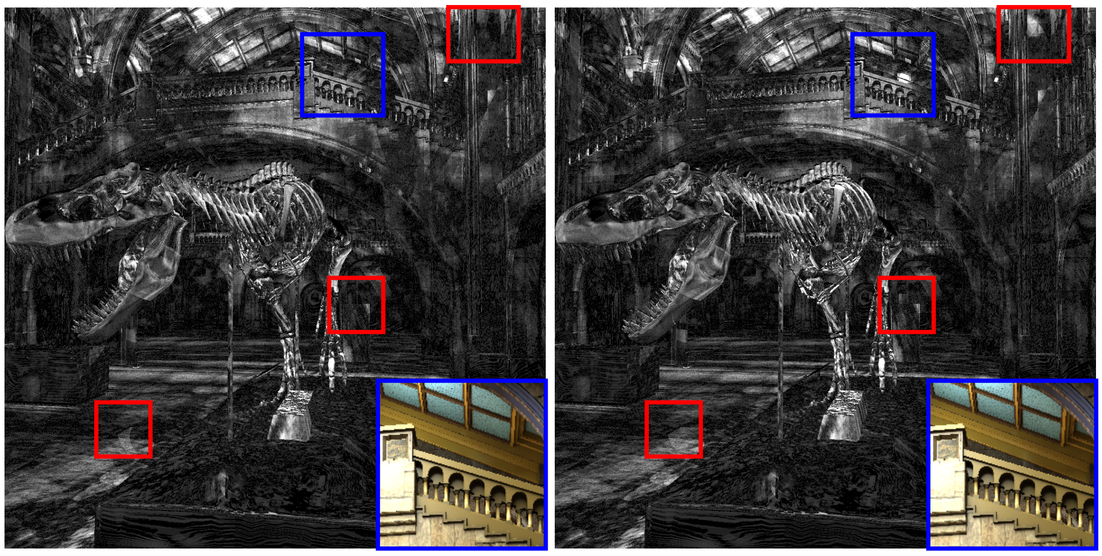


FIGURE 2.17: Error images for the Museum with 10K VPLS, rendered with refinement (left) and without refinement (right), for $\epsilon = 0.25$.

bounds on the error for diffuse materials. The data structure proposed in this work gives a new perspective on how to efficiently store and retrieve a view-independent clustering for scenes. This framework is very flexible and leaves several possibilities for improving and enhancing the current solution.

One of the main limitations of our method is that it is suited towards diffuse surfaces, and the quality decreases with highly glossy surfaces. The upper-bound proved in Theorem 2.2 increases as a function of glossiness, and so higher glossiness requires smaller values of ϵ in the WSPD construction for similar error upper-bounds. One can compensate for it with decreasing ϵ, at the loss of efficiency. In order to demonstrate this, we have replaced the Phong BRDFs in the *Conference* scene with Blinn microfacet BRDFs. This BRDF results in a very significant contribution from the direction of the reflected view ray. See Figure 2.20 for the results with 1 sample per pixel. The most prominent error is around the shadows of the chairs on the highly glossy floor where the exponent for the Blinn microfacet BRDF is set to 100. Lightcuts can more efficiently adapt to highly glossy materials than our method.

		Museum	Conference
Scenes	Triangles	1.5M	0.33M
	Resolution	$1024 \times 1024@4$	$1024 \times 1024@4$
	VPLs	474K	516K
Lightcuts	Preproc. time (s)	0.48	0.44
	Render time (s)	2080.97	876.96
	Avg # of rays	7544	3828
	RMSE	0.003374	0.002119
	LMSE	0.001008	0.003789
	Rel. Error(%)	0.905316	1.208336
	Speedup	1.0	1.0
Lightslice	Preproc. time (s)	0.01	0.02
	Render time (s)	**344.73**	**262.89**
	Avg # of rays	1696	1792
	RMSE	**0.005662**	**0.003025**
	LMSE	0.005431	0.005524
	Rel. Error(%)	1.836033	2.328994
	Speedup	6.0	3.3
WSPD 0.7	Preproc. time (s)	63.47	40.27
	Render time (s)	265.25	158.72
	Avg # of rays	2660	2528
	RMSE	0.006984	0.003378
	LMSE	0.002270	0.004733
	Rel. Error(%)	2.318444	2.425038
	Speedup	7.8	5.5
WSPD 0.5	Preproc. time (s)	68.38	43.25
	Render time (s)	**356.20**	**165.58**
	Avg # of rays	3728	2588
	RMSE	**0.005661**	**0.003029**
	LMSE	0.001876	0.005271
	Rel. Error(%)	1.809967	2.099148
	Speedup	5.8	5.3
WSPD 0.3	Preproc. time (s)	84.68	56.42
	Render time (s)	646.19	260.38
	Avg # of rays	6228	4192
	RMSE	0.005458	0.002664
	LMSE	0.001481	0.004486
	Rel. Error(%)	1.563184	1.717990
	Speedup	3.2	3.4

TABLE 2.6: Rendering statistics and images for 4 samples per pixel.

A similar problem is the need to use a small ϵ for scenes with highly varying visibility properties since in this case the number of clusters increases globally without only refining the clustering where it is necessary. Currently our definition of *well-separatedness* is purely geometric, we show in the next chapter how to create a clustering more adapted to illumination properties.

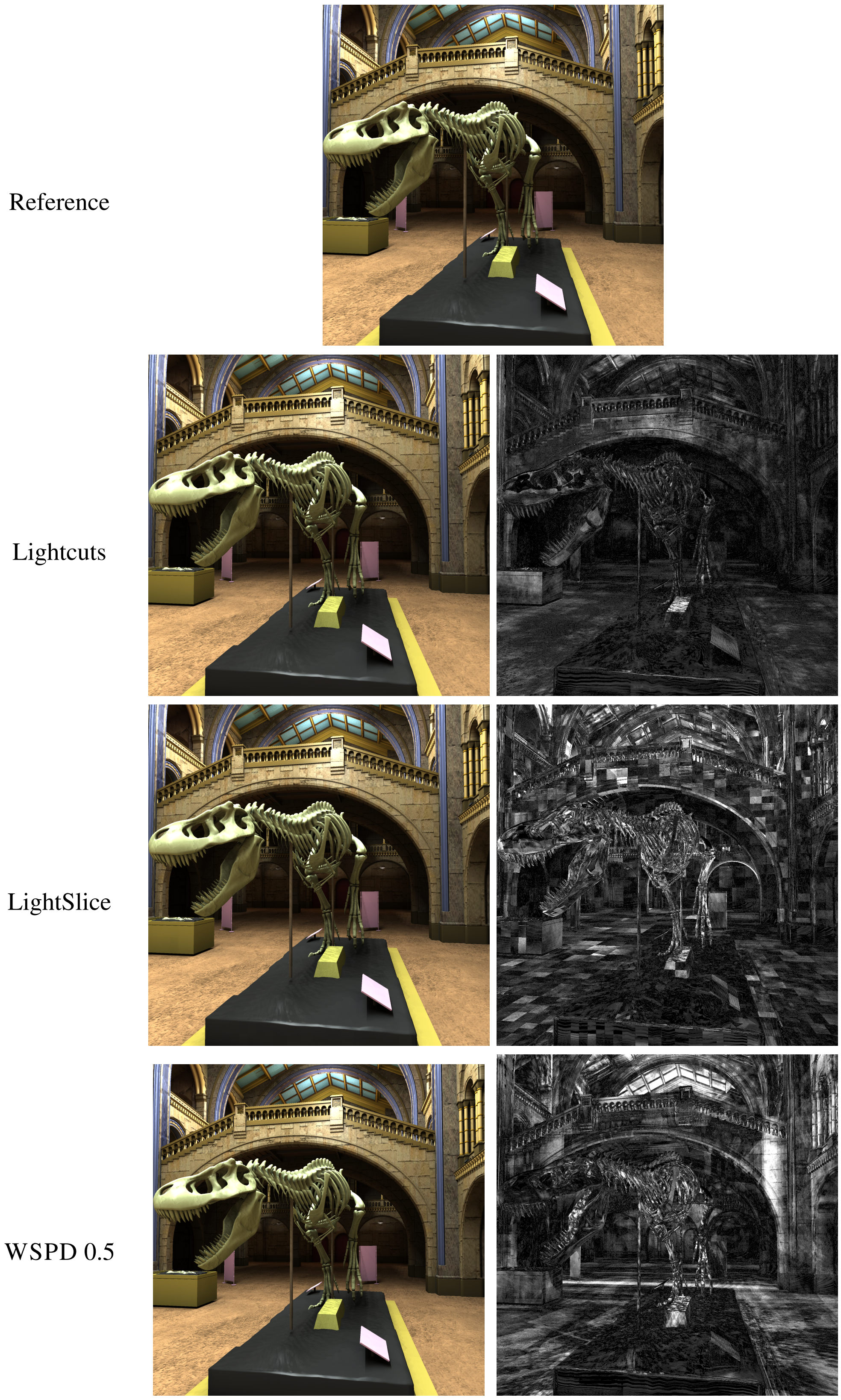


FIGURE 2.18: Museum scene rendered with 4 samples per pixel with the three different methods, Lightcuts, LightSlice and the WSPD algorithm together with the error images.

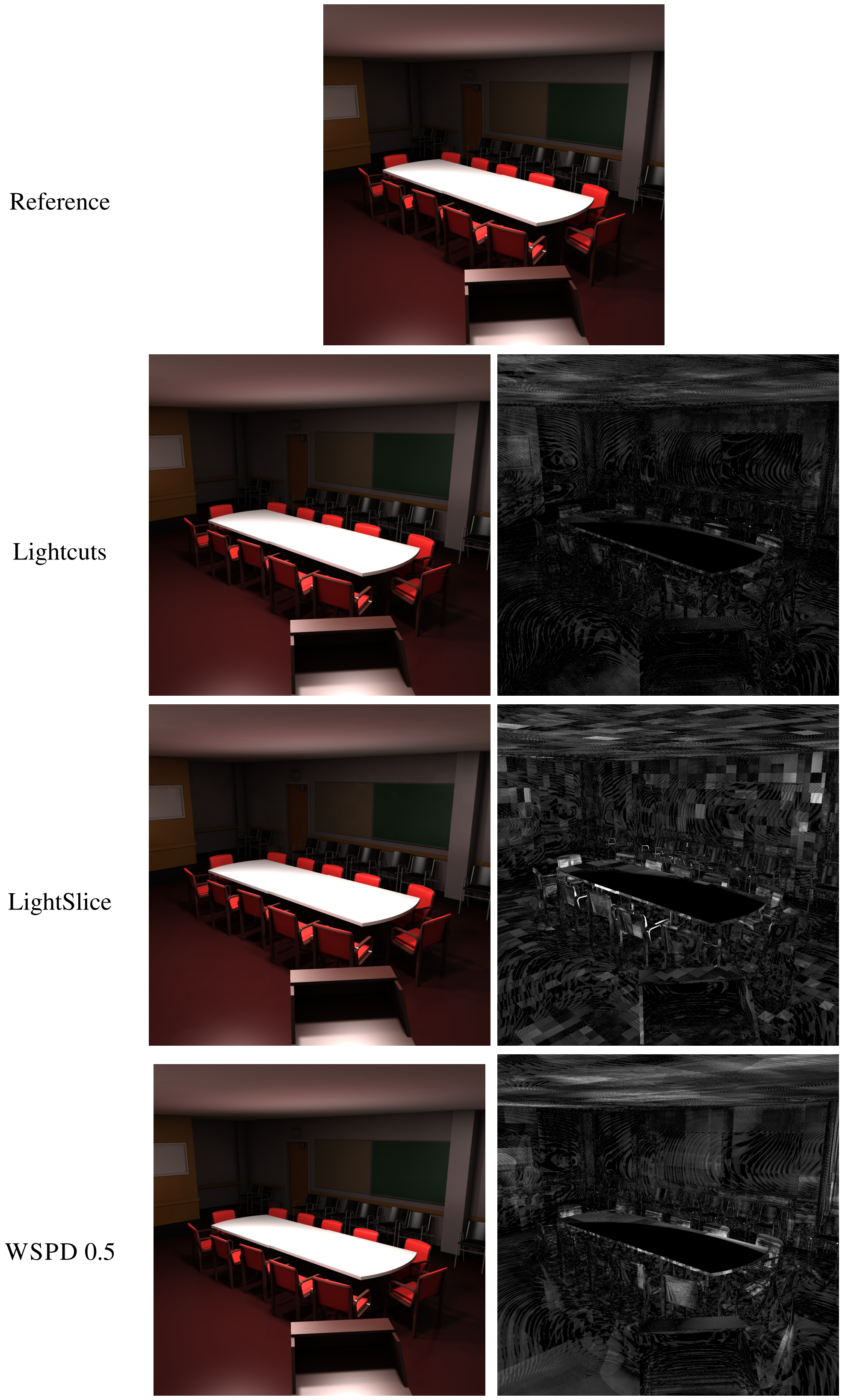


FIGURE 2.19: Conference scene rendered with the three different methods, Lightcuts, LightSlice and the WSPD algorithm together with the error images.

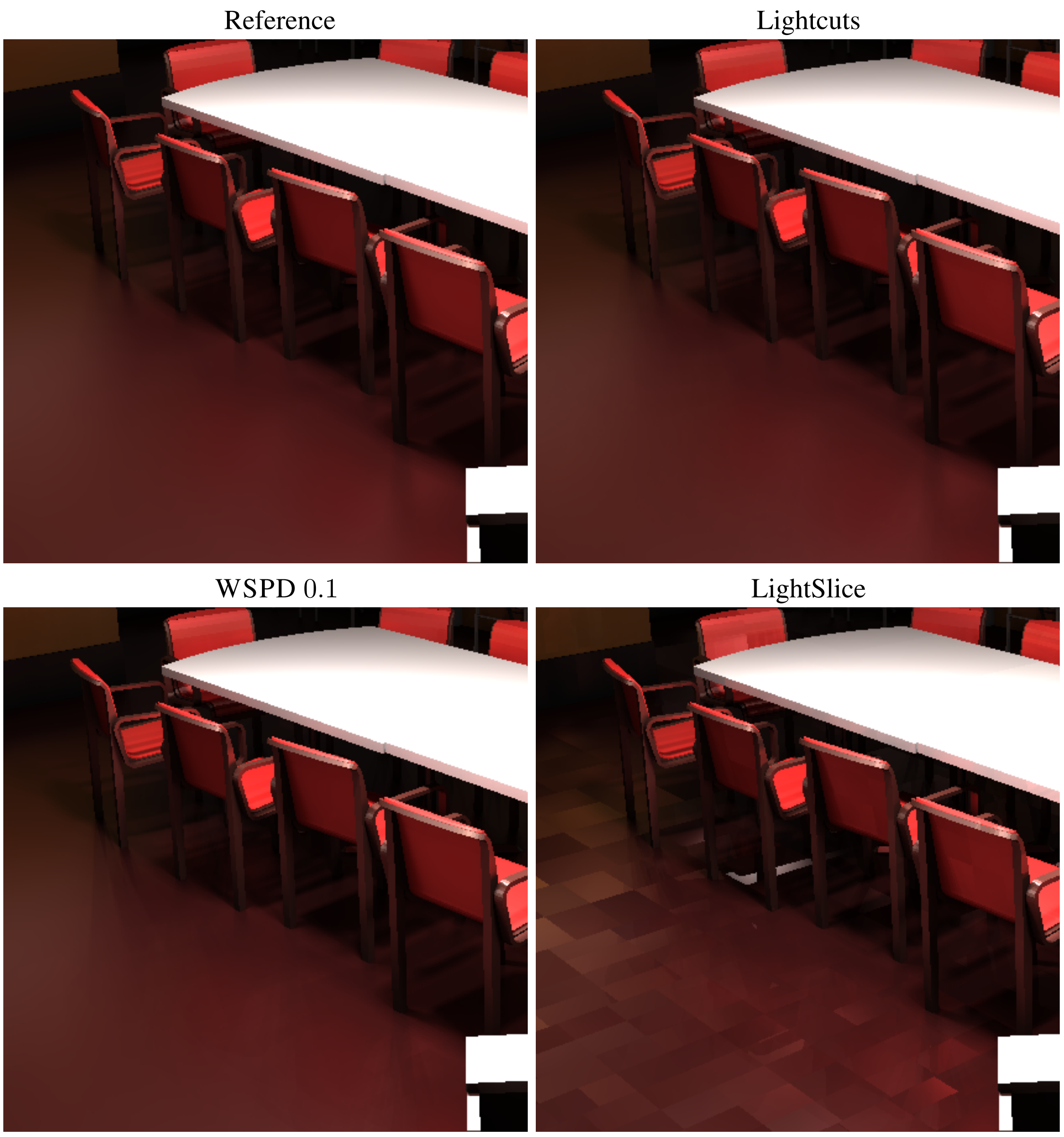


FIGURE 2.20: Part of the Conference scene with highly glossy floor using 1 sample per pixel.

Chapter 3

IlluminationCut

This chapter describes our improved many-lights algorithm achieved by clustering the product space of VPLs and shaded points, it is based on the following paper.

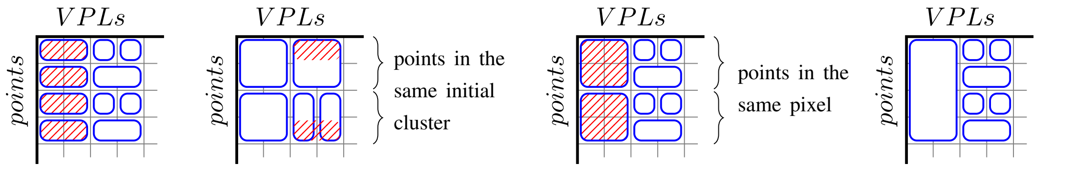
Norbert Bus, Nabil H. Mustafa, and Venceslas Biri. “IlluminationCut.” In: *Computer Graphics Forum (Proceedings of Eurographics 2015)* 34 (2), 2015

3.1 General Idea

This work proposes a new algorithm belonging to the family of instant radiosity methods. We start with a detailed overview of the different clustering techniques in the many-lights methods in order to highlight the differences between the structure of clusters. Current state-of-the-art clustering algorithms are, e.g., Lightcuts [WFA+05] and LightSlice [OP11]. Lightcuts builds a tree on the VPLs where each node of this tree represents a cluster of VPLs in the subtree of that node. For each point to be shaded, it descends in the tree to select a set of nodes (a ‘cut’) which is taken to be the VPL clustering for that point. While the method is robust and able to bound the error resulting from treating each cluster as a single VPL, the cut is recomputed for every point and descending in the tree is expensive for complex lighting situations.

LightSlice first groups all points to be shaded into a small number of roughly equal-sized clusters, called point-clusters, based on their geometric proximity. Then it uses visibility and shading information to obtain a clustering of the VPLs for each of these point-clusters. Thus all the points in the same point-cluster have the same clustering of the VPLs. The main advantage of the method is its speed, since it is able to detect occluded clusters and amortize the cost of creating VPL clusterings across points in a point-cluster. However, it has no error bound and as the construction of the point-clusters is not adapted to the illumination of the scene, it is prone to failure if the radiance of points within a group is highly varying.

Lightcuts constructs a different clustering of VPLs for each point; LightSlice clusters all the points to be shaded into a number of point-clusters for which the same clustering of VPLs is computed. Our idea is based on the following observation: instead of clustering points or VPLs independently, what is required is *clustering their product-space*, namely to cluster all point-VPL pairs. Each cluster in this product-space consists of a subset of points (to be shaded) paired with a subset of VPLs.



(a) *Lightcuts* (b) *LightSlice* (c) *Multidimensional Lightcuts* (d) *IlluminationCut*

FIGURE 3.1: Partial light transport matrices, with rectangles denoting product-space clusters. Red stripes denote parts that could be improved. (a) Lightcuts creates clusters that could be merged; (b) LightSlice creates clusters that should be merged or refined; (c) Multidimensional Lightcuts only merges and refines clusters limited to points originating from the same pixel; (d) IlluminationCut merges and refines clusters for any set of points and VPLs.

In fact, both Lightcuts and LightSlice can be seen as constructing constrained product-space clusterings. Each cluster created by Lightcuts consists of a single point paired with a set of VPLs. This constraint is wasteful as two points which are very similar could have been grouped together in many product-space clusters. See Figure 3.1 (a).

LightSlice, on the other hand, constructs a product-space clustering where the same set of points are grouped together in any cluster. For efficiency reasons each point-cluster is large, which severely limits how well the VPL-clusters paired to them can be adapted to each individual point in the point-cluster. Furthermore, as the initial clustering of points used only geometric information, these clusters cannot be completely adapted to illumination, and are likely to introduce artifacts on the cluster boundary. See Figure 3.1 (b).

Our contribution. Our proposed method, *IlluminationCut*, targets high fidelity off-line rendering by constructing an illumination-aware clustering of the product-space of all point-VPL pairs. We create illumination-aware product-space clusters without any a priori constraints on either the points or the VPLs that can appear in product-space clusters. These clusters capture similar point-VPL pairs such that shading every point in a cluster by using a single representative VPL instead of all VPLs in the cluster causes error that remains under a threshold. Treating cluster pairs enables us to amortize calculations that were previously carried out separately for each point in Lightcuts; and to construct non-uniform clusters with different subsets of points with different subsets of VPLs, which is more adaptive clustering than that of LightSlice. Our method is further extended by adaptive visibility sampling, reducing the number of rays traced for each product-space cluster without introducing high error. See Figure 3.1 (d).

IlluminationCut builds on the Multidimensional Lightcuts approach [WAB+06], in that they both utilize two hierarchies (trees), one on points and one on VPLs to construct product-space clusters with bounded error by simultaneously descending on the trees. The difference is that the latter has to maintain a heap and repeatedly builds a tree only for points that originate from the same pixel (e.g., for use in spatial anti-aliasing). It does not exploit possible similarity among points originating from different pixels and so does not improve upon Lightcuts if there is only one point per pixel. See Figure 3.1 (c).

Our results improve on both the quality and the efficiency of previous methods. We achieve $3 - 6$ times speed-up by reducing the number of visibility queries, dramatically decreasing the computations needed to construct clusters, as well as eliminating the need for maintaining a heap during rendering.

Organization. A detailed description of our algorithm is given in Section 3.2. Experimental results and comparison with state-of-the-art methods are presented in Section 3.3. Finally, limitations and future work are discussed in Section 3.4.

3.2 Algorithm

Overview. The method first constructs a cluster hierarchy on $\mathcal{P}$, called the *point tree* and on $\mathcal{S}$, called the *light tree*. Then using these trees, Phase I computes a coarse but fast approximation for every point $p \in \mathcal{P}$. In Phase II, this approximate image is used to guide a top-down search of both trees simultaneously to construct the list of desired product-space clusters $(R_1, Q_1), \ldots, (R_k, Q_k)$. Here each (R_i, Q_i) is a product-space cluster composed of the set of points $R_i \subseteq \mathcal{P}$ and the set of VPLs $Q_i \subseteq \mathcal{S}$. Finally, for all product-space clusters (R_i, Q_i), illumination contribution from Q_i is added to the radiance of each point $p \in R_i$.

Light and point trees. For the light tree we use the same structure as Lightcuts [WFA+05]. For the point tree, we use a compressed octree, which differs from a simple octree by the fact that paths without branching are contracted into a single edge. Then each node in the tree represents a unique cluster of the points in its bounding box. To ensure that the points located in the same node face approximately in the same direction, we make a slight modification: the subdivision of the first 3 levels into octants correspond to the subdivision of the space of normals of the points (these are unit vectors in $\mathbb{R}^3$) and the remaining levels follow the standard octree subdivision rule. The points in $\mathcal{P}$ are stored in an array. In our implementation the octree is constructed by repeatedly subdividing the bounding box of the scene along planes perpendicular to the three axes. Thus recursively, at each subdivision, the points corresponding to a node are partitioned in-place into two contiguous subarrays. As a result the points in the array are in z-order (also called Morton-order [Gar82]). The construction ensures that each node in the tree contains points located in a contiguous part of the array. Therefore retrieving points associated to a node is efficient since one has to only iterate over a subarray.

Our algorithm stores additional auxiliary data with the nodes of both trees. These are the bounding box of the points inside the node and representative lights/points. The latter are sampled in the same way as Multidimensional Lightcuts (the sampling ensures that the algorithm remains unbiased in the Monte Carlo sense). For each light tree node we also store the bounding cone of the light directions of VPLs associated with that node. Each node of the point tree stores the maximum/minimum BRDF components in the subtree of the node ($k_{\text{spec\_max}}, k_{\text{diff\_max}}, n_{\text{min}}, n_{\text{max}}$). We also need to associate color data ($color$) with the nodes of the point tree.

Clusters and representatives. We will identify the nodes of the tree with the points they contain, e.g., the root of the light tree is simply denoted by $v(\mathcal{S})$. Let us denote by Q a cluster of lights $Q \subseteq \mathcal{S}$ and its corresponding octree node as $v(Q)$. Denote the radiance at p caused by lights in Q as $L_Q(p, \omega)$.

$$L_Q(p, \omega) = \sum_{s \in Q} M_s(p, \omega) \cdot V_s(p) \cdot I_s \cdot G_s(p) \tag{3.1}$$

For a node $v(Q)$ let $rep(Q) \in Q$ denote its representative light, and then compute the approximate radiance at p from lights in Q with representative $rep(Q)$ as:

$$\widetilde{L}_Q(p, \omega) = M_{rep(Q)}(p, \omega) \cdot G_{rep(Q)}(p) \cdot V_{rep(Q)}(p) \cdot \sum_{s \in Q} I_s \tag{3.2}$$

For a cluster R denote the radius of the enclosing ball of its bounding box by $r(R)$, and by $d(R, Q)$ the distance between the enclosing balls of clusters R and Q.

Phase I: Computing approximate shading. Our algorithm needs an estimate of the radiance of each point $p \in \mathcal{P}$. It is computed by descending in both trees until for a pair of point and VPL nodes $(v(R), v(Q))$, the condition $max(r(R), r(Q)) < 0.1 \cdot d(R, Q)$ is satisfied and the aperture of the light node's cone is less than 20°; we then add the contribution of the VPL cluster Q to each point in R. This criteria attempts to ensure, though without any guaranteed bound on the error, that this estimated radiance roughly matches the value that would result from exhaustively evaluating the radiance for every point-VPL pair in (R, Q). Notice that for shading a pair (R, Q) we only take into account the representative point's BRDF and do not shade every point individually. This necessarily introduces error to our approximation image (e.g., the texture can vary within clusters). However, since this image is only used as an error upper bound it does not have a significant effect on the final image (Section 3.4 contains a more detailed discussion). Furthermore, instead of adding the calculated contribution of Q to all the points in $v(R)$, we can simply accumulate it in the node and later with a tree traversal, distribute it to the leaves (each containing a point of $\mathcal{P}$). We also store the minimum of the approximate radiances of the points in a node $v(R)$ as $v(R).color$. This is used in Phase II.

In Figure 3.2 we show the images rendered with this approximation to illustrate how they capture illumination.

Illumination-aware pairs. A set of points $R \subseteq \mathcal{P}$ and a set of lights $Q \subseteq \mathcal{S}$ form an illumination-aware pair if

$$\max_{p \in R} |\widetilde{L}_Q(p,\omega) - L_Q(p,\omega)| < \delta \cdot \min_{p \in R} L(p,\omega) \tag{3.3}$$

where δ is the error threshold, e.g., 1%.

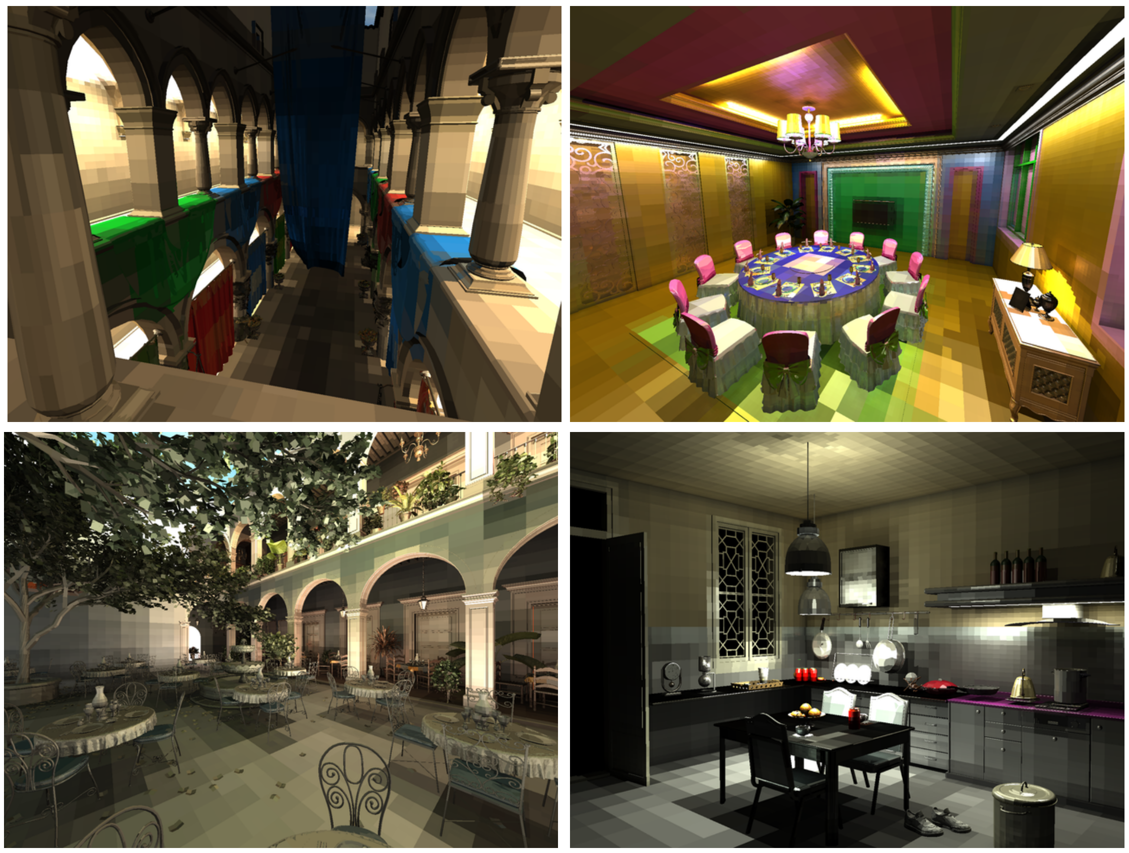


FIGURE 3.2: Approximate images for various scenes. These images are used to guide the search for product-space clusters in Phase II.

Algorithm 4: IlluminationCut

```
Data: W: stack of pairs; light and point trees for $\mathcal{S}$, $\mathcal{P}$
1  Function IlluminationCut()
2      $W \leftarrow \emptyset$
3      $W$.pushback($v(\mathcal{P}), v(\mathcal{S})$)
4      while notEmpty($W$) do
5          $(v(R), v(Q))$ = $W$.pop()
6          if IsIllumAwarePair($v(R), v(Q)$) then
7              foreach $p \in R$ do
8                  $L(p,\omega) \mathrel{+}= \tilde{L}_Q(p,\omega)$
9          else
10             if $r(Q) > r(R)$ then
11                 if $v(Q)$ has no children then
12                     foreach $p \in R$ do
13                         $L(p,\omega) \mathrel{+}= \tilde{L}_Q(p,\omega)$
14                 else
15                     foreach $u \in$ children($v(Q)$) do
16                         $W$.pushback($u, v(R)$)
17             else
18                 if $v(R)$ has no children then
19                     foreach $p \in R$ do
20                         $L(p,\omega) \mathrel{+}= \tilde{L}_Q(p,\omega)$
21                 else
22                     foreach $u \in$ children($v(R)$) do
23                         $W$.pushback($v(Q), u$)
24
25 Function IsIllumAwarePair($v(R)$,$v(Q)$)
26     return $\delta \cdot v(R).color > L_{UB}(v(R), v(Q))$
```

The definition ensures that for any point $p \in R$, using Q as a light cluster would result in a small error. This is the most conservative error bound but other variations, e.g., average of $L(p, \omega)$ could be used as well. In order to evaluate this condition one requires the knowledge of the true radiance a priori. This is estimated using the approximate image calculated in Phase I; i.e., use the minimum radiance in the approximate image of points belonging to R. This minimum was stored in each node as $v(R).color$ in Phase I.

The left-hand side of Equation (3.3), denoted $L_{UB}(v(R), v(Q))$, can be upper bounded in a similar manner as Multidimensional Lightcuts. Let M_{max} and G_{max} be the upper bounds on the material and geometric terms over all point-VPL pairs in (R, Q). As the visibility term can

Algorithm 5: Adaptive sampling for a pair (R, Q)

```
Function AdaptiveSampling(R,Q)
    if $Q$ is a leaf or $|R| < 8$ then
        foreach $p \in R$ do
            $L(p, \omega) \mathrel{+}= \tilde{L}_Q(p, \omega)$
    else
        $vs$: the number of samples
        if $|R| > 32$ then
            $vs \leftarrow 16$
        else
            $vs \leftarrow 8$
        compute $R'$, with $|R'| = vs$
        if $V_{rep(Q)}(q) = 1$ for all $q \in R'$ then
            foreach $p \in R$ do
                $L(p, \omega) \mathrel{+}= \tilde{L}_Q(p, \omega)$ with
                $V_{rep(Q)}(p)$ set to 1
        else if $V_{rep(Q)}(q) = 0$ for all $q \in R'$ then
            return
        else
            foreach $R_i$ defined by $R'$ do
                AdaptiveSampling($R_i, Q$)
```

be upper bounded by 1, $L_{UB}(v(R), v(Q))$ can be written as:

$$L_{UB}(v(R), v(Q)) = M_{max} \cdot G_{max} \sum_{s \in Q} I_s \tag{3.4}$$

We show how to calculate $G_{max} = \frac{\cos \phi_{max}}{r^2_{min}}$ where $r_{min} = \min_{p \in R, s \in Q} d(p, s)$ and ϕ_{max} is the angle between the light normals and light directions for which the cosine function attains its maximum value. r_{min} is set to $d(R, Q)$. For ϕ_{max}, we use the same technique as described in Multidimensional Lightcuts. First, we simplify the problem by calculating the bounding box of all possible light-point vectors between R and Q then apply a linear transform to this bounding box such that the direction of the light node's cone is aligned with the z axis. This transformation enables the direct evaluation of ϕ_{max}. Calculating M_{max} can be done in a similar fashion by using the surface normal and the reflected view ray ω' of $rep(R)$ in the role of the cone direction (the surface normals and reflected view rays for points in a node are located in a small cone due to our octree construction). See the details in [WAB+06].

Phase II: Rendering with illumination-aware pairs. Once we have an approximate radiance for each point $p \in \mathcal{P}$ as well as the minimum radiance of all the points in a subtree rooted at v (stored as $v.color$), we again traverse the two trees simultaneously top-down to construct the illumination-aware pairs. For each illumination-aware pair (R, Q), add the illumination contribution of the VPL cluster Q to each point in R. See Algorithm 4. The straightforward way of adding the illumination contribution of Q to each point in R is by computing $\tilde{L}_Q(p, \omega)$ separately via a shadow test from each p to $rep(Q)$ and then using Equation (3.2). We refer to this method as IlluminationCut.

Adaptive sampling. Algorithm 4 computes, for each illumination-aware pair (R, Q), the contribution $\tilde{L}_Q(p, \omega)$ of Q to each point $p \in R$ by performing a visibility query from p to $rep(Q)$. Instead, one can use an adaptive sampling technique that reduces the number of visibility queries to be considerably less than the number of points in R, denoted by $|R|$. Given an illumination-aware pair (R, Q), we have access to the points in R as a subarray in z-order; we refer to this subarray as R, this ambiguity shall not cause a problem. Take a subset R' of R dividing it into at most 16 equal length subarrays and calculate the visibility between points of R' and the representative light $rep(Q)$. If all points in R' are visible or all are occluded, use this visibility for shading R; if not, then we recurse on the subarrays R_i defined by R' . We note that choosing R' in this manner makes the algorithm biased. Though errors are unlikely, as the subarrays consist of spatially proximate points. See Algorithm 5. We refer to this method as IlluminationCut-Sampling.

3.3 Results and Discussion

In this section we present experimental results for several scenes with complex lighting, highly glossy materials and varying geometric complexity. Timings are for a server equipped with Intel(R) Xeon(R) E5-2680 CPUs utilizing in total 20 cores running at 2.8 GHz, with 74 GB of memory.

Implementation. We compare our algorithm with two well-known methods: Lightcuts [WFA+05] and LightSlice [OP11]. Since the authors of LightSlice published their code, we have ported their implementation into the ray-tracing system *Intel Embree 2.3* [WFW+13; WWB+14]. We have improved their Lightcuts implementation with the agglomerative clustering method presented in [Mik10]. The code is written in C++ with a very efficient ray-tracing engine. Due to recent advantages in packeted ray-tracing algorithms and their implementations, shading and other calculations account for a significant portion of the overall rendering time in our system; e.g., approximately 20% of the total time for Lightcuts is used by ray tracing (see [WPS+03]).

The code can be downloaded at the website of the authors.

		Banquet	San Miguel	Sponza	Kitchen
Scenes	Triangles	0.74M	10.5M	0.28M	0.17M
	VPLs	638K	677K	641K	551K
LC(1%)	Preproc. time (s)	56.05	109.22	80.57	89.80
	Render time (s)	116.71	749.41	233.31	183.07
	Avg # of rays	947	5161	2294	1597
	RMSE	0.010468	0.011369	0.005913	0.010336
	Rel. Error	2.740256	2.770336	4.316007	8.957002
	Upper bound	2878	14202	6183	4313
LS	Render time (s)	365.74(230)	227.88(41)	263.35(129)	116.61(67)
	Avg # of rays	2879	2722	2849	1362
	RMSE	0.016666	0.028958	0.011842	0.013217
	Rel. Error	3.469874	7.718524	4.048875	10.411767
	Columns	3200	3200	3200	1600
	Speedup	0.3	3.3	0.9	1.6
IC(1.5%)	Preproc. time (s)	58.28	112.48	82.07	93.02
	Render time (s)	36.94	186.12	71.83	43.04
	Avg # of rays	1301	5561	3064	1258
	RMSE	0.010401	0.012534	0.005747	0.012562
	Rel. Error	2.442368	3.137196	4.113609	9.543214
	Upper bound	90	236	69	216
	Speedup	3.1	4.0	3.3	4.2
IC-S(1%)	Preproc. time (s)	58.32	112.40	82.31	92.31
	Render time (s)	23.30	78.58	23.89	30.26
	Avg # of rays	463	1849	720	602
	RMSE	0.010097	0.012956	0.006997	0.011005
	Rel. Error	2.639261	3.807332	4.382032	9.087991
	Upper bound	155	362	102	350
	Speedup	5.0	9.6	9.7	6.1

TABLE 3.1: Rendering statistics for 1600×1200 resolution images with 1 sample per pixel. The parameters are set to achieve approximately equal RMSE error, except for LightSlice which fails to resolve certain artifacts.

Scenes. We test the algorithms on a collection of scenes, all of them having highly glossy materials except for *Sponza*, which is completely diffuse. The outdoor scene *San Miguel* is our largest scene, consisting of 10M triangles lit by an environment map. Many of the VPLs are placed on walls facing outward; therefore this scene is exploiting the weakness of our algorithm and Lightcuts, namely that occluded clusters are not quickly discarded. *Banquet* has a grid of point light sources directed towards the ceiling and lights inside the lamps, both creating a challenging global illumination setup as there is significant indirect lighting. It also contains a strip of area lights running around the ceiling. *Sponza* has all its original point light sources facing the ceilings on the side corridors and a moderately dark environment map. The light filtering through the gaps around the curtains are challenging to capture properly. *Kitchen* is lit by upward facing spotlights located inside the lamps and an area light under the shelf.

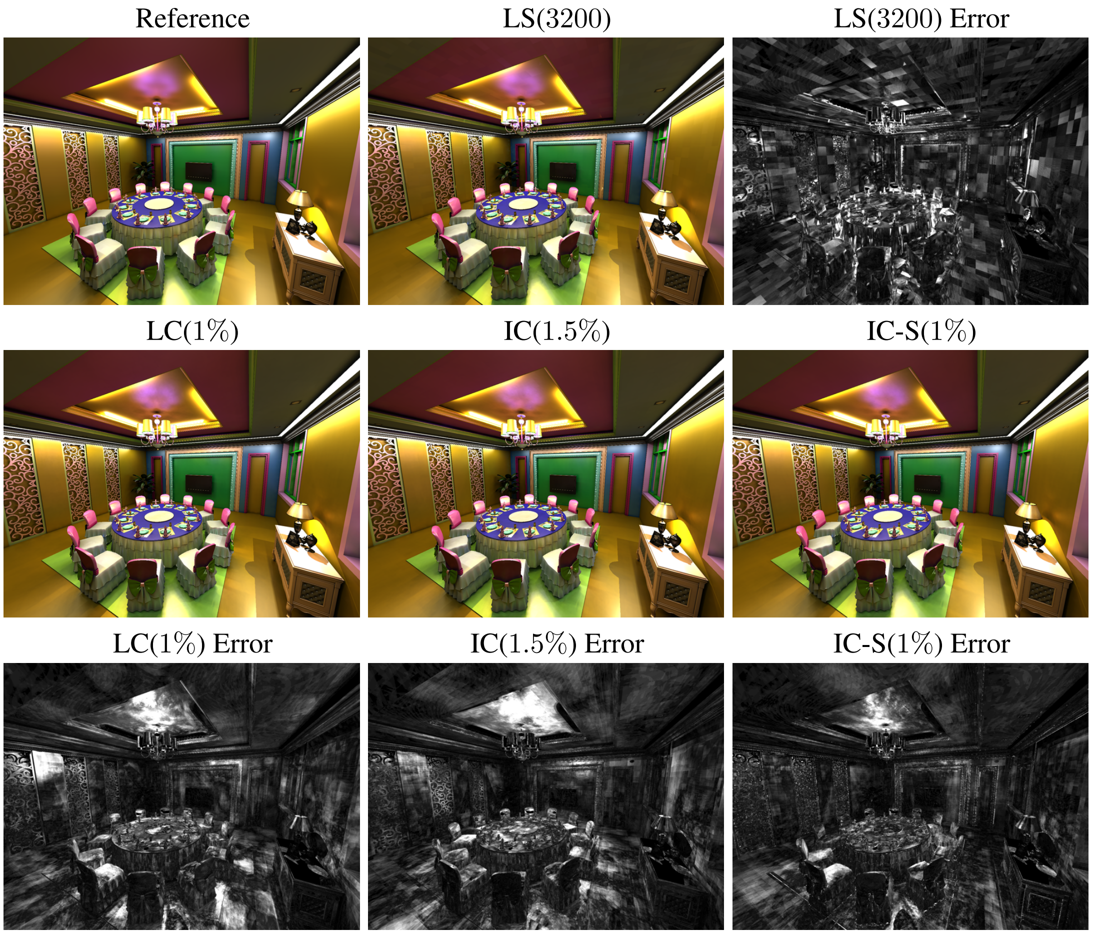


FIGURE 3.3: Banquet scene rendered with the 4 methods (LC,LS,IC,IC-S) with error images for 1600×1200 resolution with 1 samples per pixel.

Parameters. For Lightcuts the error bound is set to 1% (as in earlier work [OP11]). We give the results of both IlluminationCut and IlluminationCut-Sampling, setting the error to 1.5% in the former case and to 1% in the latter. LightSlice is run with approximately 1500 slices and with varying columns. The number of slices determines the size of the reduced light transport matrix while the number of columns determines the number of clusters used for rendering a point. For the sake of compactness we refer to these algorithms in the figures as LC(1%), LS(3200), IC(1.5%) and IC-S(1%). The images have 1600×1200 resolution and use 1 sample per pixel for a clear comparison of the quality of clusterings obtained by the different methods. We give a second table with comparison for anti-aliased images with 9 samples per pixel to show how the methods behave in this case. For each scene, around 650K VPLs are generated by tracing light paths from the original light sources up to depth 10. Our implementation uses clamping by setting the point-VPL distance not smaller than 5% of the scene radius. Due to the consequent energy loss, we use the image rendered with all VPLs as our reference image.

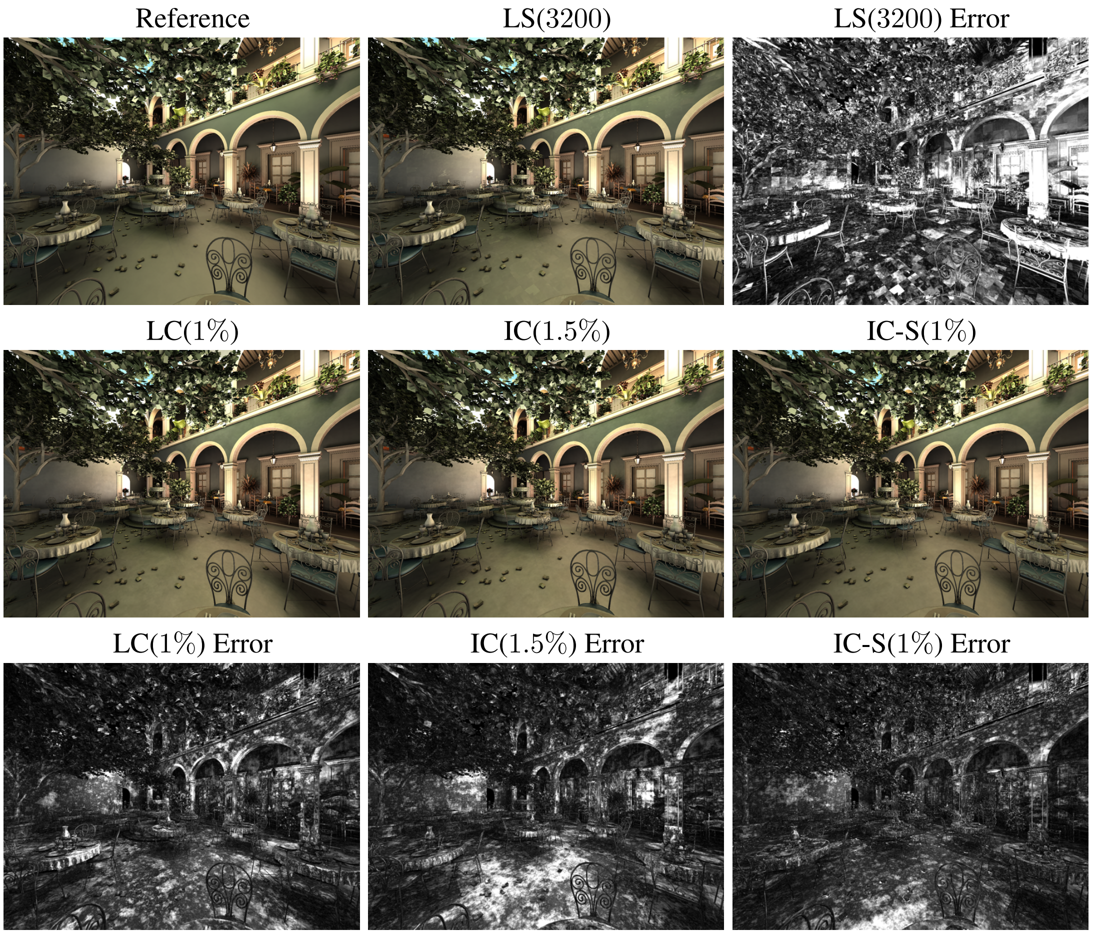


FIGURE 3.4: San Miguel scene rendered with the 4 methods (LC,LS,IC,IC-S) with error images for 1600×1200 resolution with 1 samples per pixel.

Performance. Table 3.1 shows the results with various statistics for 1 sample per pixel. Figure 3.3, 3.4, 3.5 and 3.6 contain the rendered images and error images for the four scenes. We set the error bound of Lightcuts to 1% and adjusted the other methods (error bound or column number) to provide similar *RMSE* quality. We note that all algorithms exhibit small variations in quality due to randomness. We provide the running times for the preprocessing and the rendering phase. For our method and Lightcuts, the preprocessing consists of building the light tree while for LightSlice there is no view independent preprocessing phase. The normalized *RMSE* and average relative error provides numerical difference against the VPL reference image. Both our method and Lightcuts are using similar error upper bound calculations. In order to compare the reduction of such calculations we have included in the table the number of upper bound calculations averaged over the number of pixels. We also give average number of shadow rays per pixel. Note that these are not identical, since lights facing away are not tested and additional shadow rays are traced in other parts of the algorithms. The latter happens for Phase I and for building the reduced light transport matrix in LightSlice. The error images are calculated by taking the channel-wise Euclidean distance between the image, and the VPL reference image,

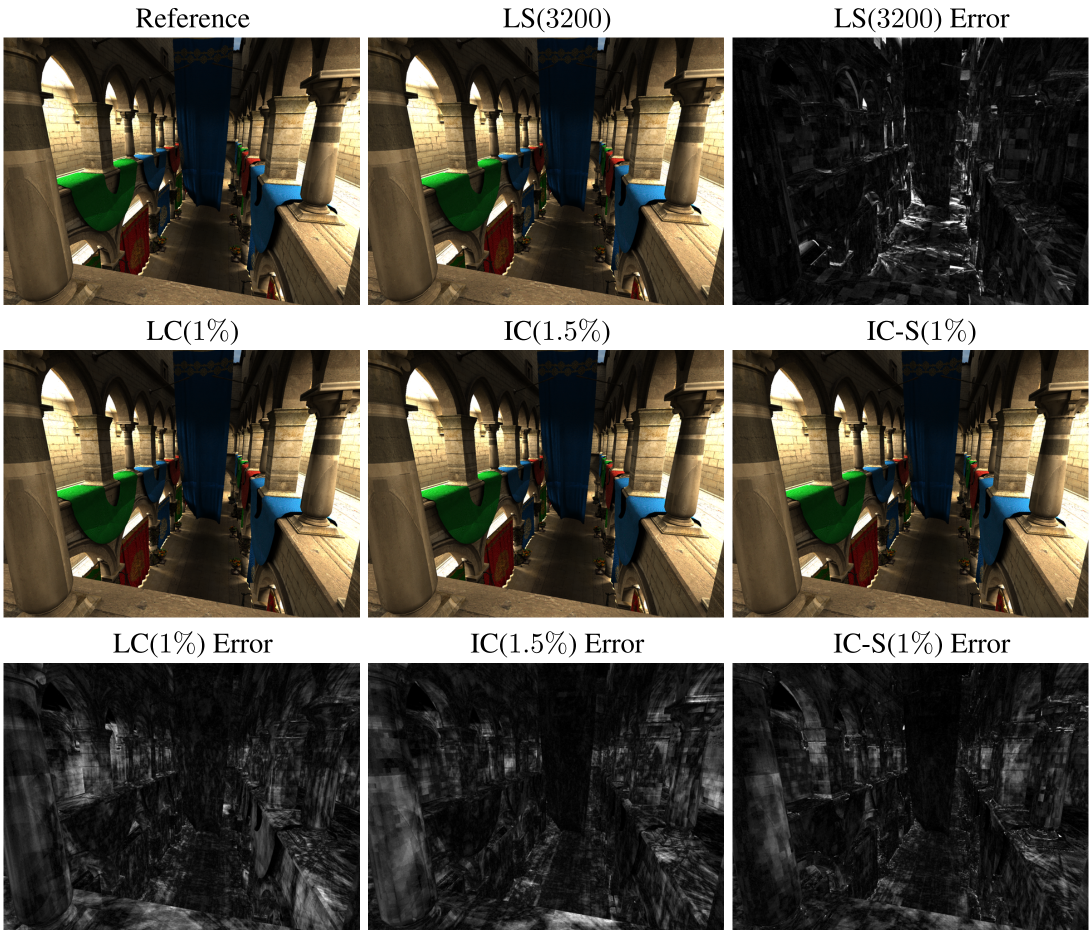


FIGURE 3.5: Sponza scene rendered with the 4 methods (LC,LS,IC,IC-S) with error images for 1600×1200 resolution with 1 samples per pixel.

and multiplying it by a factor of 16. For LightSlice we report in parenthesis the time to cluster the reduced matrix because that is single threaded and it is a significant part of the rendering causing the main bottleneck of the algorithm. We also include highlights of typical errors (see Figure 3.7).

Comparison to Lightcuts. In general, the quality of our results is similar to Lightcuts with $3-6$ times speedup. Both methods adapt well to the scenes, keeping the error low with the upper bounding methods and both methods oversample shadowed areas. The only visible artifacts are the non smooth shading of uniform surfaces, e.g., in *San Miguel*. Note that as Embree is a high performance ray tracing kernel we gain speed-up by significantly reducing the cost of clustering. In Figure 3.8 we illustrate the number of upper bound calculations for each point $p \in \mathcal{P}$. The images are using false coloring with a logarithmic scale. In Lightcuts, for each p we add 1 at each upper bound calculated while shading the point. In IlluminationCut, when an upper bounding calculation is carried out for an illumination-aware cluster we add to each p in the cluster the inverse of the number of points in the cluster. This shows the amortized cost of

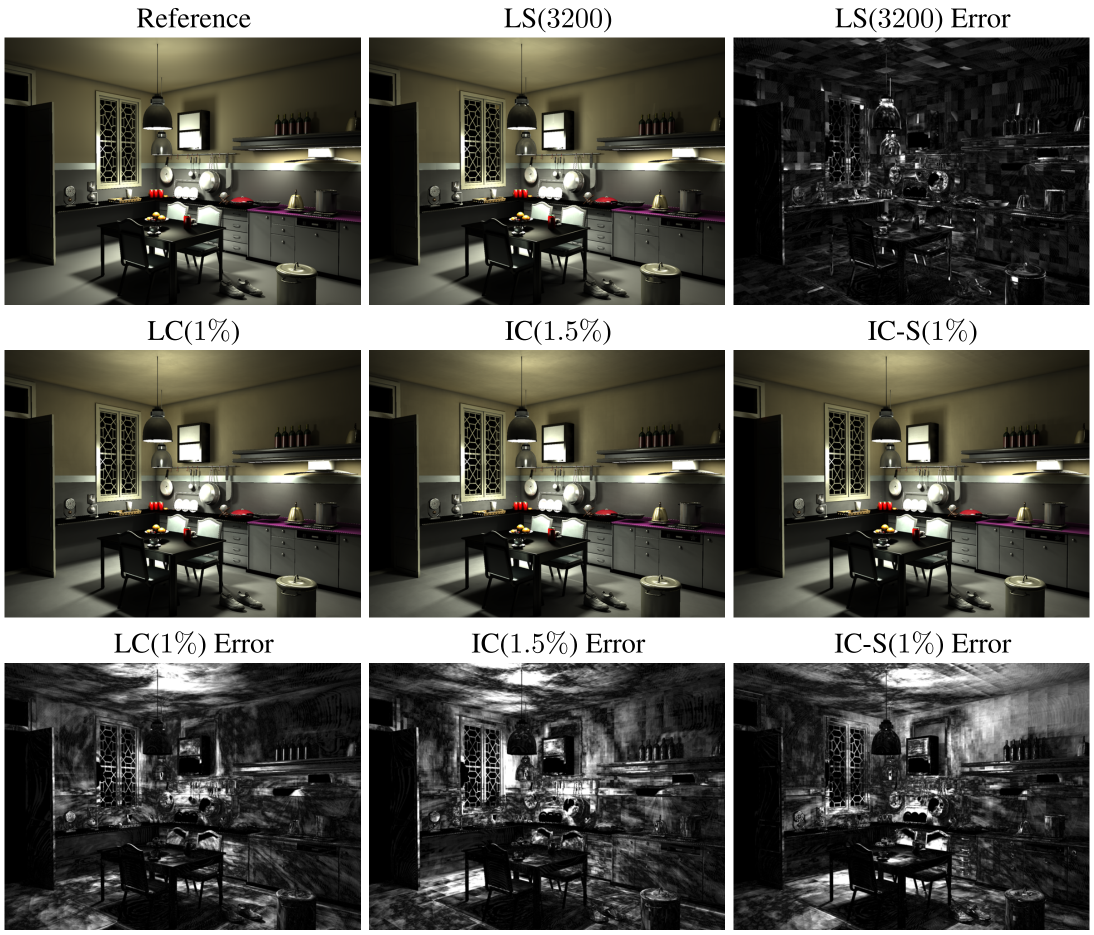


FIGURE 3.6: Kitchen scene rendered with the 4 methods (LC,LS,IC,IC-S) with error images for 1600×1200 resolution with 1 samples per pixel.

clustering and it is consistent since an upper bounding calculation still increases the total value by 1, just as in Lightcuts.

Comparison to LightSlice. LightSlice performs less efficiently in our highly glossy environments. Its ability to explore the structure of VPLs and to adapt to occluded lights reduces the number of rays traced. But this comes with a cost. Clustering the reduced matrix becomes the bottleneck and clustering the points into slices causes blocking artifacts, especially on glossy surfaces. The method fails to handle complex lighting situations, since using randomly sampled representatives easily miss important details. Consider, e.g., *Sponza*, where it is unable to calculate a good shading for the floor in a reasonable time while the other parts of the image are very close to the reference.

Maximum cut size. The original Lightcuts method sets the maximum cut size to 2000. We now present the results of Lightcuts with this additional constraint. For *Banquet* the results remained basically unchanged. For *San Miguel* the rendering time becomes similar to our method

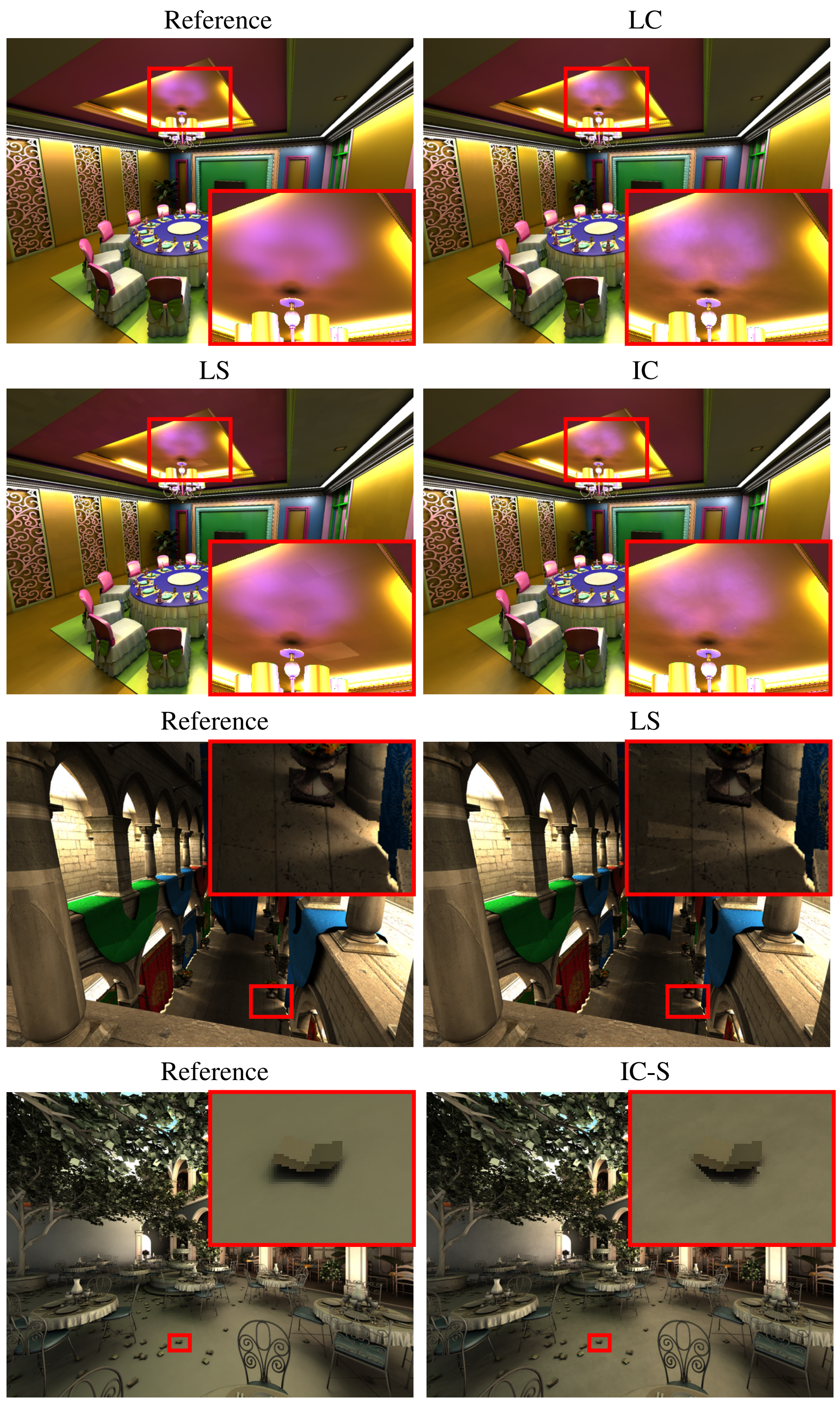


FIGURE 3.7: Typical errors of the four methods (LC, LS, IC, IC-S). The first two rows show that LC and IC both fail to reproduce smooth color gradients in complex illumination (the pink color is the result of light reflected on the lamps). LS performs better but it introduces blocking artifacts. Sponza shows that if within a point-cluster the radiance of points changes drastically LS has no means to adapt to it. San Miguel demonstrates that IC-S might fail to detect small shadows.

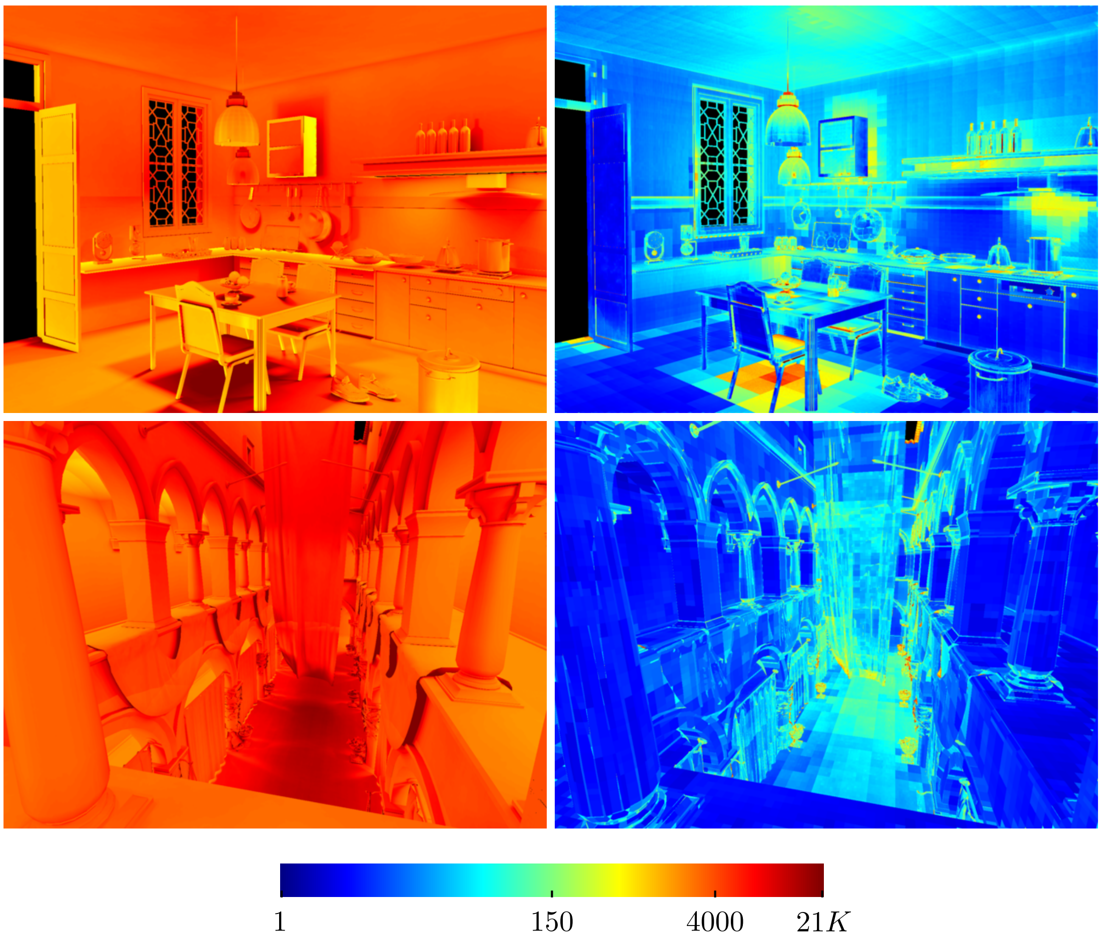


FIGURE 3.8: The logarithm of the number of upper bound calculations per point for Kitchen and Sponza for LC (left) and IC (right). IC amortizes the cost of upper bounding calculations very efficiently but suffers from descending too deep in the tree for dark areas, just as Lightcuts.

but the quality is significantly worse. For *Sponza* this results in degradation of quality while for *Kitchen* the quality remains unchanged. We believe that maximum cut size is effective mainly if one can properly set it prior to rendering a scene. In Table 3.2 we included results with setting the maximum cut size to 2000.

	Banquet	San Mig.	Sponza	Kitchen
R. time (s)	103.88	175.54	99.34	82.47
RMSE	0.01066	0.02548	0.00632	0.01581
Rel. Err.	3.04835	8.95983	5.16506	10.10555

TABLE 3.2: Lightcuts with maximum cut size set to 2000.

High fidelity images. We have set our error threshold for Lightcuts to 1% which closely matches the parameters used in previous work [WFA+05; OP11]. Despite that, the quality of our images are not matching the reference. In order to show that our methods are capable of producing nearly indistinguishable results we have set more strict error thresholds. See the result for *Banquet* in Table 3.3. Note that we still maintain our speed up over Lightcuts while the quality is the same. LightSlice converges very slowly to the correct solution if one only varies the number of columns. On the other hand, using more slices requires prohibitively large memory.

	LC(0.1%)	LS(6400)	IC(0.15%)	IC-S(0.1%)
R. time (s)	602.98	720.46	180.57	178.83
RMSE	0.00315	0.01490	0.00319	0.00425
Rel. Err.	1.12375	2.37685	1.08088	1.18171

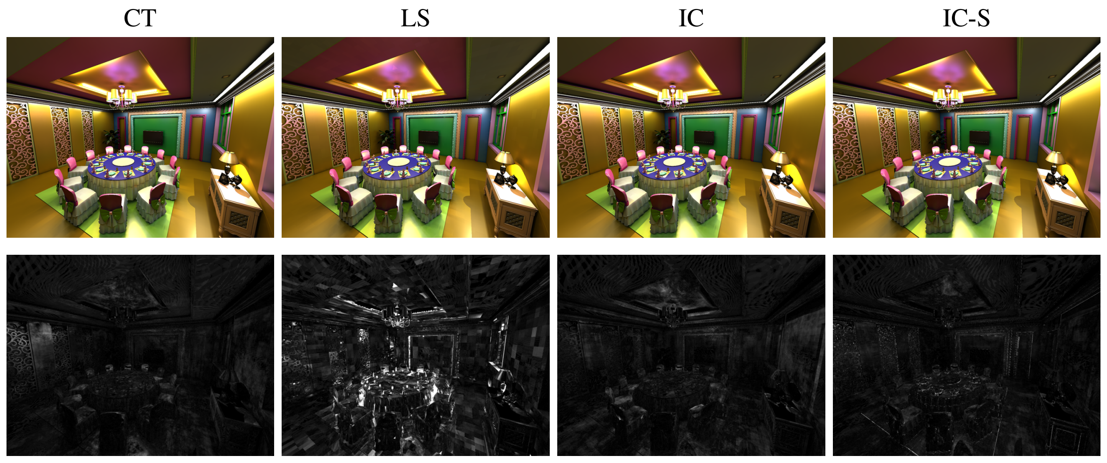


TABLE 3.3: Results for Banquet where the images are visually indistinguishable from the reference.

Speed up. The ray tracing kernel used in our implementation is more highly optimized than the renderer (see [WWB+14] for details on the ray tracing kernel). Therefore the speed up achieved by the method is less if these two components are similarly efficient. In order to have a more objective comparison we have measured the proportion of different components in our implementation of Lightcuts. Approximately 20% is spent on ray tracing, 60% on upper bounding computations and the remaining 20% is spent on shading and heap maintenance. The upper bounding calculations in our method are only a fraction of Lightcuts', therefore we achieve on average 3 times speedup over Lightcuts.

Memory. See Table 3.4 for the peak memory consumption (in GB) of the four algorithms LC(1%), LS(800), IC(1.5%), IC-S(1%) run on *Banquet* with 1 sample per pixel and 1600×1200 resolution. Lightcuts is the most efficient on memory consumption, followed by our method. For the latter the point tree consumes most memory, scaling linearly with the number of points (e.g., 9 samples per pixels would require extra 8 GB of memory). We note that in order to alleviate the memory consumption one could partition the tree into a few subtrees and process them independently. LightSlice, due to the light transport matrix storage (the size of it is the number of slices times the number of VPLs), has a very high memory consumption.

# VPLs:	50K	300K	600K	1.2M
LC(1%)	0.31	0.62	1.02	1.74
LS(800)	2.95	14.18	29.53	55.10
IC(1%)	1.35	1.65	2.24	2.70

TABLE 3.4: Peak memory requirements for Banquet (in GB).

Scalability. In Figure 3.9 we plot the rendering times of four methods (LC(1%), LS(1600), IC(1.5%), IC-S(1%)) with varying number of VPLs for *Banquet* (1 sample per pixel, 1600 × 1200 resolution). Our method consistently outperforms Lightcuts and LightSlice. The rendering times of both LC and IC are sub-linear in the number of VPLs. For LS, above a certain number of VPLs, the construction and clustering of the reduced light transport matrix becomes the dominant cost, therefore scaling approximately linearly.

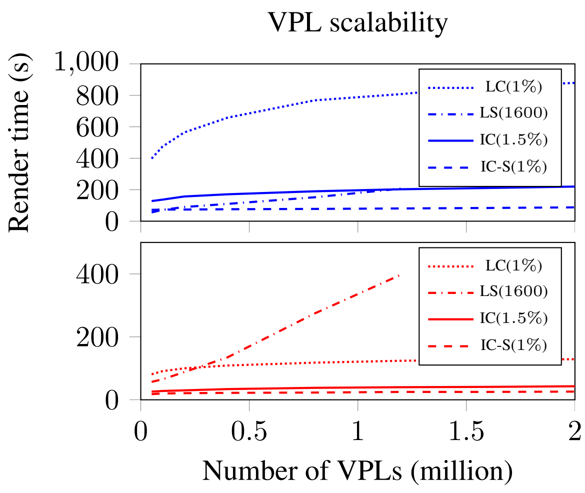


FIGURE 3.9: Render times with varying number of VPLs for San Miguel (top) and for Banquet (bottom).

Anti-aliasing. We have found that increasing the number of samples increases the quality of the images rendered with LightSlice. Given a VPL-cluster, the method uses different representative lights of the light cluster for shading the different samples in a pixel. See Table 3.5, Figure 3.10, 3.11, 3.12 and 3.13 for the results of the methods with 9 samples per pixel. This technique smooths the errors, thus enables LightSlice to achieve better quality with only 800 columns, improving its rendering time significantly. For *Kitchen*, LightSlice now outperforms our algorithm but it is still not well-suited for highly glossy environments (e.g., *Banquet*) and challenging illumination (e.g., *Sponza*). In these cases IlluminationCut-Sampling performs better. Our method can be further enhanced for the case of multiple samples per pixel, in a similar way as Multidimensional Lightcuts, by limiting the number of traced rays and shadings. Namely, in each illumination-aware pair use only a single representative for the points originating from the same pixel. This would result in fewer visibility queries but likely increase the error on surfaces with non-uniform textures.

		Banquet	San Miguel	Sponza	Kitchen
LC(1%)	Preproc. time (s)	53.70	92.53	82.36	68.16
	Render time (s)	1031.02	6665.94	2007.74	1522.25
	RMSE	0.010991	0.012007	0.006238	0.012434
	Rel. Error	2.543368	2.681210	3.279278	9.397545
LS(800)	Render time (s)	346.61	399.48	332.62	221.34
	RMSE	0.016470	0.026163	0.005793	0.008920
	Rel. Error	4.010837	6.959285	2.938075	8.923159
	Speedup	3.0	16.7	6.0	6.9
IC(1.5%)	Preproc. time (s)	55.14	95.35	84.90	70.48
	Render time (s)	281.11	1567.65	665.01	360.33
	RMSE	0.008770	0.011267	0.005986	0.011585
	Rel. Error	2.280708	2.772934	2.815512	9.152297
	Speedup	3.7	4.3	3.0	4.2
IC-S(1%)	Preproc. time (s)	55.20	95.21	83.57	70.10
	Render time (s)	117.58	444.25	143.87	170.65
	RMSE	0.008096	0.010349	0.005480	0.012178
	Rel. Error	2.257898	2.686910	2.791644	9.766247
	Speedup	8.8	15.0	14.0	8.9

TABLE 3.5: Statistics for images of 1600×1200 resolution with 9 samples per pixel.

3.4 Limitations

We have presented an implementation of a flexible and efficient framework handling highly glossy materials. It is several times faster than Lightcuts and has similar speedup as LightSlice while guaranteeing low perceptual error which can be set a priori to rendering.

The usage of an octree for the point tree can cause blocking artifacts, however with low error threshold these disappear. The approximate image used in our algorithm has errors and it is not progressively refined (contrary to Lightcuts), thus it might introduce errors in Phase II. This effect, however, is unnoticeable since even a 100% error in the approximate image only results in at most 1% additional error (per cluster) in the final image. IlluminationCut reduces calculations by exploiting the similarity of points, therefore it is less efficient for scenes where the shaded points have highly varying properties (e.g., heterogeneous BRDFs or spatial incoherence). Our current implementation requires the BRDFs to belong to a family with a low number of parameters since otherwise bounding these parameters for nodes of the octree would become prohibitively expensive.

The construction method of the point tree is not important (to some extent) for Algorithm 4 to extract pairs of clusters. Thus one step further would be to utilize a complex metric for the point tree, e.g., incorporating material properties as well. Such a strategy could be useful in a scene with highly varying materials since it would enable tighter error bounds for the individual clusters.

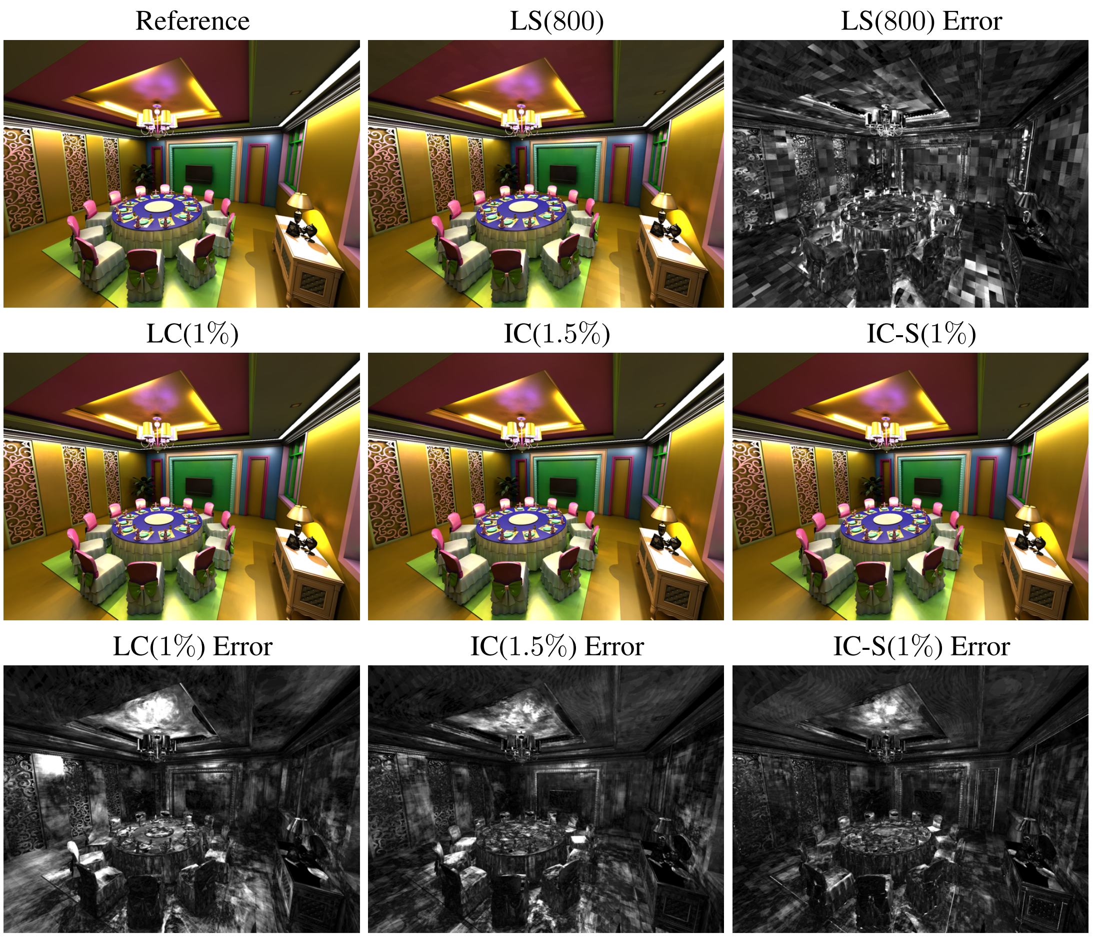


FIGURE 3.10: Banquet scene rendered with the 4 methods (LC,LS,IC,IC-S) with error images for 1600×1200 resolution with 9 samples per pixel.

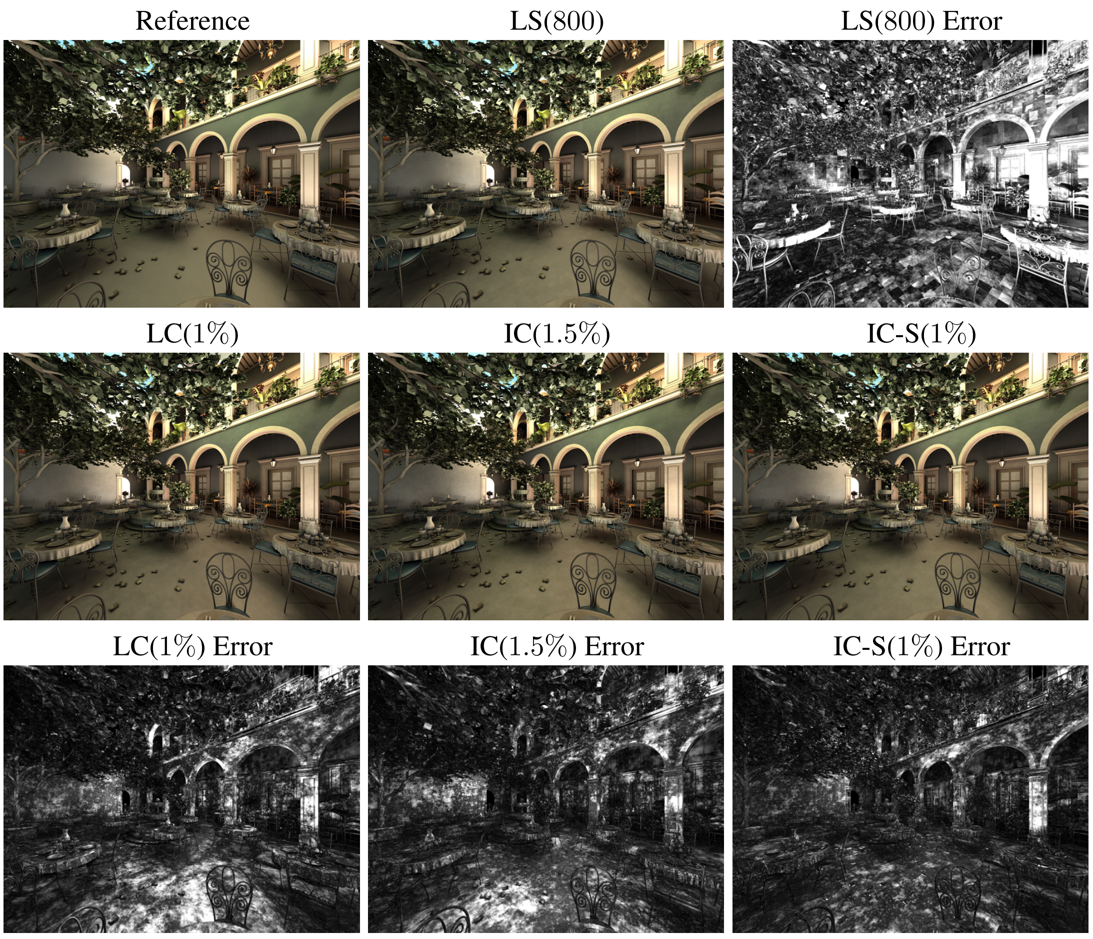


FIGURE 3.11: San Miguel scene rendered with the 4 methods (LC,LS,IC,IC-S) with error images for 1600×1200 resolution with 9 samples per pixel.

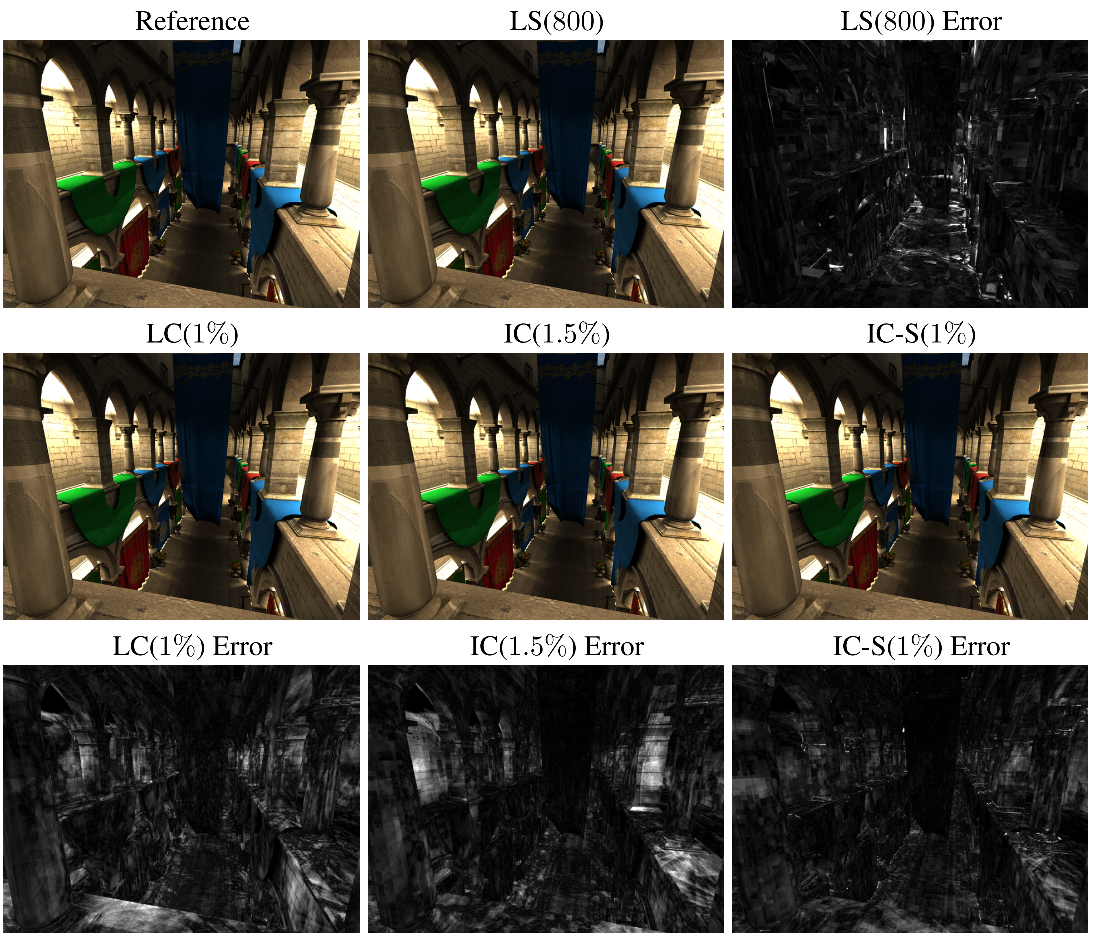


FIGURE 3.12: Sponza scene rendered with the 4 methods (LC,LS,IC,IC-S) with error images for 1600×1200 resolution with 9 samples per pixel.

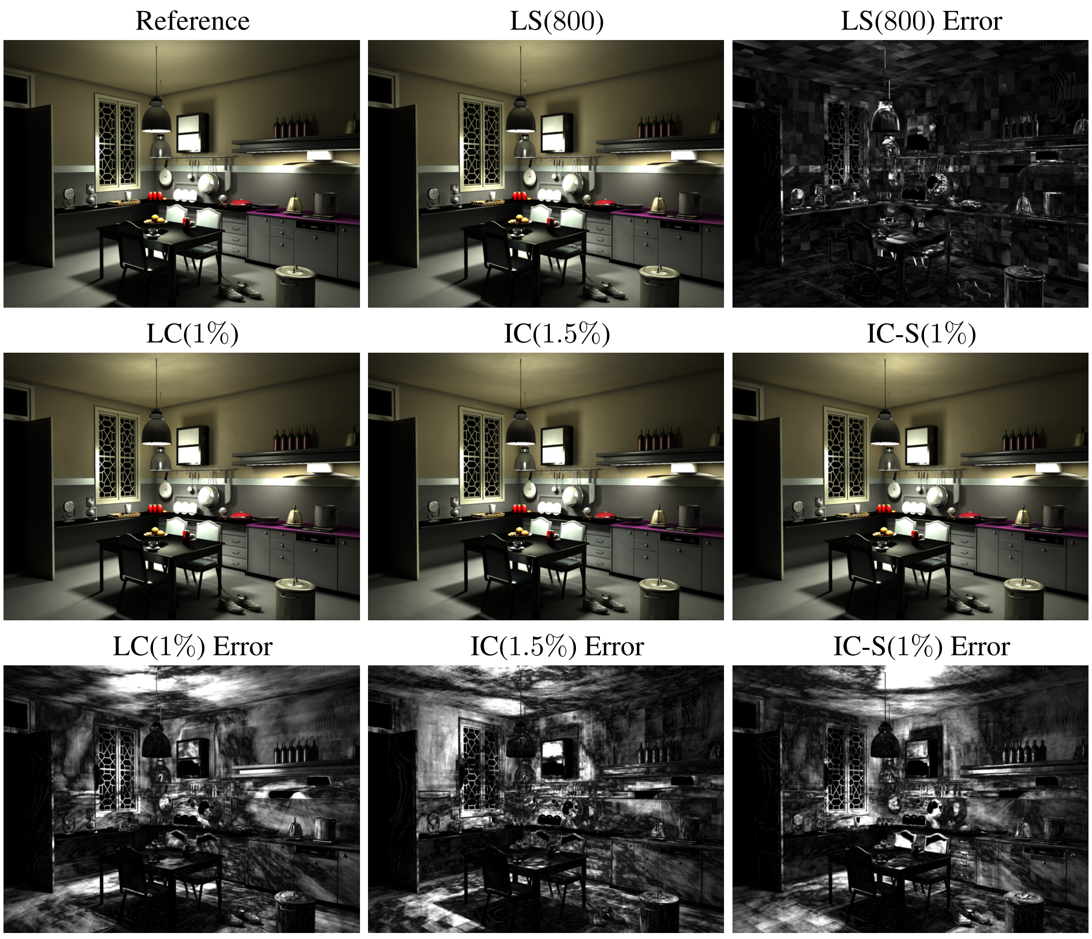


FIGURE 3.13: Kitchen scene rendered with the 4 methods (LC,LS,IC,IC-S) with error images for 1600×1200 resolution with 9 samples per pixel.

Chapter 4

Software

In our work we have introduced novel algorithms for computing global illumination in the many-lights setup and provided comparisons with state-of-the-art techniques. In order to report appropriate data and to do a fair comparison it is required that all methods share the same framework and that this framework is representative of recent professional rendering environments. In order to fulfill these requirements we have chosen a relatively simple yet fully functional rendering framework with state-of-the-art ray tracing kernels, namely the Embree rendering framework [WWB+14]. This system is capable of creating high quality images, see Figure 4.1. We have implemented (or if the code was available simply ported) all algorithms in this environment.

FIGURE 4.1: Images rendered with the Embree framework [WWB+14].

The Embree framework consists of two main parts. High performance ray tracing kernels and a simple but fully functional rendering engine with path tracing. The former was designed with a simple low level interface for ray tracing enabling any existing rendering application to simply use the ray tracing functionalities and benefit from its extreme speed. Listing 4.1 and 4.2 shows a few examples of using the Embree API.

```
unsigned geomID = rtcNewTriangleMesh(scene, geomFlags, numTriangles, ↩
    numVertices);

struct Vertex   { float x, y, z, a; };
struct Triangle { int v0, v1, v2; };

Vertex* vertices = (Vertex*) rtcMapBuffer(scene, geomID, RTC_VERTEX_BUFFER)↩
    ;
// fill vertices here
rtcUnmapBuffer(scene, geomID, RTC_VERTEX_BUFFER);

Triangle* triangles = (Triangle*) rtcMapBuffer(scene, geomID, ↩
    RTC_INDEX_BUFFER);
// fill triangle indices here
rtcUnmapBuffer(scene, geomID, RTC_INDEX_BUFFER);
```

LISTING 4.1: The Embree API: mesh creation

```
void rtcIntersect   ( RTCScene scene, RTCRay& ray);
void rtcOccluded    ( RTCScene scene, RTCRay& ray);
```

LISTING 4.2: The Embree API: ray - primitive intersection

The ray tracing kernels provide efficient intersection methods by using state-of-the-art acceleration structures (e.g., bounding volume hierarchies built using SAH [Wal07]), packeted ray tracing and low level optimization using vector instructions (both SSE and AVX). It's performance is comparable to state-of-the-art GPU ray tracing systems. The rendering engine was designed to provide a realistic environment for testing the kernel's performance for professional rendering. It only includes a path tracing implementation which is completely enough for our purposes. The renderer provides SIMD accelerated computation of vector arithmetics but does not offer parallel shading capabilities (in case this is needed it provides a renderer written in the Intel SPMD framework, a SIMD extension of the C language).

The clear and simple design of Embree enabled us to easily integrate different many-lights methods into one package enabling our ultimate goal of accurate comparison. In what follows we will give a brief description of the included algorithms. Due to the clamping of VPLs in the many-lights methods we have used a simple exhaustive rendering as the reference image. The included algorithms are Lightcuts [WFA+05], LightSlice [OP11] and our work, Illumination-Cut [BMB15b].

The source code of our project can be found on the author's website. We will here highlight some decisions and crucial parts of the algorithm. The code contained in this document is a subset of the real code only for presentation purposes.

Lightcuts

We have implemented the Lightcuts algorithm but additionally we include the implementation provided by the authors of LightSlice. Our implementation performs slightly better in the Embree framework.

We have used a heap-less agglomerative clustering algorithm for building the lighttree [WBK+08] using a fast customized kd-tree [Mik10]. See the Listing 4.3 where the kd-tree query returns the closest node using a custom metric (the one described in the original Lightcuts paper) for clustering the lights. This results in a slower construction but a better quality tree than the one used by the authors of LightSlice (built with divisive clustering based on spatial and directional splits). We have decided to use this method since the better quality tree resulted in an improvement of the rendering time.

```
Node* ClusterLightCuts::buildTreeFromLeavesLocal()
{
  // Fast Agglomerative Clustering for Rendering, Walter et al, 2008
  // Locally-ordered agglomerative clustering
  std::vector<Node*> clusters ; clusters.resize(numNodes);

  // create the initial clusters, corresponding to single vpls
  ...

  // construct kdtree
  KdTree<Node> kdtree(this->sceneradiusSqr);
  kdtree.constructTree(clusters,numCurrentClusters);

  Random rnd(RND_SEED);

  Node* na = clusters[0];
  Node* nb  = kdtree.queryNN(na);

  while ( numCurrentClusters  <  numNodes)
  {
    Node* nc = kdtree.queryNN(nb);

    if ( na == nc )
    {
      na->valid = false;
      nb->valid = false;
      Node* naOld = na;
      na = na->merge(nb,rnd);
      kdtree.invalidateNodesAndUpdate(naOld,nb,na);
      clusters[numCurrentClusters++] = na;
      nb = NULL;
      nb = kdtree.queryNN(na);
    } else {
      na = nb;
      nb = nc;
    }
  }
  Node* root = clusters[numNodes - 1];
  return root;
}
```

LISTING 4.3: Heap-less agglomerative clustering

For ensuring that the rendering phase is efficient we store the lighttree nodes in an array with a copy of the representative lights, therefore enforcing spatial locality of the nodes in memory and avoiding additional pointer indirection compared to only storing pointers to the representative lights. In Listing 4.4 we included the definition of our node class. We could have achieved a smaller memory footprint by separating the variables only needed for construction but in order to have a more readable code we have decided against it. A crucial part of the Lightcuts algorithm is the maintenance of a heap. In order to speed up this part we use thread safe preallocated memory space avoiding costly memory re-allocation for each cut.

```
/*! \brief Node for lightcuts tree*/
class Node
{
public:
  // member functions
  ...

  vplVPL repLights[REP_LIGHT_NUM];          //!< representative lights
  BBox3f cell;                              //!< bounding box
  Cone cone;                                //!< cone of directions
  Node* lc, *rc, *parent;                   //!< children and parent
  KdNode<Node>* kdnode;                     //!< kd-node for tree ↩
    creation
  bool valid;                               //!< used for tree creation
  bool repLightFromRightChild[REP_LIGHT_NUM];  //!< with which child is the↩
    representative light shared
};
```

LISTING 4.4: Lighttree node

LightSlice

We have used the code published by the authors of LightSlice. The only modification we have done is that we have adjusted it to use our framework. The code was downloaded from the authors' website.

IlluminationCut

Since IlluminationCut shares data structures and algorithms with Lightcuts we will only give some details regarding the difference, namely the pixeltree. The pixeltree was simply built by a divisive algorithm to minimize the construction time, repeatedly partitioning the set of pixels into two subsets. Each node corresponds to a group of pixels which we store in an array to have a fast access to them for shading. In order to avoid copies of the same pixels there is only one array storing all pixels and each node keeps only the beginning and ending pointer of a continuous range of this array. This is a well-known method to store a binary tree in an array. It is achieved by recursively partitioning the array, see Listing 4.5.

```
class icPixelNode
{
public:
  enum { nChildren = 2 };

  icPixelNode* parent;                 //< pointer to parent node
  icPixelNode* children[nChildren];    //< pointer to children, null if there ↩
    is no child
  BBox6f cell;                         //< bbox of the pixels in this node
  icPixel* pixel;                      //< for leaf node the pixel, otherwise ↩
    random representative
  icPixel* start;                      //< pixel range start
  icPixel* end;                        //< pixel range end
  Color maxKd, maxKs;                  //< max material coeffs
  float maxExp, minExp;                //< spec exp for ub (min is for power ↩
    of a value < 1)
    Color color;                         //< luminance used for storing approx↩
     image and ub
};
```

```

void icPixeltree::createOctreeNonRecursiveAndCompress()
{
  // create root
  this->root = ...

  std::stack<icPixelNode*> nodesToProcess;
  nodesToProcess.push(this->root);
  Random rnd(RND_SEED);

  while ( !nodesToProcess.empty() )
  {
    // get next node to process
    icPixelNode* node = nodesToProcess.top(); nodesToProcess.pop();

    // put the lights in the proper child (bb)
    size_t dim ;
    float separator;
    node->getNextDividingPlanePosition(separator,dim);
    icPixel* middle;
    separatorPredicate sp(separator,dim);
    middle = std::partition(node->start,node->end,sp);

    if (middle==node->start || middle==node->end || isLeaf(node) )
    {
      ...
    }
    // create two new children
    ...

    node->children[0]->start = node->start;
    node->children[0]->end = middle;
    node->children[1]->start = middle;
    node->children[1]->end = node->end;

    // create subtrees for children
    for (size_t i = 0; i < nChildren; i++ )
      nodesToProcess.push(nrs[i]);
  }
}
```

LISTING 4.5: Pixeltree node and in-place storage of pixels

Part II

Combinatorial Optimization

Chapter 5

Overview of the Hitting Set Problem

The minimum hitting set problem is one of the most fundamental combinatorial optimization problems: given a range space $(P, \mathcal{D})$ consisting of a set P and a set $\mathcal{D}$ of subsets of P called the *ranges*, the task is to compute the smallest subset $Q \subseteq P$ that has a non-empty intersection with each of the ranges in $\mathcal{D}$. This problem is strongly NP-hard. If there are no restrictions on the set system $\mathcal{D}$, then it is known that it is NP-hard to approximate the minimum hitting set within a logarithmic factor of the optimal [RS97]. The problem is NP-complete even for the case where each range has exactly two points since this problem is equivalent to the vertex cover problem which is known to be NP-complete [Kar72; GJ79].

The special case studied in this work – hitting sets for disks in the plane – has been the subject of a long line of research. Besides ad-hoc approaches, there are two systematic lines along which all progress on the hitting-set problem for geometric ranges has relied on: rounding via ϵ-nets, and local-search. Both these approaches have to be evaluated on the questions of computational efficiency as well as approximation quality. In spite of all the progress, there remains a large gap between theory and practice – mainly due to the ugly trade-offs between running times and approximation factors.

Rounding via ϵ-nets. Given a range space $(P, \mathcal{D})$, a positive measure μ on P (e.g., the counting measure), and a parameter $\epsilon > 0$, an ϵ-net is a subset $S \subseteq P$ such that $D \cap S \neq \emptyset$ for all $D \in \mathcal{D}$ with $\mu(D \cap P) \geq \epsilon \cdot \mu(P)$. The famous "$\epsilon$-net theorem" of Haussler and Welzl [HW87] states that for range spaces with VC-dimension d, there exists an ϵ-net of size $O(d/\epsilon \log d/\epsilon)$ (this bound was later improved to $O(d/\epsilon \log 1/\epsilon)$. The condition of having finite VC-dimension is satisfied by many geometric set systems: disks, half-spaces, k-sided polytopes, r-admissible set of regions etc. in $\mathbb{R}^d$. For certain range spaces, one can even show the existence of ϵ-nets of size $O(1/\epsilon)$ – an important case being for disks in $\mathbb{R}^2$ [PR08].

In 1994, Bronnimann and Goodrich [BG95] proved the following interesting connection between the hitting-set problem [1], and ϵ-nets: let $(P, \mathcal{D})$ be a range-space for which we want to compute a minimum hitting set. If one can compute an ϵ-net of size c/ϵ for the ϵ-net problem for $(P, \mathcal{D})$ in polynomial time, then one can compute a hitting set of size at most $c \cdot \textsc{Opt}$ for $(P, \mathcal{D})$, where $\textsc{Opt}$ is the size of the optimal (smallest) hitting set, in polynomial time. Until very recently, the best such rounding algorithms had running times of $\Omega(n^2)$, and it had been a long-standing open problem to compute a $O(1)$-approximation to the hitting-set problem for disks in the plane in near linear time. In a recent break-through, Agarwal-Pan [AP14] presented the first near-linear time algorithm that is able to do the required rounding for a broad set of geometric objects.

Bounds on ϵ-nets. The result of Agarwal-Pan [AP14] opens the way, for the first time, for near linear-time algorithms for the geometric hitting set problem. The catch is that the approximation factor depends on the sizes of ϵ-nets for disks. So far, the best constants for the ϵ-nets come from the proofs in [PR08] and [HKS+14]. These constructions give a bound that is at least $24/\epsilon$ and most likely more than $40/\epsilon$. Furthermore, there is no implementation or software solution available that can even compute such ϵ-nets efficiently.

Local search. For the hitting set problem on $(P, \mathcal{D})$, consider the following algorithm: start with any hitting set $S \subseteq P$, and repeatedly decrease the size of S, if possible, by replacing k points of S with $< k$ points of $P \setminus S$. Call such an algorithm a $(k, k-1)$-local search algorithm. Mustafa-Ray [MR10] showed that a $(k, k-1)$-local search algorithm for the hitting set problem gives a $(1 + c/\sqrt{k})$-approximation, for a fixed constant c, when the ranges are disks, or more generally, pseudo-disks in $\mathbb{R}^2$. The running time of their algorithm to compute a $(1+\epsilon)$-approximation is $O(n^{O(1/\epsilon^2)})$.

[1]They actually proved a more general statement, but the following is more relevant for our purposes.

Chapter 6

Tighter Estimates for ϵ-nets for Disks

This chapter describes a near linear algorithm to construct small sized ϵ-nets for disks. It is based on the following paper.

Norbert Bus, Shashwat Garg, Nabil H. Mustafa, and Saurabh Ray. "Tighter Estimates for epsilon-nets for Disks." In: *Computational Geometry: Theory and Applications* 53, 2016

6.1 A Near Linear Time Algorithm for Computing ϵ-nets for Disks in the Plane

We prove new improved bounds on sizes of ϵ-nets and present efficient algorithms to compute such nets. Our approach is simple: we will show that modifications to a well-known technique for computing ϵ-nets – the sample-and-refine approach of Chazelle-Friedman [CF90] – together with additional structural properties of Delaunay triangulations in fact results in ϵ-nets of surprisingly low size:

Theorem 6.1. *Given a set P of n points in $\mathbb{R}^2$, there exists an ϵ-net under disk ranges of size at most $13.4/\epsilon$. Furthermore it can be computed in expected time $O(n \log n)$.*

Together with the result of Agarwal-Pan, this immediately implies the following:

Corollary 6.2. *For any $\delta > 0$, one can compute a $(13.4 + \delta)$-approximation to the minimum hitting set for $(P, \mathcal{D})$ in time $\tilde{O}(n)$.*

Through a careful analysis, we present the algorithm for computing an ϵ-net of size $\frac{13.4}{\epsilon}$, running in near linear time. The method, shown in Algorithm 6, computes a random sample and then solves certain subproblems involving subsets located in pairs of Delaunay disks circumscribing adjacent triangles in the Delaunay triangulation of the random sample. The key to improved bounds is $i)$ considering edges in the Delaunay triangulation instead of faces in the analysis, and $ii)$ new improved constructions for large values of ϵ.

Let $\Delta(abc)$ denote the triangle defined by the three points a, b and c. D_{abc} denotes the disk through a, b and c, while $D_{ab\bar{c}}$ denotes the halfspace defined by a and b not containing the point c. Let $c(D)$ denote the center of the disk D.

Let $\Xi(R)$ be the Delaunay triangulation of a set of points $R \subseteq P$ in the plane. We will use Ξ when R is clear from the context. For any triangle $\Delta \in \Xi$, let D_Δ be the Delaunay disk of Δ, and let P_Δ be the set of points of P contained in D_Δ. Similarly, for any edge $e \in \Xi$, let Δ_e^1 and Δ_e^2 be the two triangles in Ξ adjacent to e, and $P_e = P_{\Delta_e^1} \bigcup P_{\Delta_e^2}$. If e is on the convex-hull, then one of the triangles is taken to be the halfspace defined by e not containing R.

Algorithm 6: Compute ϵ-nets

```
Data: Compute $\epsilon$-net, given $P$: set of $n$ points in $\mathbb{R}^2$, $\epsilon > 0$ and $c_1$.
1  if $\epsilon n < 13$ then
2      Return $P$
3  Pick each point $p \in P$ into $R$ independently with probability $\frac{c_1}{\epsilon n}$.
4  if $|R| \leq c_1/2\epsilon$ then
5      restart algorithm.
6  Compute the Delaunay triangulation $\Xi$ of $R$.
7  for triangles $\Delta \in \Xi$ do
8      Compute the set of points $P_\Delta \subseteq P$ in Delaunay disk $D_\Delta$ of $\Delta$.
9  for edges $e \in \Xi$ do
10     Let $\Delta_e^1$ and $\Delta_e^2$ be the two triangles adjacent to $e$, $P_e = P_{\Delta_e^1} \cup P_{\Delta_e^2}$.
11     Let $\epsilon' = (\frac{\epsilon n}{|P_e|})$ and compute a $\epsilon'$-net $R_e$ for $P_e$ depending on the cases below:
12     if $\frac{2}{3} < \epsilon' < 1$ then
13         compute using Lemma 6.3.
14     if $\frac{1}{2} < \epsilon' \leq \frac{2}{3}$ then
15         compute using Lemma 6.4.
16     if $\epsilon' \leq \frac{1}{2}$ then
17         compute recursively.
18 Return $\left(\bigcup_e R_e\right) \cup R$.
```

In order to prove that the algorithm gives the desired result, the following theorems regarding the size of an ϵ-net will be useful. Let $f(\epsilon)$ be the size of the smallest ϵ-net for any set P of points in $\mathbb{R}^2$ under disk ranges.

Lemma 6.3 ([AAG14]). *For $\frac{2}{3} < \epsilon < 1$, $f(\epsilon) \leq 2$, and can be computed in $O(n \log n)$ time.*

Lemma 6.4. *For $\frac{1}{2} < \epsilon \leq \frac{2}{3}$, $f(\epsilon) \leq 10$ and can be computed in $O(n \log n)$ time.*

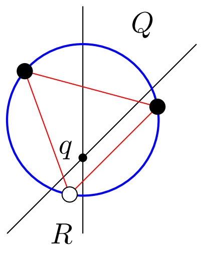


FIGURE 6.1: Setup around q.

Proof. Divide the plane into 4 quadrants with 2 lines, intersecting at a point q, such that each quadrant contains $n/4$ points. See Figure 6.1. Using the Ham-Sandwich theorem, this can be done in linear time [Mat02]. Create a $\frac{2}{3}$-net for each quadrant, using Lemma 6.3. Add these 8 points to the ϵ-net of P. If $q \in P$ then add q to the ϵ-net; otherwise let Δ be the triangle in the Delaunay triangulation of P that contains the point q. Add the two vertices of Δ that are in the opposite quadrants to the ϵ-net. The resulting size of the net is at most 10. Denote the quadrant without a vertex of the Delaunay triangle inside it by Q and its opposite quadrant by R. If a disk D intersects at most 3 quadrants and does not contain any of the points from the $\frac{2}{3}$-net in each of those quadrants, it can contain only at most $3 \cdot \frac{2}{3} \cdot \frac{n}{4} = \frac{n}{2}$ points. On the other hand, if D contains points from each of the 4 quadrants, then it must contain points from Q and R that are outside of the Delaunay disk D_Δ of Δ (as D_Δ is empty of points of P). Then if D does not contain any of the two vertices of Δ in the opposite quadrants (already added to the ϵ-net), it must pierce D_Δ, a contradiction. $\square$

Call a tuple $(\{p,q\},\{r,s\})$, where $p,q,r,s \in P$, a *Delaunay quadruple* if $int(\Delta(pqr)) \cap int(\Delta(pqs)) = \emptyset$. Define its *weight*, denoted $W_{(\{p,q\},\{r,s\})}$, to be the number of points of P in $D_{pqr} \cup D_{pqs}$. Let $\mathcal{T}_{\leq k}$ be a set of Delaunay quadruples of P of weight at most k and similarly $\mathcal{T}_k$ denotes the set of Delaunay quadruples of weight exactly k. Similarly, a *Delaunay triple* is given by $(\{p,q\},\{r\})$, where $p,q,r \in P$. Define its *weight*, denoted $W_{(\{p,q\},\{r\})}$, to be the number of points of P in $D_{pqr} \cup D_{pq\bar{r}}$. Let $\mathcal{S}_{\leq k}$ be a set of Delaunay triples of P of weight at most k, and $\mathcal{S}_k$ denotes the set of Delaunay triples of weight exactly k.

One can upper bound the size of $\mathcal{T}_{\leq k}$, $\mathcal{S}_{\leq k}$ and using it, we derive an upper bound on the expected number of sub-problems with a certain number of points.

Lemma 6.5. *$|\mathcal{T}_{\leq k}| \leq (e^3/9)nk^3$ asymptotically and $|\mathcal{T}_{\leq k}| \leq (3.1)nk^3$ for $k \geq 13$.*

Proof. The proof is an application of the Clarkson-Shor technique [Mat02]. Pick each point in P independently with probability p_{cs} to get a random sample R_{cs}. Count the expected number of edges in the Delaunay triangulation of R_{cs} in two ways. On one hand, it is simply less than

$3E[|R_{cs}|] = 3np_{cs}$. On the other hand, it is:

$$\begin{aligned}
3np_{cs} &\geq \mathbf{E}[\text{Number of Delaunay edges in } R_{cs}] = \sum_{p,q\in P} \Pr[\{p,q\} \text{ is a Delaunay edge of } R_{cs}] \\
&\geq \sum_{p,q\in P}\sum_{r,s\in P} \Pr[(D_{pqr} \cup D_{pqs}) \cap R_{cs} = \emptyset] \qquad \text{(disjoint events)} \\
&\geq \sum_{(\{p,q\},\{r,s\})\in\mathcal{T}_{\leq k}} \Pr[(D_{pqr} \cup D_{pqs}) \cap R_{cs} = \emptyset] \\
&\geq \sum_{(\{p,q\},\{r,s\})\in\mathcal{T}_{\leq k}} p_{cs}^4 \cdot (1-p_{cs})^k = |\mathcal{T}_{\leq k}| \cdot p_{cs}^4 \cdot (1-p_{cs})^k
\end{aligned}$$

Therefore $|\mathcal{T}_{\leq k}| \leq 3np_{cs}/(p_{cs}^4(1-p_{cs})^k)$ and a simple calculation gives that setting $p_{cs} = \frac{3}{k+3}$ minimizes the right hand side. Then $|\mathcal{T}_{\leq k}| \leq 3n\frac{3}{k+3}/((\frac{3}{k+3})^4(1-\frac{3}{k+3})^k) = nk^3\frac{1}{9}(1+\frac{3}{k})^{k+3}$, and the claim follows. □

Lemma 6.6. $|\mathcal{S}_{\leq k}| \leq (e^2/4)nk^2$ *asymptotically and* $|\mathcal{S}_{\leq k}| \leq (2.14)nk^2$ *for* $k \geq 13$.

Proof. Pick each point in P independently with probability p_{cs} to get a random sample R_{cs}. Count the expected number of edges in the Delaunay triangulation of R_{cs} that lie on the boundary of the Delaunay triangulation, i.e., adjacent to exactly one triangle, in two ways. On one hand, it is exactly the number of edges in the convex-hull of R_{cs}, therefore at most $E[|R_{cs}|] = np_{cs}$. Counted another way, it is:

$$\begin{aligned}
np_{cs} &\geq \mathbf{E}[\text{Number of boundary Delaunay edges in } R_{cs}] \\
&= \sum_{p,q\in P} \Pr[\{p,q\} \text{ is a boundary Delaunay edge of } R_{cs}] \\
&\geq \sum_{p,q\in P}\sum_{r\in P} \Pr[(D_{pqr} \cup D_{pq\bar{r}}) \cap R_{cs} = \emptyset] \qquad \text{(disjoint events)} \\
&\geq \sum_{(\{p,q\},\{r\})\in\mathcal{S}_{\leq k}} \Pr[(D_{pqr} \cup D_{pq\bar{r}}) \cap R_{cs} = \emptyset] \\
&\geq \sum_{(\{p,q\},\{r\})\in\mathcal{S}_{\leq k}} p_{cs}^3 \cdot (1-p_{cs})^k = |\mathcal{S}_{\leq k}| \cdot p_{cs}^3 \cdot (1-p_{cs})^k
\end{aligned}$$

Setting $p_{cs} = \frac{2}{k+2}$ gives the required result. □

Lemma 6.7.

$$\boldsymbol{E}\Big[|\{e \in \Xi \ \mid \ k_1\epsilon n \leq |P_e| \leq k_2\epsilon n\}|\Big] \leq \frac{(3.1)c_1^3}{\epsilon e^{k_1 c_1}}(k_1^3 c_1 + 3.7k_2^2) \ \textit{if} \ \epsilon n \geq 13.$$

Proof. The crucial observation is that two points $\{p,q\}$ form an edge in Ξ with two adjacent triangles $\Delta(pqr), \Delta(pqs) \in \Xi$ **iff** $\{p,q,r,s\} \subseteq R$ and none of the points of P in $D_{pqr} \cup D_{pqs}$

are picked in R (i.e, the points p, q, r, s form the Delaunay tuple $(\{p, q\}, \{r, s\})$). Or $\{p, q\}$ form an edge on the convex-hull of Ξ with one adjacent triangle $\Delta(pqr)$ **iff** $\{p, q, r\} \subseteq R$ and none of the points of P in $D_{pqr} \cup D_{pq\overline{r}}$ are picked in R.

Let $\chi_{(\{p,q\},\{r,s\})}$ be the random variable that is 1 iff $\{p, q\}$ form an edge in Ξ and their two adjacent triangles are $\Delta(pqr)$ and $\Delta(pqs)$. Let $\chi_{(\{p,q\},\{r\})}$ be the random variable that is 1 iff $\{p, q\}$ form an edge in Ξ with exactly one adjacent triangle $\Delta(pqr)$. Noting that every edge in Ξ must come from either a Delaunay quadruple or a Delaunay triple,

$$\begin{aligned}\mathbf{E}[|\{e \mid k_1\epsilon n \le |P_e| \le k_2\epsilon n\}|] &= \sum_{\substack{p,q,r,s\in P\\ k_1\epsilon n\le W_{(\{p,q\},\{r,s\})}\le k_2\epsilon n}} \Pr[\chi_{(\{p,q\},\{r,s\})}=1] \;+\\ &\quad \sum_{\substack{p,q,r\in P\\ k_1\epsilon n\le W_{(\{p,q\},\{r\})}\le k_2\epsilon n}} \Pr[\chi_{(\{p,q\},\{r\})}=1]\end{aligned}$$

The second term is asymptotically smaller, so we bound it somewhat loosely:

$$\begin{aligned}\sum_{\substack{p,q,r\in P\\ k_1\epsilon n\le W_{(\{p,q\},\{r\})}\le k_2\epsilon n}} \Pr[\chi_{(\{p,q\},\{r\})}=1] &\le \sum_{\substack{p,q,r\\ k_1\epsilon n\le W_{(\{p,q\},\{r\})}\le k_2\epsilon n}} (c_1/\epsilon n)^3(1-c_1/\epsilon n)^{W_{(\{p,q\},\{r\})}}\\ &\le |\mathcal{S}_{\le k_2\epsilon n}|\cdot(c_1/\epsilon n)^3(1-c_1/\epsilon n)^{k_1\epsilon n}\\ &\le (2.14)n(k_2\epsilon n)^2\cdot(c_1/\epsilon n)^3\cdot e^{-c_1k_1} = \frac{(2.14)k_2^2c_1^3}{\epsilon e^{c_1k_1}}.\end{aligned}$$

Now we carefully bound the first term:

$$\begin{aligned}\sum_{\substack{p,q,r,s\in P\\ k_1\epsilon n\le W_{(\{p,q\},\{r,s\})}\le k_2\epsilon n}} \Pr[\chi_{(\{p,q\},\{r,s\})}=1] &\le \sum_{i=k_1\epsilon n}^{k_2\epsilon n}\sum_{\substack{p,q,r,s\\ W_{(\{p,q\},\{r,s\})}=i}} \Pr[\chi_{(\{p,q\},\{r,s\})}=1]\\ &\le \sum_{i=k_1\epsilon n}^{k_2\epsilon n}\sum_{\substack{p,q,r,s\\ W_{(\{p,q\},\{r,s\})}=i}} (c_1/\epsilon n)^4(1-c_1/\epsilon n)^i\\ &\le \sum_{i=k_1\epsilon n}^{k_2\epsilon n} |\mathcal{T}_i|(c_1/\epsilon n)^4(1-c_1/\epsilon n)^i\end{aligned}$$

As the above summation is exponentially decreasing as a function of i, it is maximized when $|\mathcal{T}_{i_0}| = \max|\mathcal{T}_{\le i_0}|$ where $i_0 = k_1\epsilon n$, and $|\mathcal{T}_i| = \max|\mathcal{T}_{\le i}| - \max|\mathcal{T}_{\le i-1}|$ and so on. Using

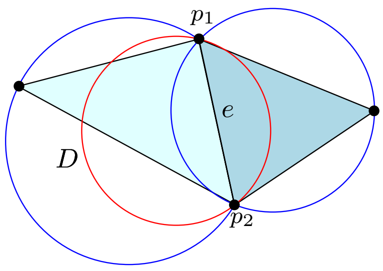
Lemma 6.5 we obtain:

$$
\begin{aligned}
&\le |\mathcal{T}_{\le k_1\epsilon n}| \cdot (c_1/\epsilon n)^4(1-c_1/\epsilon n)^{k_1\epsilon n} + \sum_{i=k_1\epsilon n+1}^{k_2\epsilon n} (|\mathcal{T}_{\le i}| - |\mathcal{T}_{\le i-1}|)\cdot (c_1/\epsilon n)^4(1-c_1/\epsilon n)^i \\
&\le (3.1)n(k_1\epsilon n)^3 \cdot (c_1/\epsilon n)^4(1-c_1/\epsilon n)^{k_1\epsilon n} + \sum_{i=k_1\epsilon n+1}^{k_2\epsilon n} (3.1)n\cdot 3i^2 \cdot (c_1/\epsilon n)^4(1-c_1/\epsilon n)^i \\
&\le (3.1)\frac{k_1^3c_1^4e^{-k_1c_1}}{\epsilon} + (3.1)\frac{3k_2^2c_1^4}{\epsilon^2 n}\sum_{i=k_1\epsilon n+1}^{k_2\epsilon n}(1-c_1/\epsilon n)^i \\
&\le (3.1)\frac{k_1^3c_1^4e^{-k_1c_1}}{\epsilon} + (3.1)\frac{3k_2^2c_1^4}{\epsilon^2 n}\frac{(1-c_1/\epsilon n)^{k_1\epsilon n}}{c_1/\epsilon n} \le \frac{(3.1)c_1^3}{\epsilon e^{k_1c_1}}(k_1^3c_1+3k_2^2).
\end{aligned}
$$

The proof follows by summing up the two terms. □

Using the above facts we can prove the main result.

Lemma 6.8. *Algorithm* COMPUTE ϵ-NET *computes an ϵ-net of expected size* $13.4/\epsilon$.



Proof. First we show that the algorithm computes an ϵ-net. Take any disk D with center c containing ϵn points of P, and not hit by the initial random sample R. Increase its radius while keeping its center c fixed until it passes through a point, say p_1 of R. Now further expand the disk by moving c in the direction $\vec{p_1 c}$ until its boundary passes through a second point p_2 of R. The edge e defined by p_1 and p_2 belongs to Ξ, and the two extreme disks in the pencil of empty disks through p_1 and p_2 are the disks $D_{\Delta_e^1}$ and $D_{\Delta_e^2}$. Their union covers D, and so D contains ϵn points out of the set P_e. Then the net R_e computed for P_e must hit D, as $\epsilon n = (\epsilon n/|P_e|)\cdot|P_e|$.

For the expected size, clearly, if $\epsilon n < 13$ then the returned set is an ϵ-net of size $\frac{13}{\epsilon}$. Otherwise we can calculate the expected number of points added to the ϵ-net during solving the sub-problems. We simply group them by the number of points in them. Set $E_i = \{e \mid 2^i\epsilon n \le |P_e| < 2^{i+1}\epsilon n\}$, and let us denote the size of the ϵ-net returned by our algorithm with $f'(\epsilon)$. Then

$$
\begin{aligned}
\mathbf{E}\left[f'(\epsilon)\right] = \mathbf{E}[|R|] + \mathbf{E}\Big[|\bigcup_{e\in\Xi} R_e|\Big] \quad = \quad & \frac{c_1}{\epsilon} + \mathbf{E}[|\{e \mid \epsilon n \le |P_e| < 3\epsilon n/2\}|]\cdot f(2/3) \\
& + \mathbf{E}[|\{e \mid 3\epsilon n/2 \le |P_e| < 2\epsilon n\}|]\cdot f(1/2) \\
& + \sum_{i=1}\mathbf{E}\left[\sum_{e\in E_i} f'\left(\frac{\epsilon n}{|P_e|}\right)\right]
\end{aligned}
$$

Noting that $\mathbf{E}[\sum_{e\in E_i} f'(\frac{\epsilon n}{|P_e|}) \mid |E_i| = t] \le t\mathbf{E}[f'(1/2^{i+1})]$, we get

$$\mathbf{E}\left[\sum_{e\in E_i} f'\left(\frac{\epsilon n}{|P_e|}\right)\right] = \mathbf{E}\left[\mathbf{E}[\sum_{e\in E_i} f'\left(\frac{\epsilon n}{|P_e|}\right)|E_i]\right] \le \mathbf{E}\left[|E_i|\cdot \mathbf{E}[f'(1/2^{i+1})]\right] = \mathbf{E}[|E_i|]\cdot\mathbf{E}[f'(1/2^{i+1})]$$

as $|E_i|$ and $f'(\cdot)$ are independent. As $\epsilon' = \frac{\epsilon n}{|P_e|} > \epsilon$, by induction, assume $\mathbf{E}[f'\left(\epsilon'\right)] \le \frac{13.4}{\epsilon'}$. Then

$$\begin{aligned}\mathbf{E}\left[f'(\epsilon)\right] &\le \frac{c_1}{\epsilon} + \frac{(3.1)\cdot c_1^3(c_1+8.34)}{\epsilon e^{c_1}}\cdot 2 + \frac{(3.1)\cdot c_1^3((3/2)^3 c_1 + 14.8)}{\epsilon e^{3c_1/2}}\cdot 10 \\ &\quad + \sum_i \frac{(3.1)\cdot c_1^3(2^{3i}c_1 + 3.7\cdot 2^{2i+2})}{\epsilon e^{c_1 2^i}}\cdot 13.4\cdot 2^{i+1} \le \frac{13.4}{\epsilon}\end{aligned}$$

by setting $c_1 = 12$. □

Finally, we bound the expected running time of the algorithm.

Lemma 6.9. *Algorithm* COMPUTE ϵ-NET *runs in expected time $O(n\log n)$.*

Proof. Note that $\mathbf{E}[|R|] = c_1/\epsilon$. First we bound the expected total size of all the sets P_e:

$$\begin{aligned}\mathbf{E}\Big[\sum_{e\in\Xi}|P_e|\Big] &\le \mathbf{E}[|\{e \mid 0\le |P_e| < \epsilon n\}|]\cdot\epsilon n + \sum_{i=0}\mathbf{E}[|\{e \mid 2^i\epsilon n \le |P_e| < 2^{i+1}\epsilon n\}|]\cdot 2^{i+1}\epsilon n \\ &\le O(\frac{\epsilon n}{\epsilon}) + \sum_{i=0} O\left(\frac{(2^i)^3}{\epsilon e^{2^i c_1}}\right)\cdot 2^{i+1}\epsilon n = O(n),\end{aligned}$$

as the last summation is a geometric series. This implies that the expected total number of incidences between points in P, and Delaunay disks in Ξ is $O(n)$. The Delaunay triangulation of R can be computed in expected time $O(1/\epsilon\log 1/\epsilon)$. Steps 5-6 compute, for each Delaunay disk $D\in\Xi$, the list of points contained in D. This can be computed in $O(n\log 1/\epsilon)$ time by instead finding, for each $p\in P$, the list of Delaunay disks in Ξ containing p, as follows. First do point-location in Ξ to locate the triangle Δ containing p, in expected time $O(\log 1/\epsilon)$. Clearly D_Δ contains p. Now starting from Δ, do a breadth-first search in the dual planar graph of the Delaunay triangulation to find the maximally connected subset of triangles (vertices in the dual graph) whose Delaunay disks contain p. As each vertex in the dual graph has degree at most 3, this takes time proportional to the discovered list of triangles, which as shown earlier is $O(n)$ over all $p\in P$. The correctness follows from the following:

Lemma 6.10. *Given a Delaunay triangulation Ξ on R and any point $p\in\mathbb{R}^2$, the set of triangles in Ξ whose Delaunay disks contain p form a connected sub-graph in the dual graph to Ξ.*

Proof. This can be seen by lifting P to $\mathbb{R}^3$ via the Veronese mapping, where it follows from the fact that the faces of a convex polyhedron that are visible from any exterior point are connected. $\square$

Note that by the ϵ-net theorem, the probability of restarting the algorithm (lines 4-5) at any call is at most a constant. Therefore it is re-started expected at most a constant number of times, and so the expected running time, denoted by $T(n)$:

$$\mathbf{E}[T(n)] = O(1/\epsilon \log 1/\epsilon) + O(n \log 1/\epsilon) + \sum_{e \in \Xi} \mathbf{E}[T(|P_e|)] \leq O(n \log 1/\epsilon) + \sum_{e \in \Xi} \mathbf{E}[T(|P_e|)]$$

Similarly to previous calculations we have that

$$\begin{aligned} \mathbf{E}[T(n)] &\leq O(n \log 1/\epsilon) + \frac{(3.1) \cdot c_1^3 (c_1 + 8.34)}{\epsilon e^{c_1}} \cdot O(3\epsilon n/2 \log(3\epsilon n/2)) \\ &+ \frac{(3.1) \cdot c_1^3((3/2)^3 c_1 + 14.8)}{\epsilon e^{3c_1/2}} \cdot O(2\epsilon n \log(2\epsilon n)) \\ &+ \sum_{i=1} \frac{(3.1) \cdot c_1^3 (2^{3i} c_1 + 3.7 \cdot 2^{2i+2})}{\epsilon e^{c_1 2^i}} \cdot \mathbf{E}[T(2^{i+1} \epsilon n)] \\ &\leq dn \log n + \sum_{i=1} \frac{(3.1) \cdot c_1^3 (2^{3i} c_1 + 3.7 \cdot 2^{2i+2})}{\epsilon e^{c_1 2^i}} \cdot \mathbf{E}[T(2^{i+1} \epsilon n)] \end{aligned}$$

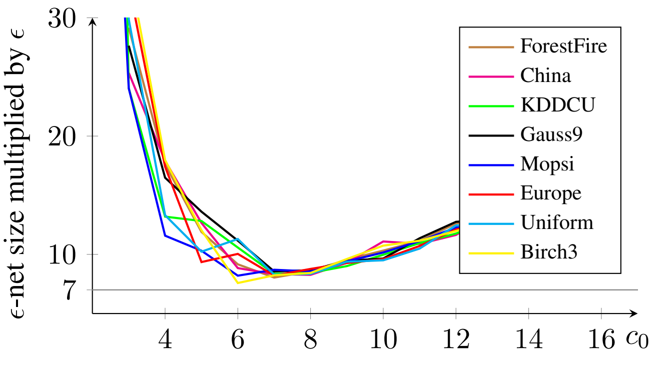
for a constant d coming from the constants above, as well as in Delaunay triangulation, point-location and list-construction computations. Setting $\mathbf{E}[T(k)] = ck \log k$ satisfies the above inequality for $c \geq 2d$, since

$$\begin{aligned} \mathbf{E}[T(n)] &\leq dn \log n + \sum_{i=1} \frac{(3.1) \cdot c_1^3 (2^{3i} c_1 + 3.7 \cdot 2^{2i+2})}{\epsilon e^{c_1 2^i}} \cdot c(2^{i+1} \epsilon n) \log(2^{i+1} \epsilon n) \\ &\leq dn \log n + (cn \log n) \sum_{i=1} \frac{2^{i+1} \cdot (3.1) \cdot 12^3 (2^{3i} \cdot 12 + 3.7 \cdot 2^{2i+2})}{e^{12 \cdot 2^i}} \\ &\leq dn \log n + cn \log n \cdot \frac{1}{2} \leq cn \log n, \text{ for } c \geq 2d. \end{aligned}$$

$\square$

6.2 Implementation and Experimental Evaluation

In this section we present experimental results for our algorithms implemented in $\mathcal{C}$ and running on a machine equipped with an Intel Core i7 870 processor (2.93 GHz) and with 16 GB main memory. All our implementations are single-threaded. The source code can be obtained from



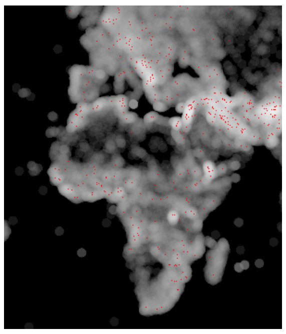


FIGURE 6.2: ϵ-net size multiplied by ϵ for the datasets, $\epsilon = 0.01$ (left) and a subset of the ϵ-net for the *World* dataset (right).

the authors' website. For nearest-neighbors and Delaunay triangulations, we use CGAL. It computes Delaunay triangulations in expected $O(n \log n)$ time.

Datasets. In order to empirically validate our algorithms we have utilized several real-world point sets. All our experiments' point sets are scaled to a unit square. The *World* dataset [Ngs] contains locations of cities on Earth (except for the US) having around 10M records. For our experiments we use only the locations of cities in China having 1M records (the coordinates have been obtained from latitude and longitude data by applying the Miller cylindrical projection). The dataset *ForestFire* contains 700K locations of wildfire occurrences in the United States [Fwf]. The *KDDCUP04Bio* dataset [Cd] (*KDDCU* for short) contains the first 2 dimensions of a protein dataset with 145K entries. We have also created a random data set *Gauss9* with 90K points sampled from 9 different Gaussian distributions with random mean and covariance matrices.

Sizes of ϵ-nets. Setting the probability for random sampling to $\frac{12}{\epsilon \cdot n}$ results in approximately $\frac{12}{\epsilon}$ sized nets for nearly all datasets, as expected by our analysis. We note however, that in practice setting c_0 to 7 gives smaller size ϵ-nets, of size around $\frac{9}{\epsilon}$. See Figure 6.2 for the dependency of the net size on c_0 for $\epsilon = 0.01$. It also includes an ϵ-net calculated with our algorithm for a subset of the *World* data (red points denote the ϵ-net and each pixel's color is the logarithm of the number of disks it is contained in). See Table 6.1 for the ϵ-net sizes for different values of ϵ as c_0 varies from 7 to 12. This table also includes the size of the first random sample (R), which shows that the number of subproblems to solve increases as the random sample is more sparse.

The results for our ϵ-net algorithm indicate that our theoretical analysis in Section 6.1 closely tracks the actual size of the nets. This can additionally be seen as continuing the program for better analysis of basic geometric tools; see, e.g., Har-Peled [HP00] for analysis of algorithms and Matousek [Mat98] for detailed analysis, both for a related structure called cuttings in the plane.

TABLE 6.1: The size of the ϵ-net multiplied by ϵ (left value in a column for a fixed ϵ) and the size of R, the first random sample multiplied by ϵ (right value in a column) for various point sets with $c_0 = 7$ or 12.

	$c_0 = 7$								$c_0 = 12$							
ϵ	0.2		0.1		0.01		0.001		0.2		0.1		0.01		0.001	
China	7.8	6.6	8.3	6.1	8.28	6.80	8.426	7.090	14.2	14.2	10.6	10.6	12.33	12.33	12.152	12.138
ForestFire	7.4	7.4	8.3	7.3	8.46	7.46	8.522	6.892	13	13	11.6	11.6	12.01	12.01	12.103	12.077
KDDCU	7.4	7.4	8.4	7.4	8.31	7.29	8.343	6.989	10.2	10.2	9.8	9.8	11.65	11.57	12.006	11.978
Gauss9	7.4	5.8	7.8	7.6	8.00	7.18	8.100	6.882	9.8	9.8	12.0	12.0	11.61	11.43	11.969	11.965
Europe	6.2	5	8.7	5.9	7.93	7.39	8.300	7.048	14.2	14.2	12.2	12.2	11.95	11.75	11.777	11.741
Birch 3	6.2	6.2	6.9	6.7	7.91	7.09	8.134	6.923	8.6	8.6	11.3	11.3	11.73	11.71	11.792	11.782
Uniform	9.4	5.4	6.5	6.5	9.00	7.08	8.014	6.940	14	14	10.9	10.9	12.45	12.43	12.034	12.014
MOPSI	9.6	7.2	9.4	7.6	7.85	6.95	7.119	7.093	13.8	13.8	14.5	14.5	12.10	11.98	12.017	12.017

Chapter 7

Engineering the Agarwal-Pan Algorithm

In this chapter we present our algorithm for computing hitting sets via the rounding technique. This work is based on the following paper.

Norbert Bus, Nabil H. Mustafa, and Saurabh Ray. "Geometric Hitting Sets for Disks: Theory and Practice." In: *23rd European Symposium on Algorithms (ESA)*. 2015

7.1 General Idea

The breakthrough algorithm of Agarwal-Pan [AP14] uses sophisticated data-structures that have large constants in the running time. In particular, it uses the $O(\log n + k)$-time algorithm for range reporting for disk ranges in the plane (alternatively, for halfspaces in $\mathbb{R}^3$) as well as a dynamic data-structure for maintaining approximate weighted range-counting under disk ranges in polylogarithmic time. We have not been able to find efficient implementations of any of these data-structures.

This work is an attempt to address this shortcoming by observing that in this specific application (i.e., minimum hitting set problem for disks), we don't need general range-searching and dynamic range counting tools, we give a new modified elementary algorithm. More specifically, rur contributions are:

1. Engineering a hitting-set algorithm (Section 7.2). Together with the result of Agarwal-Pan, Theorem 6.1 immediately implies, that for any $\delta > 0$, one can compute a $(13.4 + \delta)$-approximation to the minimum hitting set for $(P, \mathcal{D})$ in time $\tilde{O}(n)$. We present a modification of the algorithm of Agarwal-Pan that does not use any complicated data-structures – just Delaunay triangulations, ϵ-nets and binary search (e.g., it turns out that output sensitive range reporting is not required). This comes with a price: although experimental results indicate a near-linear running time, we have been unable to theoretically prove that the algorithm runs in expected near-linear time.

2. Implementation and experimental evaluation (Section 7.3). A major advantage of Delaunay triangulations is that their behavior has been extensively studied, there are many efficient implementations available, and they exhibit good behavior for various real-world data-sets as well as random point sets. We present `dnet`, a public source-code module that incorporates these ideas and an implementation of our ϵ-net algorithm to efficiently compute small-sized hitting sets in practice. We give detailed experimental results on both synthetic and real-world data sets, which indicates that the algorithm computes, on average, a 1.3-approximation in near-linear time.

7.2 Algorithm

The Agarwal-Pan (AP) algorithm (shown in Algorithm 7) uses an iterative reweighing strategy, where the idea is to assign a weight $w(\cdot)$ to each $p \in P$ such that the total weight of points contained in each $D \in \mathcal{D}$ is relatively high. It starts by setting $w(p) = 1$ for each $p \in P$. If there exists a disk D with small weight, it increases the weight of the points in D until their total weight exceeds a threshold of $cW/\textsc{Opt}$, where c is some constant and $W = \sum_{p \in P} w(p)$ is the current total weight. If after any iteration, all disks have weight above the threshold $\frac{cW}{2e\textsc{Opt}}$, return a $\frac{c}{2e\textsc{Opt}}$-net with respect to these weights, ensuring that every disk is hit.

For the purpose of analysis, Agarwal and Pan conceptually divide the reweighings into $O(\log n)$ phases, where each phase (except perhaps the last) performs $\Theta(\textsc{Opt})$ reweighings. The implementation of the AP algorithm requires two ingredients: **A)** a range reporting data structure and **B)** a dynamic approximate range counting data structure. The former is required for figuring out whether a disk needs reweighing and the latter is used to construct the set of points to be reweighed. As a pre-processing step, the AP algorithm first computes a $1/\textsc{Opt}$-net Q to be returned as part of the hitting set. This ensures that the remaining disks not hit by Q contain less than $n/\textsc{Opt}$ points. Additionally they observe that in any iteration, if less than $\textsc{Opt}$ disks are reweighed, then all disks have weight more than $\frac{cW}{2e\textsc{Opt}}$.

The AP algorithm is simple and has a clever theoretical analysis. Its main drawback is that the two data structures it uses are sophisticated with large constants in the running time. This unfortunately renders the AP algorithm impractical. Our goal is to find a method that avoids these sophisticated data structures and to develop additional heuristics which lead to not only a fast implementation but also one that generally gives an approximation ratio smaller than that

Algorithm 7: AP algorithm for computing hitting sets

```
Data: A point set $P$, a set of disks $\mathcal{D}$, a fixed constant $c$, and the value of OPT.
1  Compute a $(1/\text{OPT})$-net, $Q$, of $P$ and remove disks hit by $Q$
2  Set $w(p) = 1$ for all $p \in P$
3  repeat
4      foreach $D \in \mathcal{D}$ do
5          if $w(D) \leq cW/\text{OPT}$ then
6              reweigh $D$ repeatedly until the weight $w(D)$ exceeds $cW/\text{OPT}$
7      flag = false
8      foreach $D \in \mathcal{D}$ do
9          if $w(D) < (c/2e) \cdot W/\text{OPT}$ then flag = true;
10 until flag = true
11 return $(c/2e\text{OPT})$-net $R$ along with $Q$
```

guaranteed by the theoretical analysis of the AP algorithm. As part of the algorithm, we use the algorithm for constructing ϵ-nets described in the previous section, which already reduces the approximation factor significantly.

Removing A). Just as Agarwal and Pan do, we start by picking a c_1/OPT-net, for some constant c_1. The idea for getting rid of range-reporting data-structure is to observe that the very fact that a disk D is not hit by Q, *when Q is an ϵ-net*, makes it possible to use Q in a simple way to efficiently enumerate the points in D. We will show that D lies in the union of two Delaunay disks in the Delaunay triangulation of Q, which, as we show later, can be found by a simple binary search. The resulting algorithm still has worst-case near-linear running time.

Removing B). Our approach towards removing the dependence on dynamic approximate range counting data structure is the following: at the beginning of *each* phase we pick a c_2/OPT-net R, for some constant c_2. The set of disks that are not hit by R are then guaranteed to have weight at most c_2W/OPT, which we can then reweigh during that phase. While this avoids having to use data-structure **B)**, there are two problems with this: a) disks with small weight hit by R are not reweighed, and b) a disk whose initial weight was less than c_2W/OPT could have its current weight more than c_2W/OPT in the middle of a phase, and so it is erroneously reweighed.

Towards solving these problems, the idea is to maintain an additional set S which is empty at the start of each phase. When a disk D is reweighed, we add a random point of D (sampled according to $w(\cdot)$) to S. Additionally we maintain a nearest-neighbor structure for S, enabling us to only reweigh D if it is not hit by $R \cup S$. Now, if during a phase, there are $\Omega(\text{OPT})$ reweighings, then as in the Agarwal-Pan algorithm, we move on to the next phase, and a) is not a problem. Otherwise, there have been less than OPT reweighings, which implies that less than OPT disks were not hit by R. Then we can return R together with the set S consisting of one point from each of these disks. This will still be a hitting set.

Algorithm 8: Algorithm for computing small-sized hitting sets.

```
Data: A point set $P$, a set of disks $\mathcal{D}$, and the size of the optimal hitting set $\textsc{Opt}$.
1  Compute a $(c_1/\textsc{Opt})$-net $Q$ of $P$ and the Delaunay triangulation $\Xi(Q)$ of $Q$.
2  foreach $q \in Q$ do construct $\Psi(Q)(q)$.
3  foreach $D \in \mathcal{D}$ do
4      if $D$ not hit by $Q$ then add $D$ to $\mathcal{D}_1$. // using $\Xi(Q)$
5  $P_1 = P \setminus Q$.
6  foreach $p \in P_1$ do set $w(p) = 1$.
7  repeat
8      Compute a $(c_2/\textsc{Opt})$-net, $R$, of $P$ and the Delaunay triangulation $\Xi(R)$ of $R$.
9      Set $S = \emptyset$, $\Xi(S) = \emptyset$.
10     foreach $D \in \mathcal{D}_1$ in a random order do
11         if $D$ not hit by $R \cup S$ then // using $\Xi(R)$ and $\Xi(S)$
12             foreach $p \in D$ do set $w(p) = w(p) + c_3 w(p)$. // using $\Psi(Q)$
13             Add a $O(1)$-net to $S$; update $\Xi(S)$.
14 until $|S| \leq c_4\textsc{Opt}$
15 return $\{Q \cup R \cup S\}$
```

To remedy b), before reweighing a disk, we compute the set of points inside D, and only reweigh if the total weight is at most $c_2 W/\textsc{Opt}$. Consequently we sometimes waste $O(n/\textsc{Opt})$ time to compute this list of points inside D without performing a reweighing. Due to this, the worst-case running time increases to $O(n^2/\textsc{Opt})$. In practice, this does not happen for the following reason: in contrast to the AP algorithm, our algorithm reweighs any disk *at most once* during a phase. Therefore if the weight of any disk D increases significantly, and yet D is not hit by S, the increase must have been due to the increase in weight of many disks intersected by D which were reweighed before D *and* for which the picked points (added to S) did not hit D. Reweighing in a random order makes these events very unlikely (in fact we suspect this gives an expected linear-time algorithm, though we have not been able to prove it).

See Algorithm 8 for the new algorithm (the data-structure $\Psi(Q)$ will be defined later).

Lemma 7.1. *The algorithm terminates, $Q \cup R \cup S$ is a hitting set, of size at most $(13.4+\delta)\cdot\textsc{Opt}$, for any $\delta > 0$.*

Proof. By construction, if the algorithm terminates, then $Q \cup R \cup S$ is a hitting-set. Set $c_1 = 13.4 \cdot 3/\delta$, $c_2 = 1/(1+\delta/(13.4\cdot 3))$, $c_3 = \delta/10000$ and $c_4 = \delta/3$. By the standard reweighing argument, we know that after t reweighings, we have:

$$\textsc{Opt}\,(1+c_3)^{\frac{t}{\textsc{Opt}}} \leq n \cdot (1 + \frac{c_2 c_3}{\textsc{Opt}})^t \tag{7.1}$$

which solves to $t = O(\frac{\textsc{Opt}\log n}{\delta})$. Each iteration of the repeat loop, except the last one, does at least $c_4\textsc{Opt}$ reweighings. Then the repeat loop can run for at most $O(\frac{\textsc{Opt}\log n}{c_4 \textsc{Opt}\delta}) = O(\log n/\delta)$ times.

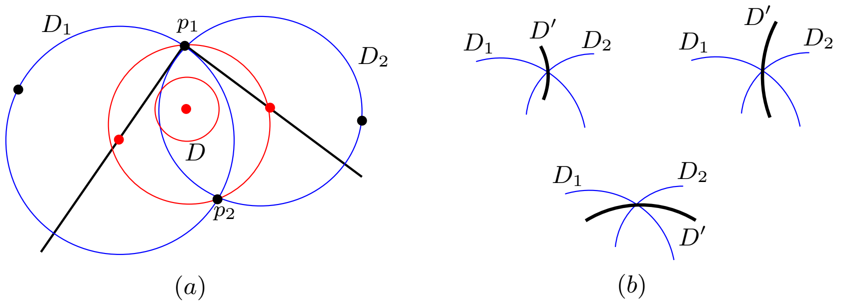


FIGURE 7.1: Computing D_1 and D_2.

By Theorem 6.1, $|Q| \leq (13.4/c_1)\text{OPT}$, $|R| \leq (13.4/c_2)\text{OPT}$, and $|S| \leq c_4\text{OPT}$. Thus the overall size is $13.4\text{OPT} \cdot \big(1/c_1 + 1/c_2 + c_4/13.4\big) \leq (13.4 + \delta) \cdot \text{OPT}$. □

Algorithmic details. Computing an ϵ-net takes $O(n \log n)$ time using Theorem 6.1. Checking if a disk D is hit by an ϵ-net (Q, R, or S) reduces to finding the closest point in the set to the center of D, again accomplished in $O(\log n)$ time using point-location in Delaunay/Voronoi diagrams $\Xi(\cdot)$. It remains to show how to compute, for a given disk $D \in \mathcal{D}_1$, the set of points of P contained in D:

Lemma 7.2. *Given a disk $D \in \mathcal{D}_1$, the set of points of P contained in D can be reported in time $O(n/\text{OPT} \log n)$.*

Proof. Each disk in $\mathcal{D}_1$ is not hit by Q, and so contains at most $c_1 n/\text{OPT}$ points of P. We now show how, given any disk D with $D \cap Q = \emptyset$, one can find two disks whose union covers D in $O(\log n)$ time. Given D, compute, using $\Xi(Q)$, the nearest neighbor $p \in Q$ to the center of D. Consider the list of Delaunay triangles incident to p, sorted by their circumcenters radially around p. Denote this list for the point p by $\Psi(Q)(p)$. Let D'_1 and D'_2 be the two Delaunay disks of $\Xi(Q)$ whose triangles are adjacent to p, and whose circumcenters are immediately before and after the center of D in this radially sorted order.

Lemma 7.3. $D \subseteq D'_1 \cup D'_2$.

Proof. Lemma 6.8 proves that for any disk D not hit by the ϵ-net Q, there exist two Delaunay disks of $\Xi(Q)$, say D_1 and D_2, such that $D \subseteq D_1 \cup D_2$. In particular, the proof shows that given D, D_1 and D_2 are circumcircles of two adjacent Delaunay triangles, say Δ_e^1 and Δ_e^2, where $e = \{p_1, p_2\} \in \Xi(Q)$ is the shared Delaunay edge. Moreover, one of the vertices of e, say p_1, is the closest point in Q to the center of D.

We finish the proof by showing that $i)$ the circumcenters of D_1 and D_2 are consecutive in the radially sorted list $\Psi(Q)(p_1)$, and $ii)$ the center of D lies between $c(D_1)$ and $c(D_2)$ in this consecutive order. Thus $\{D'_1, D'_2\} = \{D_1, D_2\}$. See Figure 7.1(a) for the geometric configuration.

For contradiction assume a disk D' whose $c(D')$ lies between $c(D_1)$ and $c(D_2)$. D' passes through p_1, and there are two cases:

- D' enters/leaves p_1 through D_1/D_2 (Figure 7.1(b) bottom). Then either $D' \subset D_1 \cup D_2$, and the two points of Q on $\partial D'$ lie inside $D_1 \cup D_2$, contradicting emptiness of $int(D_1 \cup D_2)$. Or D' must contain p_2, contradicting emptiness of D'.
- D' enters/leaves p_1 through outside ((Figure 7.1(b) top). Then $c(D')$ must lie radially outside the interval of $c(D_1)$ and $c(D_2)$.

□

Finally note that $\Psi(Q)$ can be constructed in the pre-processing phase in expected $O(n \log n)$ time: for each point $q \in Q$, extract its set of adjacent Delaunay triangles from $\Xi(Q)$, and radially sort their circumcenters around q to get the list $\Psi(Q)(q)$. As the number of triangles is $O(|Q|)$, this takes time $O(|Q| \log |Q|) = O(n/\text{OPT} \log n)$. And for each D, one can find the two Delaunay disks D'_1 and D'_2 by binary search in $\Psi(Q)(q)$ in time $O(\log n)$. □

7.3 Implementation and Experimental Evaluation

In this section we present experimental results for our algorithms implemented in $\mathcal{C}$ and running on a machine equipped with an Intel Core i7 870 processor (2.93 GHz) and with 16 GB main memory. All our implementations are single-threaded, but we note that our hitting set algorithm can be easily multi-threaded. The source code can be obtained from the authors' website. For nearest-neighbors and Delaunay triangulations, we use CGAL. It computes Delaunay triangulations in expected $O(n \log n)$ time. To calculate the optimal solution for the hitting set problem we use the IP solver SCIP (with the linear solver SoPlex).

Datasets. In order to empirically validate our algorithms we have utilized several real-world point sets. All our experiments' point sets are scaled to a unit square. The *World* dataset [Ngs] contains locations of cities on Earth (except for the US) having around 10M records. For our experiments we use only the locations of cities in China having 1M records (the coordinates have been obtained from latitude and longitude data by applying the Miller cylindrical projection). The dataset *ForestFire* contains 700K locations of wildfire occurrences in the United States [Fwf]. The *KDDCUP04Bio* dataset [Cd] (*KDDCU* for short) contains the first 2 dimensions of a protein dataset with 145K entries. We have also created a random data set *Gauss9* with 90K points sampled from 9 different Gaussian distributions with random mean and covariance matrices.

TABLE 7.1: Hitting sets. From top to bottom, $RND(0.1)$, $RND(0.01)$ and $FIX(0.001)$.

	# of points	# of disks	Q size	R size	S size	# of phases	IP solution	*dnet* solution	ap-prox.	IP time(s)	*dnet* time(s)
China	50K	50K	367	809	604	11	1185	1780	1.5	60	12
ForestFire	50K	16K	43	85	224	11	267	352	1.3	54.3	6.9
KDDCU	50K	22K	171	228	786	11	838	1185	1.4	40.9	9.8
Gauss9	50K	35K	322	724	1035	11	1493	2081	1.4	52.5	11.7
Europe	50K	31K	185	322	419	11	630	926	1.5	87.6	9.8
Birch3	50K	29K	166	233	1036	11	1026	1415	1.4	51.4	10.4
Uniform	50K	48K	665	1109	2169	11	2824	3943	1.4	49.8	12.3
Mopsi	13K	5K	40	55	197	10	228	292	1.3	2.4	1.7
China	50K	49K	673	1145	4048	11	4732	5862	1.2	4.5	14.5
ForestFire	50K	25K	162	268	1021	11	1115	1451	1.3	6.2	9.5
KDDCU	50K	102K	1326	2492	6833	11	8604	10651	1.2	12.5	22.2
Gauss9	50K	185K	2737	6636	9867	11	15847	19239	1.2	22.4	36.0
Europe	50K	85K	683	1491	3138	11	4211	5312	1.3	13.2	18.1
Birch3	50K	117K	1359	3358	7223	11	9683	11940	1.2	15.4	25.6
Uniform	50K	387K	5549	13081	16826	11	31787	35446	1.1	34.7	66.3
MOPSI	13K	6K	179	314	656	10	762	1009	1.3	0.6	2.5
China	50K	50K	2765	7376	7851	11	–	17329	–	–	19.8
ForesFire	50K	50K	331	602	1273	11	–	2206	–	–	11.9
KDDCU	50K	50K	2764	5824	15734	11	22368	24318	1.1	8.9	22.3
Gauss9	50K	50K	5380	13321	19153	11	36302	37827	1.0	19.1	26.7
Europe	50K	50K	1376	2644	5161	11	–	9181	–	–	16.2
Birch3	50K	50K	2709	7492	14434	11	–	24630	–	–	22.1
Uniform	50K	50K	5442	13417	27573	11	46124	46420	1.0	19.4	30.4
MOPSI	13K	13K	354	673	870	10	1294	1646	1.3	127.2	2.9

Approximate hitting sets. For evaluating the practical usability of our approximate hitting set algorithm we compare it to the optimal solution calculated by an IP solver. Our algorithm needs a guess for OPT, and so we run it with $O(\log n)$ guesses for the value of OPT. The parameters are set as follows: $c_0 = 10, c_1 = 30, c_2 = 12, c_3 = 2, c_4 = 2$ and $c_5 = 0.6$.

Our datasets only contain points and in order to create disks for the hitting set problem we have utilized two different strategies. In the first approach we create uniformly distributed disks in the unit square with uniformly distributed radius within the range $[0, r]$. Let us denote this test case as $RND(r)$. In the second approach we added disks centered at each point with a fixed radius of 0.001. Let us denote this test case by $FIX(0.001)$. The results are shown in Table 7.1 for two values $r = 0.1$ and $r = 0.01$. Our algorithm provides a 1.3 approximation on average. With small radius the solver seems to outperform our algorithm but this is most likely due to the fact that the problems become relatively simpler and various branch-and-bound heuristics become efficient. With bigger radius and therefore more complex constraint matrix our algorithm clearly outperforms the IP solver. Our method obtains a hitting set for all point sets, while in some of the cases the IP solver was unable to compute a solution in reasonable time (we terminate the solver after 1 hour).

In Table 7.2 we have included the memory consumption of both methods and statistics for range reporting. It is clear that the IP solver requires significantly more memory than our method. The statistics for range reporting includes the total number of range reportings (calculating the points

TABLE 7.2: Memory usage in MB (left) and range reporting statistics (right).

	RND(0.01)		RND(0.1)		FIX(0.001)	
	IP	*dnet*	IP	*dnet*	IP	*dnet*
China	243	21	4282	19	434	20
ForesFire	524	28	3059	18	5470	24
KDDCU	458	30	2999	23	175	22
Gauss9	569	33	3435	24	158	24
Europe	734	30	4418	25	659	24
Birch3	523	31	3655	24	960	26
Uniform	693	39	4083	25	155	24
MOPSI	30	11	294	8	1735	10

	RND(0.01)		RND(0.1)		FIX(0.001)	
	total	doubling	total	doubling	total	doubling
China	44014	43713	9406	9184	96335	95846
ForesFire	11167	11086	2767	2728	15648	15020
KDDCU	75448	75016	8485	8364	173147	173044
Gauss9	121168	120651	14133	13906	217048	217019
Europe	38112	37886	5224	5160	58644	57014
Birch3	85399	85063	11219	11049	167384	167197
Uniform	198168	197388	26970	26431	309286	309284
MOPSI	1883	1863	2283	2249	10360	10235

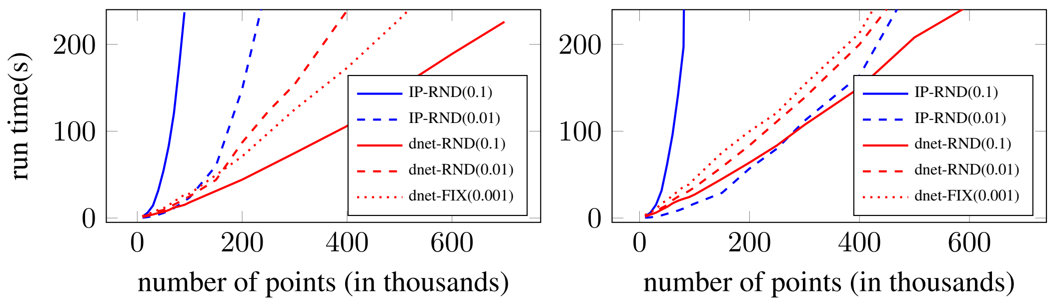


FIGURE 7.2: Different point set sizes for the *ForestFire* (left) and *China* (right) datasets.

inside a disk) and the number of range reportings when the algorithm doubles the weight of the points inside a disk (the doubling column in the table). It can be seen that only a fraction of the computations are wasted since the number of doublings is almost as high as the total number or range reportings. This in fact shows that the running time of our algorithm is near-linear in n.

In order to test the scalability of our method compared to the IP solver we have used the *Forest-Fire* and *China* dataset with limiting the number of points to 10K, 20K, 30K... and repeating exactly the same experiments as above (while increasing the number of disks in a similar manner). In Figure 7.2 we plot the running time of the methods. The solid lines represent the case $RND(0.1)$ while the dashed ones denote $RND(0.01)$. One can see that as the number of points and disks increases our method becomes more efficient even though for small instances this might not hold. It can be seen that for the *China* dataset and $RND(0.01)$ the IP solver is faster than our method but after 500K points our method becomes faster. In Figure 7.2 the dotted line represents the running time of our algorithm for $FIX(0.001)$. In this case the IP running time is not shown because the solver was only able to solve the problem with 10K points within a reasonable time (for 20K and 30K points it took 15 and 21 hours respectively).

We have varied the radius of the disks for the fixed radius case to see how the algorithms behave. See the Figure 7.3. With bigger radius the IP solver becomes very quickly unable to solve the problem (for radius 0.002 it was unable to finish within a day), showing that our method is more robust.

In order to test the extremes of our algorithm we have taken the *World* dataset containing 10M records. Our algorithm was able to calculate the solution of the $FIX(0.001)$ problem of size

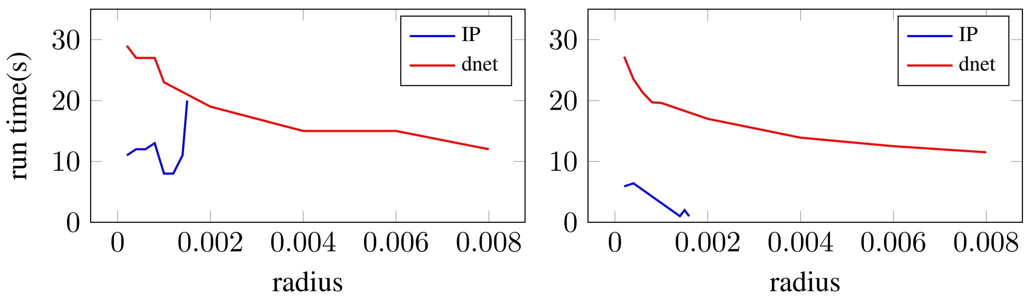


FIGURE 7.3: Different radii settings for the *KDDCU* (left) and *China* (right) datasets.

around 100K in 3.5 hours showing that the algorithm has the potential to calculate results even for extremely big datasets with a more optimized (e.g., multi-threaded) implementation.

Chapter 8

Local Search

This chapter aims to describe the technical details of our local search algorithm, it is based on the following publication.

Norbert Bus, Shashwat Garg, Nabil H. Mustafa, and Saurabh Ray. "Improved Local Search for Geometric Hitting Set." In: *32st International Symposium on Theoretical Aspects of Computer Science (STACS).* 2015

8.1 General Idea

Let R be a region in the plane. We say that a point $p \in \mathbb{R}^2$ hits R if $p \in R$, and that a set of points X hits a set of regions $\mathcal{R}$ if each region in $\mathcal{R}$ is hit by some point in X. We denote by $\mathcal{H}(P, \mathcal{R})$ the set system $(P, \{R \cap P : R \in \mathcal{R}\})$ induced by P and $\mathcal{R}$. A hitting set for $\mathcal{H}(P, \mathcal{R})$ is a subset of P which hits $\mathcal{R}$. A hitting set of the smallest cardinality is called the minimum hitting set and its size is denoted $\text{OPT}(P, \mathcal{R})$ (or simply OPT when it is clear from the context).

From now onwards, P denotes a set of points and $\mathcal{D}$ denotes a set of (circular) disks in the plane. Our goal is to compute a hitting set for $\mathcal{H}(P, \mathcal{D})$ of a small size efficiently. Our algorithm is based on local search. It starts with a hitting set and repeatedly tries to make local improvements. Let S be a hitting set for $\mathcal{H}(P, \mathcal{D})$. Let $X \subseteq S$ and $Y \subseteq P$. We say that (X, Y) is a *local improvement pair* with respect to S and $\mathcal{H}(P, \mathcal{D})$ if $|Y| < |X|$ and $(S \setminus X) \cup Y$ is a hitting set for $\mathcal{H}(P, \mathcal{D})$. Such a local improvement reduces the size of the hitting set by $|X| - |Y|$. We will refer to this quantity as the *profit* of the local improvement and the local improvement pair. We say that $X \subseteq S$ is *locally improvable* with respect to S and $\mathcal{H}(P, \mathcal{D})$ if there exists a $Y \subseteq P$ such that (X, Y) is a local improvement pair. If (X, Y) is a local improvement pair, we say that Y can *locally replace* X.

As our first result, we determine the exact limits of $(3, 2)$-local search:

Theorem 8.1 (Proof in Section 8.2). *A $(3, 2)$-local search algorithm returns a 8-approximation to the minimum hitting set. Furthermore, this is tight.*

Remark: In fact this can be extended to many other local search algorithms; e.g., it implies that the $(3, 2)$-local search gives 8-approximation to the independent-set problem for disks in the plane.

We then show how to perform this search in slightly more than quadratic time:

Theorem 8.2 (Proof in Section 8.3). *A $(3, 2)$-local search can be performed in expected time $O(n^{2.34})$.*

In fact, our techniques can be generalized for larger values of k. In particular, it can be shown that $(4, 3)$-local search gives a 5-approximation in time $\tilde{O}(n^{3.75})$.

8.2 Analysis of Quality for Local Search

The PTAS of Mustafa and Ray [MR10] uses a $(k, k-1)$-local search in which, for as long as possible, we try to swap some set of at most k points in the current hitting set with a smaller set of points while maintaining a hitting set. We stop when no such local improvement is possible.

The analysis of the approximation factor achieved by a $(k, k-1)$-local search depends on the following theorem on planar bipartite graphs.

Theorem 8.3. *[MR10] Let $G = (R, B, E)$ be a bipartite planar graph on red and blue vertex sets R and B, $|R| \geq 2$, such that for every subset $B' \subseteq B$ of size at most k, where k is a large enough number, $\left|N_G(B')\right| \geq \left|B'\right|$. Then, $|B| \leq (1 + c/\sqrt{k})\,|R|$, where c is a constant.*

Here $N_G(B')$ denotes the set of neighbors of the vertices in B' in G. The proof of the above theorem, which relies on planar graph separators, requires k to be quite large, thereby limiting the practical utility of the above theorem. A priori, it is not clear whether the theorems holds at all for small values of k. For instance, one can easily see that for $k = 2$ there is no upper bound on $|B|/|R|$ (e.g., consider complete bipartite graph where B is arbitrarily large and $|R| = 2$). However, for $k = 3$, we show a small bound of 8 on $|B|/|R|$, and then prove that it is, in fact, optimal.

Theorem 8.4. *Let $G = (R, B, E)$ be a bipartite planar graph on red and blue vertex sets R and B, $|R| \geq 2$, such that for every subset $B' \subseteq B$ of size at most 3, $\left|N_G(B')\right| \geq \left|B'\right|$. Then, $|B| \leq 8\,|R|$ and this bound is tight.*

Proof. Let $n_b = |B|$ and $n_r = |R|$. Our goal is to prove that $n_b \leq 8n_r$. Note that no vertex in B can have degree 0, otherwise the neighborhood of such a vertex is of size 0, violating the conditions of the theorem. We make a new graph G' by adding edges in G to all vertices of B which have degree 1 in G. This can always be done while maintaining the planarity and bipartiteness of the graph as any such vertex v must lie in a face which has at least two vertices of R, at least one of which is not adjacent to v. Thus in G' every vertex in B has degree at least 2. Let n_{b_2} be the number of vertices of B which have degree 2 and $n_{b_{\geq 3}} = n_b - n_{b_2}$ be the number of vertices of B which have degree at least 3 in G'. Since G' is planar and bipartite the number of edges in $G' \leq 2(n_b + n_r)$. This implies that $2n_{b_2} + 3n_{b_{\geq 3}} \leq 2n_b + 2n_r$. Since $n_b = n_{b_2} + n_{b_{\geq 3}}$, we obtain $n_{b_{\geq 3}} \leq 2n_r$.

We now show that $n_{b_2} \leq 6n_r$. To do that we construct a graph H with vertex set R as follows: two vertices $r_1 \in R$ and $r_2 \in R$ are adjacent in H iff there is at least one vertex $b \in B$ of degree 2 which is adjacent to both r_1 and r_2 in G'. Note that H is planar since the edge between r_1 and r_2 can be routed via one such b. Note that for the same pair $\{r_1, r_2\}$ there cannot be three vertices $b_1, b_2, b_3 \in B$ of degree 2 each that are adjacent to both r_1 and r_2 since in that case the neighborhood of the set $\{b_1, b_2, b_3\}$ is of size 2 violating the conditions of the theorem. Therefore, each vertex $b \in B$ of degree 2 corresponds to an edge in H and each edge has at most two vertices in B that correspond to it. Since the number of edges in H is at most $3|R| = 3n_r$, we conclude that $n_{b_2} \leq 6n_r$. Thus $n_b = n_{b_2} + n_{b_{\geq 3}} \leq 6n_r + 2n_r = 8n_r$. $\square$

We now show that the bound given above is tight. However, that still leaves open the possibility that, by exploiting other properties of disks, a $(3,2)$-local search could give a better approximation for the problem of computing minimum hitting sets for disks in the plane. The following theorem rules this out.

Theorem 8.5. *For any $\delta > 0$, one can construct a set $\mathcal{D}$ of disks in the plane, a set of points P and a subset $B \subseteq P$ s.t. i)B is a hitting set for $\mathcal{H}(P, \mathcal{D})$, ii) $|B| \geq (8-\delta)$*OPT *and iii) there are no subsets $X \subseteq B$ and $Y \subseteq P \setminus B$, $|Y| < |X| \leq 3$, s.t. $(B \setminus X) \cup Y$ is a hitting set for $\mathcal{H}(P, \mathcal{D})$.*

Proof. We first construct a bipartite graph $G = (R, B, E)$ that satisfies the conditions of Theorem 8.4 and $|B| \geq (8-\delta)|R|$. Let L be the triangular lattice, and take a large equilateral triangle Δ aligned with the edges of L (so that $L \cap \Delta$ triangulates Δ) and containing many faces of the lattice. Then replace each face of the lattice by the block of the type shown in Figure 8.1(left). The corner vertices (unshaded) of the block map to the corner vertices of the face, while the other vertices (shaded) in the block lie in the interior of the face (note that each edge of the face has two shaded vertices near it). Let R be the set of vertices of L lying in Δ and let B be the set of vertices lying in the interior of the faces in $L \cap \Delta$. The blocks together define a bipartite graph (see Figure 8.1(right) for a small example with four blocks put together). The dotted edges and the edges of the lattice L are not part of the graph. If Δ is large enough, the number of faces of L in Δ is nearly twice the number of vertices of L in Δ. Thus by making Δ large enough, we can ensure that $|B| \geq (8-\delta)|R|$ since each face in $L \cap \Delta$ contains four points of B. It can be verified by inspection that there is no subset of B of size at most 3 with a smaller neighborhood. This shows that the bound in Theorem 8.4 is tight within additive constants.

Now, we extend G to a triangulation by including the dotted edges in the blocks. Note that there are some dotted edges going between blocks. We also put an additional vertex in the outer face and connect it to all vertices in the outer face of G (i.e. we stellate the outer face). The resulting graph, call it Ξ, is triangulated (i.e., each face is of size 3) and furthermore it is 4-connected since, as can be verified by inspection, there is no separating triangle (a non-facial cycle of length 3). By a theorem of Dillencourt and Smith (Theorem 3.5 in [DS96]), there exists an embedding of Ξ in the plane so that Ξ is the Delaunay triangulation of its vertices. Abusing notation, we refer to the embedding as Ξ and we refer to the embedding of a vertex v in Ξ as v.

R and B are thus two sets of points. We set $P = R \cup B$, and construct $\mathcal{D}$ by taking for each edge e in G a disk that contains a disk that contains exactly the two end points of e among all the vertices in Ξ. This is possible because Ξ is now a Delaunay triangulation of the points in P. By construction, each disk in $\mathcal{D}$ contains exactly one point from each of the sets R and B and thus both the sets are hitting sets for $\mathcal{H}(P, \mathcal{D})$. Since $\textsc{Opt}$ is the size of the smallest hitting set, $\textsc{Opt} \leq |R|$ and therefore $|B| \geq (8-\delta)\textsc{Opt}$. Consider a local improvement step where we seek to decrease the size of the hitting set B by removing some subset $X \subseteq B$ of size at most 3 and adding a smaller set Y outside B (i.e., $Y \subseteq R$) so that $(B \setminus X) \cup Y$ is a hitting set for $\mathcal{D}$. Let x be one of the points in X. Observe that then all neighbors of x in G must be in Y since for each neighbor y of x, there is a disk in $\mathcal{D}$ which contains only the two points x and y among all the points in $R \cup B$. This means that $|Y| \geq |N_G(X)|$. Since for any X of size at most 3, $|N_G(X)| \geq |X|$, we have that $|Y| \geq |X|$ implying that such a local improvement is not possible. $\square$

8.3 An $\tilde{O}(n^{7/3})$-time Algorithm for Local Search

Let S be a hitting set for $\mathcal{H}(P, \mathcal{D})$. For any $s \in S$, we denote by $\mathcal{D}(s)$ the set of disks in $\mathcal{D}$ that are hit by s but not by any other point in S. We will call the disks in $\mathcal{D}(s)$ the *personal disks* of s. We will denote the region $\bigcap_{D \in \mathcal{D}(s)} D$ by $R(s)$ and call it the *personal region* of s. The notations $\mathcal{D}(s)$ and $R(s)$ are always with respect to a set system $\mathcal{H}(P, \mathcal{D})$ and a hitting set S. These things that are not explicit in the notation will be clear from the context. We also extend the same definitions for sets of points: for a set $X \subseteq S$, let $\mathcal{D}(X)$ be the set of disks in $\mathcal{D}$ which

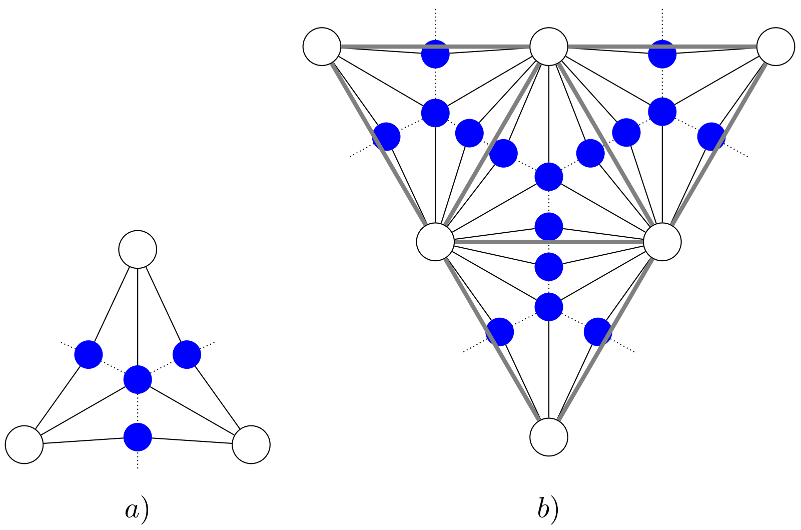


FIGURE 8.1: Unshaded vertices correspond to red and shaded to blue vertices. The dotted lines show a triangulation. We tile the triangles in (left) as shown in (right). The ratio of shaded to unshaded vertices goes to 8 as size of the tiling is increased. Connecting the vertices at the boundary of the tiling to a new vertex gives a 4-connected graph.

contain only points of X. We call these the personal disks of X. The personal region of X is $R(X) = \bigcap_{D \in \mathcal{D}(X)} D$.

Before presenting our algorithm, we prove a few results that will be useful for describing the algorithm.

Lemma 8.6. *Let S be a hitting set for $\mathcal{H}(P, \mathcal{D})$. If $|S| > 8 \cdot \textsc{Opt}(P, \mathcal{D}) + 3t$, for some integer $t \geq 0$, then there exist $t + 1$ disjoint subsets $X_0, \ldots, X_t$ of S, each of which is of size 3 and is locally improvable with respect to S in $\mathcal{H}(P, \mathcal{D})$.*

Proof. The proof is by induction. The statement is true for $t = 0$: if there is no locally improvable set X of size 3, then taking $B = S$ and $R = O$, where O is the optimal hitting set for $\mathcal{H}(P, \mathcal{D})$ and applying Theorem 8.4, we get that $|S| \leq 8\textsc{Opt}(P, \mathcal{D})$. Assume inductively that the lemma is true for $t - 1$, and let the t disjoint sets of S be $X_0, \cdots, X_{t-1}$. It remains to construct the set X_t. Let $Z = \bigcup_{i=0}^{t-1} X_i$. Let $P' = P \setminus Z$, $\mathcal{D}' = \{D \in \mathcal{D} : D \cap Z = \emptyset\}$ and $S' = S \setminus Z$. Clearly S' is a hitting set for $\mathcal{H}(P', \mathcal{D}')$. Moreover,

Lemma 8.7. $\textsc{Opt}(P', \mathcal{D}') \leq \textsc{Opt}(P, \mathcal{D})$.

Proof. Take any hitting set A for $\mathcal{H}((P, \mathcal{D}))$. Then any point $a \in A$ that hits a disk in $\mathcal{D}'$ must belong to P': otherwise $a \in P \setminus P' = Z$, and we had constructed $\mathcal{D}'$ by removing all the disks hit by Z from $\mathcal{D}$. Therefore all the points in A hitting $\mathcal{D}'$ belong to P', and form a hitting set in P' for $\mathcal{H}((P', \mathcal{D}'))$ of size at most $|A|$. □

Therefore, $|S'| = |S| - 3t > 8 \cdot \textsc{Opt}(P, \mathcal{D}) \geq 8 \cdot \textsc{Opt}(P', \mathcal{D}')$, and any hitting set for $\mathcal{H}(P, \mathcal{D})$ contains a hitting set for $\mathcal{H}(P', \mathcal{D}')$. Now, using the Theorem 8.4 for $t = 0$ on S' and $(P', \mathcal{D}')$, the fact that $|S'| > 8 \cdot \textsc{Opt}(P', \mathcal{D}')$ implies that there is set $X_t \subseteq S'$ of size 3 and a set $Y \subseteq P'$ of size 2 such that $(S' \setminus X_t) \cup Y$ is a hitting set for $\mathcal{H}(P', \mathcal{D}')$. This means that $(S' \setminus X_t) \cup Y \cup Z$ is a hitting set for $\mathcal{H}(P, \mathcal{D})$ since all disks in $\mathcal{D} \setminus \mathcal{D}'$ intersect Z. In other words, $(S \setminus X_t) \cup Y$ is a hitting set for $\mathcal{H}(P, \mathcal{D})$ since $S' \cup Z = S$ and $X_t \cap Z = \emptyset$. That is, X_t is locally improvable with respect to S in $\mathcal{H}(P, \mathcal{D})$. Since $X_t \subseteq S'$ and $S' \cap Z = \emptyset$, X_t is disjoint from the other X_i's. □

The following key structural property is crucial for the efficiency of the algorithm:

Lemma 8.8. *Let S be a hitting set for $\mathcal{H}(P, \mathcal{D})$. Then the personal regions of the points in S form a collection of pseudodisks.*

Proof. First observe that since each personal region $R(s)$ is an intersection of disks, it is convex. We show that for any two points $x, y \in S$, $R(x)$ and $R(y)$ are non-piercing i.e., the regions $R(x) \setminus R(y)$ and $R(y) \setminus R(x)$ are connected. Since the boundaries of two convex regions that are non-piercing cannot intersect in more than two points, we conclude that the regions form a collection of pseudodisks.

Assume for contradiction that $R(x)$ and $R(y)$ are piercing and without loss of generality that $R(x) \setminus R(y)$ has at least two connected components. The point x lies in one of these components and let x' be a point in a different component of $R(x) \setminus R(y)$. Since x' does not lie in $R(y)$, there must be a disk $D_y \in \mathcal{D}(y)$ that does not contain x'. Since no disk in $\mathcal{D}(y)$ contains x, D_y also does not contain x. Since D_y contains $R(y)$, this implies that x and x' are in different connected components of $R(x) \setminus D_y$. In other words $R(x) \setminus D_y$ is not connected. There are two cases to consider now:

Case 1: $D_y \setminus R(x)$ **is not connected.** In this case y lies in one of the connected components of $D \setminus R(x)$. Let y' be a point in one of the other components. Since $y' \notin R(x)$, there is some disk $D_x \in \mathcal{D}(x)$ which does not contain y' and by assumption does not contain y. Since D_x contains $R(x)$, this implies that $D_y \setminus D_x$ is not connected, a contradiction for disks.

Case 2: $D_y \setminus R(x)$ **is connected.** In this case, since $R(x) \setminus D_y$ is not connected, the situation is as shown in Figure 8.2. There is at least one point y' where the boundaries of D_y and $R(x)$ intersect but do not cross. Furthermore, y' does not lie on the boundary of $D_y \setminus R_x$. Since y' lies on the boundary of R_x, it lies on the boundary of some disk $D_x \in \mathcal{D}(x)$. Note that D_x contains R_x but does not contain y. Therefore, D_x must intersect the boundary of D_y at least twice in the portion of ∂D_y that lies outside R_x, i.e., the arc between a and b (shown in the figure) in counterclockwise direction. The boundaries of D_x and D_y then intersect at least three times, a contradiction for disks. □

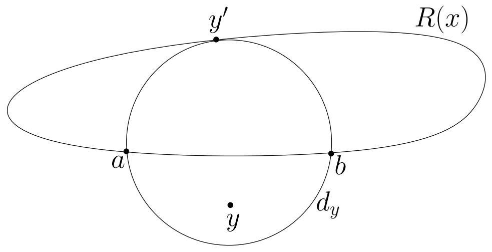


FIGURE 8.2: Case 2 in the proof of Lemma 8.8

Remark: Lemma 8.8 holds even when D is a set of pseudodisks. In the proof we have only used the fact that boundaries of disks cannot intersect more than twice.

Lemma 8.9. *Let S be a hitting set for $\mathcal{H}(P, \mathcal{D})$. Suppose that we are given two sets $X \subseteq S$ and $Y \subseteq P$ such that $|Y| = O(1)$, $|X| > 4|Y|$ and for each $x \in X$, Y hits $\mathcal{D}(x)$, the personal disks of x. Then there exists a set $X' \subseteq X$ of size $\Omega(|X|)$ such that (X', Y) is a local improvement pair with respect to S and $\mathcal{H}(P, \mathcal{D})$. Furthermore, given X and Y, X' can be computed in time $O(|X| \log |X|)$.*

Proof. Consider the Delaunay triangulation of the points in X, and let X' be an independent-set in this Delaunay graph. First we show that $(S \setminus X') \cup Y$ is a hitting set for $\mathcal{H}(P, \mathcal{D})$. Consider a disk D that is not hit by $S \setminus X'$. Since D is hit by S (S being a hitting set for $\mathcal{H}(P, \mathcal{D})$), D contains at least one point of X'. If D contains exactly one point $x' \in X'$ then D is hit by Y since $D \in \mathcal{D}(x')$ and Y hits $\mathcal{D}(x')$. Otherwise, D contains at least two points of X' in which case it must contain some point of $x \in X \setminus X' \subseteq S \setminus X'$ since X' is an independent set in the Delaunay triangulation of X.

The Delaunay triangulation can be constructed in $O(|X| \log |X|)$ time. If $|X| \leq 5|Y|$, i.e. $|X| = O(1)$, we find an independent set of size at least $\lceil |X|/4 \rceil > |Y|$ in the Delaunay graph in $O(1)$ time by brute force; the existence of such an independent set follows from the 4-color theorem on planar graphs. If $|X| > 5|Y|$, we compute a 5-coloring of the Delaunay graph in $O(|X|)$ time and take the largest color class as X'. Thus $|X'| \geq \lceil |X|/5 \rceil > |Y|$.

Therefore $|X'| > |Y|$, and so (X', Y) is a local improvement pair. □

Lemma 8.10. *Let $\mathcal{D}$ be a set of m disks in the plane having a common intersection region, say R. Then the boundary of R is composed of $O(m)$ circular arcs, and can be computed in $O(m \log m)$ expected time. We can also construct, in $O(m \log m)$ time, a data structure which, for any given query point q, answers whether $q \in R$ in $O(\log m)$ time.*

Proof. Let p be a point in the region R and D be disk in $\mathcal{D}$. Consider the function $f_D(\theta)$, for $\theta \in [0, 2\pi)$, to be the distance from p to the boundary of D in the direction θ (i.e., along a ray emanating from p at an angle θ to the positive x axis). Clearly the graph of $f_D(\theta)$ with

θ plotted along the x-axis is an x-monotone curve which we denote by Γ_D. Furthermore, for any two disks $D, D' \in \mathcal{D}$ the curves Γ_D and $\Gamma_{D'}$ intersect at most twice since the boundaries of any two disks intersect at most twice. The curves in $\{\Gamma_D : D \in \mathcal{D}\}$ therefore form a set pseudo-parabolas. The number of arcs in the boundary of R is equal to the size of the lower envelope of the arrangement of curves, which is $O(m)$ due to the linear union complexity of pseudo-parabolas [PS06; APS07].

The boundary of R can be computed by a randomized incremental construction in $O(m \log m)$ in the same way as the union of a set of m regions with linear union complexity is computed in the same amount of time. There is also a deterministic algorithm which takes $O(m \log^2 m)$ time. It basically splits the disks in $\mathcal{D}$ into disjoint sets of half the size, computes the intersection region for each, and then takes the intersection of the two intersection regions - which can be computed using a circular sweep. For details see [OWW85; EHS04; APS07] and the references therein.

Once R has been computed, we can easily set up a data structure that checks for a query point q whether $q \in R$. The region R is almost like a convex polygon except that its sides are circular arcs. We can take one vertex v of the region R and join it with a chord to each of the other vertices. The chords define a linear order. Given a query point q, we can determine, by binary search, two consecutive chords so that the cone defined by v and the rays emanating from v along the two chords contains q. Next we just need to check whether q lies on the same side of the circular arc bounding R in that cone, as v. Setting up this data structure takes $O(m)$ time and the query time is $O(\log m)$ as required.

□

Lemma 8.11. *Let P be a set of n points in the plane and let $\mathcal{D}$ be a set of pseudodisks, the boundary of each being composed of circular arcs. For any constant C, we can compute, for each $p \in P$ that lies in at most C pseudodisks, the exact set of pseudodisks it hits in $O(n \ \log \ m)$ time, where m is the total number of arcs in all the pseudodisks.*

Proof. We first compute the shallow levels (of depth at most C) of the arrangement of pseudodisks $\mathcal{D}$. This can be done using a randomized incremental construction in exactly the same way as the union of a set of pseudodisks is computed. This takes $O(m \log m)$ time since the overall complexity of the shallow levels (for constant C) is linear. For details see [OWW85; EHS04; APS07]. Once the shallow levels are computed, we set up a point location data structure. Then in $O(\log m)$ time per point, we can determine for each point exactly which of the at most C pseudodisks it is contained in. □

We now describe our algorithm for computing a small hitting set for $\mathcal{H}(P, \mathcal{D})$. We first compute a hitting set S of size $O(\textsc{Opt})$. This can be done using the near linear time algorithm of

Agarwal-Pan [AP14]. We also assume that we know the value of $\textsc{Opt} = \textsc{Opt}(P, D)$ although it suffices to guess the value of $\textsc{Opt}$ within a $(1+\epsilon)$ factor which can be done in $O(1/\log(1+\epsilon))$ guesses since we know $\textsc{Opt}$ within a constant factor. Throughout this section, we will use n as the total input size. We therefore upper bound $|P|$ and $|D|$ by n.

We first prune the input so that no point is contained in more than $\Delta = n/(\epsilon \cdot \textsc{Opt})$ disks. This can be done by iterating over each point $p \in P$ and computing the number of disks $\mathcal{D}' \subseteq \mathcal{D}$ that contain p. If $|\mathcal{D}'| \geq \Delta$, remove the disks in $\mathcal{D}'$ from $\mathcal{D}$ and add the point p to the set Q (which is initially empty). Note that as we go over the points the set $\mathcal{D}$ changes but we do not change the value of Δ. Since each time we add a point to Q, we remove at least Δ disks from $\mathcal{D}$, $|Q| \leq n/\Delta = \epsilon \cdot \textsc{Opt}$. We can thus add the set Q to our hitting set at the cost of an added ϵ in our approximation factor. This preprocessing procedure takes $O(n^2)$ time (this will not be the bottleneck of our algorithm).

After preprocessing, we pass P and D to Algorithm 9 which we describe now. It requires an initial hitting set S of size $O(\textsc{Opt})$ which we obtain from [AP14]. The goal of Algorithm 9 is to compute a hitting set whose size is at most $(8+\epsilon) \cdot \textsc{Opt}$. We compute a value $t = |S| - 8 \cdot \textsc{Opt}$ which indicates how far we are from the solution we seek. As we will see, when t is large, progress can be made quickly. However as we approach the quantity $8 \cdot \textsc{Opt}$, progress becomes slower and slower. The algorithm uses only local improvements of the type (X, Y) where $|Y| \leq 2$. Throughout the algorithm we maintain for each $D \in \mathcal{D}$, the number of points N_D it contains from S. Initially computing N_D for each disk takes $O(n^2)$ time. After that we need to update these quantities only when a local improvement (X, Y) happens. We update N_D as follows: $N_D = N_D - |D \cap X| + |D \cap Y|$. Since $|Y|$ is always at most 2 in our algorithm, naively this takes time $O(n|X|)$. Since such a local improvement decreases the size of the hitting set S by $|X| - 2 = \Omega(|X|)$, the overhead for maintaining N_D is $O(n)$ per improvement. Let `LocallyImprove(X,Y)` be the procedure that updates S to $(S \setminus X) \cup Y$ and updates N_D for each disk as mentioned above.

In each iteration of the `while` loop in Algorithm 9, we first construct a range reporting data structure [AC09] for the points in S so that given any disk D, we can find the set of points in $D \cap S$ in time $O(\log n + |D \cap S|)$. We then use this data structure to compute the personal disks of each $s \in S$ as follows. Iterate over each disk $D \in \mathcal{D}$ and if $N_D = 1$, use the reporting data structure to find the single point $s \in S$ that is contained by D. We then add D to the (initially empty) list of personal disks of s. Since each query takes $O(\log n)$ time, the total time taken to compute the personal disks is $O(n \log n)$. If we find some point $s \in S$ for which $\mathcal{D}(s) = \emptyset$, we can just remove s from the current hitting set. In other words we do a local improvement $(\{s\}, \emptyset)$.

The algorithm iterates over the points in P in random order, considering the possibility of replacing each point in a local-improvement step. Say the current point being considered is p_1;

Algorithm 9: Algorithm for $(8+\epsilon)$-approximation.

```
Data: A point set $P$, a set of disks $\mathcal{D}$, a hitting set $S$ of $\mathcal{H}(P,\mathcal{D})$ with
      $|S| = O(\textsc{Opt}(\mathcal{H}(P,\mathcal{D})))$, the size of the optimal hitting set $\textsc{Opt} = \textsc{Opt}(\mathcal{H}(P,\mathcal{D}))$, and
      a parameter $\epsilon > 0$.
1  For each disk $D \in \mathcal{D}$ compute $N_D = |D \cap S|$ // takes $O(n^2)$ time
2  while $t = |S| - 8 \cdot \textsc{Opt} > \epsilon \cdot \textsc{Opt}$ do
3      Construct a range reporting data structure for $S$ for disk ranges
4      For each $s \in S$ compute $\mathcal{D}(s) = \{D \in S : D \cap s = \{s\}\}$// use range reporting
5      if $\mathcal{D}(s) = \emptyset$ for some $s \in S$ then
6          LocallyImprove($\{s\}, \emptyset$) // $s$ is dropped from the hitting set
7          continue // with the next iteration of the while loop on line 2.
8      $\pi =$ A random permutation of the points in $P$
9      for $i$ = 1 to $|P|$ do
10         $p_1 = \pi_i$
11         for each $s \in S$ do
12             Compute: $\mathcal{D}'(s) = \{D \in \mathcal{D}(s) : p_1 \notin D\}$, $R'(s) = \bigcap_{D \in \mathcal{D}'(s)} D$
           // The above loop takes $O(n \log n)$ time
13         Let $\mathcal{R}' = \{R'(s) : s \in S\}$ // $\mathcal{R}'$ is a set of pseudodisks
14         $M = \{s \in S : R'(s) = \emptyset\}$
15         for each $p \in P$ do
16             Compute $\alpha(p)$ s.t. $0.9 \cdot \text{depth}(p,\mathcal{R}') \le \alpha(p) \le \text{depth}(p,\mathcal{R}')$
               // depth(p,$\mathcal{R}'$) denotes the number of regions in $\mathcal{R}'$ containing $p$
17         Let $q = \arg\max_{p \in P} \alpha(p)$
18         Set $\beta = \max\{\sqrt{t}, \epsilon t \cdot \textsc{Opt}/n\}$
19         if $|M|/5 + \alpha(q) \ge \frac{i\beta}{Cn \log n}$ then
20             Compute $S'(q) = \{s \in S : q \in R'(s)\}$ // Note that $|S'(q)| = \text{depth}(q,\mathcal{R}')$
21             if $|S'(q) \cup M| \ge 9$ then
22                 Compute an independent set $X \subseteq S'(q) \cup M$ in the Delaunay triangulation of
                       $S'(q)$ of size at least 3 and $\Omega(|S'(q) \cup M|)$ // $O(n \log n)$ time
23                 LocallyImprove($X, \{p_1, p_2 = q\}$)
24                 break // exit for loop
25             else
26                 For each $p_2 \in P$, set $S'(p_2) = \{s \in S : p_2 \in R'(s)\}$ // $O(n \log n)$ time
                   // Since $|S'(q) \cup M| \le 8$, $|S'(p_2) \cup M| \le \lfloor 8/0.9 \rfloor = 8$ for all $p_2 \in P$
27                 Enumerate all pairs $(X, p_2)$ where $p_2 \in P$, $X \subseteq S'(p_2) \cup M$ and $|X| \le 3$
28                 if for any $(X, p_2)$ enumerated, $(X, \{p_1, p_2\})$ is a local improvement pair then
29                     LocallyImprove($X, \{p_1, p_2\}$))
30                     break // exit for loop
```

the goal is to find a point p_2 so that $\{p_1, p_2\}$ can replace a large set X, i.e., a local improvement pair $(X, \{p_1, p_2\})$ of large profit. If we can find such a profitable local improvement, we make the improvement, exit from the `for` loop, and continue with the next iteration of the `while` loop. Otherwise, we continue with the next point in the random ordering. For any pair of points $Y = \{p, q\} \subseteq P$, denote by $\rho(Y)$ the number of points in S all of whose personal disks are hit by Y. For a point $p \in P$, we use $\rho(p)$ to denote $\max_{q \in P \setminus S}\{\rho(\{p, q\})\}$. Call a point $p \in P$ *useful* if there exists some $q \in P$ so that for some $X \subseteq S$, $(X, \{p, q\})$ is a local improvement pair.

Lemma 8.12. *If p_1 is useful, we can compute in $O(n \log^2 n)$ time a local improvement of profit $\Omega(\rho(p_1))$.*

Proof. Let us start by considering how we could compute $\rho(p_1)$. In order to compute $\rho(p_1)$, we need to find a point q so that the number of points $s \in S$ whose personal disks are hit by $\{p_1, q\}$ is maximized. To do this, we first compute for each $s \in S$, the set $\mathcal{D}'(s)$ of disks in $\mathcal{D}(s)$ that are not hit by p_1. For each $s \in S$, we then construct the region $R'(s)$ by taking the intersection of the disks in $\mathcal{D}'(s)$. Let $\mathcal{R}' = \{R'(s) : s \in S\}$. For some $s \in S$, $\mathcal{D}'(s)$ may be empty and consequently some of the regions in $\mathcal{R}'$ are empty. Let $M = \{s \in S : \mathcal{D}'(s) = \emptyset\}$. The personal disks of the points in M are hit by p_1 alone. The regions in $\mathcal{R}'$ define an arrangement of pseudodisks (Lemma 8.8). In this arrangement we seek to find a point $q \in P$ of the maximum depth. However, instead of finding a point with the maximum depth, we find a point whose depth is within a constant factor of the maximum. We construct, in $O(n \log n)$ time, an approximate depth query data structure for the pseudodisks in $\mathcal{R}'$ using Corollary 5.9 in [AHP08] with a constant $\epsilon \leq 0.1$. This takes $O(n \log n)$ time. Then, for each point $p \in P$, we compute a value $\alpha(p)$ s.t. $0.9\ \text{depth}(p, \mathcal{R}') \leq \alpha(p) \leq \text{depth}(p, \mathcal{R}')$ where $\text{depth}(p, \mathcal{R}')$ denotes the depth of p in the arrangement of regions in $\mathcal{R}'$. This takes $O(\log^2 n)$ time per point and so the overall time taken is $O(n \log^2 n)$. We then take the point p with the maximum $\alpha(p)$ as q. Observe that $|M| + \alpha(q) = \Theta(\rho(p_1))$.

We first compute the set $S'(q) = \{s \in S : q \in R'(s)\}$. Note that $|S'(q)| = \text{depth}(q, \mathcal{R}') \geq \alpha(q)$. There are two cases to consider:

Case 1: $|S'(q) \cup M| > 8$. In this case, we set $p_2 = q$ and let $Y = \{p_1, p_2\}$. Using Lemma 8.9, we can find a subset $X \subseteq S'(q) \cup M$ so that $X = \Omega(|S'(q) \cup M|)$ so that $(X, \{p_1, p_2\})$ is a local improvement pair. Note that $|X|$ is $\Omega(\rho(p_1))$. Thus in this case, we conclude that p_1 is useful and indeed we have found a local improvement that decreases the size of the current hitting set by $\Omega(\rho(p_1))$.

Case 2: $|S'(q) \cup M| \leq 8$. In this case $S'(q) \leq \lfloor 8/0.9 \rfloor = 8$ for all $p \in P$. This means that $\rho(p_1) = O(1)$ and we just need to find one set X of size 3 and a point p_2 so that $(X, \{p_1, p_2\})$ is a local improvement pair. Using Lemma 8.11, we compute the set $S'(p)$ for all $p \in P$ in

$O(n \log n)$ expected time. For each $p_2 \in P$, we need to check if there is any subset X in $S'(p_2) \cup M$ of size 3 so that $(X, \{p_1, p_2\})$, is a local improvement pair. Since $|S'(p_2) \cup M| \leq 8$, there are at most a $\binom{8}{3}$ subsets $X \subseteq S'(p_2) \cup M$ for which we need to check if $(X, \{p_1, p_2\})$ is a local improvement pair. Thus there are $O(n)$ pairs of the form $(X, \{p_1, p_2\})$, where $|X| = 3$, that we need to check. For a particular pair of this form, we basically need to verify that all the disks in $\mathcal{D}$ whose intersection with S is a subset of X are hit by either p_1 or p_2. To make things simpler, we first remove from $\mathcal{D}$ all the disks that are hit by p_1 and obtain a set $\mathcal{D}' \subseteq \mathcal{D}$. Now, we need to verify for all disks in $\mathcal{D}$ whose intersection with X is a subset of X that they are hit by p_2. All the $O(n)$ pairs can be checked in $O(n \log n)$ time as follows.

We construct a data structure that will help us do the checking for all the $O(n)$ pairs of the form $(X, \{p_1, p_2\})$. We have already constructed a range reporting data structure on S for disk ranges. Additionally, use a dictionary data structure (based on balanced binary trees) in which the keys are subsets of S of size at most 3 and the value corresponding to a key U is a list of disks $D \in \mathcal{D}'$ s.t. $D \cap S = U$. We start with an empty dictionary. We then go over each disk $D \in \mathcal{D}'$ one by one and if $N_D \leq 3$, we use the range reporting data structure to get $U = D \cap S$ in $O(\log n)$ time. We search the dictionary for U and if it is found, we add D to its list. If no entry is found, we create an entry for U with a single element d in its list. Note that since the number of (≤ 3)-sets that can be obtained from set of n points by intersecting it with a set of disks is linear in the number of points [Mat02], the number of distinct keys in the dictionary is $O(n)$. We go over each key U and construct the region $R'(U)$ by taking the intersection of all the disks in the list associated with U. Note that $R'(U)$ can be constructed in $O(m \log m)$ time where m is the size of the list associated with U using Lemma 8.10. Since each disk is in the list of at most one U, the overall time is $O(n \log n)$. In the same amount of time, for each key U, we set up a data structure that allows us to check if a query point q is in $R'(U)$ using Lemma 8.10. Now, to check if a pair $(X, \{p_1, p_2\})$ is an improvement pair, we go over all subsets $U \subseteq X$ and check if $p_2 \in R'(U)$. The time spent for any pair is now $O(\log n)$. Therefore checking all the $O(n)$ pairs takes $O(n \log n)$ time.

If we find that none of the pairs we checked are local improvement pairs, then we can conclude that p_1 is not useful. □

The following lemma will allow us to find a profitable local improvement quickly. Let $\beta = \max\{\sqrt{t}, \epsilon t \cdot \textsc{Opt}/n\}$.

Lemma 8.13. *There exists a $k > 0$ such that there are at least $\Omega(\beta/k)$ useful points $p \in P$ with $\rho(p) \geq k$.*

Proof. By Lemma 8.6, there exists $\Omega(t)$ local improvement pairs $(X_0, Y_0), \ldots$ where the X_i's are disjoint subsets of S but the Y_i's need not be disjoint. Each X_i is of size 3 and each Y_i is of size 2. For any pair of points $Y = \{p_1, p_2\} \subseteq P$, if (X_i, Y) is a local improvement pair among the $\Omega(t)$ pairs, then we say that X_i is a triple assigned to the pair Y. Define the weight of Y as the number of triples assigned to it and denote it by $W(Y)$. The total weight of all pairs is then $\Omega(t)$.

Call a pair Y to be of type i if $2^{i-1} \leq W(Y) < 2^i$, for $i = 1, \ldots, O(\log t)$. If $W(Y) = 0$ then we say that Y is of type 0. Since the total weight of all pairs is $\Omega(t)$, there must be some $j > 0$ so that the total weight of the pairs of type j is $\Omega(t/2^j)$. Let $Q = \bigcup_Y \{Y \mid Y \text{ is of type } j\}$.

There are two lower bounds on the size of Q. First, since the total weight of the pairs of type j is $\Omega(t/2^j)$, and each pair has weight less than 2^j, the number of pairs is $\Omega(t/2^{2j})$, and hence $|Q| = \Omega(\sqrt{t}/2^j)$. On the other hand, for any local improvement pair (X_i, Y) where Y is of type j, take any point $x \in X_i$. Since $\mathcal{D}(x)$ is non-empty, any disk $D \in \mathcal{D}(x)$ contains at least one point in Y. Therefore any such local improvement pair leads to an incidence between a point in Q and a disk in $\mathcal{D}$. Note that since the X_i's are disjoint these are distinct incidences. Thus there are $\Omega(t/2^j)$ incidences. Since by assumption no point in P, and therefore no point in Q, is in more than $n/(\epsilon \cdot \text{OPT})$ disks in $\mathcal{D}$, we have that $|Q| = \Omega(\epsilon t \cdot \text{OPT}/2^j n)$.

Therefore, $|Q| = \Omega\left(\max\{\epsilon t \cdot \text{OPT}/2^j n, \sqrt{t}/2^j\}\right) = \Omega(\beta/2^j)$. Observe that each $p \in Q$ is useful and $\rho(p) \geq 3 \cdot 2^j$. The lemma is therefore true for $k = 2^j$. □

We can now analyze the running time of our algorithm.

Running time: Preprocessing takes $O(n^2)$ time but this is dominated by the running time of Algorithm 9. Consider a single iteration of the `while` loop in Algorithm 9. If we find some point $s \in S$ for which $\mathcal{D}(s) = \emptyset$, we drop s from the current hitting set. This way we have improved the size of the hitting set at the cost of $O(n \log n)$ time. The total time spent on such improvements is at most $O(\text{OPT}\, n \log n) = O(n^2 \log n)$.

Otherwise, call a single iteration of the `while` loop *lucky* if the following is true:

$$\exists i \text{ such that the point } \pi_i \text{ is useful and } \frac{i}{\rho(\pi_i)} \leq \frac{Cn}{\beta}$$

for some constant C.

Lemma 8.14. *Probability that any iteration of the* `while` *loop is lucky is at least* $1/2$.

Proof. By Lemma 8.13, there exists a k such that there are $\Omega(\beta/k)$ points, say the set U, with $\rho(p) \geq k$. Consider the smallest index i s.t. $\pi_i \in U$. The expected value of i is $O(nk/\beta)$. Therefore, with probability at least $\frac{1}{2}$, $i \leq Cnk/\beta$ for some large enough C. Then,

$$\frac{i}{\rho(\pi_i)} \leq \frac{Ckn/\beta}{k} = \frac{Cn}{\beta}$$

□

Lemma 8.15. *For a lucky iteration of the `while` loop, let λ be the reduction in size of the current hitting set, and σ the time spent in this iteration. Then $\sigma/\lambda \leq Cn^2 \log^2 n/\beta$.*

Proof. As we go over the points in random over, for the current point $\nu = \pi_i$, we estimate $\rho(\nu)$ which allows us to check if $i/\rho(\nu) \leq Cn/\beta$. If so, assuming that the point ν is useful, we decrease the size of the current hitting set by $\Omega(\rho(\nu))$. If $i/\rho(\nu) > Cn/\beta$ or we discover that ν is not useful we move to the next point in the random order. However, since the iteration of the `while` loop is lucky, we will find some point $\nu = \pi_i$ which is useful and for which $i/\rho(\nu) \leq Cn/\beta$. For this point ν, we find a local improvement involving ν of value $\Omega(\rho(\nu))$ and the current iteration of the `while` loop ends. The total time spent in this iteration is $\sigma = O(i \cdot n \log^2 n)$ since we have seen i points so far and for each point we spend $O(n \log^2 n)$ time. The reduction in the size of the current hitting set is $\lambda = \Omega(\rho(\nu))$. Thus $\sigma/\lambda \leq \frac{i}{\rho(\nu)} \cdot n \log^2 n \leq Cn^2 \log^2 n/\beta$. □

Since any iteration of the `while` loop is lucky with probaility at least 0.5 and we can assume that all the iterations are lucky. This does not change the running time by more than a factor of 2.

Lemma 8.16. *The expected time taken to halve t is $O(n^{7/3} \log^2 n\epsilon^{-1/3})$.*

Proof. Lemma 8.15 tells us that the amortized amount of time spent for the reducing the size of the current hitting set by 1 is $O(n^2 \log^2 n/\beta)$. Since β is an increasing function of t, this decreases with t. However, t does not change by more than a factor of 2 until it is halved. So, the expected time for t to be halved is $O(t/2 \cdot n^2 \log^2 n/\beta)$. Now, $t/\beta = \min\{\sqrt{t}, n/(\epsilon \cdot \textsc{Opt})\}$. Since $t = O(\textsc{Opt})$, $t/\beta = O(min\{\sqrt{\textsc{Opt}}, n/(\epsilon \cdot \textsc{Opt})\} = O((n/\epsilon)^{1/3})$. Thus the expected time to halve t is $O(n^{7/3} \log^2 n\epsilon^{-1/3})$. □

Since the initial value of t is $O(\textsc{Opt})$, there are $O(\log 1/\epsilon)$ halving rounds until $t \leq \epsilon \cdot \textsc{Opt}$. Thus, the expected running time of the Algorithm 9 is $O(n^{7/3} \log^2 n\ \epsilon^{-1/3} \log(1/\epsilon))$. Finally, since we need to run Algorithm 9 for $O(1/\log(1+\epsilon))$ guesses for $\textsc{Opt}$, the overall running time is $O(n^{7/3} \log^2 n\ \epsilon^{-1/3} \log(1/\epsilon)/\log(1+\epsilon))$. For a fixed small value of ϵ, this is $O(n^{7/3} \log^2 n)$.

Chapter 9

Dynamic Convex Hull for Simple Polygonal Chains

This chapter describes an algorithm that constructs a dynamic convex hull of points in a simple polygonal chain. This chapter is based on the following paper.

Norbert Bus and Lilian Buzer. "Dynamic Convex Hull for Simple Polygonal Chains in Constant Amortized Time per Update." In: *Proceedings of the 31th European Workshop on Computational Geometry (EUROCG)*. 2015

9.1 General Idea

There exist many algorithms for computing the convex hull in both the real-RAM and the word-RAM model. If one considers the real-RAM model, an optimal output sensitive algorithm to construct the convex hull of n points in a plane was published in [Cha96] having $O(n \log h)$ time complexity where h is the output size. If the point set is a simple polygonal chain, the best algorithm, a result of Melkman, runs in linear time [Mel87]. If one requires the data structure to be dynamic, namely to handle insertions and deletions of arbitrary points an optimal algorithm requiring $O(\log n)$ time for both operations was proposed in [BJ02]. For points in a polygonal chain there has been no work on efficient deletion of points. Changing the computational model to the word-RAM model and using Graham's scan [Gra72] to construct a convex hull the running time is essentially the time to sort the points, taking, e.g., $O(n \log \log n)$ time [Han02]. Dynamic data structures supporting deletion and insertion in the word-RAM model require an optimal $O(\frac{\log n}{\log \log n})$ time for both operations assuming that word length is $\Theta(\log n)$, see [DP07].

In this chapter we give an on-line algorithm to construct the dynamic convex hull of a simple polygonal chain in the Euclidean plane supporting deletion of points from the back of the chain and insertion of points in the front of the chain. Both operations require amortized constant time considering the real-RAM model. The main idea of the algorithm is to maintain two convex hulls, for efficiently handling insertions and deletions. Implicitly these two hulls constitute the convex hull of the polygonal chain.

9.2 Algorithm

Overview of our algorithm

Our algorithm works in phases. For a precise formulation let us first define some necessary notations. A polygonal chain S in the Euclidean plane, with n vertices, is defined as an ordered list of vertices $S = (p_1, p_2, \ldots, p_n)$ such that any two consecutive vertices, p_i and p_{i+1} are connected by a line segment. A polygonal chain is called simple when it is not self-intersecting. For simplicity, we assume that the points are in general position. Our algorithm handles insertion and deletion of points into the current convex hull in the order induced by S. This results in the fact that the current convex hull always contains a contiguous subchain of S, let us denote it by $S_i^j = (p_i, \cdots, p_j)$ where $i \leq j$ and the points are effectively inserted/deleted in a FIFO manner. Let us denote the convex hull of S_i^j with C_i^j. Therefore, given a convex hull C_i^j, inserting a point results in C_i^{j+1} while removing the first point results in C_{i+1}^j.

At the beginning of each phase, we initialize a simple data structure called the *phase convex hull* that maintains the representation of the convex hull of a subrange of the polygonal chain. Each phase handles an arbitrary number of insertions and handles deleting the points that were present when the phase started. Assuming that the phase convex hull first covered S_a^b this means we can delete the points $p_a \ldots p_b$. A phase ends, when we first delete a point that was not covered by the initial convex hull. After that, a new phase starts and we initialize a new phase convex hull. See Figure 9.1.

$$\underbrace{C_a^b \to C_a^{b+1} \to C_{a+1}^{b+1} \to \cdots \to C_b^c}_{\text{phase starting with } S_a^b} \underset{\substack{\Big\downarrow \\ \text{initialize new} \\ \text{phase convex hull}}}{\longrightarrow} \underbrace{C_{b+1}^c \to C_{b+2}^c \to \cdots \to C_c^d}_{\text{phase starting with } S_{b+1}^c}$$

FIGURE 9.1: Example of two phases

We state the main result of our algorithm in Theorem 9.1.

Theorem 9.1. *The amortized time complexity of insertion and deletion of points in a convex hull of a simple polygonal chain is constant.*

Proof. Assume that each point has been inserted and later removed. In Section 24 we show that the i-th phase runs in $O(k_i + l_i)$ time where k_i is the number of insertions in the phase and l_i is the number of deletions. During the whole algorithm each point has been inserted and deleted exactly once hence there have been n insertions and n deletions overall. Therefore the overall running time of the phases is $O(n)$, yielding the desired result. □

Definitions

In this section we introduce notations, definitions and data-structures used in our algorithms. We start with the phase convex hull representing the convex hull of the polygonal chain at any step of the algorithm. Then we describe two other data-structures representing convex hulls that are maintained during the phase. These two objects' purpose is to enable that insertion and deletion of points for the phase convex hull run in constant amortized time. Each of these data-structures have to be initialized at the beginning of the phase, details can be found at the description of them. We suggest that the reader is familiar with the Melkman algorithm [Mel87] as our method builds heavily on it. Briefly, it constructs the convex hull of a polygonal chain by iteratively (in the proper order) adding the points to the convex hull and modifying it as necessary. This operation is based on checking a simple configuration of points.

At the beginning of the phase let S_a^b be the current polygonal chain while at the end let it be S_b^c. Let us denote by S_i^j the polygonal chain at an arbitrary step during the phase and C_i^j the corresponding convex hull.

The *phase convex hull* denoted by C^* is the data structure representing the convex hull C_i^j, containing all of its points in two dequeues. Every point is contained in exactly one of the two dequeues except for two: the front of both dequeues refer to the same point of the subchain, the one contained in C_i^j with highest index and similarly, the back of both dequeues refer to the same point of the subchain, the one contained in C_i^j with lowest index. We refer to the front of both dequeues as front opening and to the back as back opening. Connecting the two dequeues gives the ordered circular list of points in C_i^j. See Figure 9.2 for an illustration. Moreover, if one considers the back opening to be *closed* (i.e., as if being glued together, removing the duplicate copy of the back point), one has the data structure used in the Melkman algorithm. On the other hand, if the front opening is considered to be closed one has again the Melkman data structure but now points can be added at the other end of the chain.

Note that connecting the two dequeues is a constant time operation as there is no copy of data happening in order to create one dequeue, it is enough to virtually handle them as one dequeue.

At the beginning of the phase C^* is built for S_a^b. This initializing step can be carried out by using the Melkman algorithm to build a convex hull of S_a^b and then creating the two dequeues by splitting up the result of the Melkman algorithm at the point with the lowest index.

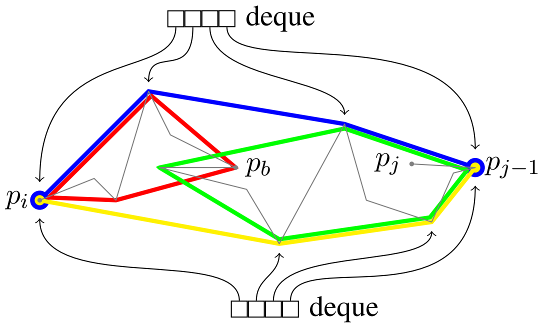


FIGURE 9.2: The two dequeues constituting C^* with corresponding points in blue and yellow. In green and red we depict C^+ and C^- respectively.

The *incremental convex hull* is the data-structure representing the convex hull of the points in the polygonal chain added after the initialization of the phase. The data-structure is a dequeue as in the Melkman algorithm and it is used the same way, namely we iteratively add the inserted points to it. Let us denote it by C^+, see Figure 9.2. At any state S_i^j, C^+ represents C_{b+1}^j. It is initialized to the empty set.

The *decremental convex hull* is a data-structure representing the convex hull of the points that can be deleted during the phase. It is initially built with the Melkman algorithm for the points present at the beginning of the phase but according to the reverse order of the points. Let us denote it by C^-, see Figure 9.2. At each deletion a point is removed from C^-. At any state S_i^j, C^- represents the convex hull of the points S_i^b, note that the data-structure corresponds to the Melkman data-structure for the subchain in reverse order. This data structure has to maintain additional information that will be used for efficient deletion of points.

First, while initializing C^-, for each point p_k in S_a^b, the list of points that were removed from the previous convex hull C_{k+1}^b in the Melkman algorithm should be kept. Let us call these points the *history* of a point and denote it by $H(p)$ for a point p. This enables one to 'rewind' the algorithm, i.e, the points of C^- can be deleted in a LIFO order while the convex hull of the current points can be maintained easily. As C^- was built in the reverse order this is exactly the deletion order we need.

Second, certain details of the polygonal regions defined by $C_k^b \backslash C_{k+1}^b$ for $a \leq k < b$, in other words the difference between consecutive convex hulls in the Melkman algorithm for building C^-, have to be kept. Clearly, some of these regions are empty as the convex hull might not change during its construction. For simplicity, let us denote *non-empty* regions by R_m such that m starts from 0 and corresponds to the order the regions appear. This way, R_0 is a simple edge,

R_1 is the first triangle, etc. In Figure 9.3 we show the regions where the red polygonal chain is S_a^b. The green line corresponds to the polygonal chain of the points inserted during the phase. Maintaining a complete description of a region is not required for our algorithm, instead, for each region only the edges (at most two) that are shared with a higher ordered region have to be kept, e.g., in Figure 9.3 R_5 has two neighboring higher ordered regions, R_6 and R_7. Let us denote the set of these edges by $E(R_m)$ and with each edge we store the two regions it belongs to. Setting up these edges for the regions can be done during the Melkman algorithm for constructing C^- and all regions have at most two such edges. As a consequence of the Melkman algorithm, these two edges are always those that contain the point p_k whose insertion (during the Melkman algorithm) created the region itself.

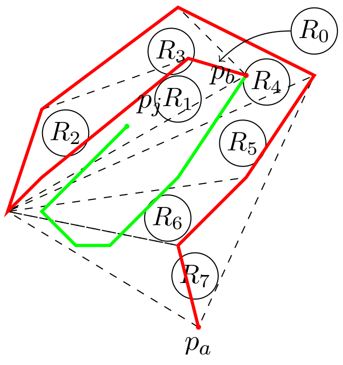


FIGURE 9.3: Regions.

We will need a simple property of the regions, namely that they form an ordered partitioning of the plane that enables a certain operation. See Lemma 9.3.

Lemma 9.2. *While initializing C^- for S_a^b one can create the ordered list of regions in $O(b-a)$ time.*

Proof. The statement is a simple consequence of the Melkman algorithm. □

Lemma 9.3. *For each phase, the ordered list of regions enables the maintenance of the highest ordered region that contains any point inserted during that phase. This operation takes $O(c-a)$ time for a phase.*

Proof. As an illustration for the lemma see Figure 9.3. The highest ordered region containing a point inserted in the phase is region no.7. The proof is by induction. Suppose we know the highest ordered region R_m containing a point of C^+. When inserting a new point p_{j+1} into C^+, one can check if this point is located in a higher ordered region by checking if the line segment between p_j and p_{j+1} intersects an edge in $E(R_m)$. If the line segment intersects one then we recursively carry out the same process for the region that shares this edge with R_m. If it does not intersect any of the edges in $E(R_m)$ then the current region is the one that is highest ordered and contains p_{j+1} Checking if a point exits one region is a constant time operation. The total number of times this operation is carried out in a phase is at most the number of regions in C^- and points in C^+ which is together linear in the number of points inserted or deleted during the phase. Proving that this method actually finds the highest ordered region is a consequence of the Melkman algorithm.

To complete our proof we have to show how to find the region of the first point p_{b+1} of C^+. In order to facilitate this search we create a list of the regions containing p_b as a vertex (this can be done during the construction of C^-). Assume we add an additional point on the line segment (p_b, p_{b+1}) infinitesimally close to p_b. Finding the region containing this point is now reduced to searching for the region whose two edges connected to p_b define the sector containing this additional point. Given this region we can find the region of p_b by applying the inductive step. Searching for the region of the additional point has $O(b - a)$ complexity.

□

Insertion

The main steps to insert points in amortized constant time is to add the point to C^* and to C^+. The latter is necessary for efficient deletions since points of C^+ might appear later on C^* after deleting a point.

In order to insert the point p_{j+1} into C^* it is sufficient to do one step of the Melkman algorithm. For that, consider the back opening of C^* to be closed, i.e., the two dequeues behave like one. One has to be cautious when the Melkman algorithm deletes the point being the back opening as the new back opening should be by our definition the point in C_i^j with least index. Indeed, this is the neighbor of the deleted opening. In order to show this, suppose that it is not true and let p_x, p_w be the back and front opening, the neighbor of the back opening p_y (the one that has not been deleted) and the new back opening p_z where $p_z \neq p_y$. If $z < y$ there were two polygonal chains S_x^z and S_y^j inside the convex hull that have to intersect, contradicting the simplicity of the chain. After C^* is updated, p_{j+1} has to be inserted into C^+ as well, which is done using the Melkman algorithm.

Moreover, as long as the front opening is one of the points in C^- i.e., $C^+ \subset C^-$ one has to update the highest ordered region of C^- containing any inserted point. We introduce *states* corresponding to the points' distribution in C^* (with respect to whether they belong to C^+ or C^-) in order to detect when is the update of the highest ordered region necessary. The following lemma describes how the points' distribution changes during the execution of the operations in a phase. It states that there cannot be arbitrary distributions, e.g., points from C^+ and C^- in an alternating order.

Lemma 9.4. *The points in C^* are partitioned into contiguous ranges according to whether they belong to C^- or C^+. At any step in a phase there are at most two such partitions, one containing points of C^+ and one containing points of C^-. The partitioning changes in a specific pattern during a phase: at the beginning there are only points from C^- in C^*; then two partitions; finally only points from C^+ are located in C^*. Let us call this the state of the algorithm and denote them by decremental, mixed or incremental respectively.*

Proof. The fact that C^* is partitioned into at most two parts is a consequence of the simplicity of the polygonal chain. Indeed, suppose the contrary. Clearly in this case there are at least 4 points A, B, C, D in C^*. Assume that $A, C \in C^+$ and $B, D \in C^-$. This implies that there is a simple polygonal chain that belongs to C^+ and connects A with C. Similarly for B, D and C^-. Since both chains are inside C^* they must intersect which contradicts the simplicity assumption of the complete chain. The strict ordering also follows easily since C^+ is monotonically growing while C^- is monotonically shrinking. □

Determining the state can be done by checking which convex hull (C^- or C^+) do the two openings of C^* belong to. Using this notation we can say that updating the highest ordered region has to be done only in the *decremental* state. For the pseudo code of inserting the next point see Algorithm 10.

Algorithm 10: Algorithm for a inserting the next point.

```
Data: $R^*$ highest ordered region containing a point of $C^+$
1 Procedure Insert ($p_{j+1}$)
2     Insert $p_{j+1}$ into $C^+$ // Melkman
3     Insert $p_{j+1}$ into $C^*$ // Melkman
4     if state == decremental then
5         while $(p_i, p_{j+1})$ exits $R^*$ into higher ordered region do
6             Update $R^*$
```

Deletion

The main idea to delete a point from the convex hull is to remove it from both C^* and C^- and reconstruct the convex hull in this area. To delete the point p_i there are several scenarios that have to be handled differently. As a common point in all cases, p_i has to be removed from C^- and its history has to be added to C^- ('rewinding' the Melkman algorithm). For a point p in C^- let us denote the set of its two neighbor vertices in the convex hull by $N^-(p)$. Similarly for the other hulls we use $N^+(p)$ and $N^*(p)$. Let us first group the different cases according to Lemma 9.4 for an illustration see Figure 9.4. There is only one case when the algorithm is nontrivial and we show that the method results in the desired convex hull. The running time analysis is left for Section 24.

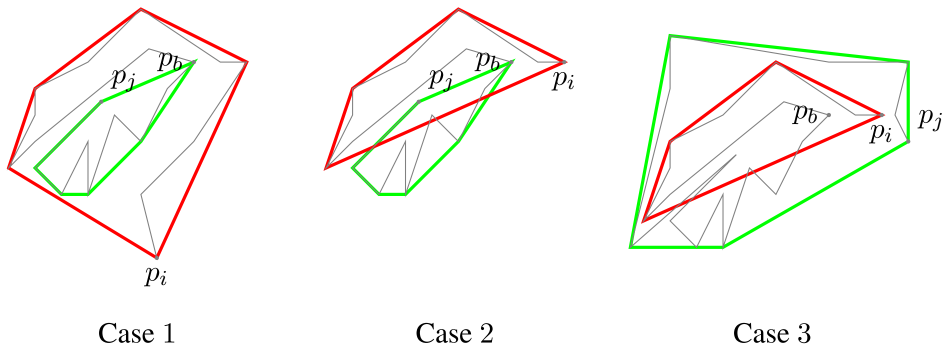


FIGURE 9.4: The three different cases for deleting the point p_i. C^+ and C^- is depicted green and red respectively. C^* is not illustrated.

Case 1 (decremental): If the phase convex hull contains only points from C^- then there are two possibilities, namely whether a point of C^+ has to be added to C^* or not. This can be simply checked since the highest ordered region is maintained exactly for this purpose. If there is no point of C^+, then one can simply remove p_i from C^* and add the points in $H(p_i)$ to C^* using the Melkman algorithm. If there is then an expensive operation is required, namely to add all the points of C^+ to C^*. This can be done with the Melkman algorithm considering the back opening to be closed. Even though this operation might require linear time in the number of insertions it can happen only once for each phase (due to Lemma 9.4) therefore its amortized time complexity is constant.

Case 2 (mixed): If the phase convex hull contains points from both C^+ and C^-, we have the most complicated case. Obviously the point p_i to be removed is in C^-. Let us denote the points in $N^*(p_i)$ by x and y. The edge between x and y would become an edge of C^* if there are no points in the triangle defined by p_i, x and y. Let us denote this triangle by Δ. If this is not the case one has to create new edges of C^* that correspond to the vertices located in Δ. Adding these vertices is done as follows. We further categorize this case into three sub cases depending on the neighbors of p_i.

Case 2a: If both x and y belong to C^- then clearly C^* can only be modified by the points from the history of the currently deleted point p_i. In such a situation using the Melkman algorithm one can add the ordered history of p_i to C^*.

Case 2b: If one point, e.g., x belongs to C^+ then there might be points of C^+ that have to be inserted into C^*. In this case first the history of p_i and the point q such that $q \in N^-(p_i), q \notin C^*$ should be inserted into C^* and then starting from x we shall add the vertices of C^+ in the circular order (starting with the point in $N^+(x)$ not being a vertex of C^*) using the Melkman algorithm. Adding points should be continued only as long as they create new vertices on C^*.

Case 2c: If both x and y belong to C^+ then a similar process has to be carried out namely first inserting the points of the history (with the points in $N^-(p_i)$) and then the vertices of C^+ in the proper circular order starting from x and y. Note that x and y define two different parts of C^+ that have to be inserted into C^* and to maintain a low running time one has to insert points from these two parts in an alternating order (one cannot proceed with points from the part of x after finishing the points starting from y). Denote these two parts by A_x and A_y. In order to be able to utilize the Melkman algorithm the added points have to belong to a simple polygonal chain, otherwise using Melkman would be impossible. This can be ensured by first creating the part of C^* within Δ and adding it to C^* by simply copying it. See Figure 9.5 for a schematic illustration of this setup. The details are as follows. We use the same double queue implementation of the Melkman algorithm as for C^*. This enables us to add points on both ends of the polygonal chain. Let us denote the points in $N^-(p_i)$ by x' and y' such that x' is on the 'side' of x. Add points in this structure in the following order $x', H(p_i), y'$ and then in an alternating order points from C^+ starting from x and y. The process continues until a new point is inside the convex hull. If one is inside we stop adding points from its part (e.g. A_x) and continue with points from the other part as long as an added point is inside the convex hull.

Proof. In Case 2a, after the removal of the point p_i, only points of C^- can be added to C^* otherwise it would contradict Lemma 9.4. It is trivial that these new points are exactly the history of p_i. In Case 2b and Case 2c we need a simple fact, namely that after deleting p_i only points of its history and $N^-(p_i)$ can become part of C^* in other words no other point of C^- can appear in C^*. This is implied by the fact that C^- is in the cone defined by p_i and $N^-(p_i)$ which is inside the cone defined by p_i, x and y (with p_i as apex in both cones). In Case 2b and Case 2c one has to argue that it is valid to apply the Melkman algorithm. This is trivial as the parts of the polygonal chains that are actually inserted are always simple since only the point located inside the convex hull could cause a self-intersection but at this point the algorithm stops.

□

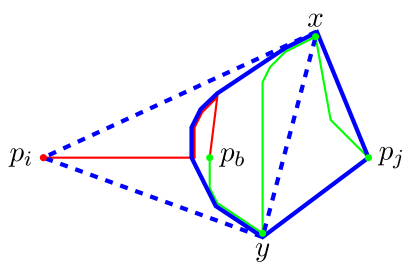


FIGURE 9.5: Deleting the point p_i. Solid blue lines denote the convex hull after deleting p_i. Gray arrows denote the points that have to be inserted into C^*.

Case 3 (incremental): If C^* contains only points of C^+ than C^- is inside C^+ therefore there is no change in C^*.

The pseudo code for deleting the next point is presented in Algorithm 11.

Algorithm 11: Algorithm for a deleting the next point.

```
Procedure Delete($p_i$)
    Delete $p_i$ from $C^-$and insert $H(p_i)$ into $C^-$ using Melkman
    // checking the state is based on whether the front and back openings
       belong to $C^+$ or $C^-$
    if state $==$ $decremental$ then                                  // Case 1
        if $R(p_i)$ not empty then
            Insert $S_{b+1}^{j+1}$ into $C^*$ using Melkman
        else
            Insert $H(p_i)$ into $C^*$ using Melkman
    if state $==$ $mixed$ then                                        // Case 2
        $x, y \leftarrow N^*(p_i)$
        if $x \in C^+$ and $y \in C^-$ then swap(x,y) // for Case 2b
        if $x \in C^-$ and $y \in C^-$ then                           // Case 2a
            Insert $H(p_i)$ into $C^*$
        else if $x \in C^-$ and $y \in C^+$ then                      // Case 2b
            Insert first $q \in N^-(p_i)/\{x\}$ and then $H(p_i)$ into $C^*$
            do
                Insert points from $C^+$ not in $C^*$ starting with $y$
            while $C^*$ changes
        else                                                          // Case 2c
            $T \leftarrow$ empty convex hull with two dequeues
            Insert $x', H(p_i), y'$ into $T$
            do
                Insert points from $C^+$ (not in $C^*$) starting from $y$ or $x$ in alternating order
            while $T$ changes
            Copy $T$ into $C^*$
```

Complexity

In this section we show that each phase takes time linear in the number of insertions and deletions. Let us denote them by k and l respectively.

Initialization: Clearly, initializing C^*, C^+ and C^- is linear in the number of points present at the beginning of the phase since we only utilize a modified Melkman algorithm. This indeed is the same as the number of points deleted in the phase. Therefore the complexity of initialization is $O(l)$.

Insertion: For all insertion using the Melkman algorithm for both the phase convex hull and the incremental convex hull takes $O(k)$ time. During insertion of points one has to also maintain the highest ordered region containing any point of C^+. This can be done in $O(k)$ time as well.

Deletion: Clearly, during executing the deletions, the number of points inserted into C^* using the Melkman algorithm is not more than a constant times the points in C^- which is $O(l)$ (this follows since the a point can appear as history only once) and the points of C^+ added to C^* which is less than $O(k)$ as no point can be added more than once to C^* due to the monotonicity of the operations.

This results in the following theorem.

Theorem 9.5. *The running time of one phase is $O(k + l)$ given that there are k insertions and l deletions.*

Chapter 10

Software

In our work we have introduced efficient algorithms for computing ϵ-nets and hitting sets for disks. In this section we will describe our code. Our methods rely on the Delaunay triangulation. Instead of implementing an algorithm to construct Delaunay triangulations we have decided to use the well-known CGAL library [The15]. It provides a rich source of efficient and reliable geometric algorithms and data structures written in C++. One of the unique features it provides is the possibility to carry out *exact computations* i.e., without numerical errors.

The source code can be found on the author's website. All of our code relies on the data structures and algorithms provided by CGAL, therefore it is necessary to use our software. We will here highlight some decisions and crucial parts of the algorithms. The code contained in this document is a subset of the real code only for presentation purposes.

The `enet` library

The library `enet` provides the implementation of our algorithm to construct small sized ϵ-nets for disks. In Listing 10.1 we give the interface of the algorithm. The comments describe the function of the parameters. We have defined a global variable `eps_c` for setting the probability with which the random sample is created. Setting it to 12 results in the smallest ϵ-nets theoretically while according to our experiments it worth lowering it to 7, resulting in even smaller ϵ-nets in practice. The parameter `counter` holds the number of points in the ϵ-net resulting from subproblems with $\epsilon' > 0.5$. The algorithms proposed for this case are not implemented (due to the lack of software for calculating centerpoints) therefore temporarily the same sampling algorithm is applied recursively instead of the more advanced constructions.

```
extern double eps_c;  //probability constant (eps_c/eps) for random sample ↩
    in the eps-net algo
```

```

/*!
* \brief Construct eps−net with random sampling.
* \param               input_points   The input points.
* \param [in, out] weights        The weights of input points.
* \param               epsilon        The epsilon.
* \param [out]        epsnet         The epsnet.
* \param [out]        counter        The counter for epsnet points resulting ↩
    from subproblems with epsilon >0.5.
* \param [in]         currentPoints The current points, to denote that only a ↩
    subset of points should be used (recursively calling the algorithm).
* return size of first random sample
*/
size_t randomSampleEpsnet(
            const std::vector<Point> & input_points,
            const std::vector<double> & weights,
            const double epsilon,
            std::vector<size_t>& epsnet,
            size_t& counter,
            std::vector<size_t>& currentPoints = std::vector<size_t>()
                 );
```

LISTING 10.1: `enet` interface

The algorithm only relies on CGAL for computing the Delaunay triangulation and locating the disks containing each point.

The `dnet` library

The library `dnet` provides the implementation of our algorithm to construct small sized hitting sets od disks. In Listing 10.2 we give the interface of the algorithm. The comments describe the function of the parameters. We have defined global variables controlling the different parameters of our algorithm, the best setting that we have applied was to set `bg_c1=30`, `bg_c2=12`, `bg_c3=2`, `bg_c4=2`, `bg_c5=0.6`.

```
extern double bg_c1;  //size of global epsnet is with eps=bg_c1/OPT.
extern double bg_c2;  //size of phase_epsnet is with eps=bg_c2/OPT.
extern double bg_c3;  //reweigh a disk only if weight < bg_c3/OPT
extern double bg_c4;  //reweighing factor
extern double bg_c5;  //finish algorithm if remaining_hittingset.size < ↩
    bg_c5*OPT.

/*!
```

```
* \brief Computes the hitting set with out simplified Bronnimann—Goodrich ↩
    implementation
* \param [in,out]  input_points   The input points.
* \param           input_disks    The input disks.
* \param [in,out]  Hittingset     The hittingset (indices).
* \param           OPT            The optimal value if known (if zero the ↩
    algo will guess).
*/
void bg_solve(
              std::vector<Point> & input_points,
              const std::vector<Circle> & input_disks,
              std::vector<size_t> & Hittingset,
              size_t OPT = 0
                );
```

LISTING 10.2: `enet` interface

This library relies on `enet` and the Delanuay triangulation provided by CGAL along the locating the disks in the Delanuay triangulation that contain certain points.

Conclusion

We have presented algorithms for various problems that have sometimes seemingly little connection to each other. But they all share one component, namely that determining exact solutions seem to be currently unfeasible therefore good approximation algorithms are highly desired. We have presented several improvements relying on the study of the geometric and combinatorial structures of them. Even for problems like global illumination, it turns out that with a suitable formulation, like the many-lights framework that discretizes the problem, geometric and combinatorial structures become a key tool for efficient algorithms delivering state-of-the-art performance. Besides the theoretical aspects of designing efficient algorithms we have empirically validated our results by implementing them and testing them on various data. Most of the software written is published online.

Bibliography

[AAG14] Pradeesha Ashok, Umair Azmi, and Sathish Govindarajan. “Small strong epsilon nets.” In: *Comput. Geom.* 47 (9), 2014, pp. 899–909.

[ABD13] Rashmisnata Acharyya, Manjanna Basappa, and Gautam K. Das. “Unit Disk Cover Problem in 2D.” English. In: *Computational Science and Its Applications - ICCSA 2013*. Vol. 7972. 2013, pp. 73–85.

[AC09] Peyman Afshani and Timothy M. Chan. “Optimal halfspace range reporting in three dimensions.” In: *SODA*. 2009, pp. 180–186.

[AEM+06] Christoph Ambühl, Thomas Erlebach, Matús Mihalák, and Marc Nunkesser. “Constant-Factor Approximation for Minimum-Weight (Connected) Dominating Sets in Unit Disk Graphs.” In: *APPROX-RANDOM*. 2006, pp. 3–14.

[AES12] Pankaj K. Agarwal, Esther Ezra, and Micha Sharir. “Near-Linear Approximation Algorithms for Geometric Hitting Sets.” In: *Algorithmica* 63 (1-2), 2012, pp. 1–25.

[AGK+01] Vijay Arya, Naveen Garg, Rohit Khandekar, Adam Meyerson, Kamesh Munagala, and Vinayaka Pandit. “Local search heuristic for k-median and facility location problems.” In: *STOC*. 2001, pp. 21–29.

[AHP08] Boris Aronov and Sariel Har-Peled. “On Approximating the Depth and Related Problems.” In: *SIAM J. Comput.* 38 (3), 2008, pp. 899–921.

[AM06] Pankaj K. Agarwal and Nabil H. Mustafa. “Independent set of intersection graphs of convex objects in 2D.” In: *Comput. Geom.* 34 (2), 2006, pp. 83–95.

[AP14] Pankaj K. Agarwal and Jiangwei Pan. “Near-Linear Algorithms for Geometric Hitting Sets and Set Covers.” In: *Symposium on Computational Geometry*. 2014, p. 271.

[APS07] Pankaj K. Agarwal, János Pach, and Micha Sharir. *State of the Union (of Geometric Objects): A Review*. 2007.

[BB15] Norbert Bus and Lilian Buzer. “Dynamic Convex Hull for Simple Polygonal Chains in Constant Amortized Time per Update.” In: *Proceedings of the 31th European Workshop on Computational Geometry (EUROCG)*. 2015.

[BD08] Thomas Bodt and Philip Dutré. “Coherent Lightcuts.” In: *PhD thesis, Katholieke Universiteit Leuven*, 2008.

[BEL+13] Niels Billen, Björn Engelen, Ares Lagae, and Philip Dutré. “Probabilistic Visibility Evaluation for Direct Illumination.” In: *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering 2013)* 32 (4), 2013, pp. 39–47.

[BG95] Hervé Brönnimann and Michael T. Goodrich. “Almost Optimal Set Covers in Finite VC-Dimension.” In: *Discrete & Computational Geometry* 14 (4), 1995, pp. 463–479.

[BGM+15] Norbert Bus, Shashwat Garg, Nabil H. Mustafa, and Saurabh Ray. “Improved Local Search for Geometric Hitting Set.” In: *32st International Symposium on Theoretical Aspects of Computer Science (STACS)*. 2015.

[BGM+16] Norbert Bus, Shashwat Garg, Nabil H. Mustafa, and Saurabh Ray. “Tighter Estimates for epsilon-nets for Disks.” In: *Computational Geometry: Theory and Applications* 53, 2016.

[BJ02] Gerth Stølting Brodal and Riko Jacob. “Dynamic Planar Convex Hull.” In: *FOCS*. 2002, pp. 617–626.

[BMB15a] Norbert Bus, Nabil H. Mustafa, and Venceslas Biri. “Global Illumination Using Well-Separated Pair Decomposition.” In: *Computer Graphics Forum* 34 (8), 2015.

[BMB15b] Norbert Bus, Nabil H. Mustafa, and Venceslas Biri. “IlluminationCut.” In: *Computer Graphics Forum (Proceedings of Eurographics 2015)* 34 (2), 2015.

[BMR15] Norbert Bus, Nabil H. Mustafa, and Saurabh Ray. “Geometric Hitting Sets for Disks: Theory and Practice.” In: *23rd European Symposium on Algorithms (ESA)*. 2015.

[Cd] *Clustering Datasets*. http://cs.joensuu.fi/sipu/datasets/. Accessed: 2015-02-05.

[CDD+10] Francisco Claude, Gautam K. Das, Reza Dorrigiv, Stephane Durocher, Robert Fraser, Alejandro López-Ortiz, Bradford G. Nickerson, and Alejandro Salinger. “An Improved Line-Separable Algorithm for Discrete Unit Disk Cover.” In: *Discrete Math., Alg. and Appl.* 2 (1), 2010, pp. 77–88.

[CF90] Bernard Chazelle and Joel Friedman. “A deterministic view of random sampling and its use in geometry.” In: *Combinatorica* 10 (3), 1990, pp. 229–249.

[Cha96] Timothy M. Chan. “Optimal Output-Sensitive Convex Hull Algorithms in Two and Three Dimensions.” In: *Discrete & Computational Geometry* 16, 1996, pp. 361–368.

[CHP09] Timothy M. Chan and Sariel Har-Peled. “Approximation algorithms for maximum independent set of pseudo-disks.” In: *Symposium on Computational Geometry*. 2009, pp. 333–340.

[Chr08] Per H. Christensen. *Point-Based Approximate Color Bleeding*. Tech. rep. Pixar, 2008.

[CK95] Paul B. Callahan and S. Rao Kosaraju. “A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields.” In: *J. ACM* 42 (1), 1995, pp. 67–90.

[CKLT07] Paz Carmi, Matthew J. Katz, and Nissan Lev-Tov. “Covering Points by Unit Disks of Fixed Location.” In: *ISAAC*. 2007, pp. 644–655.

[CMW+04] Gruia Călinescu, Ion I. Mandoiu, Peng-Jun Wan, and Alexander Zelikovsky. “Selecting Forwarding Neighbors in Wireless Ad Hoc Networks.” In: *MONET* 9 (2), 2004, pp. 101–111.

[DFLO+12] Gautam K. Das, Robert Fraser, Alejandro López-Ortiz, and Bradford G. Nickerson. “On the Discrete Unit Disk Cover Problem.” In: *International Journal on Computational Geometry and Applications* 22 (5), 2012, pp. 407–419.

[DKH+10] Tomáš Davidovič, Jaroslav Křivánek, Miloš Hašan, Philipp Slusallek, and Kavita Bala. “Combining global and local virtual lights for detailed glossy illumination.” In: *ACM Trans. Graph.* 29, 6 2010, 143:1–143:8.

[DKH+13] Carsten Dachsbacher, Jaroslav Křivánek, Miloš Hašan, Adam Arbree, Bruce Walter, and Jan Novák. “Scalable Realistic Rendering with Many-Light Methods.” In: *Eurographics 2013 – State of the Art Reports*. 2013, pp. 23–38.

[DKL10] Holger Dammertz, Alexander Keller, and Hendrik Lensch. “Progressive Point-Light-Based Global Illumination.” In: *Computer Graphics Forum* 29 (8), 2010, pp. 2504–2515.

[DP07] Erik D. Demaine and Mihai Patrascu. “Tight Bounds for Dynamic Convex Hull Queries (Again).” In: *Proceedings of the Twenty-third Annual Symposium on Computational Geometry*. 2007, pp. 354–363.

[DS96] Michael B. Dillencourt and Warren D. Smith. “Graph-theoretical conditions for inscribability and Delaunay realizability.” In: *Discrete Mathematics* 161 (1–3), 1996, pp. 63 –77.

[EHS04] Eti Ezra, Dan Halperin, and Micha Sharir. “Speeding up the incremental construction of the union of geometric objects in practice.” In: *Comput. Geom.* 27 (1), 2004, pp. 63–85.

[ERS05] G. Even, D. Rawitz, and S. Shahar. “Hitting sets when the VC-dimension is small.” In: *Inf. Process. Lett.* 95, 2005, pp. 358–362.

[FBD15] Roald Frederickx, Pieterjan Bartels, and Philip Dutré. “Adaptive LightSlice for Virtual Ray Lights.” In: *EG 2015 – Short Papers*. 2015, pp. 61–64.

[Fra12] Robert Fraser. “Algorithms for Geometric Covering and Piercing Problems.” PhD thesis. University of Waterloo, 2012.

[Fwf] *Federal Wildland Fire Occurence Data, United States of America Geological Survey*. http://wildfire.cr.usgs.gov/firehistory/data.html. Accessed: 2015-02-05.

[Gan11] Shashidhara K. Ganjugunte. “Geometric Hitting Sets and Their Variants.” PhD thesis. Duke University, 2011.

[Gar82] Irene Gargantini. “An Effective Way to Represent Quadtrees.” In: *Communications of the ACM* 25 (12), 1982, pp. 905–910.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GKP+12] Iliyan Georgiev, Jaroslav Křivánek, Stefan Popov, and Philipp Slusallek. “Importance Caching for Complex Illumination.” In: *Computer Graphics Forum* 31 (2), 2012. EUROGRAPHICS 2012.

[Gra72] Ronald L. Graham. “An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set.” In: *Inf. Process. Lett.* 1 (4), 1972, pp. 132–133.

[GS10] Iliyan Georgiev and Philipp Slusallek. “Simple and Robust Iterative Importance Sampling of Virtual Point Lights.” In: *Proceedings of Eurographics (short papers)*, 2010.

[Han02] Yijie Han. “Deterministic Sorting in $O(n \log \log n)$ Time and Linear Space.” In: *Proceedings of the Thirty-fourth Annual ACM Symposium on Theory of Computing*. 2002, pp. 602–608.

[HKS+14] Sariel Har-Peled, Haim Kaplan, Micha Sharir, and Shakhar Smorodinsky. “Epsilon-Nets for Halfspaces Revisited.” In: *CoRR* abs/1410.3154, 2014.

[HKW+09] Miloš Hašan, Jaroslav Křivánek, Bruce Walter, and Kavita Bala. “Virtual spherical lights for many-light rendering of glossy scenes.” In: *ACM Trans. Graph.* 28 (5), 2009, 143:1–143:6.

[HM85] Dorit S. Hochbaum and Wolfgang Maass. “Approximation Schemes for Covering and Packing Problems in Image Processing and VLSI.” In: *J. ACM* 32 (1), 1985, pp. 130–136.

[HM87] D. S. Hochbaum and W. Maass. "Fast Approximation Algorithms for a Nonconvex Covering Problem." In: *J. Algorithms* 8 (3), 1987, pp. 305–323.

[HP00] Sariel Har-Peled. "Constructing Planar Cuttings in Theory and Practice." In: *SIAM J. Comput.* 29 (6), 2000, pp. 2016–2039.

[HPB07] Miloš Hašan, Fabio Pellacini, and Kavita Bala. "Matrix row-column sampling for the many-light problem." In: *ACM Trans. Graph.* 26 (3), 2007.

[HRE+11] Matthias Holländer, Tobias Ritschel, Elmar Eisemann, and Tamy Boubekeur. "ManyLoDs: Parallel Many-view Level-of-detail Selection for Real-time Global Illumination." In: *Proceedings of the 22nd Eurographics Conference on Rendering*. 2011, pp. 1233–1240.

[HVAP+08] Miloš Hašan, Edgar Velázquez-Armendariz, Fabio Pellacini, and Kavita Bala. "Tensor Clustering for Rendering Many-light Animations." In: *Proceedings of the Nineteenth Eurographics Conference on Rendering*. 2008, pp. 1105–1114.

[HW87] D. Haussler and E. Welzl. "Epsilon-nets and simplex range queries." In: *Discrete Comput. Geom.* 2, 1987, pp. 127–151.

[Jen01] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., 2001.

[Kaj86] James T. Kajiya. "The Rendering Equation." In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. 1986, pp. 143–150.

[Kar72] R. M. Karp. "Reducibility Among Combinatorial Problems." In: *Complexity of Computer Computations*. 1972, pp. 85–103.

[Kar84] N. Karmarkar. "A New Polynomial-time Algorithm for Linear Programming." In: *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*. 1984, pp. 302–311.

[Kel97] Alexander Keller. "Instant radiosity." In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 1997, pp. 49–56.

[KFB10] Jaroslav Křivánek, James A. Ferwerda, and Kavita Bala. "Effects of global illumination approximations on material appearance." In: *ACM Trans. Graph.* 29 (4), 2010. SIGGRAPH '10, 112:1–112:10.

[KHA+12] Jaroslav Křivánek, Miloš Hašan, Adam Arbree, Carsten Dachsbacher Alexander Keller, and Bruce Walter. "Optimizing Realistic Rendering with Many-Light Methods." In: *SIGGRAPH 2012 Course*. 2012.

[KK06] Thomas Kollig and Alexander Keller. "Illumination in the Presence of Weak Singularities." In: *Monte Carlo and Quasi-Monte Carlo Methods 2004*. 2006, pp. 245–257.

[KM72] Victor Klee and George J. Minty. "How good is the simplex algorithm?" In: *Inequalities*. Vol. III. 1972, pp. 159–175.

[KMN+02] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. "A local search approximation algorithm for k-means clustering." In: *Symposium on Computational Geometry*. 2002, pp. 10–18.

[LJ03] Li Lin and Yunfei Jiang. "The Computation of Hitting Sets: Review and New Algorithms." In: *Inf. Process. Lett.* 86 (4), 2003, pp. 177–184.

[LN12] B. Jaya Lakshmi and M. Neelima. "Maximising Wireless sensor Network life time through cluster head selection using Hit sets." In: *International Journal of Computer Science Issues*, 2012.

[LSK+07] Samuli Laine, Hannu Saransaari, Janne Kontkanen, Jaakko Lehtinen, and Timo Aila. "Incremental Instant Radiosity for Real-Time Indirect Illumination." In: *Proceedings of Eurographics Symposium on Rendering*. 2007, pp. 277–286.

[LW93] Eric P. Lafortune and Yves D. Willems. "Bi-Directional Path Tracing." In: *Proceedings of the 3rd International Conference on Graphics and Visualization Techniques (COMPUGRAPHICS '93*. 1993, pp. 145–153.

[Mat02] J. Matousek. *Lectures in Discrete Geometry*. Springer-Verlag, 2002.

[Mat98] Jirí Matousek. "On Constants for Cuttings in the Plane." In: *Discrete & Computational Geometry* 20 (4), 1998, pp. 427–448.

[Mel87] Avraham A. Melkman. "On-line Construction of the Convex Hull of a Simple Polyline." In: *Inf. Process. Lett.* 25 (1), 1987, pp. 11–12.

[Mik10] Miroslav Miksik. "Implementing Lightcuts." In: *CESCG*. 2010.

[MR10] Nabil H. Mustafa and Saurabh Ray. "Improved Results on Geometric Hitting Set Problems." In: *Discrete & Computational Geometry* 44 (4), 2010, pp. 883–895.

[Ngs] *Geographic Names Database, maintained by the National Geospatial-Intelligence Agency*. http://geonames.nga.mil/gns/html/. Accessed: 2015-02-05.

[NND+12a] Jan Novák, Derek Nowrouzezahrai, Carsten Dachsbacher, and Wojciech Jarosz. "Progressive Virtual Beam Lights." In: *Computer Graphics Forum (Proceedings of EGSR)* 31 (4), 2012.

[NND+12b] Jan Novák, Derek Nowrouzezahrai, Carsten Dachsbacher, and Wojciech Jarosz. "Virtual Ray Lights for Rendering Scenes with Participating Media." In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 31 (4), 2012.

[OP11] Jiawei Ou and Fabio Pellacini. "LightSlice: matrix slice sampling for the many-lights problem." In: *ACM Trans. Graph.* 30 (6), 2011, p. 179.

[OWW85] Thomas Ottmann, Peter Widmayer, and Derick Wood. “A fast algorithm for the Boolean masking problem.” In: 1985, pp. 249–268.

[PA95] J. Pach and P. K. Agarwal. *Combinatorial Geometry*. John Wiley & Sons, 1995.

[PGD+96] D. G. Porter, E. Glavinas, P. Dhagat, J. A. O’sullivan, R. S. Indeck, and M. W. Muller. “Irregular Grain Structure in Micromagnetic Simulation.” In: *Journal of Applied Physics* 79, 1996, pp. 4694–4696.

[PGS+13] Stefan Popov, Iliyan Georgiev, Philipp Slusallek, and Carsten Dachsbacher. “Adaptive Quantization Visibility Caching.” In: *Computer Graphics Forum (Proceedings of Eurographics 2013)* 32 (2), 2013.

[PKD12] Roman Prutkin, Anton Kaplanyan, and Carsten Dachsbacher. “Reflective Shadow Map Clustering for Real-Time Global Illumination.” In: *Eurographics (Short Papers)*. 2012, pp. 9–12.

[PPD98] Eric Paquette, Pierre Poulin, and George Drettakis. “A Light Hierarchy for Fast Rendering of Scenes with Many Lights.” In: *Computer Graphics Forum* 17, 1998, pp. 63–74.

[PR08] E. Pyrga and S. Ray. “New existence proofs for epsilon-nets.” In: *Proceedings of Symposium on Computational Geometry*. 2008, pp. 199–207.

[PS06] János Pach and Micha Sharir. *Combinatorial Geometry with Algorithmic Applications – The Alcala Lectures*. 2006.

[PS85] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag New York, Inc., 1985.

[PW90] J. Pach and G. Woeginger. “Some New Bounds for Epsilon-Nets.” In: *Symposium on Computational Geometry*. 1990, pp. 10–15.

[Rad08] Ingo Radax. *Instant Radiosity for Real-Time Global Illumination*. Tech. rep. Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2008.

[RGK+08] Tobias Ritschel, Thorsten Grosch, Min H. Kim, Hans-Peter Seidel, Carsten Dachsbacher, and Jan Kautz. “Imperfect Shadow Maps for Efficient Computation of Indirect Illumination.” In: *ACM Trans. Graph. (Proc. of SIGGRAPH ASIA 2008)* 27 (5), 2008.

[RS97] R. Raz and M. Safra. “A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP.” In: *Proceedings of STOC*. 1997, pp. 475–484.

[Sam95] Hanan Samet. “Spatial Data Structures.” In: *Modern Database Systems*. 1995, pp. 361–385.

[SHD15] Florian Simon, Johannes Hanika, and Carsten Dachsbacher. “Rich-VPLs for Improving the Versatility of Many-Light Methods.” In: *Computer Graphics Forum (Proceedings of Eurographics 2015)* 34 (2), 2015, pp. 575–584.

[She98] Jonathan Richard Shewchuk. “Tetrahedral Mesh Generation by Delaunay Refinement.” In: *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*. 1998, pp. 86–95.

[SIM+06] B Segovia, J C Iehl, R Mitanchey, and B Péroche. “Bidirectional Instant Radiosity.” In: *EGSR*. 2006, pp. 389–398.

[SIP07] Benjamin Segovia, Jean-Claude Iehl, and Bernard Péroche. “Metropolis Instant Radiosity.” In: *Computer Graphics Forum* 26 (3), 2007, pp. 425–434.

[SPA07] E. A. Silva, K. Panetta, and S. S. Agaian. “Quantifying image similarity using measure of enhancement by entropy.” In: *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*. Vol. 6579. 2007.

[Tat09] Natalya Tatarchuk. “Advances in Real-time Rendering in 3D Graphics and Games I.” In: *ACM SIGGRAPH 2009 Courses*. 2009.

[VG95] Eric Veach and Leonidas Guibas. “Bidirectional Estimators for Light Transport.” English. In: *Photorealistic Rendering Techniques*. 1995, pp. 145–167.

[VG97] Eric Veach and Leonidas J. Guibas. “Metropolis light transport.” In: *SIGGRAPH ’97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 1997, pp. 65–76.

[VØ00] Staal Vinterbo and Aleksander Øhrn. “Minimal approximate hitting sets and rule templates.” In: *International Journal of Approximate Reasoning* 25 (2), 2000, pp. 123 –143.

[WAB+06] Bruce Walter, Adam Arbree, Kavita Bala, and Donald P. Greenberg. “Multidimensional lightcuts.” In: *ACM SIGGRAPH 2006 Papers*. 2006, pp. 1081–1088.

[Wal07] Ingo Wald. “On Fast Construction of SAH-based Bounding Volume Hierarchies.” In: *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*. 2007, pp. 33–40.

[WBK+08] Bruce Walter, Kavita Bala, Milind Kulkarni, and Keshav Pingali. *Fast Agglomerative Clustering for Rendering*. 2008.

[WC13] Yu-Ting Wu and Yung-Yu Chuang. “VisibilityCluster: Average Directional Visibility for Many-Light Rendering.” In: *Visualization and Computer Graphics, IEEE Transactions on* 19 (9), 2013, pp. 1566–1578.

[WFA+05] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. “Lightcuts: a scalable approach to illumination.” In: *ACM Trans. Graph.* 24 (3), 2005, pp. 1098–1107.

[WFW+13] Sven Woop, Louis Feng, Ingo Wald, and Carsten Benthin. “Embree Ray Tracing Kernels for CPUs and the Xeon Phi Architecture.” In: *ACM SIGGRAPH 2013 Talks*. 2013, 44:1–44:1.

[WKB12] Bruce Walter, Pramook Khungurn, and Kavita Bala. “Bidirectional Lightcuts.” In: *ACM Trans. Graph.* 31 (4), 2012, 59:1–59:11.

[WPS+03] Ingo Wald, Timothy J. Purcell, Joerg Schmittler, Carsten Benthin, and Philipp Slusallek. “Realtime Ray Tracing and its use for Interactive Global Illumination.” In: *Eurographics State of the Art Reports*. 2003.

[WWB+14] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. “Embree: A Kernel Framework for Efficient CPU Ray Tracing.” In: *ACM Trans. Graph.* 33 (4), 2014, 143:1–143:8.

[WXW11] Guangwei Wang, Guofu Xie, and Wencheng Wang. “Efficient Search of Lightcuts by Spatial Clustering.” In: *SIGGRAPH Asia 2011 Sketches*. 2011, 26:1–26:2.

[The15] The CGAL Project. *CGAL User and Reference Manual*. 4.6. CGAL Editorial Board, 2015.