Low Latency Photon Mapping with Block Hashing

Vincent C.H. Ma
Michael D. McCool
Computer Graphics Lab
University of Waterloo

Menu du Jour

Appetizer:	Motivation and Background
First course:	Locality Sensitive Hashing
Entrée:	Block Hashing
Dessert:	Results, Future Work, Conclusion

Motivation

- Trend: Rendering algorithms migrating towards hardware(-assisted) implementations
  - [Purcell et al. 2002]
  - [Schmittler et al. 2002]

➢ We wanted to investigate hardware(-assisted) implementation of Photon Mapping [Jensen95]

kNN Problem

- Need to solve for k-Nearest Neighbours (kNN)
- Used for density estimation in photon mapping
- Want to eventually integrate with GPU-based rendering
- ➢ Migrate kNN onto hardware-assisted platform
  - Adapting algorithms and data structures
  - Parallelization
  - Approximate kNN?

Applications of kNN

- kNN has many other applications, such as:
  - Procedural texture generation [Worley96]
  - Direct ray tracing of point-based objects [Zwicker et al. 2001]
  - Surface reconstruction
  - Sparse data interpolation
  - Collision detection

Hashing-Based AkNN

- Why hashing?
  - Hash function can be evaluated in constant time
  - Eliminates multi-level, serially-dependent memory accesses

  ➢ Amenable to fine-scale parallelism and pipelined memory

Hashing-Based AkNN

- Want hash functions that preserve spatial neighbourhoods
  - Points close to each other in domain space will be close together in hash space
  - Points in the same hash bucket as query point are close to query point in domain space
  - Good candidates for *k*-nearest neighbour search

Locality Sensitive Hashing

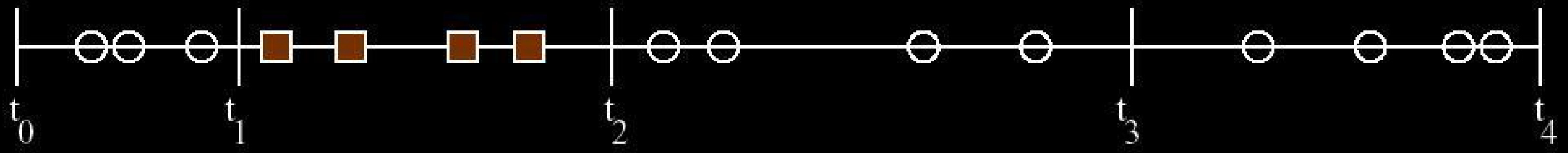
- Gionis, Indyk, and Motwani,
  Similarity Search in High Dimensions via Hashing, Proc. VLDB'99
  - Hash function partitions domain space
  - Assigns one hash value per partition

  ➢ All points falling into the same partition will receive the same hash value

Mathematically…

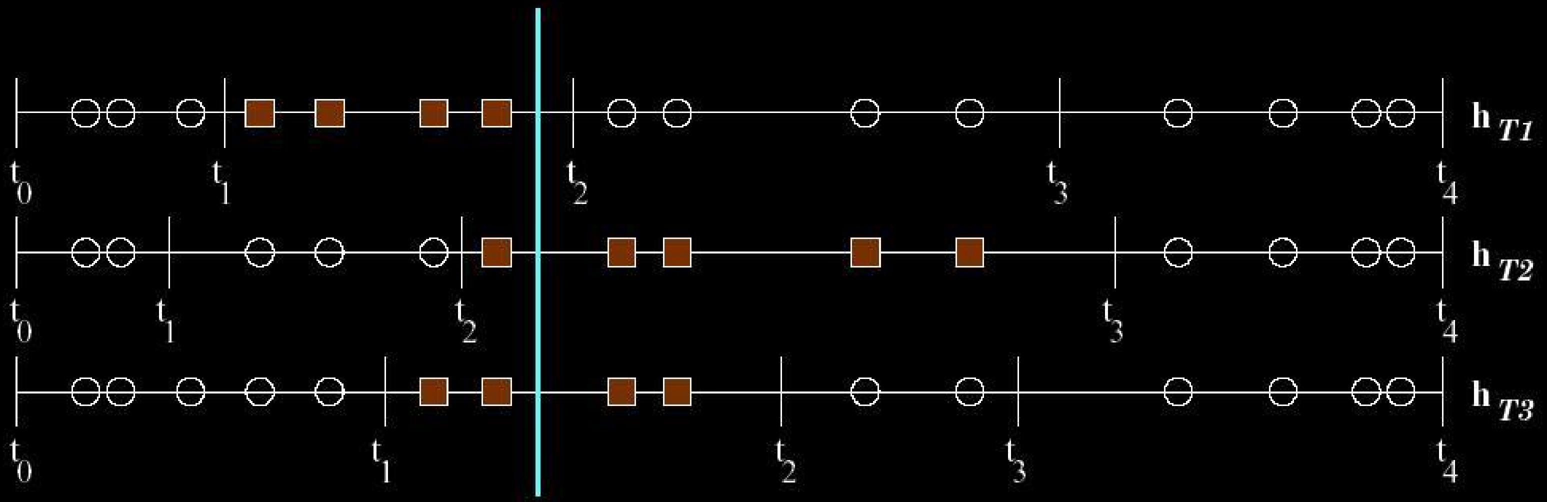
- Let $T = \{t_i \mid 0 \le i \le P\}$ be a monotonically increasing sequence of thresholds
- Define hash function to be

$$h_T(t) = i, \text{ for } t_i \le i \le t_{i+1}$$



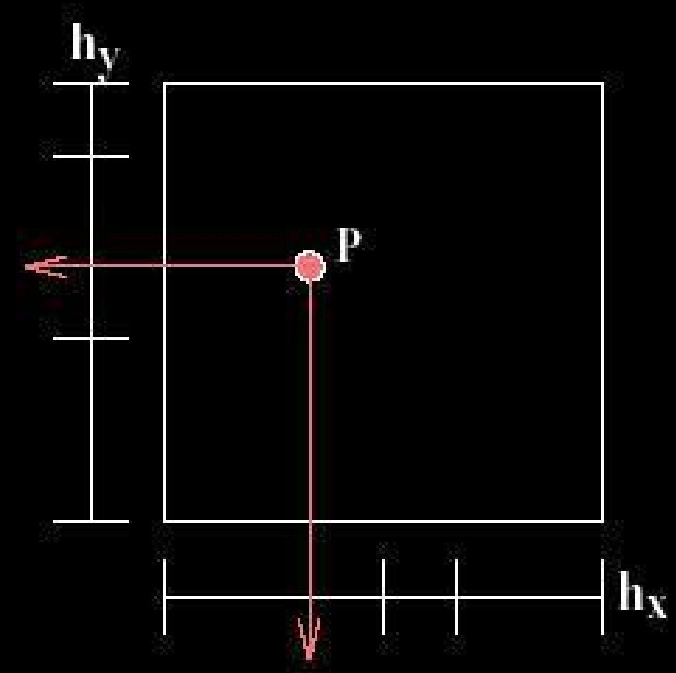
Multiple Hash Functions

- Each hash bucket stores a subset of the local neighbourhood
- Multiple hash tables are needed for retrieving a complete neighbourhood



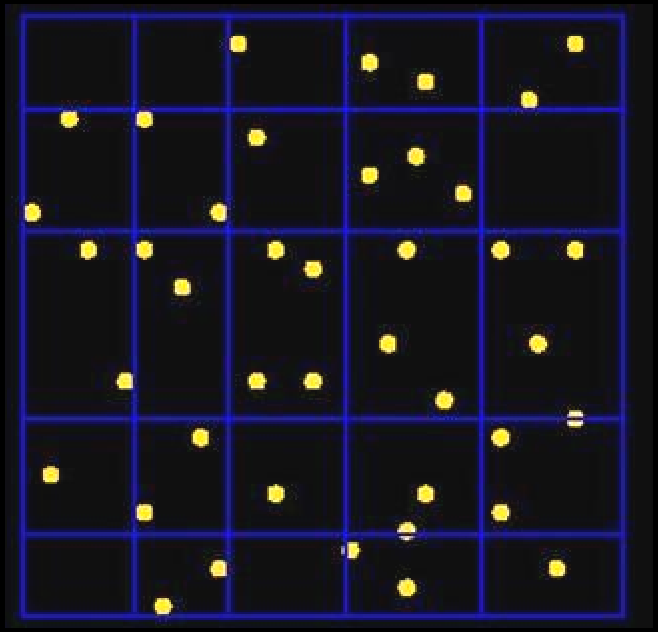
Higher Dimensions

- Multidimensional points are handled by using one hash function per dimension

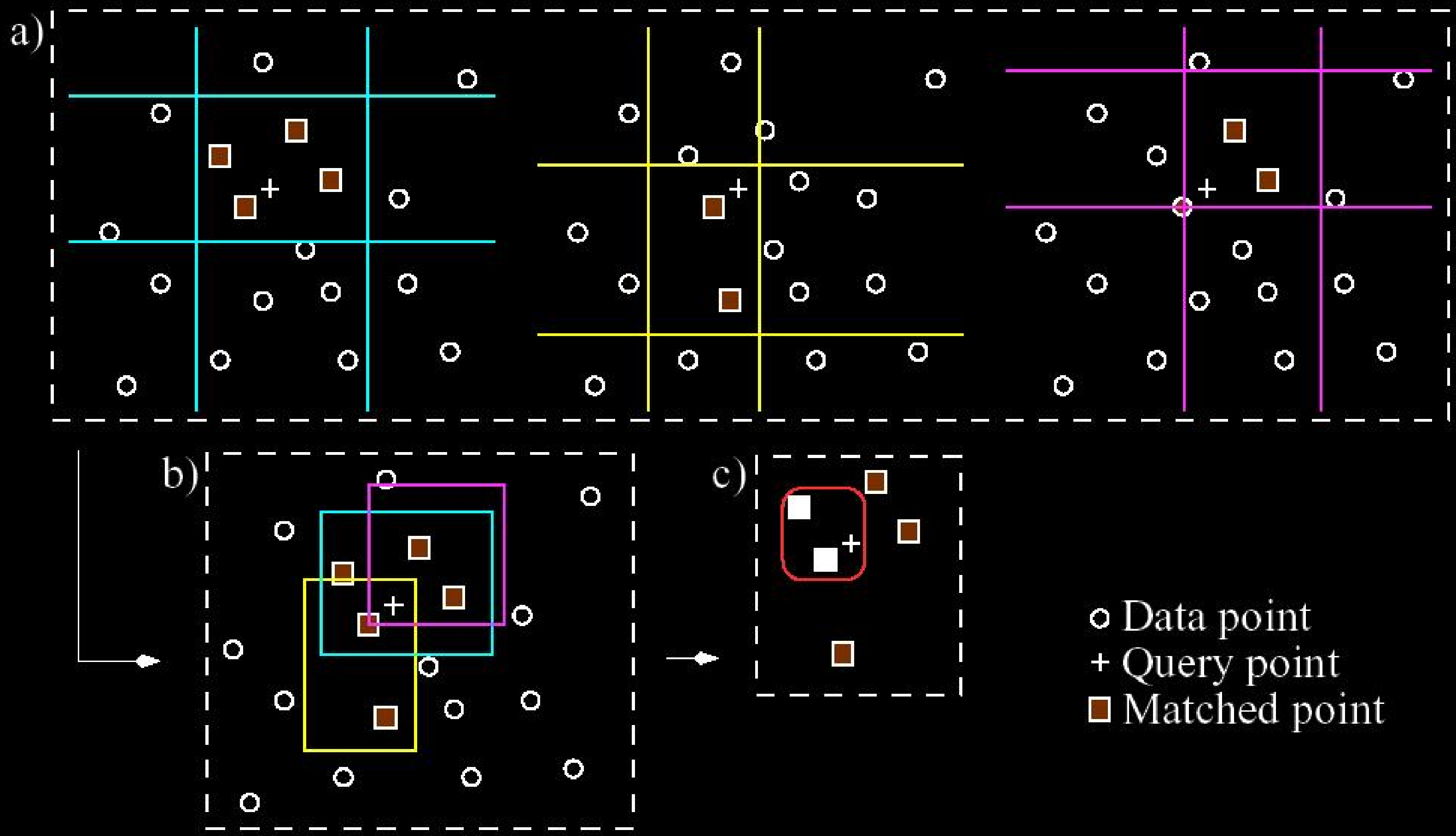


Higher Dimensions

- Together, hash functions partition space into variable-sized rectangular cells



Higher Dimensions



Block Hashing Essentials

- Photons are grouped into spatially-coherent memory blocks
- Entire blocks are inserted into the hash tables

Why Blocks of Photons?

- More desirable to insert blocks of photons into the hash table (instead of individual photons)
  - Fewer references needed per hash table bucket
  - Fewer items to compare when merging results from multiple hash tables during query
  - Photon records are accessed once per query
  - Memory block-oriented anyways

Block-Oriented Memory Model

- Memory access is via burst transfer
  - Reading any part of a fixed-sized memory block implies the access to the rest of this block is virtually zero-cost
- 256-byte chosen as size of photon blocks
  - Photons are 24-bytes
  - X = 10 photons fit in each block

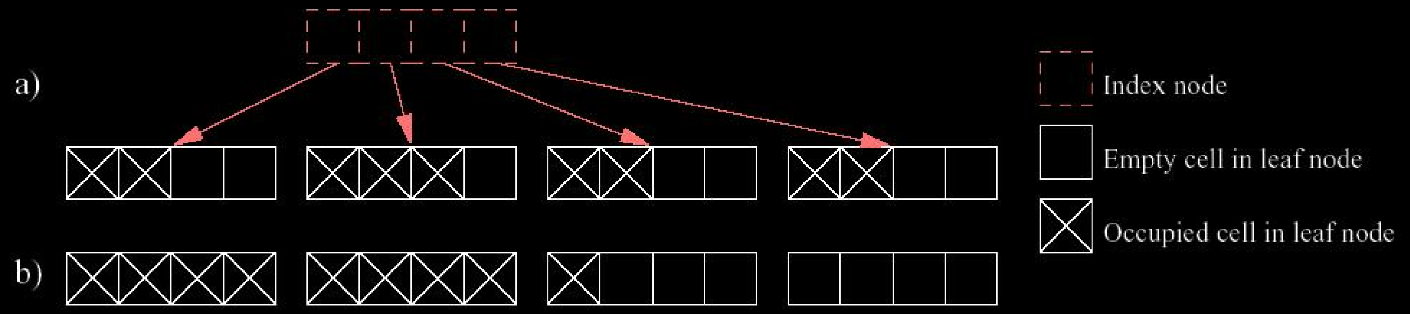
Block Hashing

- Preprocessing: before rendering
  - Organize photons into blocks
  - Create hash tables
  - Insert blocks into hash tables

- Query phase: during rendering

Organize Photon Blocks

- Want to sort photons by spatial location
  - Hilbert curve generates 1D key from photon location
  - Insert photon records into B+ tree
- Leaf nodes of B+ tree becomes photon blocks
- Compact leaf nodes to minimize blocks required



Create Hash Tables

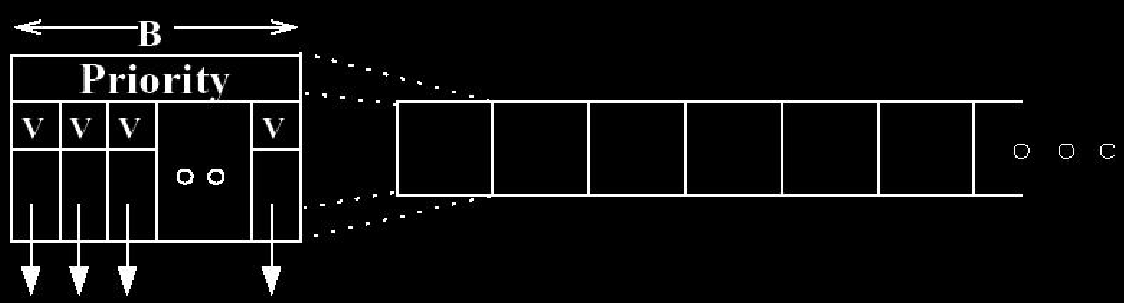
- Based on LSH:
  - L hash tables
  - Each hash table has three hash functions
  - Each function has P thresholds

Create Hash Tables

- Generate thresholds adaptively
  - Create one photon-position histogram per dimension
  - Integrate → cumulative distribution function (*cdf*)
  - Invert *cdf*, take stochastic samples to get thresholds

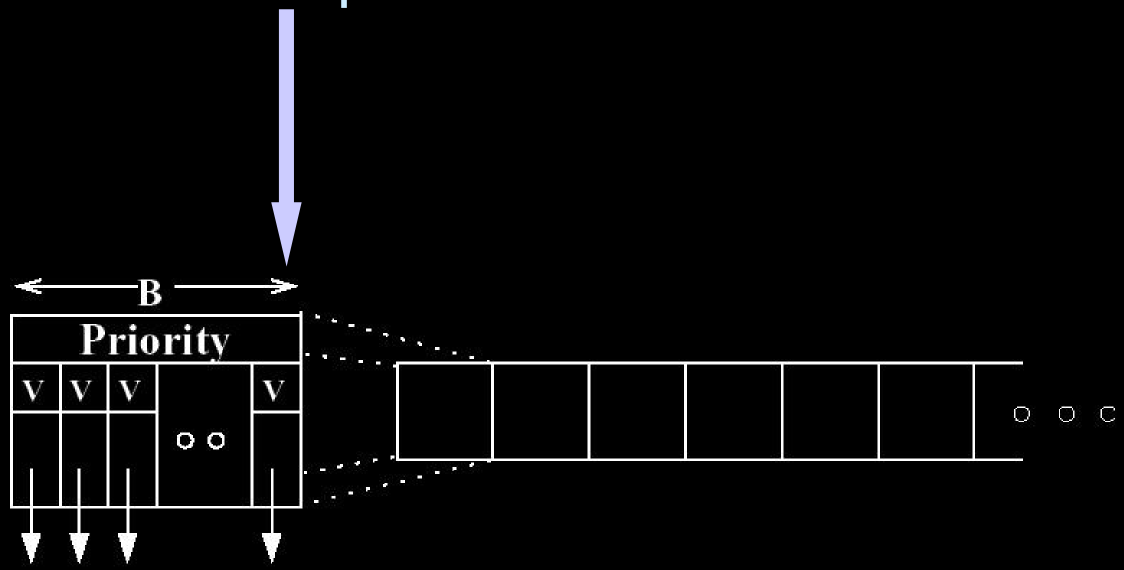
Create Hash Tables

- Hash table stored as 1D array in memory
- Each element is a hash bucket



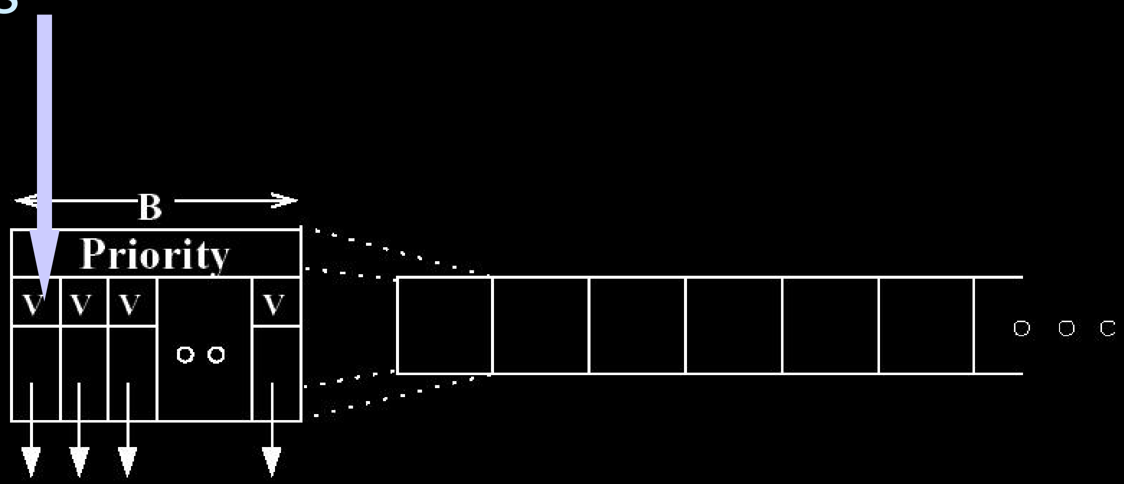
Create Hash Tables

- Hash table stored as 1D array in memory
- Each element is a hash bucket
  - B references to photon blocks



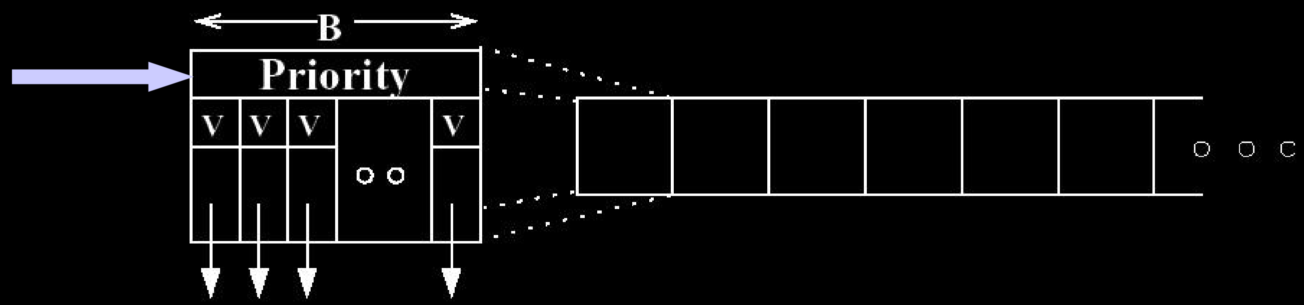
Create Hash Tables

- Hash table stored as 1D array in memory
- Each element is a hash bucket
  - B references to photon blocks
  - flags



Create Hash Tables

- Hash table stored as 1D array in memory
- Each element is a hash bucket
  - B references to photon blocks
  - flags
  - a priority value

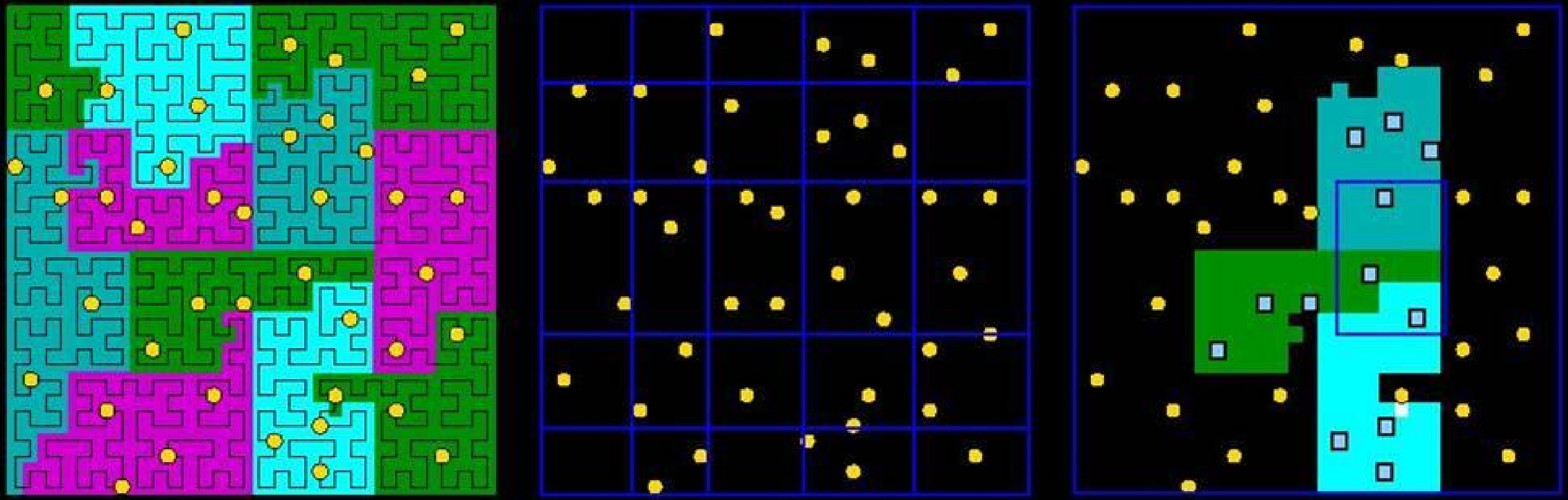


Insert Photon Blocks

- For each photon and each hash table:
  - Create hash key using 3D position of photon
  - Insert entire photon block into hash table using key
- Strategies to deal with bucket overflow

Insert Photon Blocks

- Each bucket refers to entire blocks with at least one photon that hashed into the bucket
  - Bucket also responsible for all photons in blocks it refers to



Block Hashing

- Preprocessing: before rendering
  - Organizing photons into blocks
  - Creating the hash tables
  - Inserting blocks into hash tables

- Query phase: during rendering

Querying

- Query is delegated to each hash table in parallel
- Each hash table returns all blocks contained in bucket that matched the query
- Set of unique blocks contain candidate set of photons
  - Each block contains a disjoint set of photons

Querying

- Each query yields L buckets, each bucket gives B photon blocks
- Each query retrieves at most BL blocks
- Can trade accuracy for speed:
  - User defined variable A, determines # blocks eventually included in candidate search
  - Examines at most Ak photons → Ak/X blocks

Querying

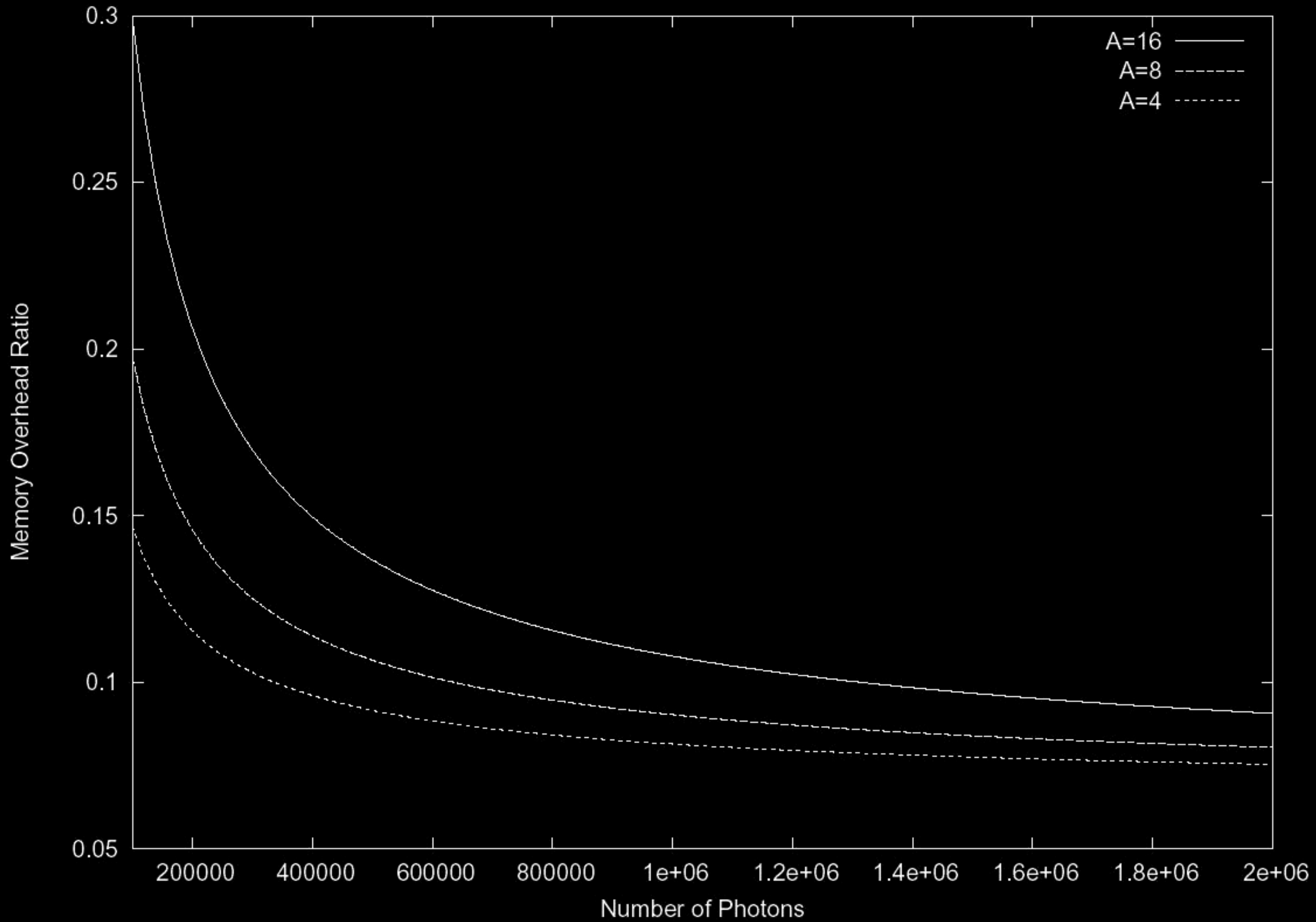
- Which photon blocks to examine first?
- Assign “quality measure” Q to every bucket in each hash table
  - Q = B - #blocks_inserted - #overflows
- Sort buckets by their “priority” |Q|

Parameter Values

- Need to express L, P, and B in terms of N, k and A
- Experiments showed lnN is a good choice for both L and P
- B is determined by k and A, given by: $$B > \frac{Ak}{X \ln N}$$
- Memory overhead:

$$\frac{4\frac{Ak}{X}(\ln N)^3 + 12(\ln N)^2 + 1.6N}{24N} = O\left[\frac{(\ln N)^3}{N}\right]$$

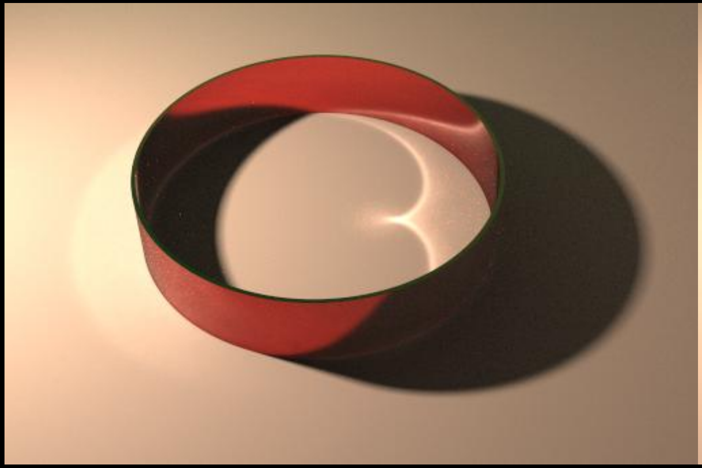
Memory Overhead



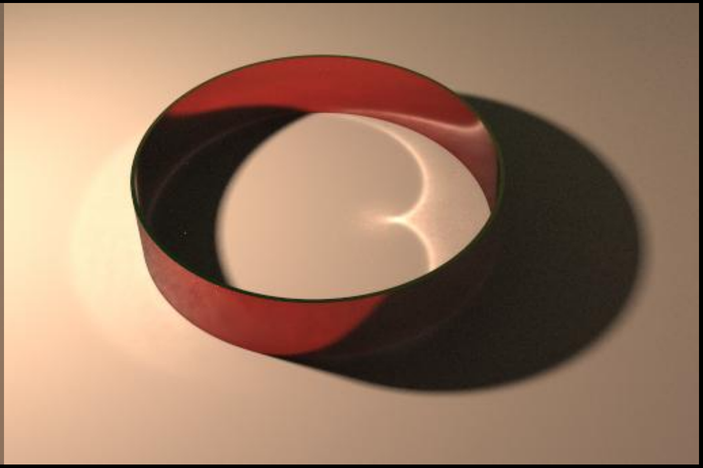
Results

- Visual quality
- Algorithmic accuracy
  - False negatives
  - Maximum distance dilation
  - Average distance dilation

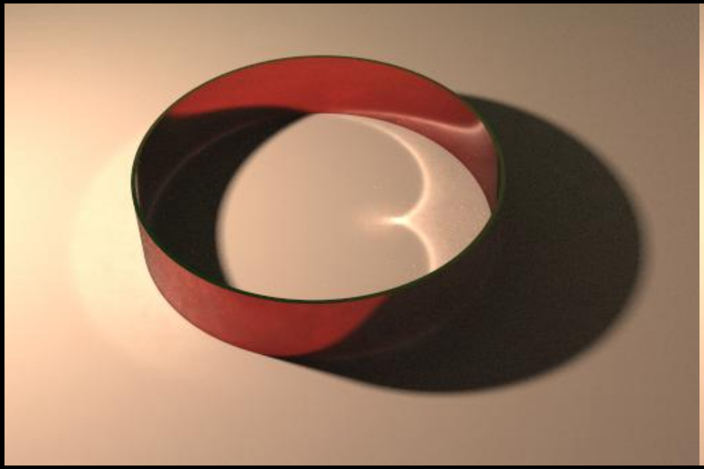
Results



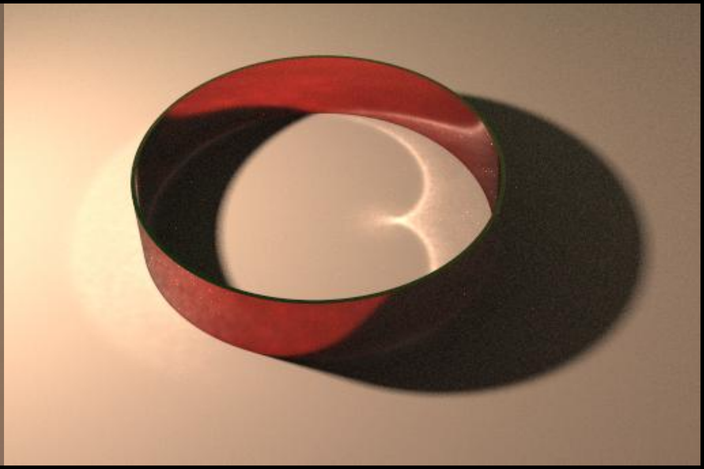
kd-tree



BH A=8

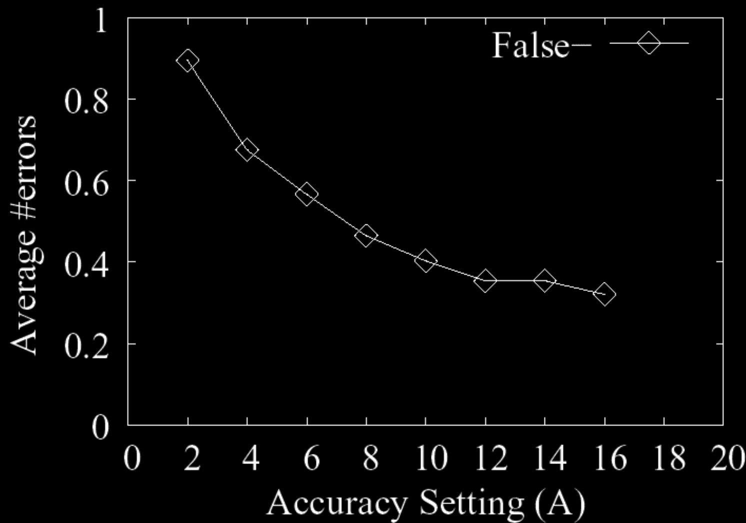


BH A=16

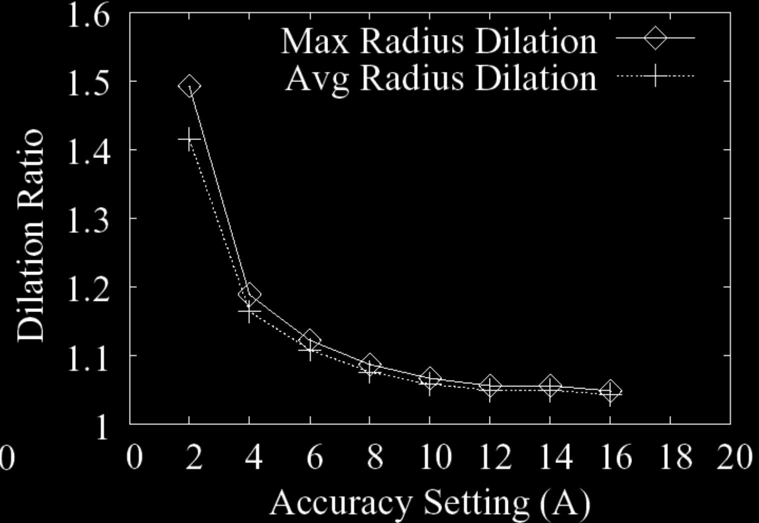


BH A=4

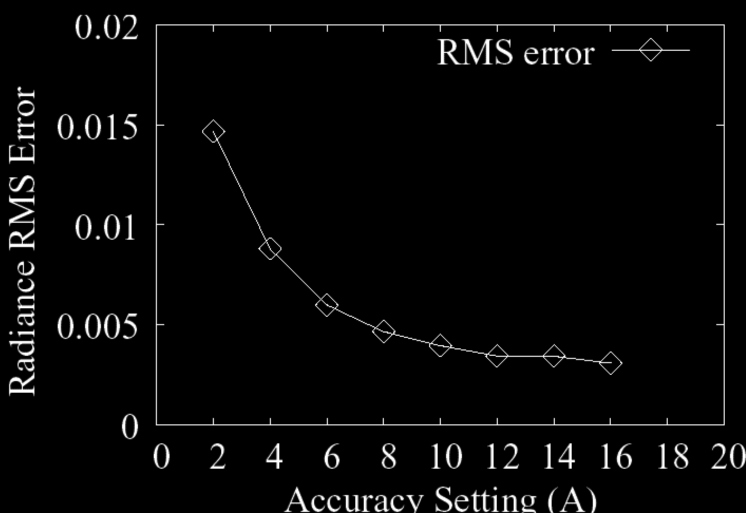
Results



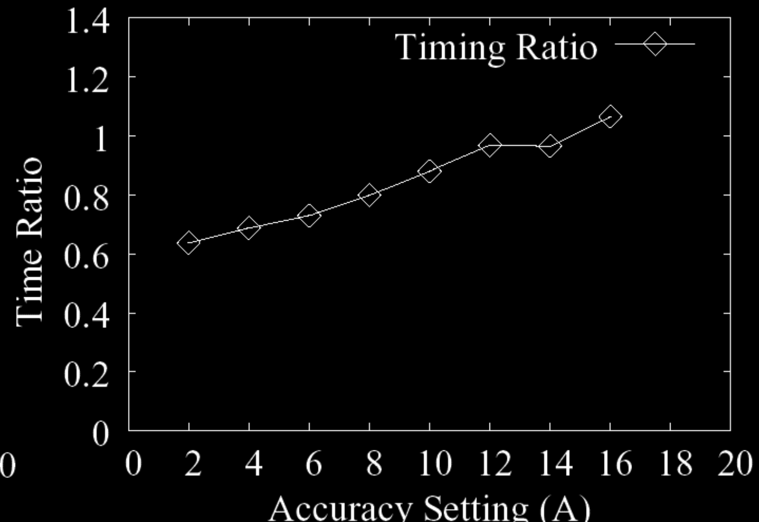
False
Average #errors
Accuracy Setting (A)



Max Radius Dilation
Avg Radius Dilation
Dilation Ratio
Accuracy Setting (A)

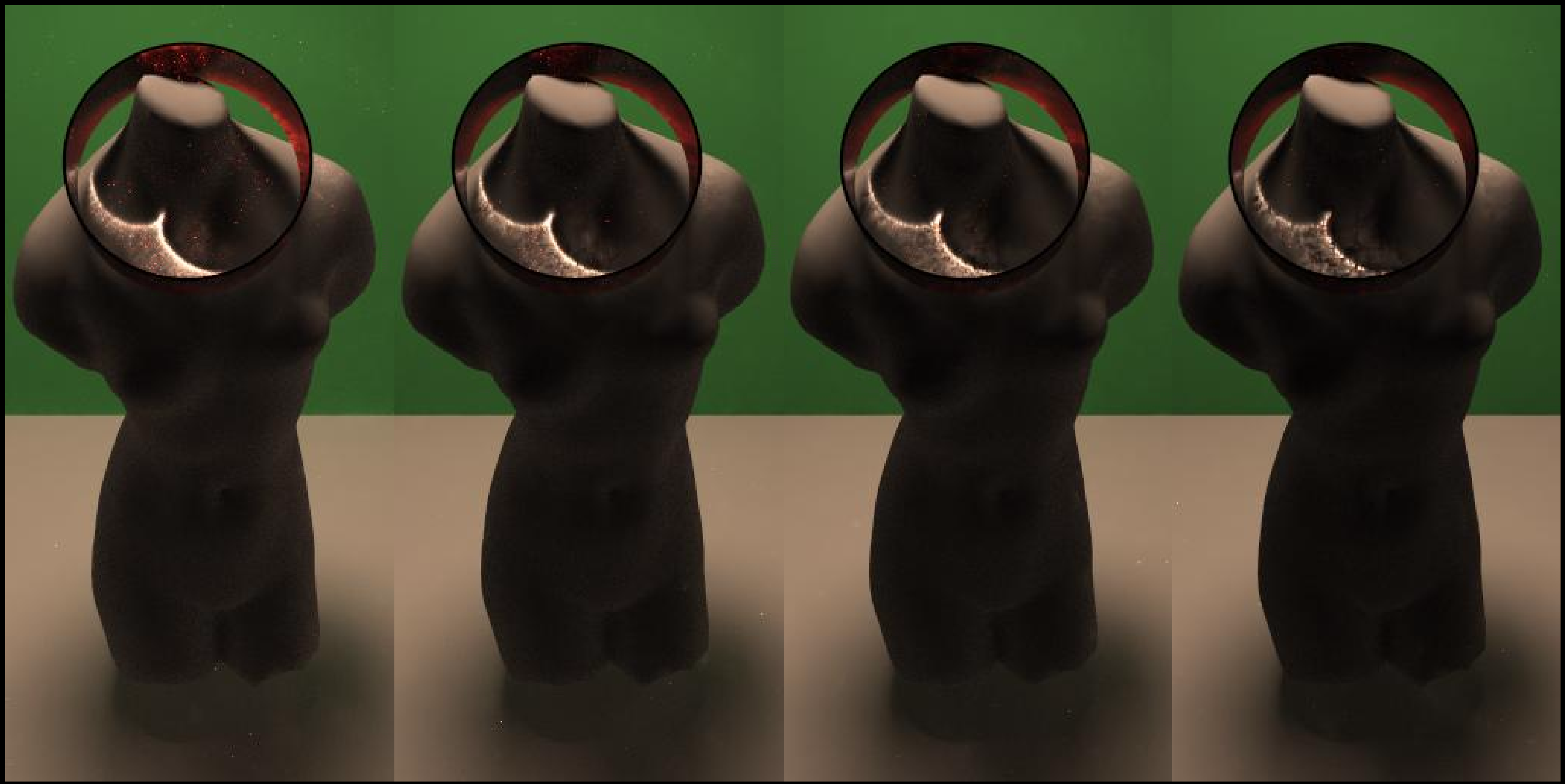


RMS error
Radiance RMS Error
Accuracy Setting (A)



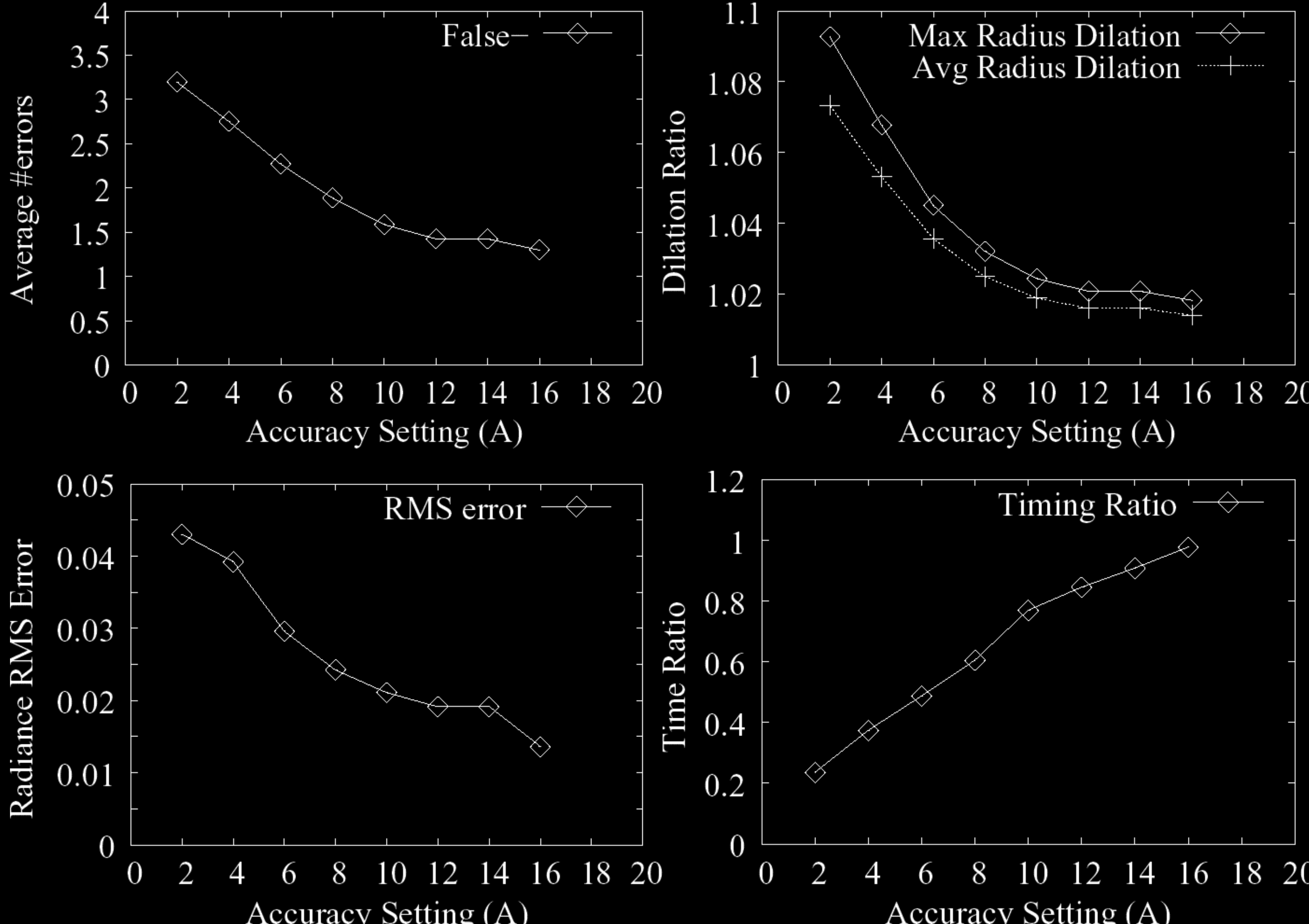
Timing Ratio
Time Ratio
Accuracy Setting (A)

Results



kd-tree BH A=16 BH A=8 BH A=4

Results


False
Average #errors
Accuracy Setting (A)
Max Radius Dilation
Avg Radius Dilation
Dilation Ratio
Accuracy Setting (A)
RMS error
Radiance RMS Error
Accuracy Setting (A)
Timing Ratio
Time Ratio
Accuracy Setting (A)

Hardware-Assisted Implementation

Query phase:

(1) Generate hash keys for 3D query position

(2) Find hash buckets that match keys

(3) Merge sets of photons blocks into unique collection

(4) Retrieve photons from blocks

(5) Process photons to find k-nearest to query position

Hardware-Assisted Implementation

(1) Generate hash keys for 3D query position

(5) Process photons to find k-nearest to query position

- Could be performed with current shader capabilities
- Loops will reduce shader code redundancy

Hardware-Assisted Implementation

(2) Find hash buckets that match keys

(4) Retrieve photons from blocks

- Amounts to table look-ups
- Can be implemented as texture-mapping operations given proper encoding of data

Hardware-Assisted Implementation

(3) Merge sets of photons blocks into unique collection

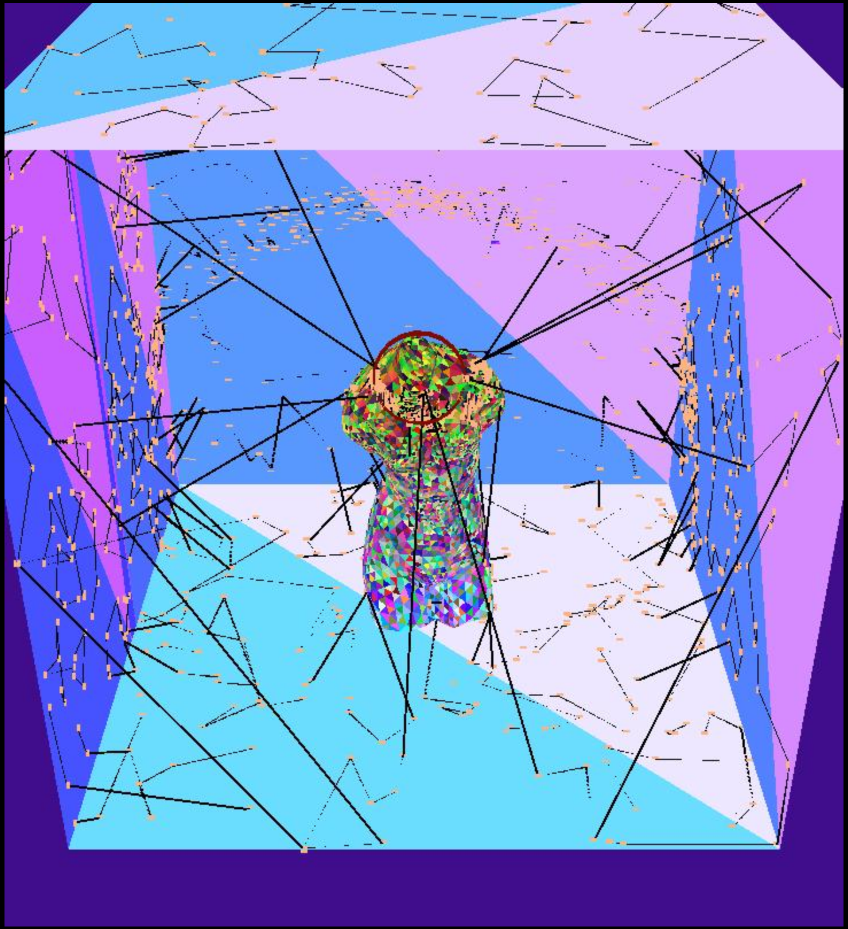
- Difficult to do efficiently without conditionals
- Generating unique collection may reduce size of candidate set
- Alternative: Perform calculations on duplicated photons anyhow, but ignore their contribution by multiplying them by zero

Future Work

- Approximation dilates radius of bounding sphere/disc of nearest neighbours
- Experiment with other density estimators that may be less sensitive to such dilation
  - Average radius
  - Variance of radii

Future Work

- Hilbert curve encoding probably not optimal clustering strategy



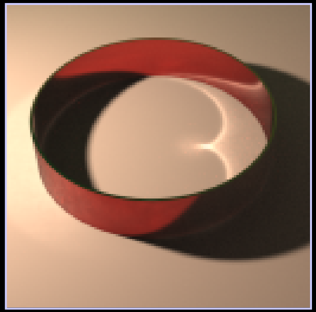
➢ Investigate alternative clustering strategies

Conclusion

- Block Hashing is an efficient, coherent and highly parallelizable approximate k-nearest neighbour scheme
- Suitable for hardware-assisted implementation of Photon Mapping

Thank you

http://www.cgl.uwaterloo.ca/Projects/rendering/



vma@cgl.uwaterloo.ca

mmccool@cgl.uwaterloo.ca