Graphics Hardware (2002)
Thomas Ertl, Wolfgang Heidrich, and Michael Doggett (Editors)

Low Latency Photon Mapping Using Block Hashing

Vincent C. H. Ma and Michael D. McCool

Computer Graphics Lab, School of Computer Science,
University of Waterloo, Waterloo, Ontario, Canada

Abstract

For hardware accelerated rendering, photon mapping is especially useful for simulating caustic lighting effects on non-Lambertian surfaces. However, an efficient hardware algorithm for the computation of the k nearest neighbours to a sample point is required.

Existing algorithms are often based on recursive spatial subdivision techniques, such as kd-trees. However, hardware implementation of a tree-based algorithm would have a high latency, or would require a large cache to avoid this latency on average.

We present a neighbourhood-preserving hashing algorithm that is low-latency and has sub-linear access time. This algorithm is more amenable to fine-scale parallelism than tree-based recursive spatial subdivision, and maps well onto coherent block-oriented pipelined memory access. These properties make the algorithm suitable for implementation using future programmable fragment shaders with only one stage of dependent texturing.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism, Color, Shading, Shadowing, and Texture.

1. Introduction

Photon mapping, as described by Jensen[25], is a technique for reconstructing the incoming light field at surfaces everywhere in a scene from sparse samples generated by forward light path tracing. In conjunction with path tracing, photon mapping can be used to accelerate the computation of both diffuse and specular global illumination . It is most effective for specular or glossy reflectance effects, such as caustics[24].

The benefits of migrating photo-realistic rendering techniques towards a real-time, hardware-assisted implementation are obvious. Recent work has shown that it is possible to implement complex algorithms, such as ray-tracing, using the programmable features of general-purpose hardware accelerators[35] and/or specialised hardware [41]. We are interested in hardware support for photon mapping: specifically, the application of photon maps to the direct visualisation of caustics on non-Lambertian surfaces, since diffuse global illumination effects are probably best handled in a real-time renderer using alternative techniques such as irradiance volumes[18].

Central to photon mapping is the search for the set of photons nearest to the point being shaded. This is part of the interpolation step that joins light paths propagated from light sources with rays traced from the camera during rendering, and it is but one application of the well-studied k nearest neighbours (kNN) problem.

Jensen uses the kd-tree[5, 6] data structure to find these nearest photons. However, solving the kNN problem via kd-trees requires a search that traverses the tree. Even if the tree is stored as a heap, traversal still requires random-order memory access and memory to store a stack. More importantly, a search-path pruning algorithm, based on the data already examined, is required to avoid accessing all data in the tree. This introduces serial dependencies between one memory lookup and the next. Consequently, a hardware implementation of a kd-tree-based kNN solution would either have high latency, or would require a large cache to avoid such latency. In either case a custom hardware implementation would be required. These properties motivated us to look at alternatives to tree search.

Since photon mapping is already making an approximation by using kNN interpolation, we conjectured that an approximate kNN (AkNN) solution should suffice so long as visual quality is maintained. In this paper we investigate a hashing-based AkNN solution in the context of high-performance hardware-based (specifically, programmable shader-based) photon mapping. Our major contribution is an A*k*NN algorithm that has bounded query time, bounded memory usage, and high potential for fine-scale parallelism. Moreover, our algorithm results in coherent, non-redundant accesses to block-oriented memory. The results of one memory lookup do not affect subsequent memory lookups, so accesses can take place in parallel within a pipelined memory system. Our algorithm is based on array access, and is more compatible with current texture-mapping capabilities than tree-based algorithms. Furthermore, any photon mapping acceleration technique that continues to rely on a form of *k*NN (such as irradiance caching[7]) can still benefit from our technique.

In Section 2, we first review previous work on the *k*NN and the approximate *k*-nearest neighbour (A*k*NN) problems. Section 3 describes the context and assumptions of our research and illustrates the basic hashing technique used in our algorithm. Sections 4 and 5 describe the details of our algorithm. Section 6 presents numerical, visual quality, and performance results. Section 7 discusses the mapping of the algorithm onto a shader-based implementation. Finally, we conclude in Section 8.

2. Previous Work

Jensen's book[25] covers essentially all relevant previous work leading up to photon mapping. Due to space limitations, we will refer the reader to that book and focus our literature review on previous approaches to the *k*NN and A*k*NN problems.

Any non-trivial algorithm that claims to be able to solve the *k*NN problem faster than brute-force does so by reducing the number of candidates that have to be examined when computing the solution set. Algorithms fall into the following categories: *recursive spatial subdivision*, *point location*, *neighbourhood graphs*, and *hashing*.

Amongst algorithms based on recursive spatial subdivision, the *kd*-tree [5] method is the approach commonly used to solve the *k*NN problem[14]. An advantage of the *kd*-tree is that if the tree is balanced it can be stored as a heap in a single array. While it has been shown that *kd*-trees have optimal expected-time complexity[6], in the worst case finding the *k* nearest neighbours may require an exhaustive search of the entire data structure via recursive decent. This requires a stack the same size as the depth of the tree. During the recursion, a choice is made of which subtree to search next based on a test at each internal node. This introduces a dependency between one memory access and the next and makes it hard to map this algorithm into high-latency pipelined memory accesses.

Much work has been done to find methods to optimise the *kd*-tree method of solving the *k*NN and A*k*NN problems. See Christensen[26], Vanco *et al.*[44], Havran[19], and Sample *et al.*[39]. Many other recursive subdivision-based techniques have also been proposed for the *k*NN and A*k*NN problems, including *kd*-B-trees[36], BBD-trees[4], BAR-trees[9], Principal-Axis Trees[33], the R-tree family of data structures[27], and ANN-trees[30]. Unfortunately, all schemes based on recursive search over a tree share the same memory dependency problem as the *kd*-tree.

The second category of techniques are based on building and searching graphs that encode sample-adjacency information. The randomised neighbourhood graph approach[3] builds and searches an approximate local neighbourhood graph. Eppstein *et al.*[11] investigated the fundamental properties of a nearest neighbour graph. Jaromczyk and Toussaint surveyed data structures and techniques based on Relative Neighbourhood Graphs[23]. Graph-based techniques tend to have the same difficulties as tree-based approaches: searching a graph also involves stacks or queues, dependent memory accesses, and pointer-chasing unsuited to high-latency pipelined memory access.

Voronoi diagrams can be used for optimal 1-nearest neighbour searches in 2D and 3D[10]. This and other point-location based techniques[17] for solving nearest neighbour problems do not need to calculate distances between the query point and the candidates, but do need another data structure (like a BSP tree) to test a query point for inclusion in a region.

Hashing approaches to the *k*NN and A*k*NN problems have recently been proposed by Indyk *et al.*[20, 21] and Gionis *et al.*[16]. These techniques have the useful property that multi-level dependent memory lookups are not required. The heart of these algorithms are simple hash functions that preserve spatial locality, such as the one proposed by Linial and Sasson[31], and Gionis *et al.*[16]. We base our technique on the latter. The authors also recognise recent work by Wald *et al.*[45] on real-time global illumination techniques where a hashing-based photon mapping technique was apparently used (but not described in detail).

Numerous surveys and books [1, 2, 15, 42, 43, 17] provide further information on this family of problems and data structures developed to solve them.

3. Context

We have developed a novel technique called *Block Hashing* (BH) to solve the approximate *k*NN (A*k*NN) problem in the context of, but not limited to, photon mapping.

Our algorithm uses hash functions to categorise photons by their positions. Then, a *k*NN query proceeds by deciding which hash bucket is matched to the query point and retrieving the photons contained inside the hash bucket for analysis. One attraction of the hashing approach is that evaluation of hash functions takes constant time. In addition, once we have the hash value, accessing data we want in the hash table takes only a single access. These advantages permit us to avoid operations that are serially dependent on one another, such as those required by *kd*-trees, and are major stepping stones towards a low-latency shader-based implementation.

Our technique is designed under two assumptions on the behaviour of memory systems in (future) accelerators. First, we assume that memory is allocated in fixed-sized blocks . Second, we assume that access to memory is via burst transfer of blocks that are then cached. Under this assumption, if any part of a fixed-sized memory block is "touched", access to the rest of this block will be virtually zero-cost. This is typical even of software implementations on modern machines which rely heavily on caching and burst-mode transfers from SDRAM or RDRAM. In a hardware implementation with a greater disparity between processing power and memory speed, using fast block transfers and caching is even more important. Due to these benefits, in BH all memory used to store photon data is broken into fixed-sized blocks.

3.1. Locality-Sensitive Hashing

Since our goal is to solve the *k*NN problem as efficiently as possible in a block-oriented cache-based context, our hashing technique requires hash functions that preserve spatial neighbourhoods. These hash functions take points that are close to each other in the domain space and hash them close to each other in hash space. By using such hash functions, photons within the same hash bucket as a query point can be assumed to be close to the query point in the original domain space. Consequently, these photons are good candidates for the *k*NN search. More than one such scheme is available; we chose to base our technique on the Locality-Sensitive Hashing (LSH) algorithm proposed by Gionis *et al.*[16], but have added several refinements (which we describe in Section 4).

The hash function in LSH groups one-dimensional real numbers in hash space by their spatial location. It does so by partitioning the domain space and assigning a unique hash value to each partition. Mathematically, let $\mathcal{T} = \{t_i \mid 0 \leq i \leq P\}$ be a monotonically increasing sequence of $P+1$ thresholds between 0 and 1. Assume $t_0 = 0$ and $t_P = 1$, so there are $P-1$ degrees of freedom in this sequence. Define a one-dimensional Locality-Sensitive Hash Function $h_{\mathcal{T}} : [0,1] \to \{0 \ldots P-1\}$ to be $h_{\mathcal{T}}(t) = i$, where $t_i \leq t < t_{i+1}$. In other words, the hash value i can take on P different values, one for each "bucket" defined by the threshold pair $[t_i, t_{i+1})$. An example is shown in Figure 1.

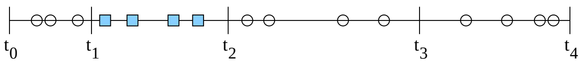


Figure 1: *An example of $h_{\mathcal{T}}$. The circles and boxes represent values to be hashed, while the vertical lines are the thresholds in $\mathcal{T}$. The boxes lie between t_1 and t_2, thus they are given a hash value of 1.*

The function $h_{\mathcal{T}}$ can be interpreted as a monotonic non-uniform quantisation of spatial position, and is characterised by P and the sequence $\mathcal{T}$. It is important to note that $h_{\mathcal{T}}$ gives each partition of the domain space delineated by $\mathcal{T}$ equal representation in hash space. Depending on the location of the thresholds, $h_{\mathcal{T}}$ will contract some parts of the domain space and expand other parts. If we rely on only a single hash table to classify a data set, a query point will only hash to a single bucket within this table, and the bucket may represent only a subset of the true neighbourhood we sought. Therefore, multiple hash tables with different thresholds are necessary for the retrieval of a more complete neighbourhood around the query point (See Figure 2.)

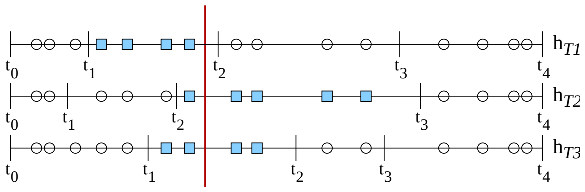


Figure 2: *An example of multiple hash functions classifying a dataset. The long vertical line represents the query value. The union of results multiple hash tables with different thresholds represents a more complete neighbourhood.*

To deal with *n*-dimensional points, each hash table will have one hash function per dimension. Each hash function generates one hash value per coordinate of the point (See Figure 3.) The final hash value is calculated by $\sum_{i=0}^{n-1} h_i P^i$, where h_i are the hash values and P is the number of thresholds. If P were a power of two, then this amounts to concatenating the bits. The only reason we use the same number of thresholds for each dimension is simplicity. It is conceivable that a different number of thresholds could be used for each dimension to better adapt to the data. We defer the discussion of threshold generation and query procedures until Sections 4.2 and 4.4, respectively.

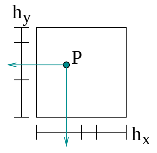


Figure 3: *Using two hash functions to handle a 2D point. Each hash function will be used to hash one coordinate.*

LSH is very similar to grid files[34]. However, the grid file was specifically designed to handle dynamic data. Here, we assume that the data is static during the rendering pass. Also, the grid file is more suitable for range searches than it is for solving the *k*NN problem.

3.2. Block-Oriented Memory Model

It has been our philosophy that hardware implementations of algorithms should treat off-chip memory the same way software implementations treat disk: as a relatively slow, "out-of-core", high-latency, block-oriented storage device. This analogy implies that algorithms and data structures designed to optimise for disk access are potentially applicable to hardware design. It also drove us to employ fixed-sized blocks to store the data involved in the *k*NN search algorithm, which are photons in the context of this application.

In our prototype software implementation of BH, each photon is stored in a structure similar to Jensen's "extended" photon representation[25]. As shown in Figure 4, each component of the 3D photon location is represented by a 32-bit fixed-point number. The unit vectors representing incoming direction $\hat{\mathbf{d}}$ and surface normal $\hat{\mathbf{n}}$ are quantised to 16-bit values using Jensen's polar representation. Photon power is stored in four channels using sixteen-bit floating point numbers. This medium-precision signed representation permits other A*k*NN applications beyond that of photon mapping. Four colour channels are also included to better match the four-vectors supported in fragment shaders. For the photon mapping application specifically, our technique is compatible with the replacement of the four colour channels with a Ward RGBE colour representation[46]. Likewise, another implementation might use a different representation for the normal and incident direction unit vectors.

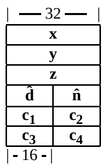


Figure 4: *Representation of a photon record. The 32-bit values* (x,y,z) *denote the position of a photon and are used as keys. Two quantised 16-bit unit vectors* $\hat{\mathbf{d}}$, $\hat{\mathbf{n}}$ *and four 16-bit floating point values are carried as data.*

All photon records are stored in fixed-sized memory blocks. BH uses a 64 32-bit-word block size, chosen to permit efficient burst-mode transfers over a wide bus to transaction-oriented DRAM. Using a 128-bit wide path to DDR SDRAM, for instance, transfer of this block would take eight cycles, not counting the overhead of command cycles to specify the operation and the address. Using next-generation QDR SDRAM this transfer would take only four cycles (or eight on a 64-bit bus, etc.)

Our photon representation occupies six 32-bit words. Since photon records are not permitted to span block boundaries, ten photons will fit into a 64-word block with four words left over. Some of this extra space is used in our implementation to record how many photons are actually stored in each block. For some variants of the data structures we describe, this extra space could also be used for flags or pointers to other blocks. It might be possible or desirable in other implementations to support more or fewer colour channels, or channels with greater or lesser precision, in which case some of our numerical results would change.

4. Block Hashing

Block Hashing (BH) contains a preprocessing phase and a query phase. The preprocessing phase consists of three steps. After photons have been traced into the scene, the algorithm organises the photons into fixed-sized memory blocks, creates a set of hash tables, and inserts photon blocks into the hash tables.

In the second phase, the hash tables will be queried for a set of candidate photons from which the *k* nearest photons will be selected for each point in space to be shaded by the renderer.

4.1. Organizing Photons into Blocks

Due to the coherence benefits associated with block-oriented memory access, BH starts by grouping photons and storing them into fixed-sized memory blocks. However, these benefits are maximised when the photons within a group are close together spatially.

We chose to use the Hilbert curve[13] to help group photons together. The advantage of the Hilbert curve encoding of position is that points mapped near each other on the Hilbert curve are guaranteed to be within a certain distance of each other in the original domain[22]. Points nearby in the original domain space have a high probability of being nearby on the curve, although there is a non-zero probability of them being far apart on the curve. If we sort photons by their Hilbert curve order before packing them into blocks, then the photons in each block will have a high probability of being spatially coherent. Each block then corresponds to an interval of the Hilbert curve, which in turn covers some compact region of the domain (see Figure 7a). Each region of domain space represented by the blocks is independent, and regions do not overlap.

BH sorts photons and inserts them into a B$^+$-tree[8] using the Hilbert curve encoding of the position of each photon as the key. This method of spatially grouping points was first proposed by Faloutsos and Rong[12] for a different purpose. Since a B$^+$-tree stores photon records only at leaves, with a compatible construction the leaf nodes of the B$^+$-tree can serve as the photon blocks used in the later stages of BH. One advantage of using a B$^+$-tree for sorting is that insertion cost is bounded: the tree is always balanced, and in the worst case we may have to split *h* nodes in the tree, when the height of the tree is *h*. Also, B$^+$-trees are optimised for block-oriented storage, as we have assumed.

The B$^+$-tree used by BH has index and leaf nodes that are between half full to completely full. To minimise the final number of blocks required to store the photons, the leaf nodes can be compacted (see Figure 5.) After the photons are sorted and compacted, the resulting photon blocks are ready to be used by BH, and the B$^+$-tree index and any leaf nodes that are made empty by compaction are discarded. If the complete set of photons is known *a priori*, the compact B$^+$-tree[37] for static data can be used instead. This data structure maintains full nodes and avoids the extra compaction step.

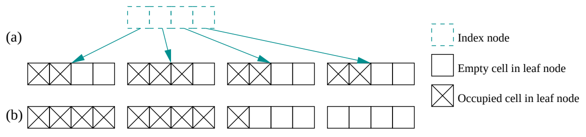


Figure 5: *Compaction of photon blocks. (a) B⁺-tree after inserting all photons. Many leaf nodes have empty cells. (b) All photon records are compacted in the leaf nodes.*

Regardless, observe that each photon block contains a spatially clustered set of photons disjoint from those contained in other blocks. This is the main result we are after; any other data structures that can group photons into spatially-coherent groups, such as grid files[34], can be used in place of the B⁺-tree and space-filling curve.

4.2. Creating the Hash Tables

The hash tables used in BH are based on the LSH scheme described in Section 3.1. BH generates L tables in total, serving as parallel and complementary indices to the photon data. Each table has three hash functions (since photons are classified by their 3D positions), and each hash function has $P+1$ thresholds.

BH employs an adaptive method that generates the thresholds based on the photon positions. For each dimension, a histogram of photon positions is built. Then, the histogram is integrated to obtain a cumulative distribution function (*cdf*). Lastly, stratified samples are taken from the inverse of the *cdf* to obtain threshold locations. The resulting thresholds will be far apart where there are few photons, and close together where photons are numerous. Ultimately this method attempts to have a similar number of photons into each bucket.

Hash tables are stored as a one-dimensional array structure, shown in Figure 6. The hash key selects a bucket out of the P^n available buckets in the hash table. Each bucket refers up to B blocks of photons, and has space for a validity flag per reference, and storage for a priority value. We defer the discussion on the choice of P, L and B until Section 5.

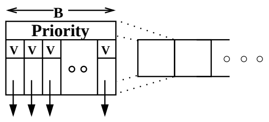


Figure 6: *Hash table bucket layout*

4.3. Inserting Photon Blocks

In BH, references to entire photon blocks, rather than individual photons, are inserted into the hash tables. One reason for doing so is to reduce the memory required per bucket. Another, more important, reason is that when merging results from multiple hash tables (Section 3.1), BH needs to compare only block addresses instead of photons when weeding out duplicates as each block contains a unique set of photons. This means fewer comparisons have to be made and the individual photons are only accessed once per query, during post-processing of the merged candidate set to find the k nearest photons. Consequently, the transfer of each photon block through the memory system happens at most once per query. All photons accessed in a block are potentially useful contributions to the candidate set, since photons within a single block are spatially coherent. Due to our memory model assumptions, once we have looked at one photon in a block it should be relatively inexpensive to look at the rest.

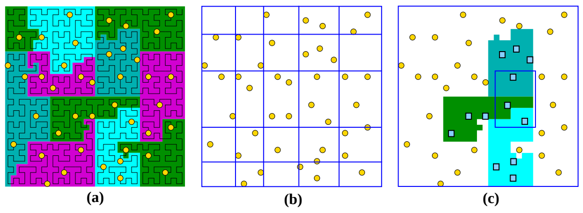


Figure 7: *Block hashing illustrated. (a) Each block corresponds to an interval of the Hilbert curve, which in turn covers some compact region of the domain. Consequently, each bucket (b) represents all photons (highlighted with squares) in each block with at least one photon hashed into it (c).*

Each bucket in a hash table corresponds to a rectangular region in a non-uniform grid as shown in Figure 7b. Each block is inserted into the hash tables once for each photon within that block, using the position of these photons to create the keys. Each bucket of the hash table refers to not only the photons that have been hashed into that bucket, but also all the other photons that belong to the same blocks as the hashed photons (see Figure 7c.)

Since each photon block is inserted into each hash table multiple times, using different photons as keys, a block may be hashed into multiple buckets in the same hash table. Of course, a block should not be inserted into a bucket more than once. More importantly, our technique ensures that each block is inserted into at least one hash table. Orphaned blocks are very undesirable since the photons within will never be considered in the subsequent AkNN evaluation and will cause a constant error overhead. Hence, our technique does not naïvely drop a block that causes a bucket to overflow.

However, there may be collisions that cause buckets to overflow, especially when a large bucket load factor is chosen to give a compact hash table size, or there exists a large variation in photon density (which, of course, is typical in this application). Our algorithm uses two techniques to address this problem. The first technique attempts to insert *every* block into *every* hash table, but in different orders on different hash tables, such that blocks that appear earlier in the ordering are not favoured for insertion in all tables. BH uses a technique similar to disk-striping[38], illustrated by the pseudo code in Figure 8. An example is given in the diagram in the same figure.

```
for h from 0 to (number_of_hash_tables-1)
    for b from 0 to (number_of_blocks-1)
        idx = (h+b) modulo L
        insert block[b] into hashtable[idx]
    endfor
endfor
```

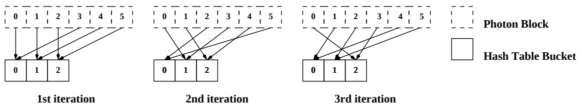


Figure 8: *Striping insertion strategy*

The second technique involves a strategy to deal with overflow in a bucket. For each photon block, BH keeps the count of buckets that the block has been hashed into so far. When a block causes overflow in a bucket, the block in the bucket that has the maximum count will be bumped if that count is larger than one, and larger than that of the incoming block. This way we ensure that all blocks are inserted into at least one bucket, given adequate hash table sizes, and no block is hashed into an excessive number of buckets.

4.4. Querying

A query into the BH data structure proceeds by delegating the query to each of the L hash tables. These parallel accesses will yield as candidates all photon blocks represented by buckets that matched the query. The final approximate nearest neighbour set comes from scanning the unified candidate set for the nearest neighbours to the query point (see Figure 9.) Note that unlike kNN algorithms based on hierarchical data structures, where candidates for the kNN set trickle in as the traversal progresses, in BH all candidates are available once the parallel queries are completed. Therefore, BH can use algorithms like *selection*[29] (instead of a priority queue) when selecting the k nearest photons.

Each query will retrieve one bucket from each of the L hash tables. If the application can tolerate elevated inaccuracy in return for increased speed of query (for example, to pre-visualise a software rendering), it may be worthwhile to consider using only a subset of the L candidate sets. Block hashing is equipped with a user-specified accuracy setting: Let $A \in \mathbb{N}$ be an integer multiplier. The block hashing algorithm will only consider Ak candidate photons in the final scan to determine the k nearest photons to a query. Obviously the smaller A is, the fewer photons will be processed in the final step; as such, query time is significantly reduced, but with an accuracy cost. Conversely, a higher A will lead to a more accurate result, but it will take more time. Experimental results that demonstrate the impact of the choice of A will be explored in Section 6.

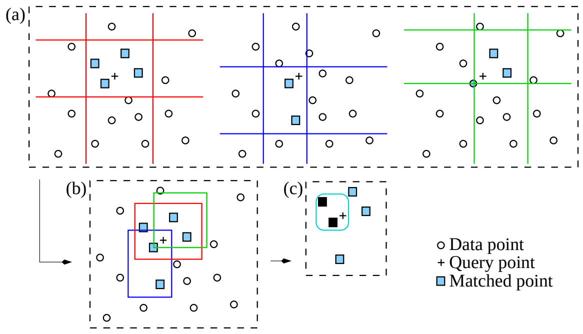


Figure 9: *Merging the results from multiple hash tables. (a) the query point retrieves different candidates sets from different hash tables, (b) the union set of candidates after merging, and (c) the two closest neighbours selected.*

There needs to be a way to select the buckets from which the Ak candidate photons are obtained. Obviously, we want to devise a heuristic to pick the "best" candidates. Suppose every bucket in every hash table has a *priority* given by

$$\alpha = |\text{bucket\_capacity} - \#\text{entries} - \#\text{overflows}|$$

where "#overflows" is the number of insertion attempts after the bucket became full. The priority can be pre-computed and stored in each bucket of each hash table during the insertion phase. The priority of a bucket is smallest when the bucket is full but never overflowed. Conversely, when the hash bucket is underutilised or overflow has occurred, α will be larger. If a bucket is underutilised, it is probably too small spatially (relative to local sample density). If it has experienced overflow, it is likely too large spatially, and covers too many photon block regions.

During each query, BH will sort the L buckets returned from the hash tables by their priority values, smallest values of α first. Subsequently, buckets are considered in this order, one by one, until the required Ak photons are found. In this way the more "useful" buckets will be considered first.

5. Choice of Parameter Values

Block Hashing is a scheme that requires several parameters: B, the bucket capacity; L, the number of hash tables whose results are merged; and P, the number of thresholds per dimension. We would like to determine reasonable values for these parameters as functions of k, the number of nearest neighbours sought, and N, the number of photons involved.

It is important to realize the implications of these parameters. The total number of 32-bit pointers to photon blocks is given by LP^3B. Along with the number of thresholds $3LP$, this gives the memory overhead required for BH. The upper bound for this value is $6N$, the number of photons multiplied by the six 32-bit words each photon takes up in our implementation. If we allow B to be a fixed constant for now, the constraint $LP^3 + 3LP \leq N$ arises from the reasonable assumption that we do not want to have more references to blocks than there are photons, or more memory used in the index than in the data.

Empirically, $L = P = \ln N$ has turned out to be a good choice. The value $\ln N$ remains sub-linear as N increases, and this value gives a satisfactory index memory overhead ratio: There are a total of $B(\ln N)^4$ block references. Being four bytes each, the references require $4B(\ln N)^4$ bytes. With each hash table, there needs to be $3LP = 3(\ln N)^2$ thresholds. Represented by a 4-byte value each, the thresholds take another $12(\ln N)^2$ bytes. Next, assuming one photon block can hold ten photons, N photons requires $N/10$ blocks; each block requires 64 words, so the blocks require $25.6N$ bytes in total. The total memory required for N photons, each occupying 6 words, is $24N$ bytes. This gives an overhead ratio of

$$(4B(\ln N)^4 + 12(\ln N)^2 + 25.6N - 24N)/24N. \qquad (1)$$

The choice of B is also dependent on the value of k specified by the situation or the user. However, since it is usual in photon mapping that k is known ahead of time, B can be set accordingly. B should be set such that the total number of photons retrieved from the L buckets for each query will be larger than k. Mathematically speaking, each photon block in our algorithm has ten photons, hence $10LB \gg k$. In particular, $10LB > Ak$ should also be satisfied. Since we choose $L = \ln N$, rearranging the equation yields: $B > Ak/(10 \ln N)$ For example, assuming $A = 16$, $N = 2000000$, $k = 50$, then $\lceil B \rceil = 6$.

If we substitute B back into Equation 1, we obtain the final overhead equation

$$(4(Ak/10)(\ln N)^3 + 12(\ln N)^2 + 1.6N)/24N. \qquad (2)$$

Figure 10 plots the number of photons versus the memory overhead. For the usual range of photon count in a photon mapping application, we see that the memory overhead, while relative large for small numbers of photons, becomes reasonable for larger numbers of photons, and has an asymptote of about 6%. Of course, if we assumed different block size (cache line size), these results would vary, but the analysis is the same.

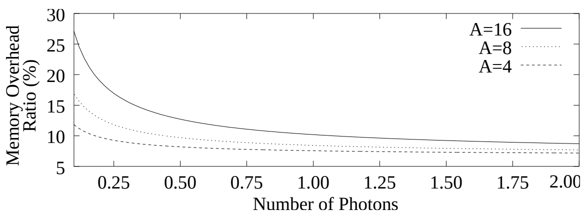


Figure 10: *Plot of photon count vs. memory overhead incurred by BH, assuming $k = 50$.*

6. Results

For BH to be a useful AkNN algorithm, it must have satisfactory algorithmic accuracy. Moreover, in the context of photon mapping, BH must also produce good visual accuracy. This section will demonstrate that BH satisfies both requirements, while providing a speed-up in terms of the time it takes to render an image, even in a software implementation.

To measure algorithmic accuracy, our renderer was rigged to use both the *kd*-tree and BH based photon maps. For each kNN query the result sets were compared for the following metrics:

False negatives: # photons incorrectly excluded from the kNN set.

Maximum distance dilation: the ratio of bounding radii around the neighbours reported by the two algorithms.

Average distance dilation: the ratio of the average distances between the query point and each of the nearest neighbours reported by the two algorithms.

To gauge visual accuracy, we calculate a **Caustic RMS Error** metric, which compares the screen space radiance difference between the caustic radiance values obtained from *kd*-tree and BH.

A timing-related **Time Ratio** metric is calculated as a ratio of the time taken for a query into the BH data structure versus that for the *kd*-tree data structure. Obviously, as A increases, the time required for photon mapping using BH approaches that for a *kd*-tree based photon mapping.

Our first test scene, shown in Figure 11, with numerical results in Figure 12, consists of a highly specular ring placed on top of a plane with a Modified Phong[28] reflectance model. This scene tests the ability of BH to handle a caustic of varying density, and a caustic that has been cast onto a non-Lambertian surface.

Figure 13 shows a second scene consisting of the Venus bust, with a highly specular ring placed around the neck of Venus. Figure 14 shows the numerical statistics of this test scene. The ring casts a cardioid caustic onto the (non-planar) chest of the Venus. This scene demonstrates a caustic on a highly tessellated curved surface. Global illumination is also used for both scenes, however the times given are only for the query time of the caustic maps.

The general trend to notice is that for extremely low accuracy (A) settings the visual and algorithmic performance of BH is not very good. The candidate set is simply not large enough in these cases. However, as A increases, these performance indicators drop to acceptable levels very quickly, especially between values of A between 2 and 8. After $A = 8$ diminishing returns set in, and the increase in accuracy incurs a significant increase in the cost of the query time required. This numerical error comparison is parallelled by the

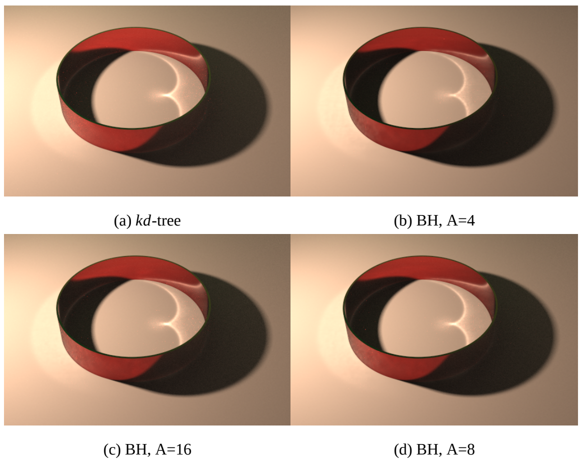


Figure 11: *"Ring" image*

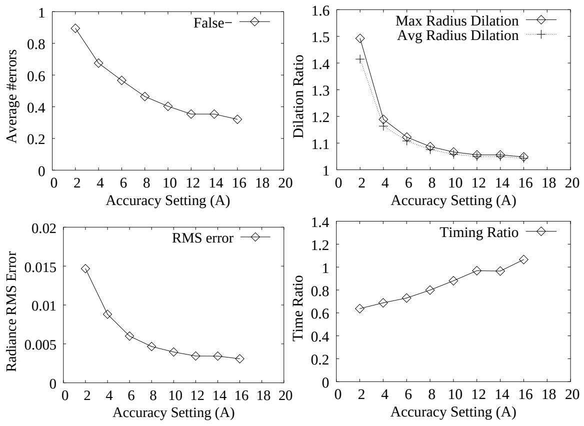


Figure 12: *"Ring" numerical statistics*

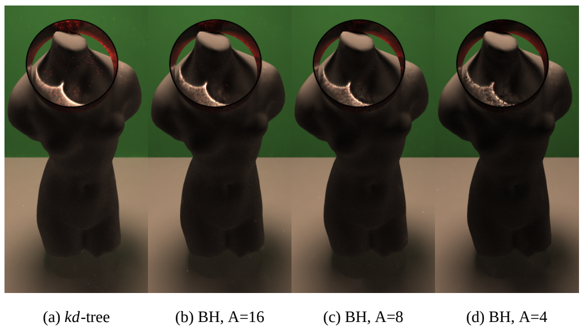


Figure 13: *"Venus with Ring" images*

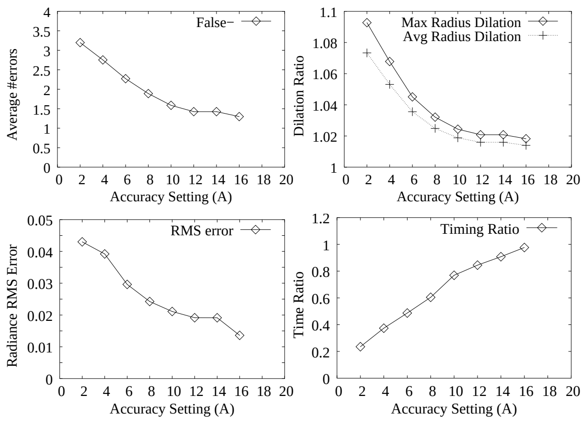


Figure 14: *"Venus with Ring" numerical statistics*

visual comparison of the images: the image rendered with $A = 8$ and $A = 16$ are virtually indistinguishable. These results suggest that intermediate values of A, between 8 to 10, should be used as a good compromise between query speed and solution accuracy.

It is apparent from the query time ratio plots that there exists a close-to-linear relationship between values of A and time required for a query into BH. This is consistent with the design of the A parameter; it corresponds directly to the number of photons accessed and processed for each query.

Another important observation that can be made from the visual comparisons is that images with greater approximation value A look darker. This is because the density estimate is based on the inverse square of the radius of the sphere enclosing the k nearest neighbours. The approximate radius is always larger than the true radius. This is an inherent problem with any approximate solution to the kNN problem, and indeed even with the exact kNN density estimator: as k goes to infinity, the kNN density estimator does converge on the true density, but always from below.

7. Hardware Implementation

There are two ways to approach a hardware implementation of an algorithm in a hardware accelerator: with a custom hardware implementation, or as an extension or exploitation of current hardware support. While there would be certain advantages in a full custom hardware implementation of BH, this would probably lead to a large chunk of hardware with low utilisation rates. Although there are many potential applications of AkNN search beyond photon mapping (we list several in the conclusions), it seems more reasonable to consider first if BH can be implemented using current hardware and programmable shader features, and if not, what the smallest incremental changes would be. We have concluded that BH, while not quite implementable on today's graphics hardware, should be implementable in the near future.

We consider only the lookup phase here, since the preprocessing would indeed require some custom hardware support, but support which perhaps could be shared with other useful features. In the lookup phase, (1) we compute hash keys, (2) look up buckets in multiple hash tables, (3) merge and remove duplicates from the list of retrieved blocks and optionally sorting by priority, (4) retrieve the photon records stored in these blocks, and (5) process the photons. Steps (1) and (5) could be performed with current shader capabilities, although the ability to loop would be useful for the last step to avoid replicating the code to process each photon. Computing the hash function amounts to doing a number of comparisons, then adding up the zero-one results. This can be done in linear time with a relatively small number of instructions using the proposed DX9 fragment shader instruction set. If conditional assignment and array lookups into the register file are supported, this could be done in logarithmic time using binary search.

Steps (2) and (4) amount to table lookups and can be implemented as nearest-neighbour texture-mapping operations with suitable encoding of the data. For instance, the hash tables might be supported with one texture map giving the priority and number of valid entries in the bucket, while another texture map or set of texture maps might give the block references, represented by texture coordinates pointing into another set of texture maps holding the photon records.

Step (3) is difficult to do efficiently without true conditionals and conditional looping. Sorting is not the problem, as it could be done with conditional assignment. The problem is that removal of a duplicate block reduces the number of blocks in the candidate set. We would like in this case to avoid making redundant photon lookups and executing the instructions to process them. Without true conditionals, an inefficient work-around is to make these redundant texture accesses and process the redundant photons anyhow, but discard their contribution by multiplying them by zero.

We have not yet attempted an implementation of BH on an actual accelerator. Without looping, current accelerators do not permit nearly enough instructions in shaders to process k photons for adequate density estimation. However, we feel that it might be feasible to implement our algorithm on a next-generation shader unit using the "multiply redundant photons by zero" approach, if looping (a constant number of times) were supported at the fragment level.

We expect that the generation that follows DX9-class accelerators will probably have true conditional execution and looping, in which case implementation of BH will be both straightforward and efficient, and will not require any additional hardware or special shader instructions. It will also only require two stages of conditional texture lookup, and lookups in each stage can be performed in parallel. In comparison, it would be nearly impossible to implement a tree-based search algorithm on said hardware due to the lack of a stack and the large number of dependent lookups that would be required. With a sufficiently powerful shading unit, of course, we could implement any algorithm we wanted, but BH makes fewer demands than a tree-based algorithm does.

8. Conclusion and Future Work

We have presented an efficient, scalable, coherent and highly parallelisable AkNN scheme suitable for the high-performance implementation of photon mapping.

The coherent memory access patterns of BH lead to improved performance even for a software implementation. However, in the near future we plan to implement the lookup phase of BH on an accelerator. Accelerator capabilities are not quite to the point where they can support this algorithm, but they are very close. What is missing is some form of true run-time conditional execution and looping, as well as greater capacity in terms of numbers of instructions. However, unlike tree-based algorithms, block hashing requires only bounded execution time and memory.

An accelerator-based implementation would be most interesting if it is done in a way that permits other applications to make use of the fast AkNN capability it would provide. AkNN has many potential applications in graphics beyond photon maps. For rendering, it could also be used for sparse data interpolation (with many applications: visualisation of sparse volume data, BRDF and irradiance volume representation, and other sampled functions), sparse and multi-resolution textures, procedural texture generation (specifically, Worley's texture[47] functions), direct ray-tracing of point-based objects[40], and gap-filling in forward projection point-based rendering[48]. AkNN could also potentially be used for non-rendering purposes: collision detection, surface reconstruction, and physical simulation (for interacting particle systems). Unlike the case with tree-based algorithms, we feel that it will be feasible to implement BH as a shader subroutine in the near future, which may make it a key component in many potential applications of advanced programmable graphics accelerators.

For a more detailed description of the block hashing algorithm, please refer to the author's technical report[32].

9. Acknowledgements

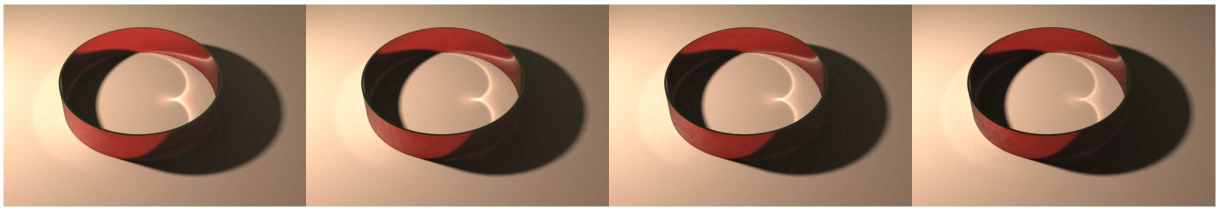
This research was funded by grants from the National Science and Engineering Research Council of Canada (NSERC), the Centre for Information Technology of Ontario (CITO), the Canadian Foundation for Innovation (CFI), the Ontario Innovation Trust (OIT), and the Bell University Labs initiative.

References

1. P. K. Agarwal. Range Searching. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press, July 1997. 2
2. P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. *Advances in Discrete and Computational Geometry*, 23:1–56, 1999. 2

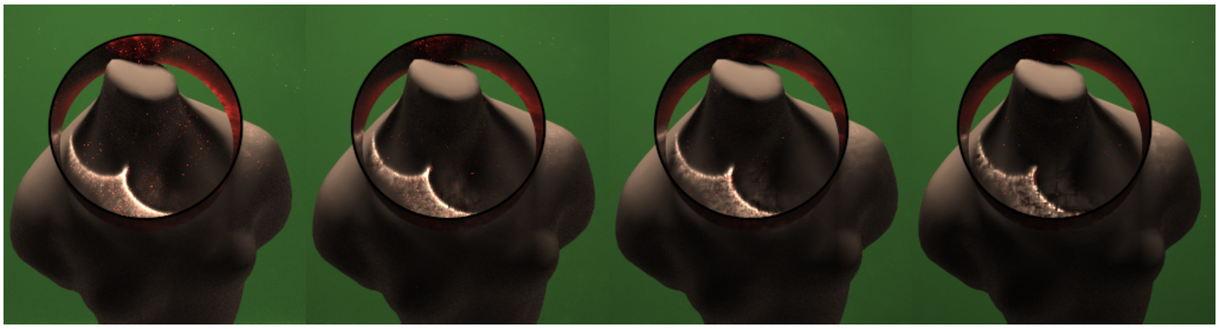
3. S. Arya and D. M. Mount. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *Proc. ACM-SIAM SODA*, 1993. 2
4. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An Optimal Algorithm for Approximate Nearest Neighbor Searching. In *Proc. ACM-SIAM SODA*, pages 573–582, 1994. 2
5. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), September 1975. 1, 2
6. J. L. Bentley, B. W. Weide, and A. C. Chow. Optimal Expected-Time Algorithms for Closest Point Problems. *ACM TOMS*, 6(4), December 1980. 1, 2
7. Per H. Christensen. Faster Photon Map Global Illumination. *Journal of Graphics Tools*, 4(3):1–10, 1999. 2
8. D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979. 4
9. C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. In *Proc. ACM-SIAM SODA*, volume 10, pages 300–309, 1999. 2
10. H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987. 2
11. D. Eppstein, M. S. Paterson, and F. F. Yao. On nearest neighbor graphs. *Discrete & Computational Geometry*, 17(3):263–282, April 1997. 2
12. C. Faloutsos and Y. Rong. DOT: A Spatial Access Method Using Fractals. In *Proc. 7th Int. Conf. on Data Engineering,*, pages 152–159, Kobe, Japan, 1991. 4
13. C. Faloutsos and S. Roseman. Fractals for Secondary Key Retrieval. In *Proc. 8th ACM PODS*, pages 247–252, Philadelphia, PA, 1989. 4
14. J. H. Freidman, J. L. Bentley, and R. A. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM TOMS*, 3(3):209–226, 1977. 2
15. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys (CSUR)*, 30(2):170–231, 1998. 2
16. A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proc. VLDB*, pages 518–529, 1999. 2, 3
17. J. E. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry*. CRC Press, July 1997. ISBN: 0849385245. 2
18. G. Greger, P. Shirley, P. Hubbard, and D. Greenberg. The irradiance volume. *IEEE CG&A*, 18(2):32–43, 1998. 1
19. V. Havran. Analysis of Cache Sensitive Representation for Binary Space Partitioning Trees. *Informatica*, 23(3):203–210, May 2000. 2
20. P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proc. ACM STOC*, pages 604–613, 1998. 2
21. P. Indyk, R. Motwani, P. Raghavan, and S. Vempala. Locality-Preserving Hashing in Multidimensional Spaces. In *Proc. ACM STOC*, pages 618–625, 1997. 2
22. H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proc. Acm-sigmod*, pages 332–342, May 1990. 4
23. J. W. Jaromczyk and G. T. Toussaint. Relative Neighborhood Graphs and Their Relatives. *Proc. IEEE*, 80(9):1502–1517, September 1992. 2
24. H. W. Jensen. Rendering Caustics on Non-Lambertian Surfaces. *Computer Graphics Forum*, 16(1):57–64, 1997. ISSN 0167-7055. 1
25. H. W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. A.K. Peters, 2001. 1, 2, 4
26. H. W. Jensen, F. Suykens, and P. H. Christensen. A Practical Guide to Global Illumination using Photon Mapping. In *SIGGRAPH 2001 Course Notes*, number 38. ACM, August 2001. 2
27. J. K. P. Kuan and P. H. Lewis. Fast k Nearest Neighbour Search for R-tree Family. In *Proc. of First Int. Conf. on Information, Communication and Signal Processing*, pages 924–928, Singapore, 1997. 2
28. E. P. Lafortune and Y. D. Willems. Using the Modified Phong BRDF for Physically Based Rendering. Technical Report CW197, Department of Computer Science, K.U.Leuven, November 1994. 7
29. C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001. 6
30. K.-I. Lin and C. Yang. The ANN-Tree: An Index for Efficient Approximate Nearest-Neighbour Search. In *Conf. on Database Systems for Advanced Applications*, 2001. 2
31. N. Linial and O. Sasson. Non-Expansive Hashing. In *Proc. acm stoc*, pages 509–518, 1996. 2
32. V. Ma. Low Latency Photon Mapping using Block Hashing. Technical Report CS-2002-15, School of Computer Science, University of Waterloo, 2002. 9
33. J. McNames. A Fast Nearest-Neighbor Algorithm Based on a Principal Axis Search Tree. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(9):964–976, 2001. 2
34. J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: an adaptable, symmetric multikey file structure. *ACM TODS*, 9(1):38–71, March 1984. 3, 5
35. T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. In *to appear in Proc. SIGGRAPH*, 2002. 1
36. J. T. Robinson. The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proc. acm sigmod*, pages 10–18, 1981. 2
37. Arnold L. Rosenberg and Lawrence Snyder. Time- and space-optimality in b-trees. *ACM TODS*, 6(1):174–193, 1981. 4
38. K. Salem and H. Garcia-Molina. Disk striping. In *IEEE ICDE*, pages 336–342, 1986. 5

39. N. Sample, M. Haines, M. Arnold, and T. Purcell. Optimizing Search Strategies in kd-Trees. May 2001. 2
40. G. Schaufler and H. W. Jensen. Ray Tracing Point Sampled Geometry. *Rendering Techniques 2000*, pages 319–328, June 2000. 9
41. J. Schmittler, I. Wald, and P. Slusallek. SaarCOR - A Hardware Architecture for Ray Tracing. In *to appear at EUROGRAPHICS Graphics Hardware*, 2002. 1
42. M. Smid. Closest-Point Problems in Computational Geometry. In J. R. Sack and J. Urrutia, editors, *Handbook on Computational Geometry*. Elsevier Science, Amsterdam, North Holland, 2000. 2
43. P. Tsaparas. Nearest neighbor search in multidimensional spaces. Qualifying Depth Oral Report 319-02, Dept. of Computer Science, University of Toronto, 1999. 2
44. M. Vanco, G. Brunnett, and T. Schreiber. A Hashing Strategy for Efficient k-Nearest Neighbors Computation. In *Computer Graphics International*, pages 120–128. IEEE, June 1999. 2
45. I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek. Interactive global illumination. Technical report, Computer Graphics Group, Saarland University, 2002. to be published at EUROGRAPHICS Workshop on Rendering 2002. 2
46. G. Ward. Real Pixels. In James Arvo, editor, *Graphics Gems II*, pages 80–83. Academic Press, 1991. 4
47. Steven Worley. A cellular texture basis function. In *Proc. SIGGRAPH 1996*, pages 291–294. ACM Press, 1996. 9
48. M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface Splatting. *Proc. SIGGRAPH 2001*, pages 371–378, 2001. 9



(a) *kd*-tree (b) BH, A=16 (c) BH, A=8 (d) BH, A=4

Figure 13: *"Ring"*



(a) *kd*-tree (b) BH, A=16 (c) BH, A=8 (d) BH, A=4

Figure 14: *"Venus with Ring"*