Photon Mapping on the GPU

Martin Fleisz

S0894924@sms.ed.ac.uk

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2009

Abstract

In realistic image synthesis, Photon Mapping is an essential extension to the standard ray tracing algorithm. However, recent developments in processor design, towards multi-core systems do not favor the tree-based photon map technique. We developed a novel approach for Photon Mapping, based on spatial hashing. Photon map creation and photon search are fully parallelized and take full advantage of the processing power of current GPUs. Synchronization is kept to a minimum and we are able to use texture memory to cache access to the photon information. In order to evaluate our new approach we carried out a series of benchmarks with existing, GPU based photon mapping techniques. Our spatial hashing approach is shown to be much faster than existing techniques with almost any possible configuration, while archiving the same image quality.

Acknowledgements

I would like to thank my supervisor, Taku Komura for his advice and encouragement throughout this project. I would also like to thank Vincent Garcia, who helped me to get a better understanding of his paper and also provided source code to his work.

Declaration

I declare that this thesis was composed by me, that the work contained herein is my own except where explicitly stated otherwise in the text, and that thesis work has not been submitted for any other degree or professional qualification except as specified.

(Martin Fleisz)

Table of Contents

1 Introduction 1

1.1 Problem statement 2

1.2 Motivation 3

1.3 Structure of the Report 3

2 Background 4

2.1 Theoretical Background 5

2.1.1 Ray Tracing 5

2.1.2 Photon Mapping 6

2.1.3 GPU Implementations 8

2.2 Fast k Nearest Neighbor Search using GPU 8

2.2.1 Basics 8

2.2.2 Construction 9

2.2.3 Photon Search 9

2.3 Real-Time KD-Tree Construction on Graphics Hardware 10

2.3.1 Basics 10

2.3.2 Construction 11

2.3.3 Photon Search 14

2.4 Analysis of existing Techniques 15

2.4.1 Brute Force Approach 15

2.4.2 GPU kD-Tree Approach 15

3 A new Approach 17

3.1 Photon Mapping using Spatial Hashing 18

3.1.1 Overview 18

3.1.2 Theory 19

3.1.3 Implementation 21

3.1.4 Limitations 23

4 Performance Measuring 25

4.1 Measureable Parameters 26
4.2 System Specification 27
4.3 Application Specification 28
4.4 Test Scene Specification 28

5 Results and Analysis 30

5.1 Construction Time Performance 31
5.2 Memory Requirements 32
5.3 Photon Search 34

5.3.1 Photon Map Sizes 34
5.3.2 Number of Photons 38
5.3.3 Query Radius 40
5.3.4 Query Size 43

5.4 Results with the Brute Force Approach 46
5.5 Image Quality 47
5.6 Observations 49

6 Conclusion 52

6.1 Conclusion 53
6.2 Future Work 54

7 Bibliography 55

List of Figures

Figure 1: Ray Tracing [8] 5
Figure 2: Ray Tracing on the left, Photon Mapping on the right 6
Figure 3: Photon density estimation [17] 8
Figure 4: Small node splitting [36] 13
Figure 5: Compaction of the active node list [43] 14
Figure 6: Immediate lookup of photons in the cell of the sample point x, using the hash table 18
Figure 7: After calculating the hashes the photon list is sorted and the hash table is created 19
Figure 8: Calculating neighbor cells' grid positions, using the values from the offset lookup table 21
Figure 9: Hash Table (start index, end index) and Photon Table relation 22
Figure 10: 2D view of a scene, green cells occupy geometry whereas red cells do not 23
Figure 11: Test scene 29
Figure 12: Construction Time Performance for Global Photon Map 31
Figure 13: Construction Time Performance for Caustic Photon Map 32
Figure 14: Memory Consumption 33
Figure 15: Peak Memory Consumption 33
Figure 16: Global photon map search performance with different map sizes 35
Figure 17: kD-Tree Traversal - Local Memory Access 36
Figure 18: Caustic photon map search performance with different map sizes 37
Figure 19: Photon Density in a global (left) and a caustic (right) photon map 37
Figure 20: Global photon map search performance with different k sizes 38
Figure 21: Caustic photon map search performance with different k sizes 39
Figure 22: Global photon map search performance with different query radii 41
Figure 23: Caustic photon map search performance with different query radii 41
Figure 24: Hashing Memory Consumption with different query radii 43
Figure 25: Global photon map performance with different query size 44
Figure 26: Non-Coalesced Memory Access 45
Figure 27: Caustic photon map performance with different query size 45
Figure 28: Brute Force Performance 46

Figure 29: Image quality with hashing (left) and kD-Tree (right) with 10,000 (top) and 50,000 (bottom) photons 48
Figure 30: Image quality with hashing (left) and kD-Tree (right) with a caustic photon map (10,000 photons).. 49

List of Tables

Table 1: Measurements 26
Table 2: System Specification..27
Table 3: Computation time decomposition for the Brute Force Technique [25] 47

1

Introduction

“The only way forward in terms of performance - but we also think in terms of power - is to go to multicores. Multicores put us back on the historical trajectory of Moore's Law. We can directly apply the increase in transistors to core count - if you are willing to suspend disbelief for a moment that you can actually get all those cores working together.”

- Justin R. Rattner, Chief Technical Officer at Intel [1]

Objectives

- To explain the problems we try to address with our project
- To explain our motivation behind this project
- To give an overview of the report’s structure

1.1 Problem statement

Because of recent developments in processor design, parallel computing has become the new way of creating high performance applications. Especially Graphics Processor Units (GPUs) offer amazing computation power and are available in a lot of standard consumer machines these days. However, in order to unleash this available processing power, algorithms have to be parallelized [2]. In a lot of cases, like Photon Mapping, this task can be quite difficult due to a few reasons.

The first problem arises with the creation of the photon map data structure. For performance reasons this structure is usually a balanced kD-Tree [3], used to speed up the photon search. However, it is difficult to parallelize the tree construction because all threads have to access the same data structure. This usually requires a lot of synchronization overhead and completely defeats the advantage of parallel processing. Although Jensen discusses parallelization of photon search in [4], we think that this is only a mediocre solution to the overall problem.

Another problem is the highly irregular memory access pattern, revealed during tree construction and traversing. In order to find photons in a kD-Tree, lots of scattered memory reads have to be performed which is everything but optimal for GPUs. Also, the interdependent operations that have to be performed during tree traversal entail that the hardware is not able to hide these memory latencies.

In this project we want to present and evaluate a new Photon Mapping technique for GPUs. Using a new approach, based on spatial hashing to organize the photons in the photon map, we think that we are able to utilize the available parallel computation power more efficient than existing techniques. In order to proof our proposed technique in terms of performance and quality, we will implement prototypes of the various approaches and compare them. We will also apply different parameters and data sizes during our tests, in order to evaluate the scalability of the different techniques.

1.2 Motivation

Sequential computing has reached a point where huge performance improvements are not feasible anymore. Instead, chip manufacturers are now designing multi-core processors in order to be able to deliver regular performance improvements. However, in order to utilize this available power, algorithms must be massively parallelized and, in case of GPUs, executed using thousands of threads. While some sequential algorithms can be easily changed to parallel versions, others are very difficult or even impossible to modify in such a manner.

The motivation behind this project is to come up a new technique for Photon Mapping that fits these trends in processor design. Our main idea is to develop a heavily parallelized algorithm that is designed with respect to current GPU architectures. This means we have to utilize as many threads as possible for the photon map creation and photon search, while trying to avoid any synchronization. Scattered memory access should also be kept to a minimum to achieve the highest possible performance. As we believe the future of computing is within parallelization, our technique should also be very useful for future, general purpose multi-core architectures.

1.3 Structure of the Report

After the introduction, a short explanation of Ray Tracing and Photon Mapping is provided, to give a basic understanding of these techniques. Furthermore, we take a look at existing GPU approaches, explaining how they work in detail and what shortcomings they have.

Chapter three introduces our new GPU Photon Mapping approach, based on spatial hashing, with detailed information on our current implementation.

The results of our experiments and tests are presented in chapter four, giving an overview of the performance, memory consumption and scalability of the various techniques.

In chapter five we will critically analyze the results presented in the previous chapter.

Finally, chapter six contains conclusions and directions for future research.

2

Background

"Parallel programming is perhaps the largest problem in computer science today and is the major obstacle to the continued scaling of computing performance that has fuelled the computing industry, and several related industries, for the past 40 years."

- Bill Dally, Chief Scientist and Vice President of NVIDIA Research [5]

Objectives

- To give an introduction to Ray Tracing and highlight its problems
- To explain how Photon Mapping works
- To give an overview of existing GPU Photon Mapping techniques
- To identify weaknesses in existing GPU Photon Mapping techniques

2.1 Theoretical Background

2.1.1 Ray Tracing

Ray Tracing is a technique for image synthesis, which is used to create a 2D image of a 3D world. The first Ray Tracing algorithm was introduced by Arthur Apple [6] and, with some modifications, is still used in current ray tracers. For each pixel on our viewing plane we shoot one or more rays into the scene, testing if they intersect with any object (Figure 1). We might find a few objects that our ray intersects but we will only consider the object, closest to the viewer. Now we have to distinguish between three different kinds of rays that are generated next, depending on the object's material properties: shadow, reflection and refraction rays [7].

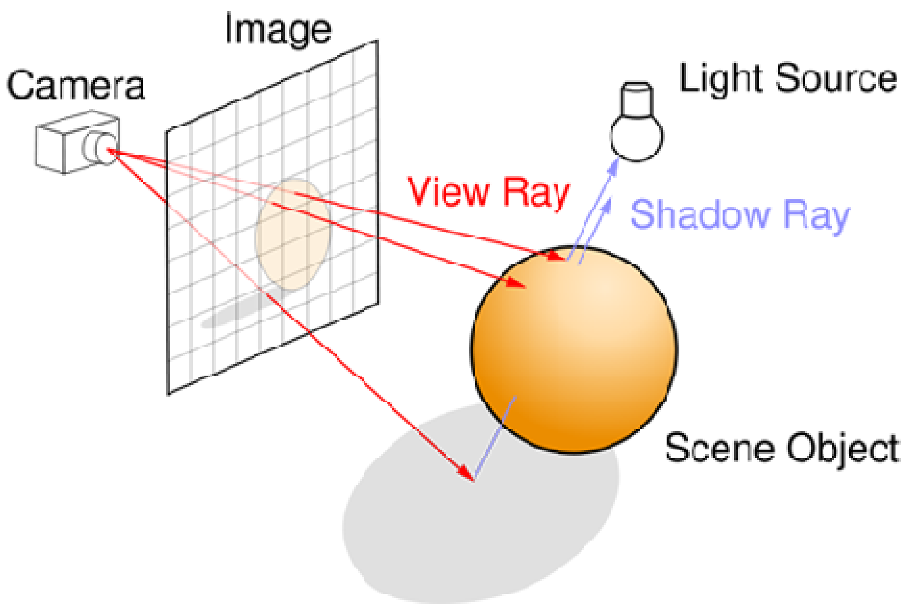


Figure 1: Ray Tracing [8]

Shadow rays simply check if our intersection point faces any of the light sources in the scene. In order to determine the amount of light received, we shoot another ray from this point to each light source in our scene. If all rays reach the light source without intersecting another object, we know that our point is fully lit and we have to color that pixel on screen. In case all of these rays intersect other objects, we know that our point is within a shadowed area and we color that pixel very dark or just black.

Reflection rays are cast when our initial ray hits a reflective surface, i.e. a mirror. Leaving from the mirroring surface, we test for object intersection again. If we find an intersecting object, we pick the closest one to our reflective surface and color the pixel, using the reflected intersection point's color.

The third ray type is cast when we hit a refractive surface, i.e. a glass full of water. Light changes its direction according to Snell's Law [9] when it travels between two media with different refraction indexes. The coloring for refractive objects works similar to reflective objects in the previous paragraph.

2.1.2 Photon Mapping

The problem with traditional methods, like Ray Tracing and Radiosity is that they are not able to model all possible lighting effects in a scene (Figure 2). Ray Tracing only simulates indirect illumination by adding a constant ambient term in the lighting calculation, whereas Radiosity only simulates diffuse reflections, completely ignoring mirrored surfaces. Therefore, approaches were made to combine both methods, where each technique renders the effects the other fails to. However, such approaches are is still not sufficient as they both fail to model focused light effects, like caustics. Solutions to this problem were presented by [10], [11] and [12] but they introduced other problems as [13] explains.

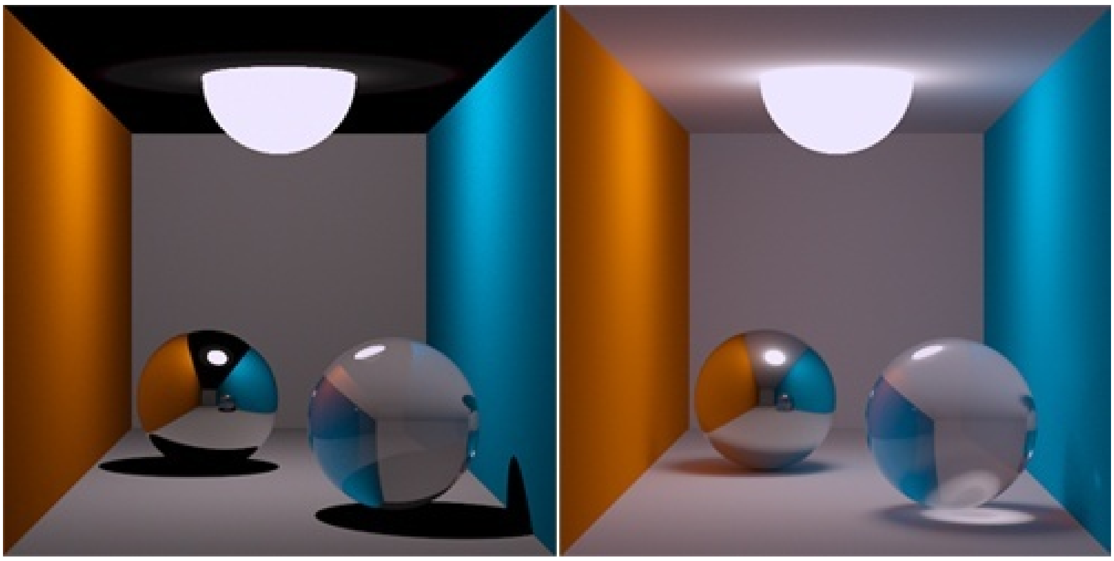


Figure 2: Ray Tracing on the left, Photon Mapping on the right

Jensen presented a new extension to Ray Tracing, based on the concept of photon maps [14], that was able to overcome all these problems. In a first pass, the photon map is constructed by emitting light rays from each light source into the scene. In order to emit photons more efficiently, projection maps can be used as described in [15]. Whenever a photon hits a diffuse surface, its position, incident angles and power are stored in the photon map. Jensen uses a balanced kD-Tree [3] to organize this data, which is very useful to locate photons during the radiance estimation. Afterwards, we decide if the photon is absorbed or reflected, using Russian roulette [16]. Photon hits are not stored for specular objects because the chance, of having incoming photons from the specular direction is almost zero. Instead, these surfaces are rendered using standard Ray Tracing techniques, as explained earlier. An important property of the photon map is that it stores lighting information decoupled from the scene geometry. This means that the photon map's lookup time for complex scenes with many polygons, is the same as it is for simple scenes with just a few polygons.

In the second pass, the information in the photon map is used to calculate the effects of indirect lighting and caustics. Direct illumination and specular surfaces are rendered, using standard Ray Tracing because these effects would require a huge amount of photons in the photon map, to be rendered correctly. In order to calculate the reflected radiance for any given point *x* in a scene, Jensen uses density estimation. As shown in Figure 3, by expanding a sphere around *x*, until it contains *n* photons, we are able to collect photon samples for the estimation. This yields the following equation for estimating the reflected radiance L_r at a given point *x*:

$$L_r(x,\vec{\omega}) \approx \frac{1}{\pi r^2}\sum_{p=1}^{N} f_r\left(x,\vec{\omega}_p,\vec{\omega}\right)\Delta\Phi_\mathrm{p}\left(x,\vec{\omega}_p\right) \tag{2.1}$$

where f_r is the surface's bidirectional reflectance distribution function (BRDF) and Φ_p is the power of photon *p*, stored in the photon map.

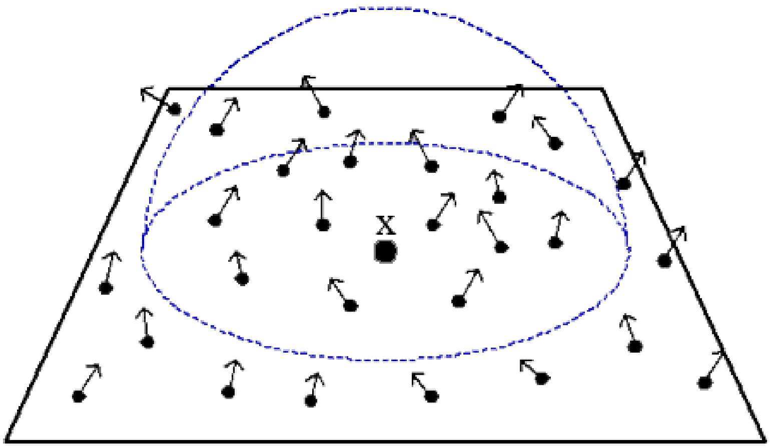


Figure 3: Photon density estimation [17]

2.1.3 GPU Implementations

There have already been a couple of successful attempts made, to implement Ray Tracing efficiently on GPUs. Purcell et al [18] presented the first ray tracer, running entirely on the GPU, using a uniform grid for acceleration. The first implementations that achieved better performance than CPU based ray tracers were presented in [19] and [20]. Unfortunately, both these techniques work with static scenes only. The latest work from Luebke et al [21] presents a technique, combining Ray Tracing and rasterization methods to obtain real-time performance on dynamic scenes. Photon mapping has been implemented for multi-core CPUs in [22] and for older GPU generations in [23]. In the following sections, we take a look at the available techniques that can be used for Photon Mapping on current GPU hardware.

2.2 Fast k Nearest Neighbor Search using GPU

2.2.1 Basics

As described earlier, calculation of the illumination of a distinct point requires a certain amount of closely located photons. This kind of search is called k-nearest neighbor search [24], which is a special variation from the family of nearest neighbor search algorithms. In our case, the kNN search problem consists of finding the k nearest photons, for a set of query points that we want to calculate the illumination for.

In [25], Garcia et al present a brute force based approach to the kNN search problem. Their work is kept very general and supports points with arbitrary dimensions, as well as differently sized reference and query point sets. As they write in their report, this exhaustive search method is by nature highly parallelizable and therefore, is perfectly suitable for a GPU implementation. Their solution runs completely on the graphics device, completely offloading all work from the CPU.

2.2.2 Construction

All that needs to be done during the construction or setup phase is allocation of device memory and copying the required data to the device. For a given set of m photons and n query points, the memory requirements are pretty high with $O(nm)$. In order to improve performance, data is loaded into different kinds of memory on the device. The query set is stored in global memory, which has a huge bandwidth but performs bad if access is not coalesced [2]. Photons will be stored in texture memory which provides better performance on non-coalesced accesses.

2.2.3 Photon Search

Photon search is split into the following three steps:

- Compute all distances between a query point q_i and all reference points r_j with $j \in [1, m]$
- Sort the computed distances
- Select the first k reference points, corresponding to the k smallest distances

These steps are repeated for all query points in the query set and can be performed in parallel on the GPU. Distances are computed and stored in a way similar to a matrix. Because our query point set will be usually quite large (i.e. more than 300.000 for an image resolution of 640 x 480), it is necessary to split our query, due to memory constraints.

The sorting part is, by its nature, very problematic as it compares and exchanges many distances in non-predictable order. This means memory access is non-coalesced which results in a performance hit, when using global memory. Texture memory would be a good alternative, but unfortunately it is read-only memory and therefore, cannot be used for sorting.

For the sorting step, Garcia et al tried out a couple of algorithms in their paper. Quicksort [26] is very popular but cannot be used with CUDA, due to the lack of support for recursive functions. Therefore, their first implementation used the comb sort algorithm [27], which is able to sort the calculated n distances in $O(n \log n)$ time. However, because it is not necessary to sort the complete data set but only the first k elements, Garcia et al finally used a modified version of insertion sort [28]. Insertion sort proofed to be faster for finding up to $k = 100$ neighbors, before being outperformed by the comb sort implementation. Because insertion sort itself cannot be parallelized efficiently, sorting for all query points is performed simultaneous instead.

Obviously, this solution demands a huge amount of processing power. Complexity is $O(nmd)$ for the n times m distances computed and $O(nm \log m)$ for the n sorts, performed to find the nearest reference points. However, according to Garcia et al, their brute force approach performs faster than a kD-Tree based software implementation. For their comparison they used the ANN C++ library [29], which implements the kD-Tree based nearest neighbor search method, presented in [30].

2.3 Real-Time KD-Tree Construction on Graphics Hardware

2.3.1 Basics

Obviously, one way to move the Photon Mapping technique onto graphics hardware is to do the creation and traversal of the kD-Tree on the GPU. The first attempt to accomplish this was presented by Purcell et al in [23]. However, their work is a bit outdated because this technique is based on graphics hardware that is far less flexible and programmable than current devices.

More general work on parallel kD-Tree construction was published by Popov et al [31] and Shevtsov et al [32]. Both approaches are based on multi-core CPUs and therefore, have design issues when used on a GPU. The first problem is that kD-Tree construction can easily become bandwidth limited on large input data sets, due to its random memory access pattern. Therefore, the construction switches from breadth first search (BFS), over to depth first search (DFS) manner at deeper nodes. This means that these approaches keep the number of concurrently running threads pretty low. Graphics hardware however, has a much higher memory bandwidth and requires at least $10^3 \sim 10^4$ threads for optimal performance [2]. Another important factor during construction is the balance of the tree and therefore, finding the right splitting position for a node. Both papers use the Surface Area Heuristic (SAH) [33] [34] to evaluate the costs for a splitting candidate. Even though the SAH improves the quality of trees significantly [35], its calculation consumes a lot of time.

Finally, parallelizing the photon search or tree traversing is pretty easy, as the tree is accessed read-only. However, for performance reasons it is important that the tree is well balanced and stored efficiently. Storing the tree efficiently means to keep scattered memory accesses as low as possible, by placing child nodes close to their parents. The traversal algorithm itself is not a good candidate for parallelization. Instead, performing multiple traversals simultaneously is a much better way, in order to obtain good performance.

In [36], Zhou et al present a new approach to kD-Tree construction and traversal on GPUs, using CUDA. Even though their main focus is on SAH kD-Tree construction for Ray Tracing, they also provide information on adapting the technique for photon mapping, which we will concentrate on.

2.3.2 Construction

Zhou et al build their kD-Tree completely in breadth first search manner, distinguishing between two different node stages. During the initialization stage, global memory is allocated for the tree construction and the root node is created. For the photon mapping implementation we also have to create three sorted order lists (one for each dimension) for all point coordinates, using the sort function from [37]. Using the sorted order, we are able to compute bounding boxes in *O(1)* time and we avoid to use segmented reduction, which compensates for the sorting. Additionally, we maintain three associated point ID lists (one for each coordinate axis) which have to fulfill the following criteria:

- Points in the same node are contiguous in all lists
- Points in the same node start at the same offset in all lists

In the first step, the so called Large Node Stage is executed. This stage splits nodes using a combination of spatial median splitting and "cutting off empty space", as described in [38]. Because in the Large Node Stage the number of nodes is naturally smaller, computation is parallelized over all points rather than over nodes. First we need to find the splitting plane (the plane that splits the longest axis in the middle), after repeatedly applying empty space splitting before. Then each photon is classified as being either left (1) or right (0) of the splitting plane. Finally, we perform the scan operation from [39], in order to use the split operation in [40] to split the current node. As Zhou et al mention in their paper, the sorted coordinate and ID lists maintain all their properties after splitting. After the split we check if the amount of photons in our child nodes is below the threshold *T = 32*. If the number is smaller, the node is added to the small node list. Otherwise it is added to the active list and scheduled for the next iteration of the Large Node Stage. The Large Node Stage finishes as soon as there are no more nodes in the active list.

The next phase is the Small Node Stage and begins with a preprocessing step for all nodes in the small node list, created during the previous stage. In this step we collect all splitting plane candidates and calculate the resulting split sets, which define photon distribution after a split. It should be noted that splitting planes are restricted to initial photon positions in this stage.

After the preprocessing has finished, we process all small nodes in parallel and split them, until each node contains one photon. Since we need to build a kD-Tree for points (or photons), rather than triangles, we are now using the Voxel Volume Heuristic (VVH) [41] for split cost evaluation, instead of the SAH. Given a split position *x*, we can calculate the VVH as follows:

$$VVH(x) = \frac{C_L(x)\, Vol(V_L \pm R)}{Vol(V \pm R)} + \frac{C_R(x)\, Vol(V_R \pm R)}{Vol(V \pm R)} \tag{2.2}$$

where C_L and C_R is the number of photons in the left and right node, after the split and $Vol(V \pm R)$ is the volume of node *V*, extended by the maximum query radius *R*. Wald et al approximate $Vol(V \pm R)$ using the following formula:

$$Vol(V \pm R) \approx \prod_{i=x,y,z} (V_{i,max} - V_{i,min} + 2R) \tag{2.3}$$

After we found the best split candidate (the one with the lowest VVH cost) we can split the small node into two sub nodes. To do so we need the current node's photon set, which is a bit mask representation of the photons inside the node. In order to complete our split, we simply perform a logical AND between the current photon set and the precalculated result split sets of the root small node. This is illustrated in Figure 4 where we split node A into two sub nodes (B and C) and the symbols (~, #, o, +, *) represent photons in node A.

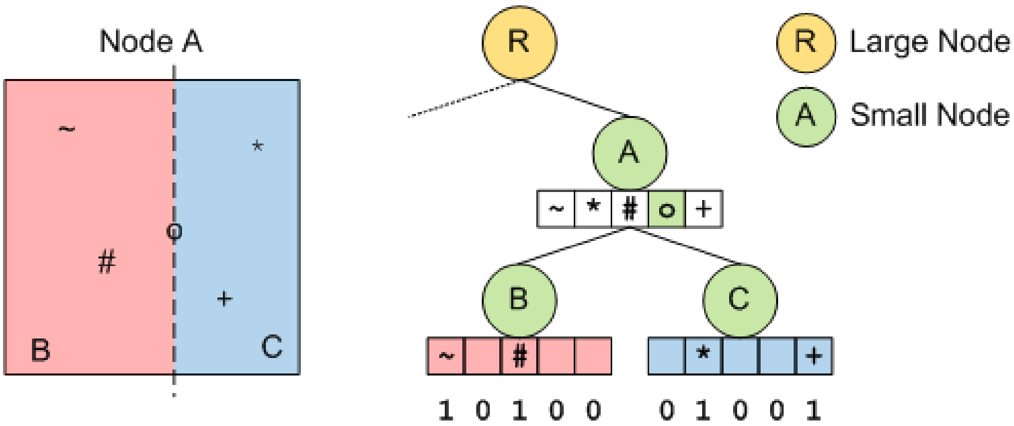


Figure 4: Small node splitting [36]

Besides easing node splitting, the binary photon representation also helps us calculating the number of photons in a node, which we need for VVH computation and to stop node splitting. All we have to do is to count the bits in the current photon set, using the parallel bit counting routine from [42].

After splitting is done, the new nodes are added to the active node list, in order to be processed during the next iteration step. Of course it can and will happen that some nodes will not create any new child nodes. Therefore, we have to add another step to compact our active node list and remove empty space, as is illustrated in Figure 5 [43]. If there are no more nodes left in the active list, we can finish the Small Node Stage and proceed to the final construction stage.

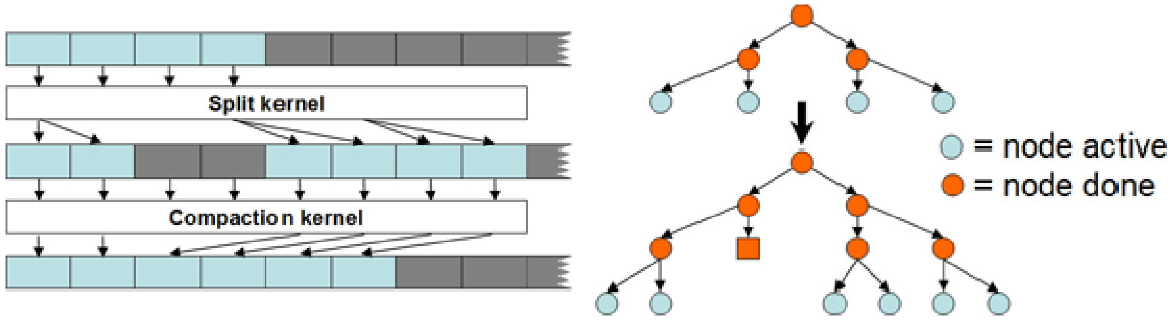


Figure 5: Compaction of the active node list [43]

In the final kD-Tree Output Stage, the tree is reorganized to change its layout to a preorder traversal of nodes, in order to improve memory access performance. First, we calculate the memory requirements for each node and its sub-tree by traversing the tree bottom-up. Finally, we are able to calculate each node's address, using the size information from the previous step in a top-down traversal pass. After reordering, each node stores its bounding box, split plane and the references to its children, as well as its photon's position and power.

2.3.3 Photon Search

A natural choice for locating the k nearest neighbors in a kD-Tree is the priority queue method, described in [44]. Unfortunately, it is not possible to implement a priority queue in CUDA efficiently, because memory access is incoherent and almost all arithmetic is interdependent, making it difficult for the hardware to hide memory latency. Therefore, Zhou et al propose an iterative kNN search algorithm, based on range searching [45].

Starting from an initial, conservative search radius r_0, they try to find the query radius r_k through a couple of iterations. During each iteration, a histogram of photon numbers over different radius ranges is created and the final search radius is reduced from it. The final radius r_k is then used for range search which returns all photons within that radius. Parts of these computations are performed on the CPU and, according to Zhou et al, the resulting error of the final kNN radius is less than 0.1%. Range search is implemented using the depth first search kD-Tree traversal algorithm from [45].

2.4 Analysis of existing Techniques

2.4.1 Brute Force Approach

First we take a look at the approach presented by Garcia et al [25], using a brute force technique to find the k nearest neighbors for a set of query points. On the plus side, this technique is very easy to understand and implement. It also supports arbitrary point dimensions, which is useful for scientific applications but not relevant to Photon Mapping.

The first problem with this technique is obviously the huge memory consumption. Basically the method grabs all available memory and uses it for its distance matrix computation. Even though you can theoretically scale the memory consumption down, this will have a bad impact on the performance as the supported query size is minimized.

Another problem, as already mentioned before, is the limited query size. Even with a lot of memory available, the query set size is limited to 2^{16} elements, due to limitations in the CUDA hardware API [47]. This is not a lot, considering that even for a low image resolution of 320 x 240 pixels we need to find neighbors for approximately 76,800 points.

Finally, the greatest disadvantage is the huge time complexity of *O(nm)* for *n* reference points (photons) and *m* query points. Especially low range graphics cards do not offer as much computation power as the top class devices and will not be able to achieve a good performance.

2.4.2 GPU kD-Tree Approach

The GPU kD-Tree approach from Zhou et al [36] has a couple of advantages compared to the brute force technique. Using a kD-Tree has been the first choice for all available software implementations of Photon Mapping. The main reason for this is that nearest neighbor search can be done pretty fast with this data structure, having a worst case time complexity of $O(k \cdot n^{1-\frac{1}{k}})$ [48], where $k = 3$ specifies the dimension of the tree. Another advantage of the kD-Tree is that memory consumption is quite moderate with *O(n)*.

However, we also experienced a couple of problems with the kD-Tree approach on the GPU. First, parallel tree construction requires a certain level of synchronization, because we are writing to a single data instance. This is done explicitly by the CUDA API which waits for a previous kernel to finish, before the next kernel is executed on the GPU. Even though kernel calls are asynchronous we have to synchronize and stall the CPU at some point, because the kD-Tree technique maintains a list of active nodes used for the next processing step. This synchronization is done explicitly as well, within the CUDA API layer.

This leads us directly to the next problem, the use of dynamic lists. The kD-Tree paper uses dynamic lists extensively for storing active nodes, tree nodes, small nodes and so on. CUDA only supports static arrays and therefore, additional work has to be done to grow lists by reallocating and copying memory. To avoid a high overhead caused by this memory management, Zhou et al double list sizes every time they run out of space. However, this leads to increased memory consumption during construction. Even though the final tree is stored without wasting any memory, the memory management during the construction stage constraints the supported maximum kD-Tree size.

Another problem is that traversing the kD-Tree is, by its nature, non-predictable. This means we cannot place the tree in memory without having a non-coalesced access patterns. In order to decrease random memory access during photon search, Zhou et al also use an iterative kNN search approach at the cost of utilizing the CPU.

As already explained in the previous chapter, Zhou et al also utilize the CPU for coordination work, during the tree construction. This means that the CPU and GPU are both busy when using the kD-Tree technique. Therefore, the CPU cannot be used for different tasks, like it is possible with other techniques that run completely on the GPU. Finally, we also think that the GPU kD-Tree approach is pretty complex and not that easy to implement. Also the use of many other parallel algorithms like scan, split and sort increase the effort required to use this technique.

3

A new Approach

"There will be the developers that go ahead and have a miserable time and do get good performance out of some of these multi-core approaches."

- John Carmack, Technical Director at idSoftware [46]

Objectives

- Introduce the Spatial Hashing technique
- Give detailed information on our Spatial Hashing implementation
- Highlight limitations and possible solutions for them

3.1 Photon Mapping using Spatial Hashing

3.1.1 Overview

Our new Photon Mapping approach is based on the CUDA particles paper by Green [49]. Green shows how to perform fluid simulation, based on particle systems, efficiently on GPUs. The central part of this technique is to simulate the interaction between all the particles in the system. Therefore, it is necessary to locate neighbor particles for each fluid particle and test for collisions between them. This technique can be directly mapped to our photon search problem. Instead of finding neighbor particles we have to find neighbor photons for a set of sampling points.

Our new approach uses a hash table, that allows us to look up a set of potential neighbor photons in *O(1)* time and which can easily be created and accessed in parallel. Each entry in the hash table references a spatial cell in the scene (numbered from 1 to 16 in the example below), containing photons as shown in Figure 6. In order to find the right cell, all we have to do is to calculate the hash value for the sample point and locate the right cell using the hash table.

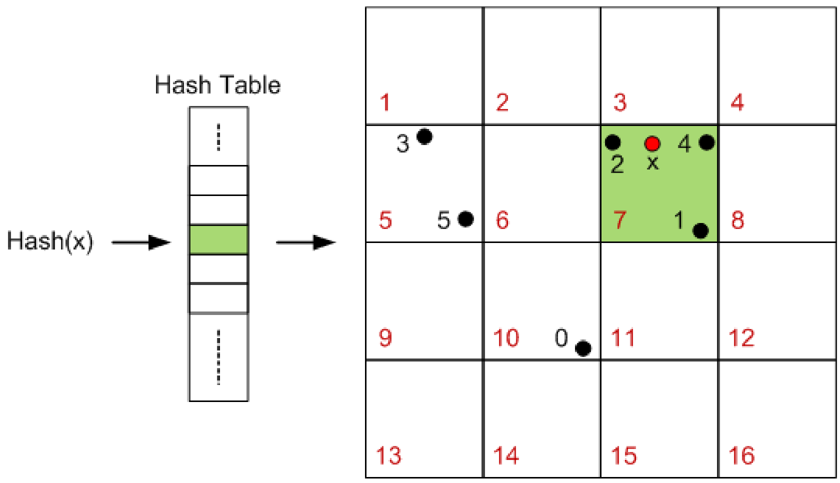


Figure 6: Immediate lookup of photons in the cell of the sample point x, using the hash table

Because the sample point can be located close to an edge of the cell, we also have to process photons from the neighboring cells in all three dimensions. Finally, we collect the closest *k* photons, using a sorted list that we implement using the fast on-chip shared memory. Photon collection has a time complexity of *O(n)*, where *n* is the average number of photons in a cell and its neighbors. It should be noted, that the number of photons in a cell grows only at a fraction of the actual photon map size because photons will be distributed over many cells. Photon maps rarely contain more than a million photons, which means that we will not get any problems with time complexity, due to extremely large photon maps.

3.1.2 Theory

Green presents two different ways to build the hash table, depending on the available device's compute capability. The first approach uses atomic operations (compute capability 1.1 and higher) to build the final hash table and is quite easy to implement. However, because of various reasons, this approach is not as fast as the other technique in Green's paper. For our Photon Mapping approach, we will concentrate on the faster, but more complex solution, based on sorting.

This algorithm consists of several kernels that are executed after each other. The first kernel calculates the hash values for each photon in the photon map and stores the resulting values in an array, along with the associated photon's index. In the next step, Green sorts the array based on the hash values, using the radix sort from Le Grand [50]. Finally, we need to find the start indexes for each entry in the sorted array to create our hash table.

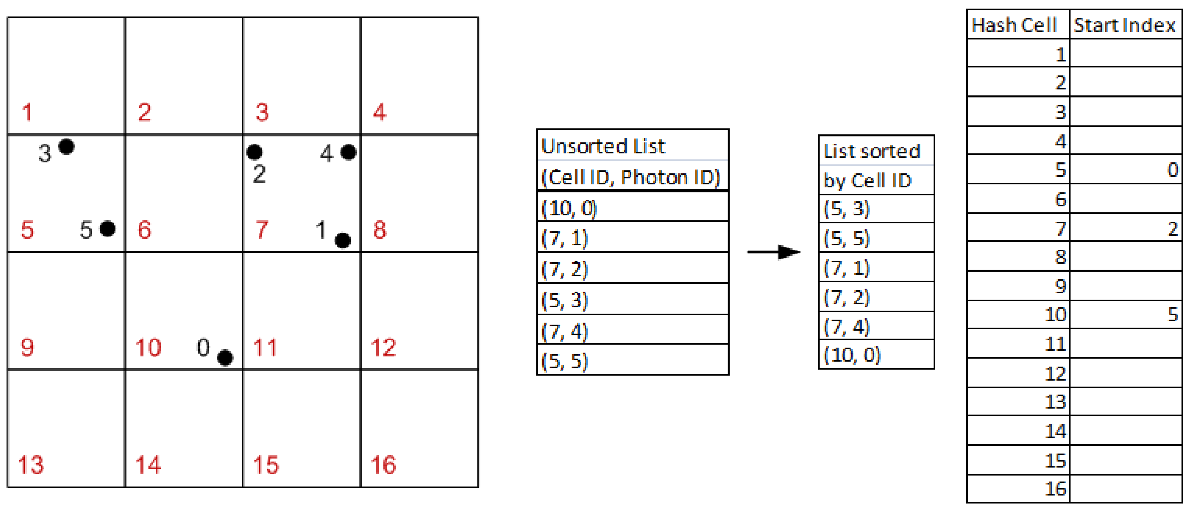


Figure 7: After calculating the hashes the photon list is sorted and the hash table is created

As can be seen in Figure 7, the sorted list introduces another level of indirection as it only contains a reference ID to the actual photon data in the photon table. In order to get rid of this overhead, we simply reorder the photons' data according to their position in the sorted array. This allows us to access the data of the first photon in a cell directly, simply by looking up its index in our hash table. The other photons in the same cell can be easily iterated, sequentially. The array containing the reordered photon data is finally bound to texture memory. Unlike global memory reads, texture lookups are cached and the sorted, sequential order will improve the coherence when accessing the photons during the photon search stage.

Choosing the right hash function is quite important for our technique to work efficiently. All hash functions require a grid position as their input parameter. The grid position p' for a photon with position p can be easily calculated using the following equation:

$$p' = \lfloor p \cdot \vec{s} \rfloor \tag{3.1}$$

where the scaling vector $\vec{s}$ specifies the scaling factors for a cell, in each dimension. Using p' we can now continue and calculate the hash value for our photon position. Green suggests two different hash functions in his paper. The first one simply calculates the linear cell id for the given grid position p using equation 3.2.

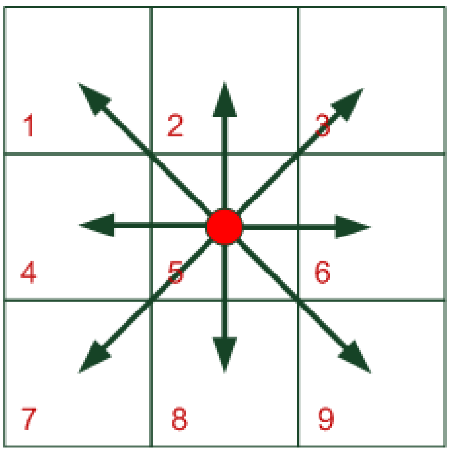
$$f_{Hash}(p) = pz \cdot gridSizeY \cdot gridSizeX + py \cdot gridSizeX + px \tag{3.2}$$

The *gridSize* factors in the formula above specify the number of grid cells along the x, y and z-axis. Alternatively, Green suggests using a hash function, based on the Z-order curve [51] to improve coherence of memory accesses.

In Green's paper, each particle has to be checked for collision with other particles. First, he calculates each particle's hash to find the cell of interest. Then he loops through all 27 neighboring cells (using a 3x3x3 pattern) to test all particles for collision. We can almost map this technique directly to our needs for photon search. Our initial point of interest is not a photon but a sampling point in our scene, whose color we want to estimate. We can simply use the sampling point's coordinate to calculate its grid position and hash value, in order to find the cell it is located in. All we have to do now is to collect the closest k photons from all neighboring cells and we are able to calculate our radiance estimation.

3.1.3 Implementation

Before we can start calculating the hash values for our photons we need to initialize a couple of parameters, required for the hashing. Because we support photons with negative coordinate values, we need to provide the minimum coordinate values for the x, y and z dimension, stored in our photon map, which we call the world origin. We will use the world origin in our grid position calculation to transform all photon positions into a positive coordinate system, by simply adding the world origin to the input position. For the grid cell size, we currently use the maximum photon query radius and therefore, our scaling vector $\vec{s}$ is simply $(1/r_{max} \quad 1/r_{max} \quad 1/r_{max})$. Finally, we calculate the number of grid cells along each axis, using the cell size and the bounding box extents of the photon map. We increase the number of grid cells along each axis by two additional cells to ease the handling of border cells during photon search. This allows us to avoid any expensive checks, otherwise required for clamping grid positions. The last step performed, during the initialization phase, is the creation of the neighbor offset lookup table. This table is used to calculate the neighbor cells' grid positions during the photon search, as shown in Figure 8. The table is placed in constant memory which caches memory reads and provides better performance, compared to global memory.



Neighbor	Offset
1	(-1/1)
2	(0/1)
3	(1/1)
4	(-1/0)
5	(0/0)
6	(1/0)
7	(-1/1)
8	(0/-1)
9	(1/-1)

Figure 8: Calculating neighbor cells' grid positions, using the values from the offset lookup table

After finishing all initialization work, we will execute the first kernel which creates the unsorted hash value array, where each entry also contains the hashed photon's index. Afterwards we sort this array, based on the photon hashes, using the radix sort algorithm from [37]. Now we have to determine the start photon index for each grid cell, in order to create the final hash table. This is done by executing a kernel function for each entry in the sorted array that checks, if the previous photon's hash value is different from its own. If it is, we know that the current entry marks the start of a cell and the previous entry marks the end of another cell. This information can be efficiently exchanged using the GPU's on-chip shared memory. After the resulting indexes have been written to our hash table we reorder the photons, as explained earlier, to improve memory access coherence. We finally end up with a hash table that can be indexed using a point's hash value and a sorted photon table, as illustrated in Figure 9.

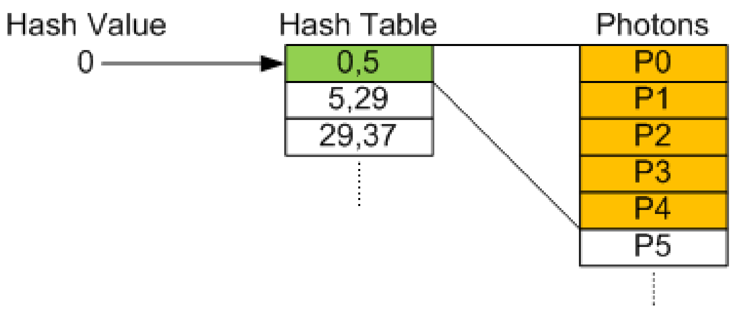


Figure 9: Hash Table (start index, end index) and Photon Table relation

In order to find the required photons, we execute our parallel search kernel for each point in the query set. First, we calculate the query point's grid position and initialize our photon list to an empty list. For efficiency reasons, we place the photon list in shared memory which is an order of magnitude faster than global memory. Next we iterate through the 27 neighbor cells of interest, using the offsets from our precomputed neighbor offset table. We simply add the values from the table to the current grid position and use that new grid position for the hash calculation. In our current implementation we use a simple sorted list approach for collecting photons. As long as the list is empty, we just keep adding photons until the list is full. If we find a closer photon and the list is already full, we insert that photon to its sorted position in the list and shift all following photons back. The last item in the list is simply lost as we do not need it any longer. While this approach works well for our prototype, we think better performance can be achieved with different methods, like a heap data structure [52]. When the kernel finishes with the processing of all cells, the remaining k photons in the shared memory list are written to the result array, stored in global memory.

3.1.4 Limitations

In this chapter, we talk about some limitations that our technique has, compared to others and present some possible workarounds for them. The first constraint is caused by the limited size of shared memory, which is used during photon collection. In our current implementation, 32 threads have to share 16 KByte of shared memory. For each photon we need 8 bytes of memory in the list, 4 bytes for its square distance to the query point and 4 bytes for its power in RGBE format. This means we can gather a maximum number of $k \approx 60$ photons per sampling point (we cannot use the full 16 KByte of shared memory because CUDA uses part of it for kernel parameters). Usually, this is not a big problem as it should be sufficient to locate 40-50 photons per query point. In case more photons are required for the radiance estimate, we can increase the maximum list size by decreasing the thread block size from 32 to 16. However, this solution should only be applied if absolutely necessary, as our experiments showed a performance decrease of around 25% with this change. Also, future GPUs most likely provide more shared memory, allowing larger photon search lists without any modification to the code.

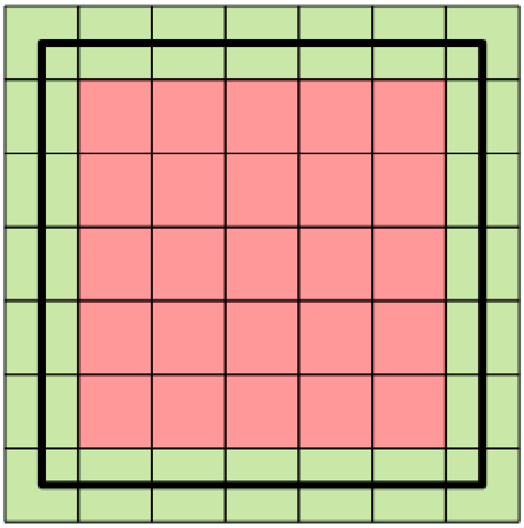


Figure 10: 2D view of a scene, green cells occupy geometry whereas red cells do not

As shown in equation 3.2, our hash function uses the number of grid cells to calculate a hash value from a grid position. Unfortunately, using these factors causes some problems and disadvantages. First, we might end up wasting a lot of memory with empty entries in our hash table because the linear hashing function is not ideal for certain scene types, as Figure 10 shows. The green cells indicate cells that contain geometry and therefore, most like photon information. Red cells are just floating around in empty space and do not contain any useful data but still occupy an entry in the hash table.

The other drawback of this hashing function is that the scene size is limited. For instance we are not able to easily model an open landscape scene because our hash function depends on the grid size. Without choosing a reasonable amount of cells along each axis we can get problems with the increasing memory requirements of our hash table. A solution to these problems is using a different hash function, like the one presented in [53]:

$$f_{Hash}(p) = (px \cdot p1 \ \boldsymbol{xor} \ py \cdot p2 \ \boldsymbol{xor} \ pz \cdot p3) \ \boldsymbol{mod} \ n \qquad (3.3)$$

where *p1*, *p2* and *p3* are large prime numbers and *n* specifies the hash table size. As you can see, this function is not based on the grid size and therefore, does not experience the aforementioned problems. In order to avoid the expensive modulo operation we suggest choosing a hash table size equal to a power of 2. In that case the modulo can be replaced by a much cheaper logical AND operation with *n – 1*.

Another limitation is that the photon query radius has to be specified at construction time and remains fixed for the hash table’s life time. This is because we use the radius as the cell size, when calculating the grid position for photons. If the radius is enlarged without recalculating the hash table, we might miss some photons during photon search. On the other hand, if the radius is decreased we will not experience any performance improvements from the smaller search radius. Other techniques like the kD-Tree also use the query radius during construction, in example for the VVH calculation. However, it is unlikely an application needs to change the query radius and even if so, both techniques should be fast enough to recreate the required data structures at run-time.

4

Performance Measuring

"There is a famous rule in performance optimization called the 90/10 rule: 90% of a program's execution time is spent in only 10% of its code. The standard inference from this rule is that programmers should find that 10% of the code and optimize it, because that's the only code where improvements make a difference in the overall system performance."

- Richard Pattis, Senior Lecturer at Carnegie Mellon University, Pittsburgh [54]

Objectives

- To specify the environment used for the performance measurements
- To specify what measurements we can use to compare Photon Mapping techniques
- To explain the different measurements' implication on performance

4.1 Measureable Parameters

In order to validate and compare our technique to others, we will use a couple of measurements which are explained in more detail in Table 1.

Table 1: Measurements

Measurement	Description
Construction Speed	Measures the time it takes for a technique to be ready to search photons. This includes memory allocations as well as creation of data structures, like trees or tables. Shorter is better.
Search Speed	Measures how fast a technique is able to locate the *k* nearest photons for *n* query points. Shorter is better.
Memory Consumption	Measures each technique's memory consumption after the construction step. Lower is better.
Memory Consumption (Peak)	Measures the peak memory consumption of each technique during the construction phase. Lower is better.
Non-Coalesced Reads/Writes	Measures the number of non-coalesced memory accesses. Lower is better.
Image Quality	Verifies if the photons returned for the radiance estimate are correct.

Of course it is important to keep photon search time as low as possible, in order to obtain good performance. However, it is just as important to keep construction time very low to get the most out of Photon Mapping. Scenes with dynamic lights, for instance require rebuilding the photon map every frame and a high construction time is unacceptable in such cases.

Memory consumption is an important factor as well. Even though GPUs have quite a bit of memory available nowadays, not all of these resources might be available to our application. The

GPU memory is also used by the OS to display the user interface or by 3D APIs to store resources, like textures and geometry data. The second important memory measurement is the peak memory usage of a technique. Even if the final memory requirements are low, if a technique requires a lot of temporal memory during construction, we are still constraint by these needs. We also take a look at the memory access pattern of the different techniques. Non-coalesced memory accesses are penalized with a severe performance hit on GPUs. Therefore it is important to keep the number of such accesses as low as possible.

Finally, we take a look at the correctness of the different GPU techniques. The resulting photon sets used for the radiance estimate should be the same with all techniques. To verify that, we will directly visualize the photon map and compare the visual quality of the resulting images.

We will perform our tests with two different photon map types. Global photon maps contain photon information for the whole scene and are primarily used to render interreflections and soft shadows. Caustic photon maps are created by emitting photons only towards reflective and refractive objects, because we need a higher number of photons to visualize caustics. Therefore, the photons in a global photon map are more scattered over the scene whereas the photons in a caustic photon map are concentrated at certain areas. Because caustic photon maps contain lighting information for smaller areas, they do not require as many photons like global photon maps, which contain lighting information for a whole scene. To see if our approach handles both map types well, we carried out each test with global and caustic photon maps.

4.2 System Specification

Our test system has the following hardware and software specifications:

Table 2: System Specification

System Specifications	
CPU	Intel Mobile Core 2 Duo P8600 @ 2.4 GHz
Memory	4 GB DDR2
GPU	nVidia GeForce 9600M GT

GPU Cores	32 Cores @ 1.25 GHz
GPU Memory	512 MB G-DDR3 @ 800 MHz
CUDA Version	2.2
Driver Version	185.85
OS	Windows Vista 64 Business Edition

4.3 Application Specification

Our test application implements the following Photon Mapping techniques:

- A software kD-Tree, based on Jensen's photon map code [44]
- A GPU kD-Tree, based on Zhou et al [36]
- A GPU brute force k nearest neighbor search, based on Garcia et al [25]
- The GPU Sptatial Hashing technique presented in this report using two different hashing functions:
    - Linear hashing function by Green [49]
    - Hashing function by Teschner et al [53] with a hash table size of 2^{14}

We used the CUDA 2.2 SDK and Microsoft's C++ Optimizing Compiler 15.0 with the amd64 release mode build and all default optimizations turned on.

4.4 Test Scene Specification

Our test scene is a simple Cornell Box scene with a shiny and a glass ball. The light source is an area light source, located right under the ceiling (not displayed in the rendered image). Notice the soft shadows and interreflections between the walls, as well as the caustic under the glass ball. Because the lighting information in the photon map is decoupled from the underlying geometry, the results of our tests are also valid for much more complex scenes with a higher polygon count.

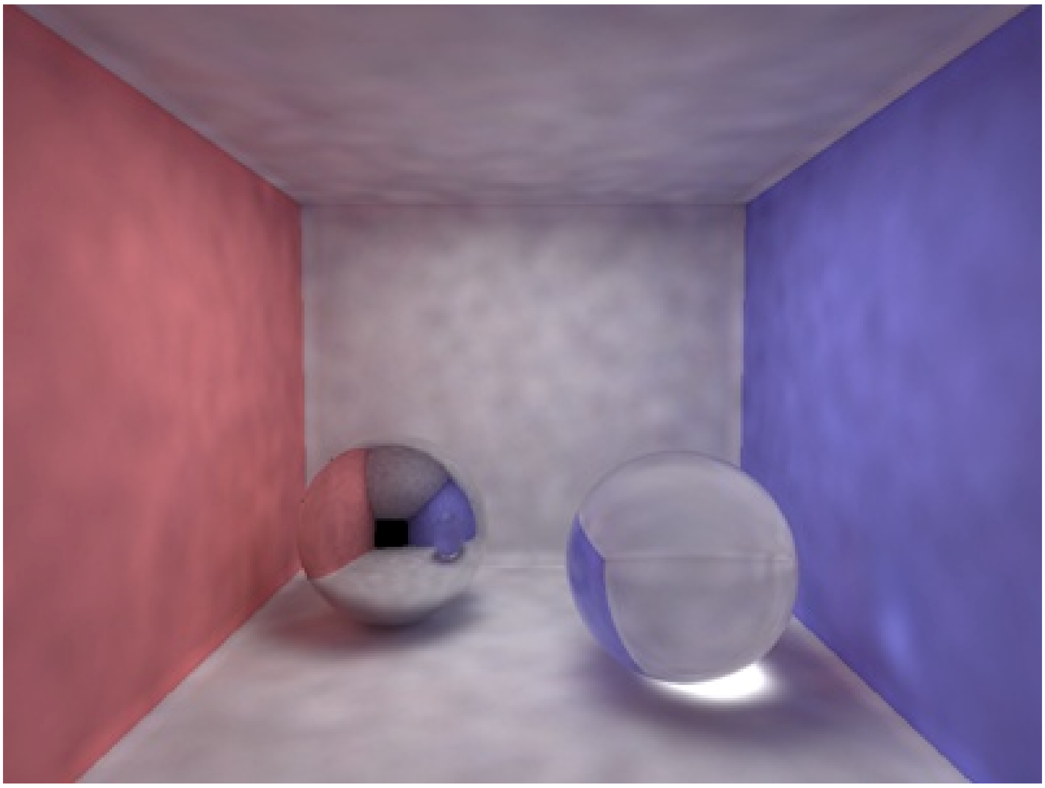


Figure 11: Test scene

5

Results and Analysis

" It's just not right that so many things don't work when they should."

- Stephen Wozniak, Chief Scientist at Fusion-io

Objectives

- To present the results of our performance measurements
- To analyze and critically evaluate the observed results

5.1 Construction Time Performance

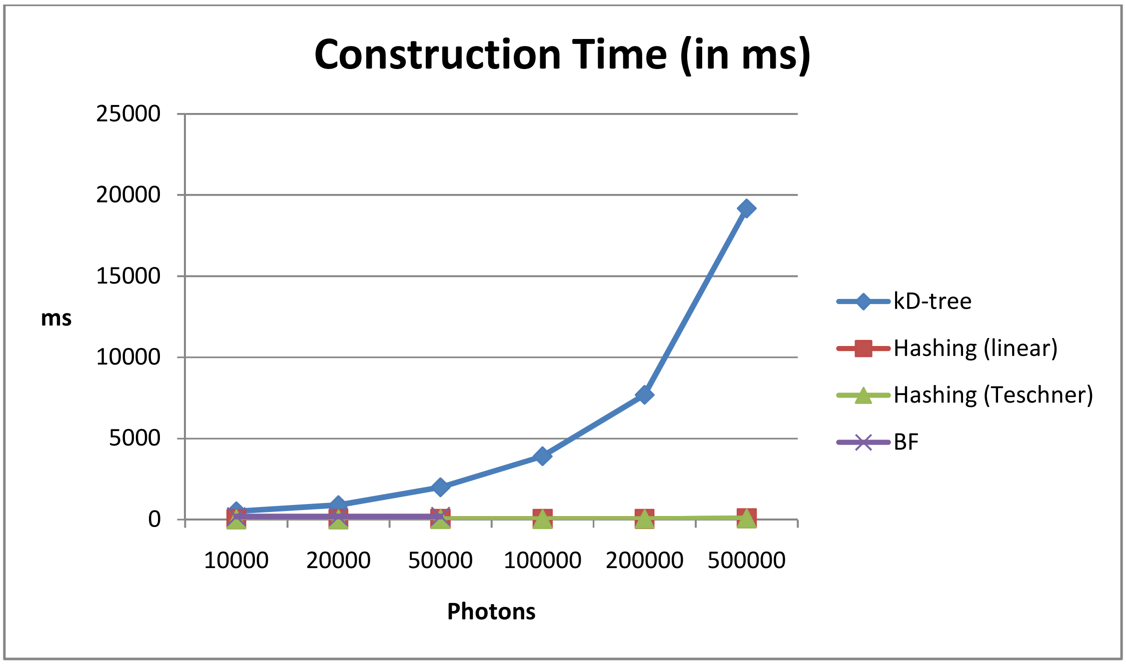


Figure 12: Construction Time Performance for Global Photon Map

As can be seen in Figure 12, our spatial hashing technique is much faster than the kD-Tree approach. The kD-Tree construction cost increases with the amount of photons in the photon map, whereas our hash table creation takes almost constant time. The reason for this behavior can be explained quite simple. Our hash table creation is fully parallelized, beginning with the hash calculation for each photon and ending with the hash table creation and the reordering of photons. In contrast, the kD-Tree technique iteratively executes several kernels for scan and split operations at each tree level. Every successive kernel invocation performs implicit synchronization because a GPU can run only one kernel function at a time. This means, before a new kernel can be started, all threads from the previous kernel function must be finished.

Another reason for the longer creation time is non-coalesced memory access. The spatial hashing technique's access pattern is completely coalesced, except for the final photon reordering. On the other hand, the kD-Tree approach performs non-coalesced memory writes after every split, when it reorders the sorted coordinate and index lists. Also, the final tree reorganization phase's access pattern is almost entirely random and therefore, non-coalesced.

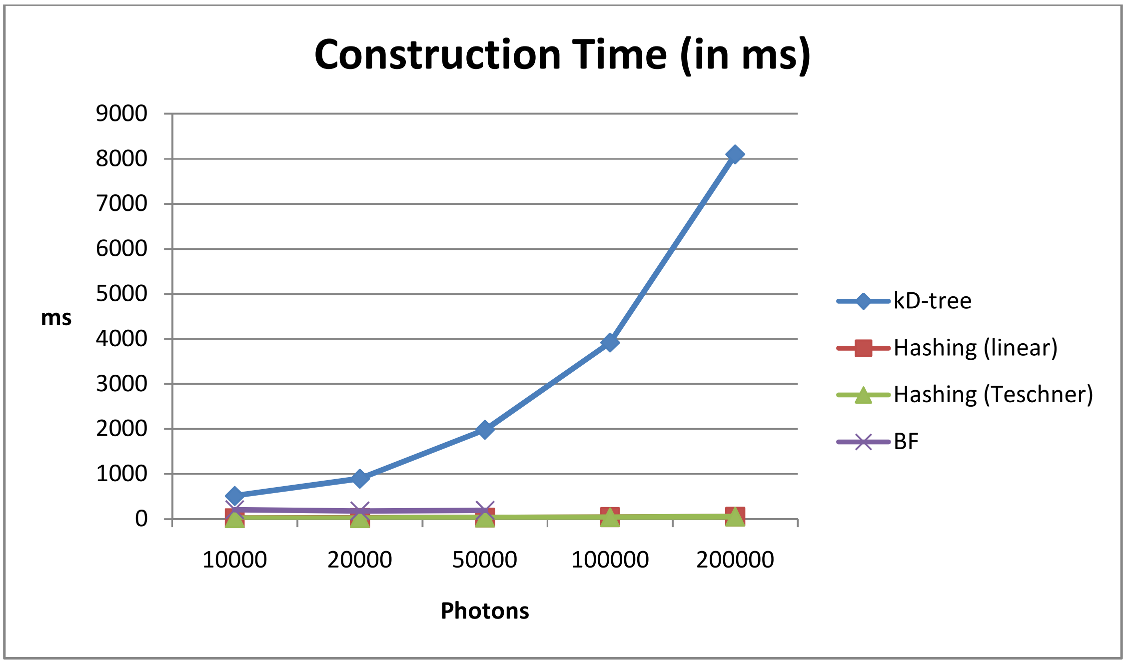


Figure 13: Construction Time Performance for Caustic Photon Map

As Figure 13 shows, the measurements are almost identical for caustic photon maps. Again our spatial hashing methods are much faster than the kD-Tree method, especially when the photon map size becomes larger. Construction time for the GPU brute force technique is pretty low as well, but considering that it is only allocating memory and copying photon data to the device, this is not a big surprise. One thing to note is that we were only able to test the brute force approach with photon maps up to a size of 50,000 photons. This is due to a limitation in the CUDA hardware API, as already mentioned in chapter 2.

5.2 Memory Requirements

It is obvious that, if we increase the amount of photons in our photon map we will need more memory to store this information. However, as Figure 14 shows, there is still a huge difference between our hashing methods and the kD-Tree. Especially with an increasing number of photons, our spatial hashing technique consumes significantly less memory than the kD-Tree approach. The reason for this behavior is that the tree structure produces a lot of overhead. For every node we have to store references to its children, the splitting plane and bounding box, along with the photon information itself. The only overhead our technique introduces is the hash table, which grows with the scene size or the number of hash table entries, depending on the used hash function.

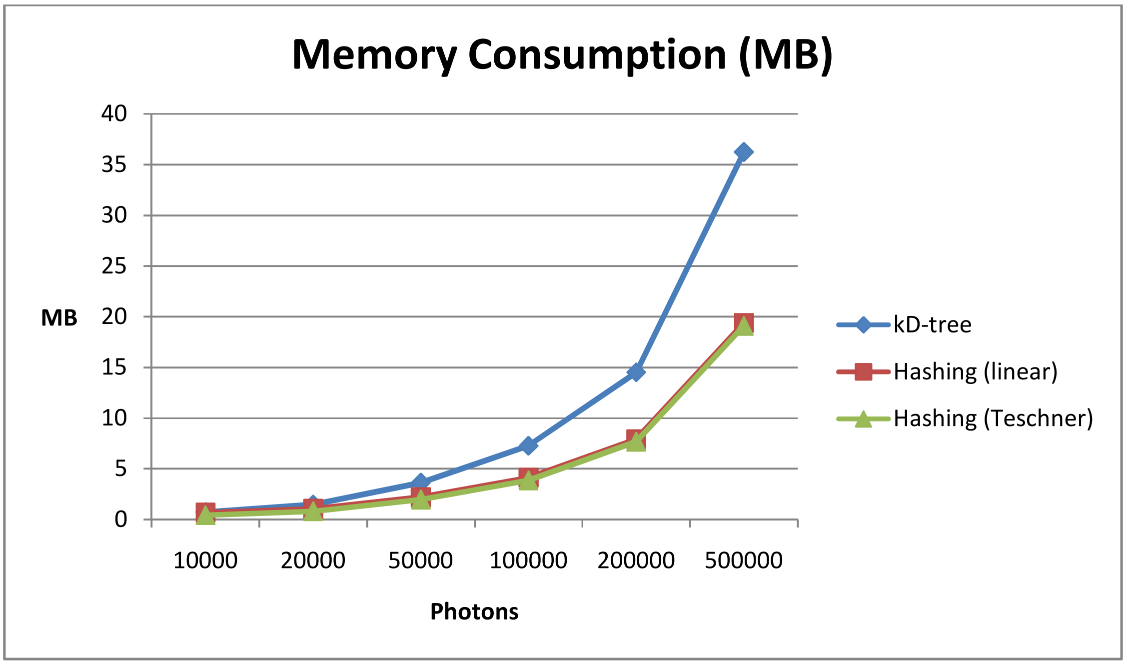


Figure 14: Memory Consumption

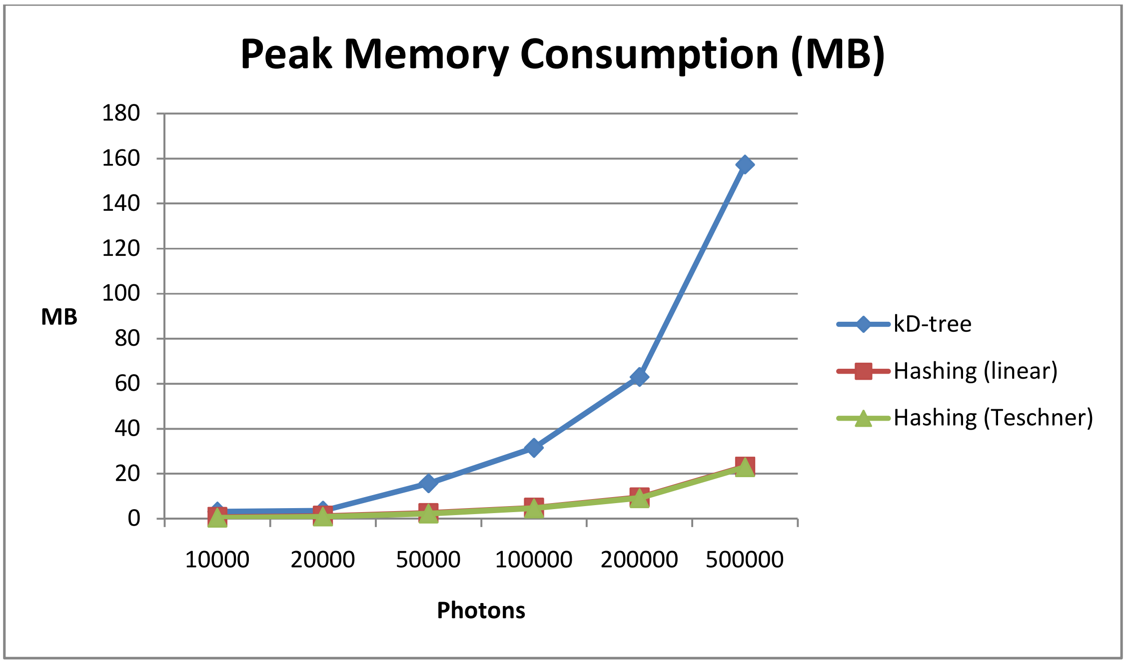


Figure 15: Peak Memory Consumption

The next thing we measured is each technique's peak memory consumption, during the creation phase. Figure 15 shows the data of our experiments with the kD-Tree and our spatial hashing approach. The results show that our technique is again, significantly better than the kD-Tree approach, with an almost 8 times lower peak consumption at 500,000 photons.

As you may have noticed, our graphs are missing the GPU brute force method. Because this technique just grabs all the available memory for the distance matrix, we thought it makes no sense to include this method in the comparison.

5.3 Photon Search

In this section we show, how the different techniques perform during the photon search. There are a couple of parameters that have an influence on the performance at this stage, apart from the different photon map types:

- The number of photons in the photon map
- The number of *k* nearest neighbor queries, which increases with higher image resolutions
- The number of photons, requested for each query (*k*)
- The maximum allowed distance between the query point and a photon

During our experiments we discovered that the brute force technique is not able to compete with the other two techniques, in terms of speed. In a best case scenario the brute force approach was almost 10 times slower than the kD-Tree and the spatial hashing methods. Therefore, we decided to omit the brute force technique from our photon search graphs in order to keep them meaningful.

5.3.1 Photon Map Sizes

In Figure 16 we see the performance graph for the kD-Tree and the hashing technique, with an increasing number of photons in the photon map. We first take a look at the test results, obtained with the hashing function from Teschner et al. As we can see there is a slight increase in search time with photon maps larger than 200,000. This performance drop is caused by collisions in the hashing function. Because of these collisions, some photons end up in the same hash table entry, even though their location is not close to that of the other photons in the entry. With an increasing number of photons, the chance for a collision increases, causing the observed performance hit. However, the performance with bigger photon maps can be greatly improved by simply using a larger hash table size.

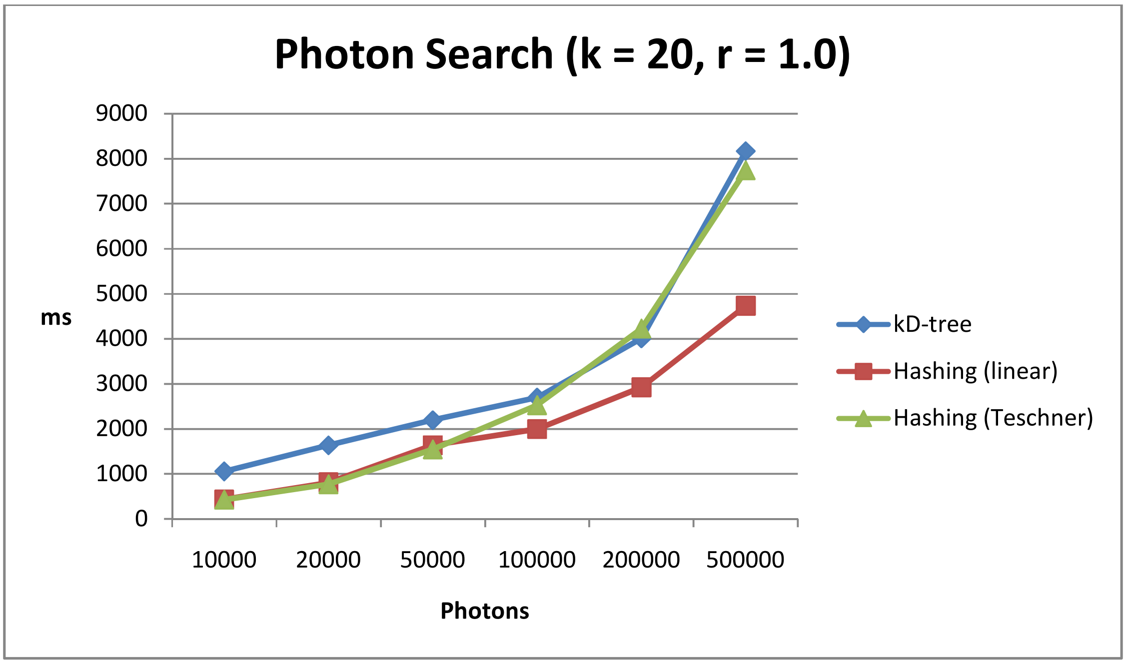


Figure 16: Global photon map search performance with different map sizes

As we can see, the increase in search time with the kD-Tree and the linear hashing function is pretty steady, before performance drops off a little bit with the kD-Tree. The reason for this performance hit is an increasing access to local memory. Local memory is basically global memory, which is only visible to a single thread, but accessing it is just as expensive as accessing normal global memory. Local memory is used for the stack-based kD-Tree traversal. With the increasing number of photons we get a deeper tree, resulting in more local memory accesses.

Because we also have to traverse the tree several times before the actual search, in order to determine the *k* nearest neighbor search radius, a deeper tree becomes more expensive to process. Figure 17 shows the increase of local memory accesses which become significantly more with a larger photon map size. Here, one of the disadvantages of using a tree structure on the GPU becomes obvious. By its nature, traversing a tree requires a lot of scattered memory accesses, which is extremely bad for performance on current GPU devices.

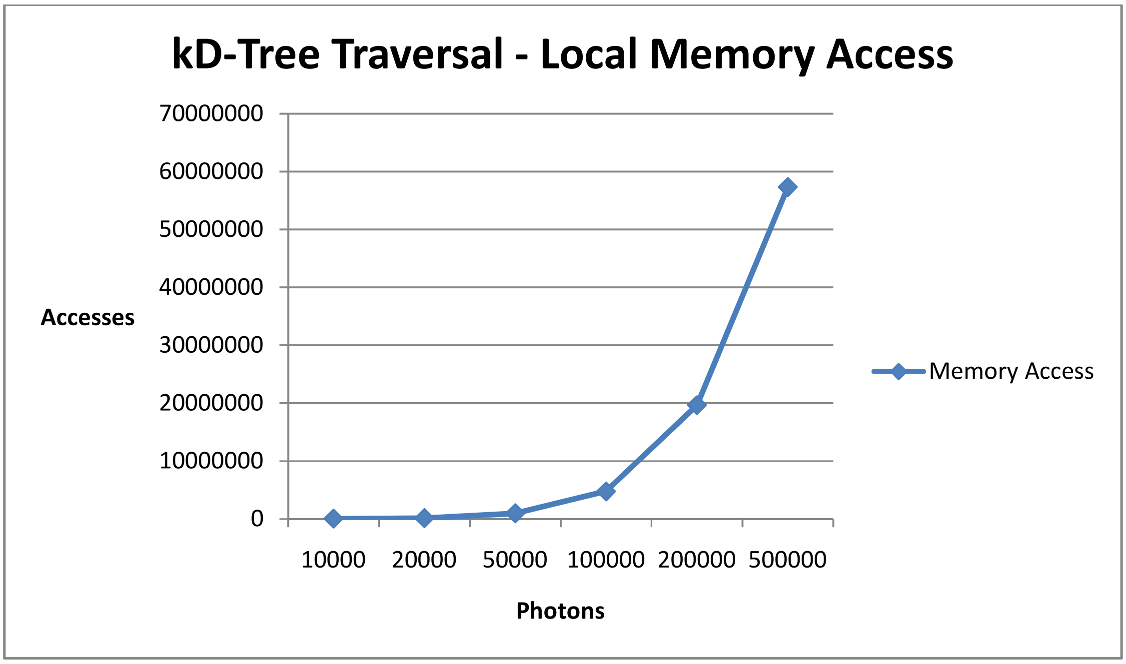


Figure 17: kD-Tree Traversal - Local Memory Access

If we take a look at our hashing method, using the linear hashing function, we can see that we have a steady growth in search time for larger photon maps. This makes perfect sense, as the number of photons in each hash cell also increases with more photons in the photon map. However, we have some advantages over the kD-Tree approach that enable us to obtain a better performance.

First, we do not have to traverse a tree or a similar data structure, using local memory. Instead, we just have to look up our grid cell in the hash table, using a single memory read. The second optimization we are able to use is texture memory, for storing the photon information. Because we are able to access the photons sequentially, we can actually benefit from the texture cache.

If we take a look at the performance graph for caustic photon maps in Figure 18, we will notice that all methods, the kD-Tree and both hashing methods, need more time to find photons, than they need with a global photon map. The reason behind this is the high photon density in small areas, compared to the photon density in global photon maps, as illustrated in Figure 19. In case of the kD-Tree, this forces us to traverse through more tree nodes because we fail to reject tree branches at higher levels.

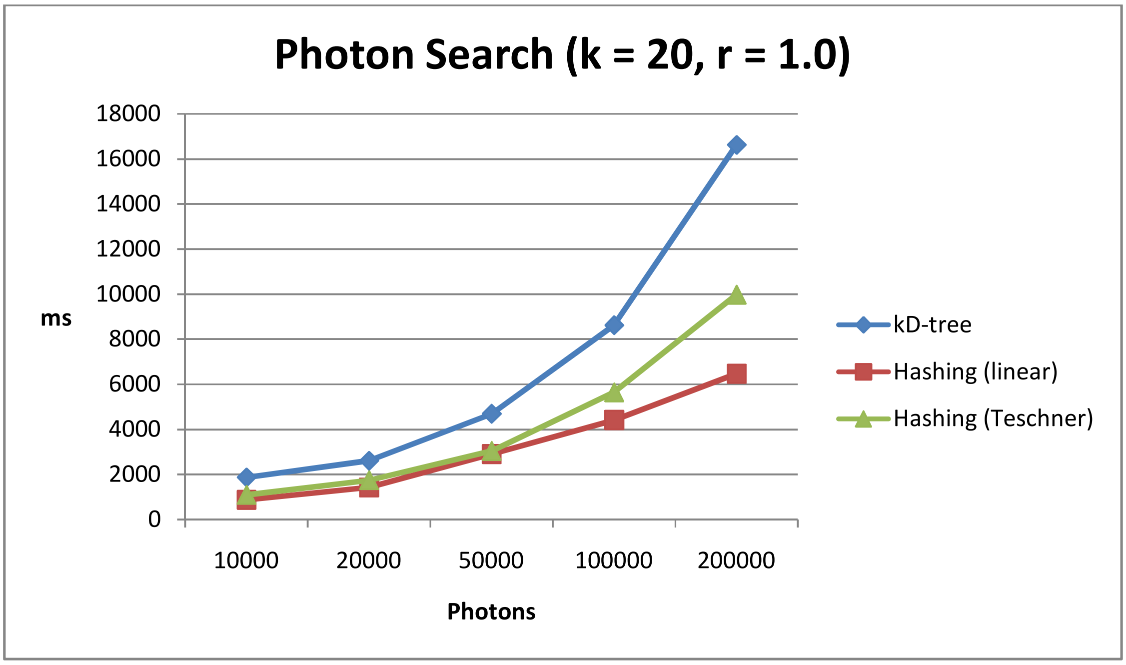


Figure 18: Caustic photon map search performance with different map sizes

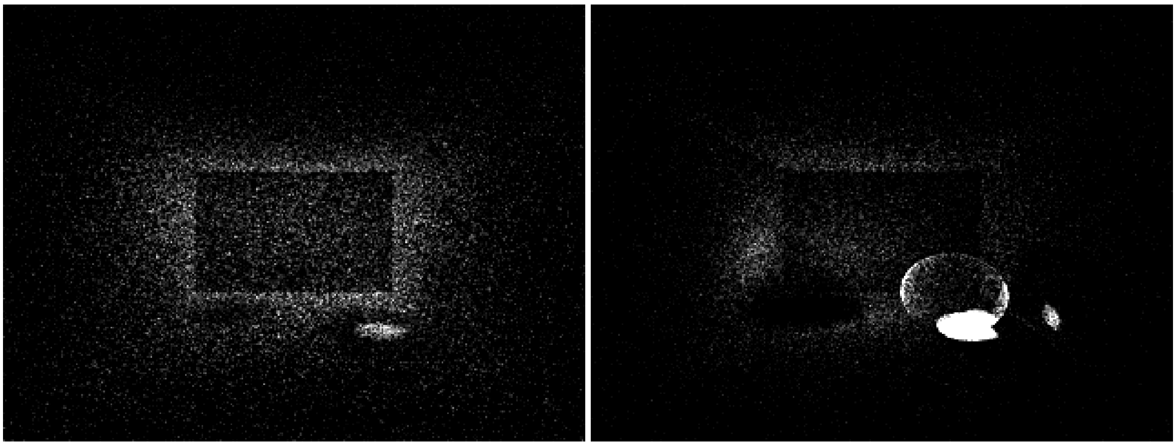


Figure 19: Photon Density in a global (left) and a caustic (right) photon map

This time, Teschner's hashing function performs much better and is significantly faster than the kD-Tree technique. This is because most photons are located very close to each other and therefore, the number of hash collisions is not as high as with a global photon map. However, while collisions decrease, the number of photons in a cell is much higher because of the high photon density in some areas. The higher photon numbers in the cells are responsible for the increased photon search time, because more photons have to be processed. Using the linear hashing function we experience a similar problem. Our final hash table will have lots of almost empty cells and a couple of cells, containing lots of photons. However, we do not have a performance hit due to hash collisions, which is why this hashing function performs better.

Another problem with our hashing technique is the sorted list that we use to collect photons. With many close photons around our sampling point, we end up reordering the list many times, before we have found our final set. One reason why our hashing method is still pretty fast is that we are able to return photon information, for areas with a low photon count much faster than the kD-Tree implementation. If we have none or only a few photons in a cell, a thread will finish much faster than if it has to traverse a whole kD-Tree, especially if the tree is very deep, as is the case with large photon maps.

5.3.2 Number of Photons

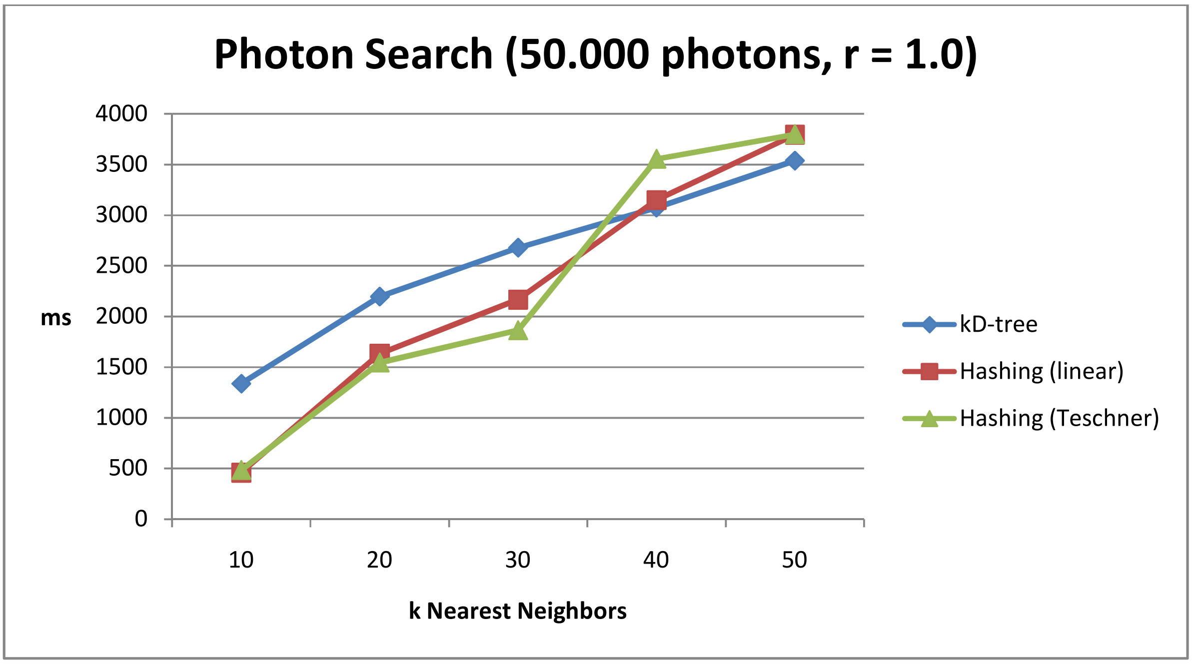


Figure 20: Global photon map search performance with different k sizes

In this section we take a look at what impact, different photon sample sizes have on the overall performance. As Figure 20 shows, the *k* parameter has a pretty big influence on the performance with both techniques. We can see that the kD-Tree's performance is slowly degrading with an increased photon search number. This slowdown is caused by the larger number of memory writes when writing the resulting photons to global memory.

If we take a look at the spatial hashing technique, we will notice that the chosen hashing function has a big impact on the performance. Our experiments showed that the main reason for these different results is our photon collection implementation. As already explained, we are using a sorted list to return the *k* closest photons when a thread finishes. However, photons are stored in random order with respect to their sampling point distance. This means we most likely end up shifting around lots of photons in our sorted array. If we take a look at the behavior of our spatial hashing technique with caustic photon maps in Figure 21, we will see a similar result.

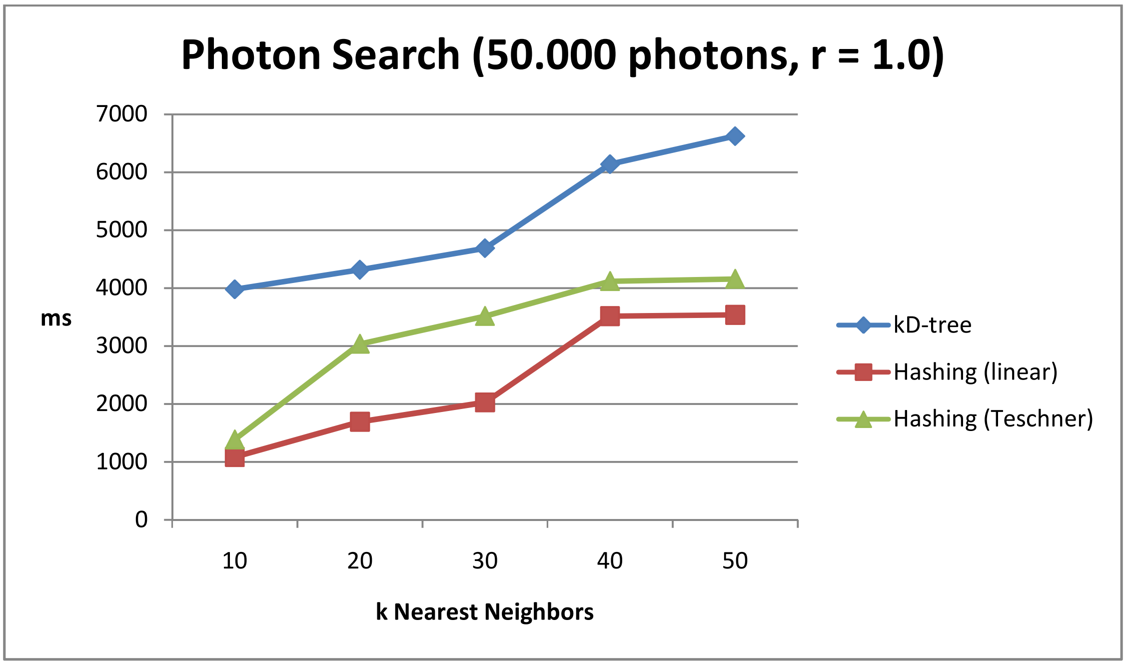


Figure 21: Caustic photon map search performance with different k sizes

Again, the performance of the kD-Tree is steadily decreasing, whereas the spatial hashing technique shows some weakness with higher sample numbers. Interestingly, our linear hashing method shows a significant performance drop when changing the sample size from 30 to 40. Our tests showed that the cause of this problem is again, the sorted list. On the other hand with 50 samples there is hardly any performance decrease noticeable. It seems that the ordering of photons allows to reject many photons earlier with list sizes smaller than 40. With a larger list size we are adding many photons at list positions 30 to 40, making it necessary to shift photons to the back.

With Teschner's hashing function we could not obtain the same performance as with the linear hashing function. Again we see a significant performance drop, this time between sample sizes of 10 and 20 photons. The reason for this behavior is again, the sorted list combined with the hash collisions. Because of the collisions, an entry might have photons which are actually far away from the sample point. This causes a lot more shifting in our sorted list, leading to the performance hit. The reason why this only happens with the caustic map is the higher photon density. With the global photon map we were able to reject the colliding photons easily by testing, if their distance to the sampling point is smaller than the maximum photon search radius. However, this check fails with the caustic photon map, where lots of photons are located within the search radius. Nevertheless, with the caustic photon map our solution is still better than the kD-Tree implementation.

5.3.3 Query Radius

The maximum query radius is an important parameter for both, the kD-Tree and the spatial hashing technique. The kD-Tree technique uses it for the initial query radius r_k, which is required during construction for the VVH calculation and during photon search for the radius estimation. For our spatial hashing technique, the query radius determines the size of our grid, cells which are used for calculating the grid positions and hash values.

Figure 22 shows the performance of both techniques with different query radii. As we can see in the graph, all techniques have an almost steady increase in search time with an increasing query radius. Because we have to visit more tree nodes when searching for photons, using a larger query radius, the kD-Tree requires more time. When using the spatial hashing technique with a growing query radius, we end up having fewer hash cells in our hash map. This means there are more photons in the remaining cells which causes the increased search time. Again the spatial hashing method, using the linear hashing function, is faster because we avoid the overhead of traversing a tree structure or having hash collisions.

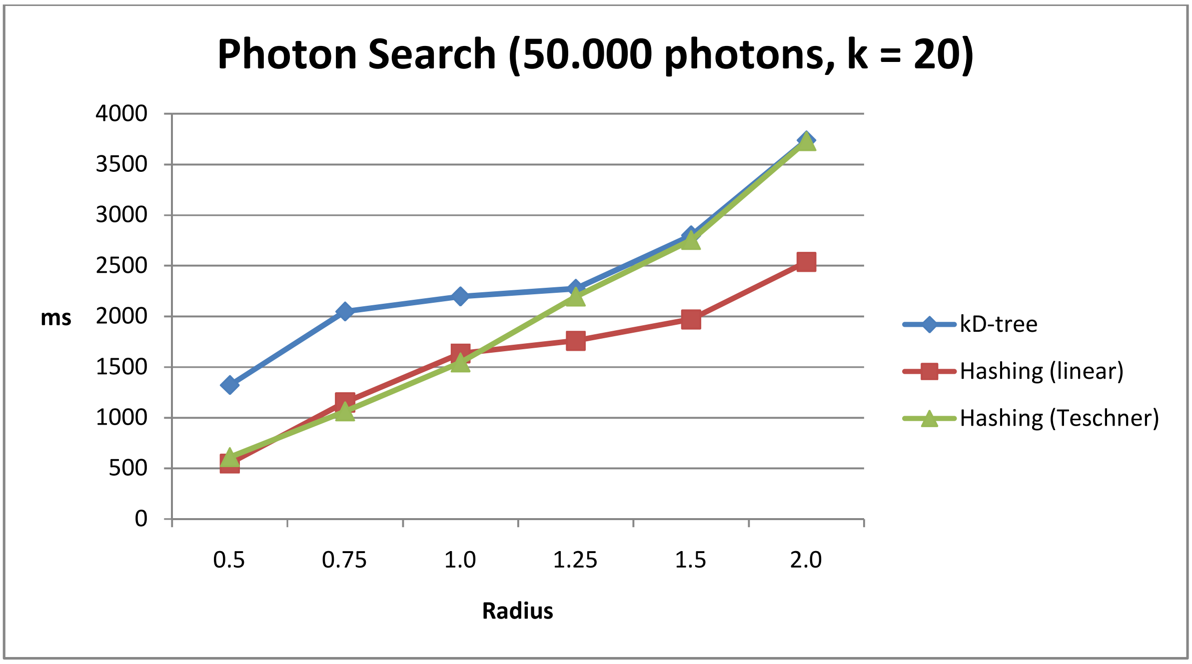


Figure 22: Global photon map search performance with different query radii

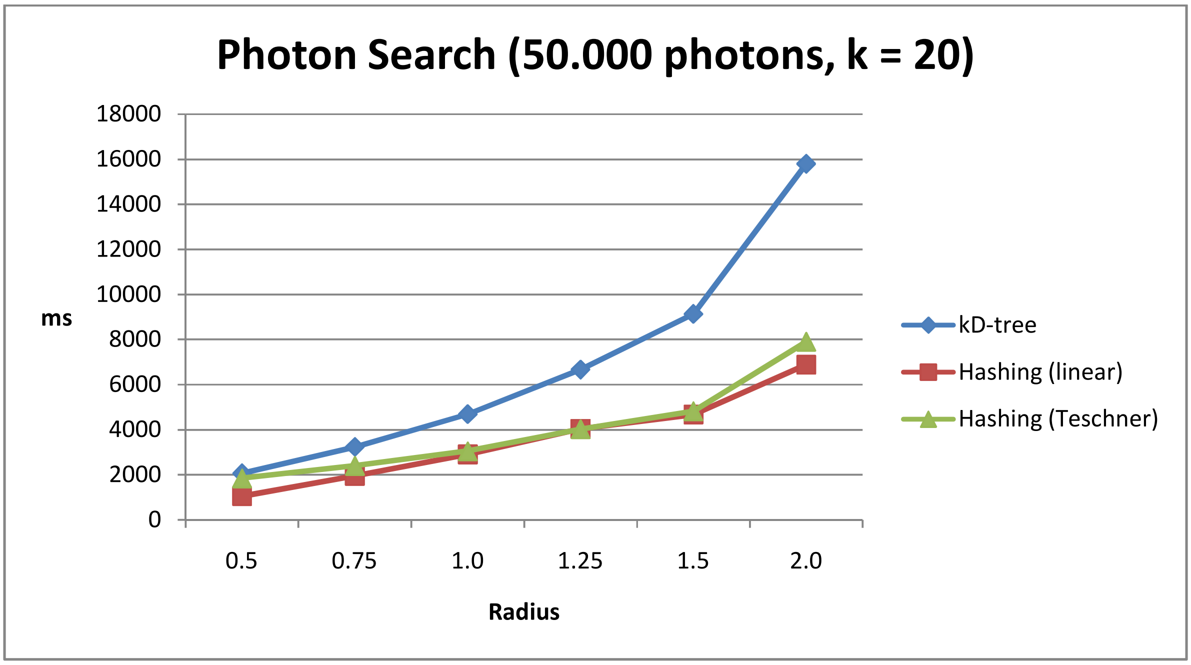


Figure 23: Caustic photon map search performance with different query radii

A similar behavior can be observed when we run our tests with a caustic photon map. In these tests it becomes clear that choosing the right query radius is very important for the kD-Tree. As Zhou et al write in their paper, a good estimation of r_k is critical to the performance of their technique. Figure 23 confirms that, showing a steeper increase of search time with an increasing search radius. Another disadvantage is the high photon density in caustic photon maps that prevents us from rejecting tree branches earlier, when using large query radii.

Our spatial hashing technique is dealing better with the high photon density in a caustic photon map. In this particular situation, the sorted list in combination with a little optimization to the neighbor offset lookup table, works very well. Obviously, we can assume that most of the photons we are interested in, are located in the same cell as our query point. Therefore, all we have to do is to swap the center cell's offset (0/0/0) to the first position in the lookup table. This ensures that each thread first checks all photons in the cell that contains the current query point, before it continues with the other neighbors. Our experiments showed that this little optimization improves search time by around 15 percent, compared to the non-optimized lookup table. Of course we still have to check all neighbor cells as well, in case the sample point is close to one of the cell corners, for instance. However, especially in the high density caustic photon maps lots of photons will be already found in the center cell, allowing us to reject other photons and avoiding list reordering. As Figure 23 shows, this holds true for the spatial hashing technique generally, independent from the used hashing function.

One thing we also have to keep in mind is that memory usage, with the linear hashing function, depends on the grid cells size and therefore, on the query radius. A smaller radius means that we have smaller cells and therefore, we need a larger grid with more cells to cover the whole scene. In Figure 24 we see the impact of the search radius on the memory consumption of our method. Especially with small photon maps, there is a noticeable difference in memory consumption. Using the smallest radius 0.5, we need more than 4 times the memory, required with a radius of 1.5 or 2. As the photon maps grow in size, the significance of the hash table overhead becomes less important because more memory is used to store the photon information itself. At around 50,000 photons, the memory requirement is roughly the same as for the kD-Tree technique. Another factor, which has an impact on our memory consumption, is the scene size. If we have a large scene we need a bigger grid to cover every part of it. However, in such situations, a different hashing function like the one presented by Teschner [53] is a much better choice, to avoid exhaustive memory usage.

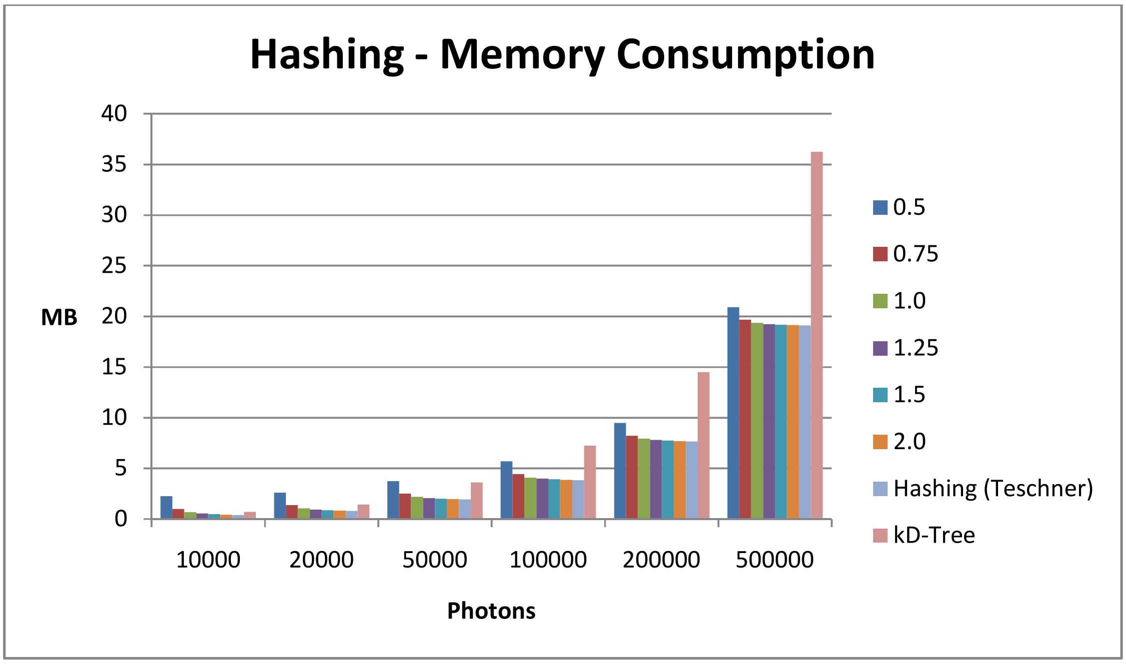


Figure 24: Hashing Memory Consumption with different query radii

Nevertheless, the increased memory consumption with smaller radii should not be a big problem, because the radius is usually in inverse proportion to the photon map size. If we have a small photon map, we want to use a large radius to find enough photons whereas if we have a large photon map, we can use a smaller radius to find the same amount. As you can see, this perfectly fits our technique to ensure an optimum memory usage.

5.3.4 Query Size

In our last series of performance tests, we evaluated how well both techniques can deal with different query sizes. The higher the resolution of our output image is going to be, the more points we have to sample in our scene. Figure 25 shows that, for low resolutions, there is hardly any difference between the different techniques. Both hashing functions obtain almost identical results, indicating that the chosen hashing function has no impact on these tests. However, the higher the resolution gets, the bigger grows the gap between the kD-Tree and both spatial hashing methods.

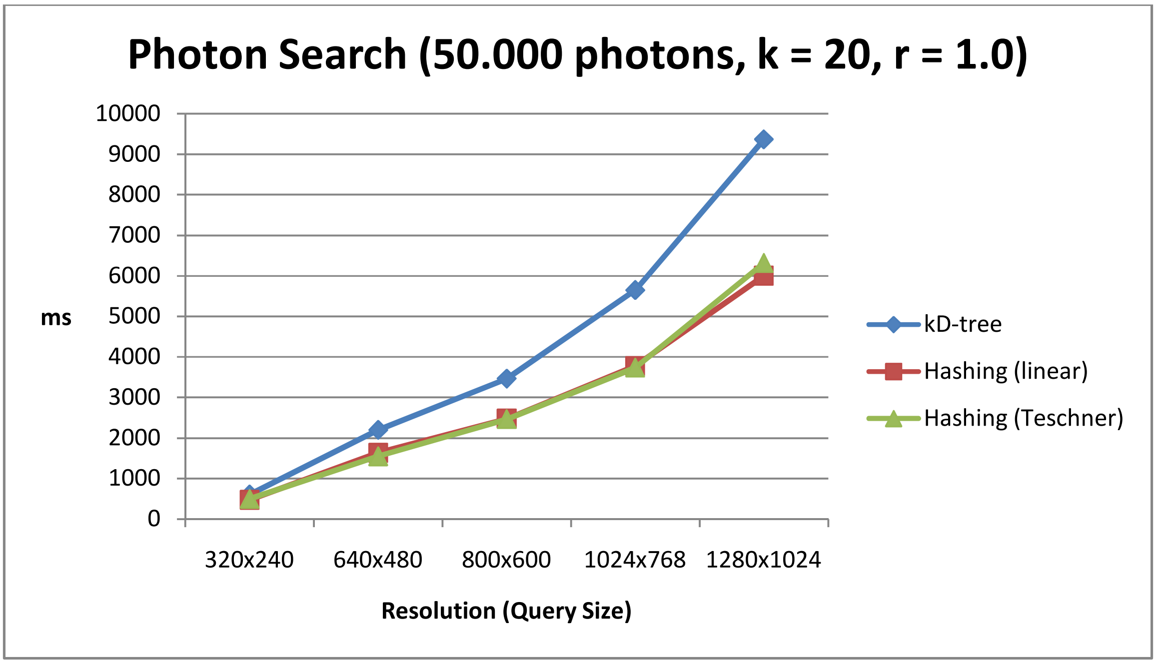


Figure 25: Global photon map performance with different query size

The reason for this difference is the number of non-coalesced memory accesses, performed by the two techniques. As Figure 26 shows, the kD-Tree technique suffers from a far higher increase in non-coalesced memory accesses than our spatial hashing approach. This graph makes the disadvantage, of having a tree-based structure that requires lots of scattered reads, obvious. The reason, why the overall performance of our technique is not as significant as the difference between the non-coalesced memory accesses is our photon collection implementation. Again the sorted list based approach prevents us from obtaining a better result in our tests.

Using a caustic photon map, we obtained a very similar result as with the global photon map. However, the gap between the two techniques is bigger than in the previous tests, with the spatial hashing methods being more than 75% faster at a resolution of 1280 x 1024 pixels. Like in previous tests, the higher photon density forces us to visit more tree nodes in the kD-Tree, further increasing the number of non-coalesced memory accesses. Overall, the random access pattern again, causes a bigger performance hit than the sorted list in our spatial hashing implementation.

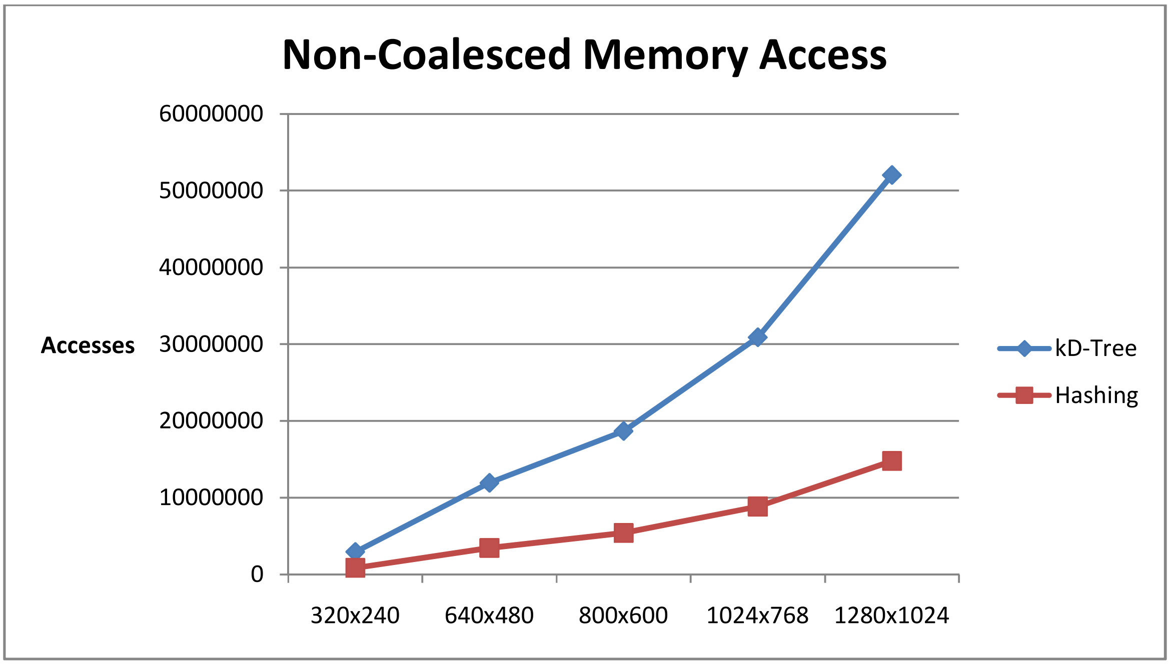


Figure 26: Non-Coalesced Memory Access

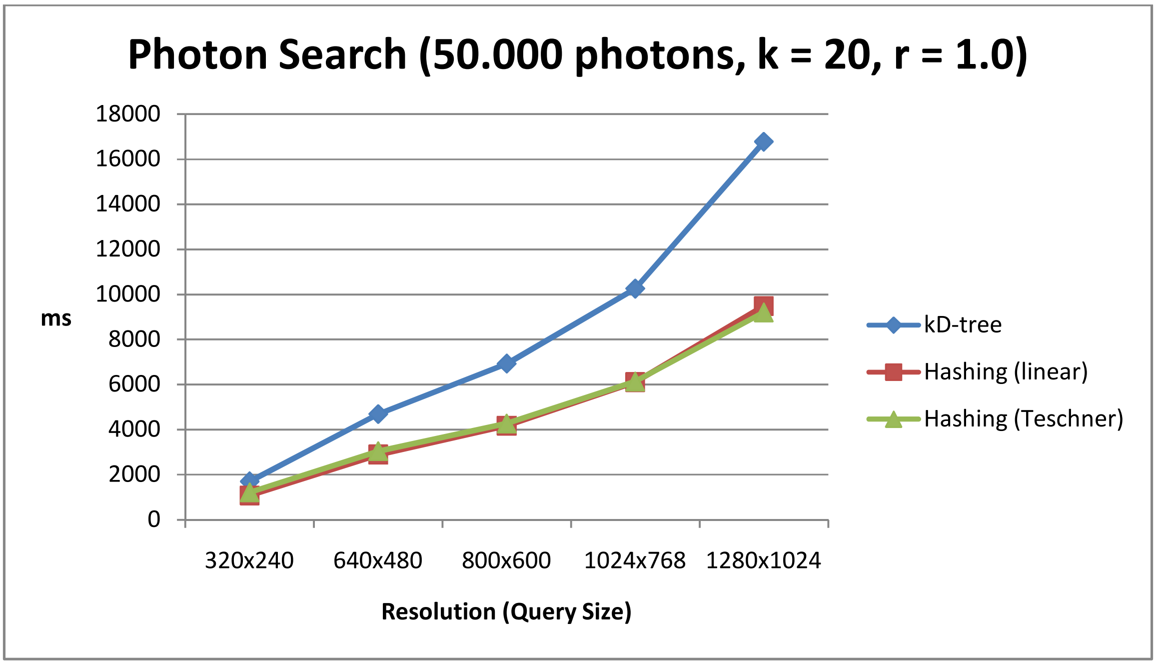


Figure 27: Caustic photon map performance with different query size

5.4 Results with the Brute Force Approach

As already mentioned earlier, the brute force approach was far behind the other two techniques in our performance comparison. However, we still want to give a short overview of how this technique performed in the different test cases. In Figure 28, we summarized the performance results for the radius, photon sample size and query size tests. We did not include any measures from the variable photon map tests, because the technique only supports a maximum photon map size of 2^{16}. There is also no need to distinguish between global and caustic photon maps, because the brute force approach is independent of the photons' distribution or density.

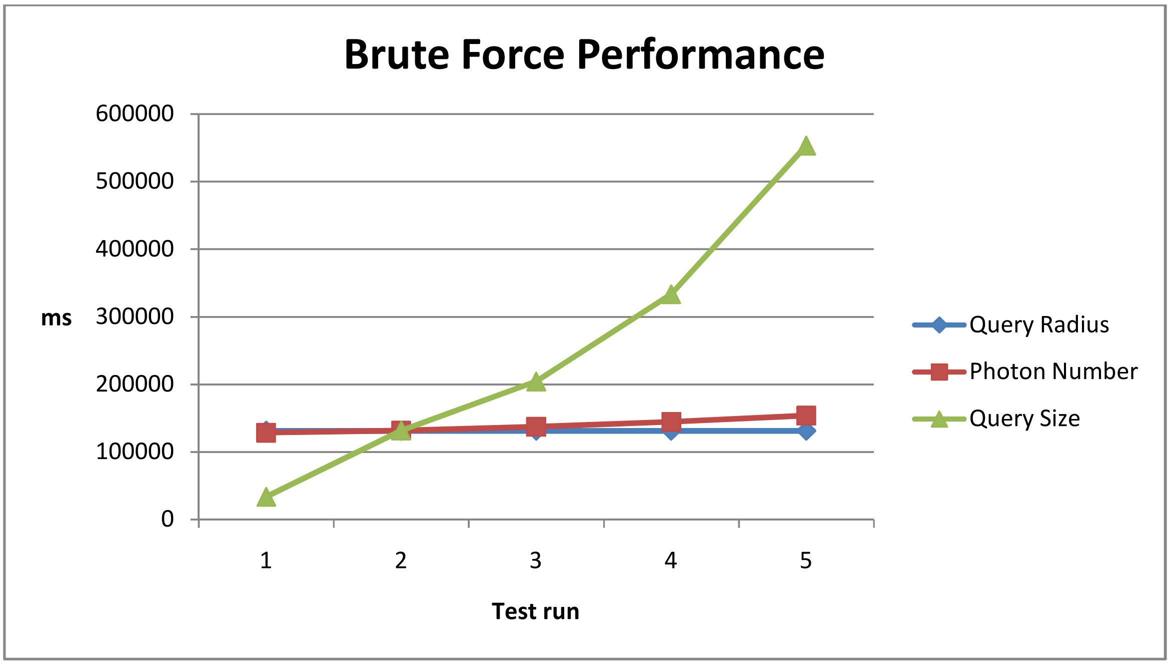


Figure 28: Brute Force Performance

As we see in the performance graph, the technique is completely independent from the query radius. In the original version of the algorithm there was no support for a maximum query radius included. Therefore, we extended the insertion sort algorithm to stop after sorting the first k elements or when the last, sorted photon's distance is outside the radius. However, the results showed that the algorithm was not able to benefit from this adaption.

The second test with variable photon sample sizes shows a slight increase in search time. Again we started with 10 samples in test 1 and increased the size by 10 photons, up to 50 for test 5.

According to Garcia et al, the sorting algorithm is the bottle neck of their technique. Table 3 shows the partitioning of the computation time, between the various steps during search. It should be noted that the data in the table is for a rather small reference point and query point set of just 4800 each.

Table 3: Computation time decomposition for the Brute Force Technique [25]

k	**5**	**10**	**20**
Distance	82%	71%	51%
Sort	15%	26%	47%
Memory Copying	3%	3%	2%

In our case, where we have thousands of photons in the photon map (reference points) and even more query points, the distance computation clearly dominates. However between the photon sample test cases with 10 and 50 photons per sample, computation time still rises by roughly 20% due to the increased sorting effort.

Finally, we also performed tests with different photon query sizes, using the brute force technique. As already mentioned in chapter 2, we had to split the photon query into multiple chunks because we can only search for 2^{16} query points at one time. Of course this introduces a slight overhead, caused by copying memory to and from the device. However, this is not as significant as the tremendous increase in distance computation, time that causes the slow down we observed during our tests.

5.5 Image Quality

In this section we take a look at the quality of the returned photon information. If a technique misses or returns false photons for a query, it will have a negative impact on the image quality. We already showed that our spatial hashing technique is pretty fast and now we take a look at the quality of the returned photon information. In order to evaluate quality, we will directly visualize the photon maps and compare them, using the method from [55]. This method can be used to determine if images are perceptually identical, even though they might contain some numerical differences.

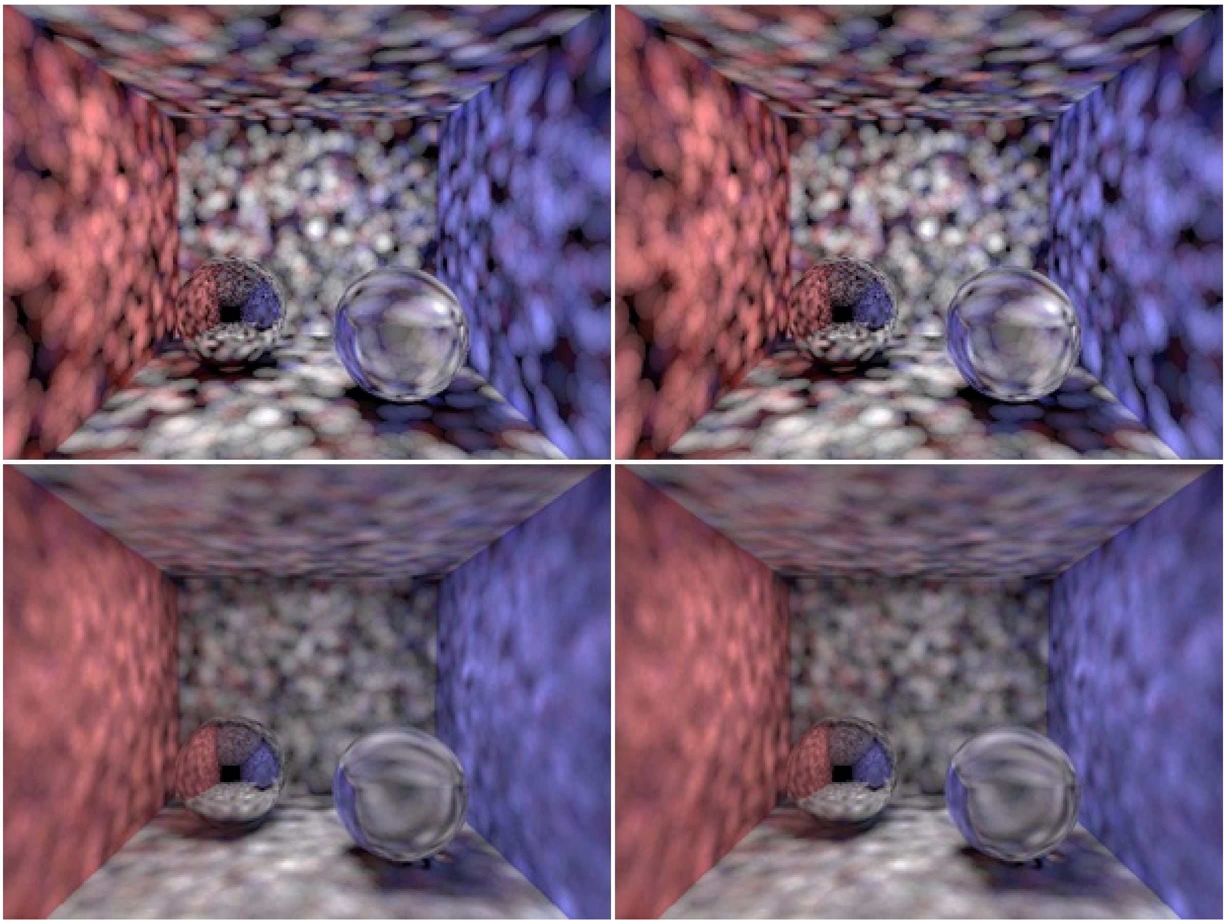


Figure 29: Image quality with hashing (left) and kD-Tree (right) with 10,000 (top) and 50,000 (bottom) photons

In Figure 29 we see four images created with the spatial hashing and the kD-Tree technique, using different global photon map sizes. The upper images were created with a global photon map, containing 10,000 photons whereas the bottom images used 50,000 photons. Looking at the results, we can say that both techniques are identical in terms of quality and visual appearance. Checking the images with the aforementioned image comparison algorithm confirms our observations. The same holds true for the two images, showing the results for a caustic map, containing 10,000 photons in Figure 30. Again, the comparison algorithm confirmed that both images are visually equal, in terms of quality.

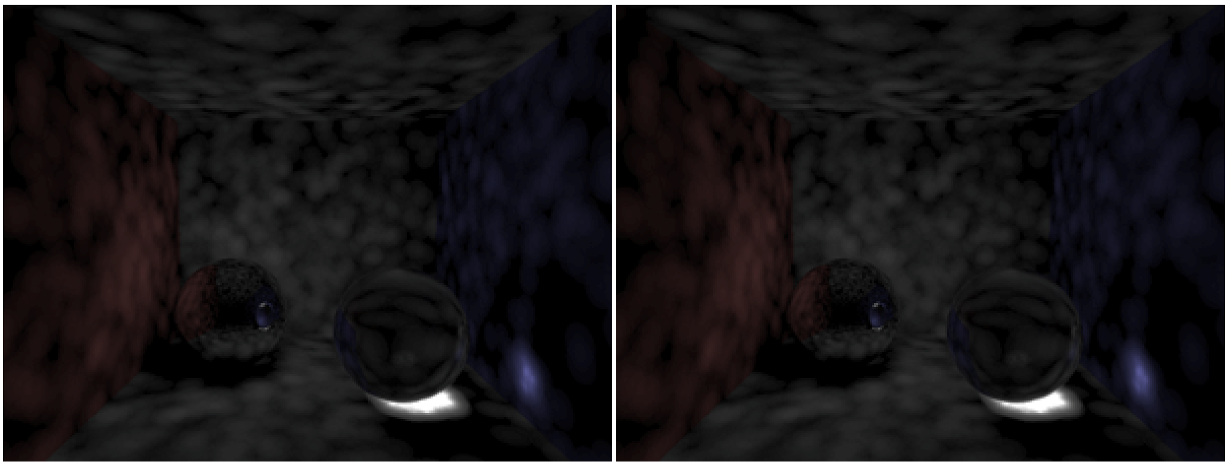


Figure 30: Image quality with hashing (left) and kD-Tree (right) with a caustic photon map (10,000 photons)

5.6 Observations

During our tests, we made a couple of observations that we want to summarize in this section. We start with the brute force approach, which proofed to be not very useful for photon mapping. The technique has a time complexity of *O(mnd)*, where *m* is the number of photons in the photon map, *n* is the number of query points and *d* is the dimension (in our case 3). This clearly shows that the brute force technique is scaling pretty bad with large photon maps or query sizes. Unfortunately, especially with photon mapping, these numbers tend to be relatively big. Also the size limitation to 2^{16} photons or query points is very constraining. The strengths of the brute force technique seem to be smaller point sets with higher dimensions, as well as its straightforward implementation.

Another observation was, that the brute force and the kD-Tree technique have problems on low performance GPUs. Compared to the results that were presented in both papers, we obtained much worse results on our system. It seems that both techniques rely heavily on a greater number of cores, a higher clock rate and higher memory bandwidth. Unfortunately, we have not had the possibility to perform benchmark tests of our spatial hashing technique on high performance GPU devices. However, we still proofed that on low-class consumer hardware, our approach is able to outperform both other techniques.

The kD-Tree was sometimes a very close competitor to our spatial hashing implementation. However, we think that besides performance there is another big advantage, our approach has over the kD-Tree, which is implementation complexity. The kD-Tree construction consists of 3 different stages, where each stage consists of multiple kernel functions. Even though, Zhou et al claim that most of the parallel primitive functions, like scan and sort, are implemented in CUDPP [56] others like the splitting kernel [40] or compaction kernel [43] are not and have to be implemented from scratch. In contrast, our spatial hashing implementation requires three very simple kernels for construction and only uses the radix sort functionality from the CUDPP library. Search is just a simple iteration over the photons within the different cells, whereas the kD-Tree requires the CPU and multiple kernels to estimate the query radius and traverse the kD-Tree.

Our experiments also showed that the chosen hashing function is very important for our hashing technique, in terms of memory consumption and search speed. With smaller scenes, the linear hashing function proofed to be the best choice in almost all test cases. However, as already explained, this function puts some constraints on the supported scene types. With the hashing function presented by Teschner et al, we are able to overcome these problems, at the cost of a sometimes longer photon search time. Our tests also showed that it is important to choose the right hash table size with Teschner's function. Especially with larger photon maps, a too small table can result in many hash collision, leading to a big performance hit. Nevertheless, performance with this hash function will certainly drop because hash collisions are inevitable, which results in cells containing more photons.

Construction time with our spatial hashing technique is almost 0, even on our low performance GPU. This is definitely one of the strengths of our technique. Another advantage is that we are able to use texture memory for our photon data and utilize the texture cache to further improve performance. The kD-Tree cannot profit from using texture memory because the memory reads are scattered and therefore, the texture cache will not improve performance. What we also observed during our experiments is that non-coalesced or local memory access literally kills performance, especially on low range GPU hardware. This is one of the reasons why the kD-Tree did not perform as well as our spatial hashing technique.

One issue we found with our spatial hashing implementation was the photon collection, using the sorted list approach. We think that there are a couple of possible solutions for this problem. The first one is to use a different data structure for the photon list, like a max heap [52]. We think it is possible to implement this data structure efficiently in shared memory. Another solution is to further parallelize the search and process each cell in a different thread, where each thread will search for the *k* closest photons in its cell, for a given query point. The problem with this solution is that we somehow have to collect the final result set from these sub results. We implemented a naïve approach that simply uses an insertion sort to find the closest *k* photons at the end. However, this approach was slower than our sorted list implementation because we were running the insertion sort in just one thread. In order to obtain a better performance, the sorting or collecting has to be parallelized as well. Finally, there is an interesting paper by Deo et al [57] that presents a parallel implementation of a priority queue. This priority queue could be created after each thread finished processing its cell, making it quite easy to return the closest *k* photons.

6

Conclusion

“After you finish the first 90% of a project, you have to finish the other 90%.”

- Michael Abrash, Programmer at Rad Game Tools [58]

Objectives

- To give an overall summary of this project
- To give a overview of the project’s outcomes
- To highlight future developments or extensions to the project

6.1 Conclusion

We started our project with the aim of developing an improved Photon Mapping technique that is tailored for parallel architectures, specifically GPUs. We began by looking at existing papers, published for Photon Mapping and *k* nearest neighbor search on the GPU and how they exploited parallelism. In our opinion, both methods had a few deficiencies in how they parallelized work, organized photon data or in their algorithm complexity. After analyzing these weaknesses, we came up with a new approach to parallelize Photon Mapping, where we try to avoid all the previously discovered shortcomings. Our new method is based on spatial hashing and grouping photons together in cells, according to their location in space. To evaluate our concept we implemented prototypes for the brute force, the kD-Tree and our spatial hashing technique and carried out a series of benchmarks.

In our tests we observed that our new approach was faster during construction and almost all the search tests, than the other two techniques. Especially construction time was greatly reduced, compared to the kD-Tree solution. We also proofed that our spatial hashing approach works well with both, global and caustic photon map types. Some problems occurred with larger photon sample sizes of up to 40 or 50 photons per query point. Along with speed measurements, we also examined the memory consumption of the various methods. Our approach consumed the least memory but, as we noted, this strongly depends on the size of the rendered scene, the maximum photon search radius and the used hashing function. Finally, we validated the quality of the returned photon samples, by directly visualizing the photon maps and comparing the resulting images. The quality of the resulting images was identical and finally proofed the usefulness of our technique.

We ended the results chapter with a summary of our observations, identifying two issues with our solution that we experienced during the tests. Memory consumption and the photon collection implementation were found to be the culprits of our technique. Finally, we presented a couple of possible solutions to these shortcomings. Nevertheless, there are a couple of areas where we think our technique might be very useful, which we will describe in the following section.

6.2 Future Work

We think there are a couple of interesting topics, upon future work can be based on. The first one is to use our spatial hashing approach for rendering participating media, using Photon Mapping [59]. Examples for participating media are smoke, dust or clouds, which all affect light, when it travels through them. When a light beam enters participating media it is either absorbed or scattered, which means the light beam’s direction changes. Because our hashing function is based on a grid of cells, building a volume, we think it is a perfect structure for storing photon information of volumetric participating media, like in the aforementioned examples.

Another interesting approach is to combine our spatial hashing method with the progressive Photon Mapping technique in [60]. As we were able to see in our experiments, our method scales pretty well with small photon map sizes and construction overhead is very low. Both these factors should favor our technique for the progressive radiance estimate, used in the paper.

Like in the paper by Garcia et al [25], we think it is also interesting to see how well our approach scales with higher dimension points. A couple of adjustments need to be done to the algorithm in order to support variable dimensions, specifically the grid and hash calculation as well as the neighbor offset lookup table.

Future developments in graphics hardware will also have an important impact on Photon Mapping techniques. With their newest range of GeForce graphics cards, nVidia relaxed the requirements for coalesced memory access. Maybe future generates will further relax these requirements or provide caches for local and global memory access. Like the kD-Tree, our technique could greatly profit from such developments.

Finally, there is also Intel’s new GPU, called Larrabee [61], which is going to be released next year, offering a many-core x86 architecture for visual computing. Our spatial hashing algorithm should be well suited for this architecture because of its high parallelization. The sequential photon access pattern in each cell should also guarantee an optimal use of Larrabee’s first and second level data caches.

Bibliography

[1] G. Anthes. (2007, November) ComputerWorldUK. [Online]. http://www.computerworlduk.com/technology/hardware/processors/in-depth/index.cfm?articleid=957

[2] nVidia Corporation, "NVIDIA CUDA Programming Guide Version 2.2," NVIDIA, 2009.

[3] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Comm. of the ACM 18*, pp. 509-517, 1975.

[4] T. Davis, A. Chalmers, and H. W. Jensen, "Practical Parallel Processing for Realistic Rendering," in *SIGGRAPH 2000 Course Notes*, New Orleans, 2000.

[5] C. James. (2008, May) V3. [Online]. http://www.v3.co.uk/vnunet/news/2215645/nvidia-touts-parallel-computing

[6] A. Appel, "Some techniques for shading machine renderings of solids," in *AFIPS Joint Computer Conferences*, Atlantic City, New Jersey, 1968, pp. 37-45.

[7] T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, vol. 23, no. 6, pp. 343-349, June 1980.

[8] J. Atwood. (2008, March) Real-Time Raytracing. [Online]. http://www.codinghorror.com/blog/images/ray-tracing-diagram.png

[9] R. Descartes, *Discourse on Method, Optics, Geometry, and Meteorology*. Indianapolis: Hackett Publishing Co., 2001.

[10] J. R. Wallace, M. F. Cohen, and D. P. Greenberg, "A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods," *Computer Graphics 21*, pp. 311-320, 1987.

[11] H. E. Rushmeier and K. E. Torrance, "Extending the Radiosity Methods to Include

Specularly Reflecting and Translucent Materials," *ACM Transaction on Graphics 9*, pp. 1-27, 1990.

[12] F. X. Sillion, J. R. Arvo, S. H. Westin, and D. P. Greenberg, "A Global Illumination Solution for General Reflectance Distributions," *Computer Graphics 25*, pp. 187-196, 1991.

[13] H. W. Jensen and N. J. Christensen, "Photon Maps in Bidirectional Monte Carlo Ray Tracing of Complex Objects," *Computers & Graphics vol. 19*, pp. 215-224, 1995.

[14] H. W. Jensen, "Global Illumination using Photon Maps," *Rendering Techniques '96*, pp. 21-30, 1996.

[15] H. W. Jensen, "Global illumination via bidirektional Monte Carlo ray tracing," Technical University of Denmark, Master thesis 1993.

[16] J. Arvo and D. B. Kirk, "Particle Transport and Image Synthesis," *Computer Graphics (Proc. SIGGRAPH '90)*, vol. 24, no. 4, pp. 63-66, August 1990.

[17] Z. Waters. (2009) Photon Mapping. [Online]. http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/photon_mapping/PhotonMapping.html

[18] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray Tracing on Programmable Graphics Hardware," *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 703-712, 2002.

[19] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-D Tree GPU Raytracing," *Proceedings of Symposium on Interactive 3D graphics and Games*, pp. 167-174, 2007.

[20] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, "Stackless KD-Tree Traversal for High Performance GPU Ray Tracing," *Computer Graphics Forum*, vol. 26, no. 3, pp. 415-424, September 2007.

[21] D. Luebke and S. Parker, "Interactive Ray Tracing with CUDA," in *NVISION08*, 2008.

[22] J. Günther, I. Wald, and P. Slusallek, "Realtime Caustics Using Distributed Photon Mapping," *Proceedings of the 15th Eurographics Symposium on Rendering*, pp. 111-121, June 2004.

[23] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon Mapping on Programmable Graphics Hardware," in *International Conference on Computer Graphics and Interactive Techniques*, Los Angeles, 2005.

[24] Y. Lifshits. (2008) The Homepage of Nearest Neighbors and Similarity Search. [Online]. http://simsearch.yury.name/

[25] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k Nearest Neighbor Search using GPU," in *CVPR Workshop on Computer Vision on GPU*, Anchorage, 2008.

[26] C. A. R. Hoare, "Quicksort: Algorithm 64," *Communications of the ACM*, vol. 4, no. 7, pp. 321-322, 1961.

[27] W. Dobosiewicz, "An efficient variation of bubble sort," *Information Processing Letters*, vol. 11, pp. 5-6, 1980.

[28] D. Knuth, "Sorting by Insertion," in *The Art of Computer Programming, Volume 3: Sorting and Searching*.: Addison-Wesley, 1998, ch. 5.2.1, pp. 80-105.

[29] D. M. Mount and S. Arya. ANN: A library for approximate nearest neighbor searching. [Online]. http://www.cs.umd.edu/˜mount/ANN/

[30] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *Journal of the ACM*, vol. 45, no. 6, pp. 891-923, 1998.

[31] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, "Experiences with Streaming Construction of SAH KD-Trees," in *Proceedings of the 2006 IEEE Symposium on Interactive*, 2006, pp. 89-94.

[32] M. Shevtsov, A. Soupikov, and A. Kapustin, "Highly Parallel Fast KD-tree Construction

for Interactive Ray Tracing of Dynamic Scenes," *Computer Graphics Forum*, vol. 26, no. 3, pp. 395-404, October 2007.

[33] J. Goldsmith and J. Salmon, "Automatic creation of object hierarchies for ray tracing," *IEEE Computer Graphics and Applications*, vol. 7, no. 5, pp. 14-20, 1987.

[34] J. D. MacDonald and K. S. Booth, "Heuristics for ray tracing using space subdivision," *The Visual Computer*, vol. 6, no. 3, pp. 153-166, 1990.

[35] I. Wald, "Realtime Ray Tracing and Interactive Global Illumination," Saarland University, PhD Thesis 2004.

[36] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time KD-tree construction on graphics hardware," *ACM Transactions on Graphics*, vol. 27, no. 5, pp. 1-11, December 2008.

[37] N. Satish, M. Harris, and M. Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs," *Proc. 23rd IEEE Int'l Parallel & Distributed Processing Symposium*, May 2009.

[38] V. Havran, "Heuristic Ray Shooting Algorithms," Czech Technical University, PhD Thesis 2000.

[39] M. Harris, S. Sengupta, and J. D. Owens, "Parallel Prefix Sum (Scan) with CUDA," in *GPU Gems 3*.: Addison Wesley, 2007.

[40] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan Primitives for GPU Computing," in *Graphics Hardware 2007*, San Diego, 2007, pp. 97-106.

[41] I. Wald, J. Günther, and P. Slusallek, "Balancing Considered Harmful - Faster Photon Mapping using the Voxel Volume Heuristic," *Computer Graphics Forum*, vol. 22, no. 3, 2004.

[42] G. S. Manku. (2008, August) Fast Bit Counting. [Online]. http://gurmeetsingh.wordpress.com/2008/08/05/fast-bit-counting-routines/

[43] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH

Construction on GPUs ," *Computer Graphics Forum*, vol. 28, no. 2, pp. 375-384, 2009.

[44] H. W. Jensen, *Realistic Image Synthesis Using Photon Mapping*.: A K Peters, 2001.

[45] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*.: Springer-Verlag New York, 1985.

[46] Edge Staff. (2005, August) Edge. [Online]. http://www.edge-online.com/features/john-carmack

[47] V. Garcia. (2008, June) CUDA Zone Forum. [Online]. http://forums.nvidia.com/index.php?s=&showtopic=69307&view=findpost&p=391299

[48] D. T. Lee and C. K. Wong, "Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees," *Acta Informatica*, vol. 9, no. 1, pp. 23-29, 1977.

[49] S. Green, "Particle-based Fluid Simulation," in *Game Developers Conference*, 2008.

[50] S. Le Grand, "Broad-Phase Collision Detection with CUDA," in *GPU Gems 3*.: Addison Wesley, 2007, pp. 697-721.

[51] G. M. Morton, "A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing," IBM Ltd., Ottawa, Technical Report 1966.

[52] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*.: MIT Press / McGraw-Hill, 1990.

[53] M. Teschner, B. Heidelberger, M. Müller, D. Pomeranets, and M. Gross, "Optimized Spatial Hashing for Collision Detection of Deformable Objects," *Proceedings of VMV'03 Munich*, pp. 47-54, 2003.

[54] R. Pattis. (2002) Framework Methodology. [Online]. http://www.phred.org/~brian/factoring/chapter2.html

[55] Y. H. Yee and A. Newman, "A perceptual metric for production testing," in *International*

Conference on Computer Graphics and Interactive Techniques, Los Angeles, 2004, p. 121.

[56] M. Harris et al. (2009, August) CUDPP homepage. [Online]. http://code.google.com/p/cudpp/

[57] N. Deo and S. Prasad, "Parallel heap: An optimal parallel priority queue," *The Journal of Supercomputing*, vol. 6, no. 1, pp. 87-98, November 2004.

[58] M. Abrash. (2000) Ramblings in Realtime. [Online]. http://www.gamedev.net/reference/articles/abrash/abrash.pdf

[59] H. W. Jensen and P. H. Christensen, "Efficient Simulation of Light Transport in Scenes with Participating Media using Photon Maps," in *In Proceedings of SIGGRAPH'98*, Orlando, 1998, pp. 311-320.

[60] T. Hachisuka, S. Ogaki, and H. W. Jensen, "Progressive Photon Mapping," in *SIGGRAPH Asia 2008*, Singapore, 2008.

[61] L. Seiler et al., "Larrabee: a many-core x86 architecture for visual computing," *Proceedings of ACM SIGGRAPH 2008*, vol. 27, no. 3, August 2008.