The RADIANCE Out-of-Core Photon Map
— Technical Report —

Roland Schregle (roland.schregle@{hslu.ch, gmail.com})
CC Envelopes and Solar Energy
Lucerne University of Applied Sciences and Arts

Revision
May 23, 2016

Abstract

Being an Account of the famous Exploits undertaken by the merry Crew aboard CC EASE in their Quest for more Photons, wherein is describ'd a Method of providing same in large(r) Numbers as requir'd.

Contents

1 Introduction

The RADIANCE photon map has recently been extended to support annual daylight simulations using contribution photons, and integrated into the official RADIANCE software distribution [Sch15]. Initial results using contribution photon mapping to compute daylight coefficients for DRCs with strong redirection indicated – unsurprisingly – that the number of required photons to adequately predict lighting levels (e.g. to evaluate daylight autonomy) scales linearly with the number of timestamps / light sources [SBGW15].

Increasing the number of photons raises the memory footprint of the photon map in the simulation, as the photons are maintained *in-core* (iC), i.e. they all reside in physical memory. Once this memory is exhausted, the operating system swaps memory pages to hard disk. This can severely degrade performance, and in extreme cases (termed *thrashing*), the simulation may grind to a virtual standstill as the system is entirely occupied with disk I/O. Tests showed that the iC photon map fails to build beyond 500M photons with 8Gb RAM available [SGW16].

This motivated the development of an *out-of-core* (ooC) photon map that maintains its photons entirely on disk. By loading photons on demand, the memory footprint is reduced to those photons which actually contribute flux to the illuminance sensor points, or to the surfaces visible in a rendered luminance map. By doing so, complex annual daylight simulations with large photon maps can be accommodated on commodity office PCs.

While the iC photon map uses a traditional k-d tree data structure [Ben75] to locate photons spatially, the ooC photon map uses a simpler octree [Mea80] that facilitates single-pass construction. The iC photon map is however retained and can be enabled via a compile-time option, as it is still suitable for smaller photon maps; tests showed that ooC has negligible benefit below 200M photons [SGW16].

In the course of revisions to the photon distribution algorithm used by ***mkpmap*** to transparently support the two optional data structures via interface routines, a long overdue parallelisation was included. Like RADIANCE, this was implemented using standard, portable UNIX ***fork()*** calls to spawn child processes, with file-based memory mapping to relay statistics back to the parent. Results of the arallel scalability were published in [SGW16].

2 Overview

Figure 1 gives an overview of the ooC photon map components and their dependencies. Photon maps are generated by ***mkpmap***, while lookups are performed by ***rtrace***, ***rpict***, ***rvu*** and ***rcontrib*** (hence-

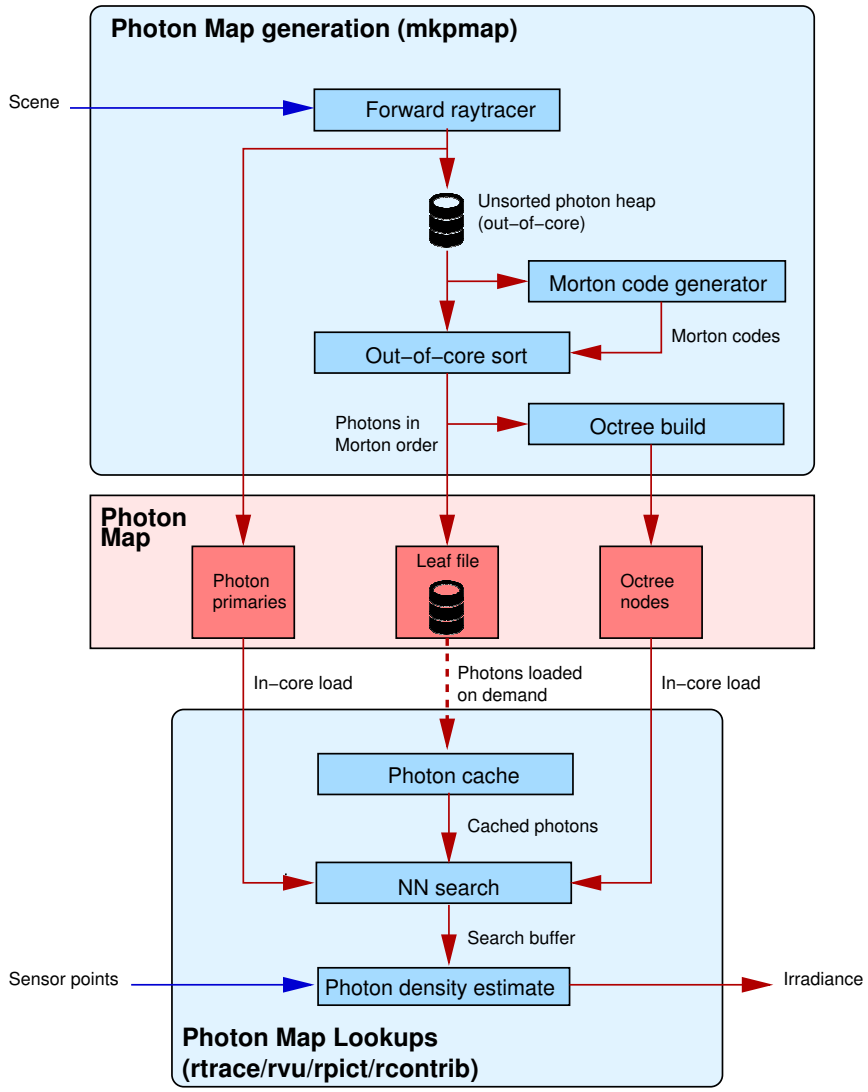

Figure 1: *Overview of ooC photon map components. Data flows along the red arrows, inputs are denoted by blue arrows.*

forth referred to as ***rtrace*** for brevity) when evaluating irradiance or coefficients/contributions at sensor points. Photons are generated by the forward raytracer and written to an out-of-core heap file. This file is then sorted according to Morton order into a *leaf file*, corresponding to the leaves of an octree. The octree is then built from the leaf file. These processes are described in detail in the following sections.

Note figure 1 shows the photon map as consisting of an octree with separately stored leaves containing the photons. In addition, contribution photon primaries are stored in a separate array. During photon map lookups, the primaries and octree nodes are maintained in-core, while the actual photons at the octree leaves are stored in the leaf file on disk. The leaf file is identified by a **.leaf** suffix to the name of the main photon map file. The main file contains the octree nodes, primaries, and all metadata, while the leaf file comprises the bulk of the photon data volume and is loaded on demand via a dedicated cache. The octree nodes supplement the photons in the leaf file by indexing them and thereby accelerating lookups.

3 Photon Map Generation

The ooC and iC photon map share the same forward raytracing algorithm, which is agnostic to the underlying data structure [Sch15].

Each deposited photon is stored in an out-of-core unsorted heap file on disk along with its position, normalised flux, and its surface normal (to arbitrate backlighting and occlusion). In addition, contribution photons contain an index to their primary, which in turn identifies the emitting source and incident direction to support annual simulations [SBGW15].

The parallelism inherent in the forward raytracing step is leveraged due to the fact that photon paths are independent. The number of photons to trace is evenly distributed among multiple processes, which atomically write to a shared heap file. I/O contention is reduced by locally buffering the photons generated by each process, and flushing these regularly. The buffers are randomly sized for each process to temporally decorrelate the flushing events.

Once forward raytracing is complete, the ooC photon map – implemented as a disk-based octree – is built to efficiently locate photons. Its construction is largely based on the method described in [KTO11]. The algorithm consists of two steps: photon sorting and octree node construction; with the presorted photons, the latter is reduced to a single-pass process.

3.1 Photon Sorting

In preparation to octree construction, the photons in the unsorted heap file are externally sorted according their *Morton codes* (see next section). External sorting algorithms for large datasets are a well studied area in computer science. The ooC photon map employs an external mergesort [Knu98, SH10] to enumerate the photons in Morton order. This involves recursively subdividing the unordered heap file into smaller subblocks until their size falls below a threshold which can be quicksorted in-core and in parallel. The sorted blocks are written out to separate temporary files which are then merged into progressively larger blocks as recursion unwinds.

3.2 Morton Code Generation

The Morton code represents a linearisation of the photons' coordinates within the scene geometry. It does so by mapping a photon's 3D Cartesian floating point coordinates $\vec{p} = [x, y, z]$ to a scalar integer $f_{M,i}(\vec{p})$, where i denotes the resolution of the code in number of bits per dimension. A $3i$-bit Morton code is thus defined as:

$$f_{M,i}(\vec{p}) : \mathbb{R}^3 \to \mathbb{N},\ [x,y,z] \quad \to \quad (Z_{i-1}Y_{i-1}X_{i-1}\ldots Z_0Y_0X_0)_2\,, \tag{1}$$

$$[X,Y,Z] \quad = \quad \left\lfloor \frac{2^i-1}{s}\,(\vec{p}-\vec{o}) \right\rfloor. \tag{2}$$

Here $\vec{p}$ is mapped to a vector $[X,Y,Z]$ of i-bit integers. This requires normalising by the size s of the axis-aligned bounding box containing the entire photon set (defining the size of the octree). Note that we subtract its origin $\vec{o}$, which can be arbitrary in a RADIANCE scene description. The bits comprising $[X,Y,Z]$ are then 3-way interleaved, denoted here as the binary representation $(Z_{i-1}Y_{i-1}X_{i-1}\ldots Z_0Y_0X_0)_2$. Note that this mapping incurs a discretisation error proportional to $2^{-i}s$ per dimension.

Graphically, the Morton code corresponds to a Z-curve which recursively traverses all octants of an octree of up to i levels (see figure 2 for the first two levels). Consequently this specifies the order in which photons are encountered at the octree leaves during a traversal, and the file containing the sorted photons becomes the octree's leaf file. Of course, depending on the photon distribution, only a subset of the 2^{3i} possible Morton codes will actually be assigned. Note that Morton codes are not stored with the photons, but generated on the fly using an optimised bitmask method [BLD13] instead of bitwise iteration. (No, we didn't understand the derivation of the bitmasks either...)

The current ooC implementation uses 63-bit Morton codes with $i = 21$ bits per dimension, which provides sufficient resolution for the photon maps used in beta tests with various DRC case studies. Based on the leaf file, the octree nodes are generated to index the photons as described in the folowing sections. The leaf file forms the bulk of the photon map's data volume and remains consistently out-of-core during photon lookups performed by ***rtrace***.

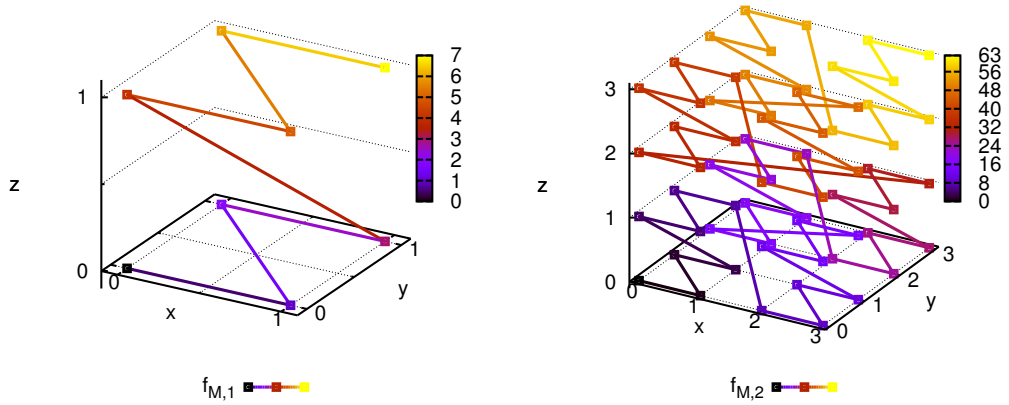

Figure 2: *Space subdividing Z-curve at levels 1 (left) and 2 (right). At level i, the curve subdivides the space into 8^i suboctants. By traversing the suboctants in a fixed Z-order pattern, the curve linearises their 3D Cartesian coordinates into discrete scalar indices. Each index corresponds to a suboctant's position along the length of the curve, and is referred to as its Morton code $f_{M,i}$.*

3.3 Octree Data Structure

The ooC photon map is organised as a sparse octree with photons stored (albeit not physically) at the leaves. Figure 3 shows the layout of an internal and leaf node. The node type is defined by the t (type) field, where $t = 0$ identifies an internal node, and $t = 1$ a leaf.

Each internal node contains a 32-bit counter n (num) quantifying the number of photons contained in its subtree. These counters are accumulated during tree traversal to index the photons in the leaves, which constitutes an implicit addressing scheme. Thus defined, the nodes can address a maximum of ca. 4.3G photons.

An internal node's children are indexed linearly with a 31-bit k (kid) field. This is the index to the first non-empty child node in Morton order, which is immediately followed by its siblings. This too constitutes an implicit node addressing scheme. An 8-bit bitmask b (branch, binary $b_7b_6b_5b_4b_3b_2b_1b_0$) indicates the presence of a child node for each octant $i \in [0,7]$ if b_i is set, i.e. $b_i \wedge 2^i \neq 0$.

In contrast, a leaf contains 8-bit counters $n_{0\ldots7}$ for the number of photons in each of its suboctants, which allows more selective testing during lookups. Unlike the counters for internal nodes, some of these may be zero.

Figure 4 shows the logical and internal physical layout of the ooC octree for a sample photon distribution. The octree nodes are stored linearly in a node array in postfix order, i.e. with the nodes reversed, and the root being the last node. This is a consequence of the single-pass octree construction algorithm detailed in section 3.4. The k field in each internal node refers to the first non-empty child node as index into the node array.

The octree node structure is sparse since an irregular photon distribution (as can be expected

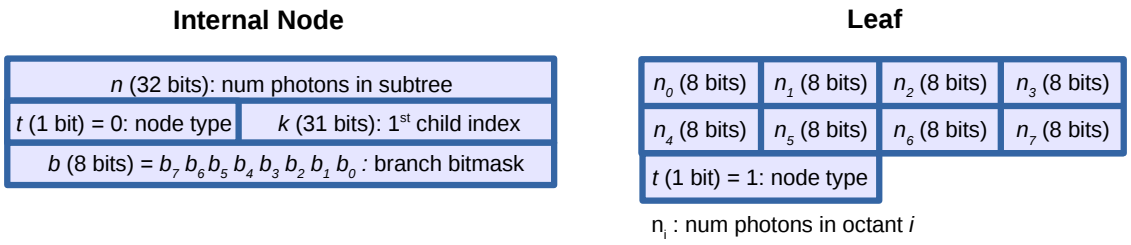

Figure 3: *Physical Layout of internal node (left) and leaf (right) in ooC octree. Each node corresponds to a bounding box in the octree containing 8 octants, with k pointing to the child node for the first non-empty octant; empty octants are omitted. An 8-bit bitmask b in the internal node indicates the presence of a child node for octant i if bit b_i is set. The 32-bit photon count n in an internal node indicates the total number of photons contained in the bounding box, including its subdivided octants, which corresponds to the subtree rooted at the node. A leaf breaks the photon count further down per octant with 8-bit counters $n_{0...7}$ for more efficient lookups.*

with caustics) would result in many redundant nodes containing no photons. The octree is therefore compacted by only storing non-empty nodes, implying n>0 for all internal nodes. An internal node therefore only references non-empty child nodes. This organisation is summarised in figure 4 for an octree with leaf capacity $N_{l,max} = 4$. Note that this is an unrealistic and inefficient example, as the low leaf capacity results in more nodes, thus mitigating the benefit of the sparse layout. In the current implementation, the ooC octree has a leaf capacity of $N_{l,max} = 255$, which uses the full 8-bit range of each per-octant counter in the leaves (note that a leaf could store more if the photons are distributed over the octants, but this is not guaranteed).

Photons in the leaf file are indexed using an implicit addressing scheme during octree traversal. When descending into a subtree rooted at a given node, the photon counters n of its lower numbered siblings in Morton order (i.e. left of a node in the logical and physical layout) are summed to obtain an offset into the leaf file for the first photon contained in that subtree. This index is accumulated recursively when descending into the subtree until a leaf is reached. A leaf's per-octant counters are used similarly to index photons in a particular octant. This addressing scheme is used during photon map lookups as described in section 4.

3.4 Octree Node Construction

The octree nodes are generated from the leaf file using the efficient single pass algorithm described by Kontkanen in [KTO11]. With the photons in Morton order, this algorithm obviates a top-down insertion from the octree root for each photon, which requires multiple traversals. Instead, the nodes are constructed bottom-up from the leaves, and visited only once.

The octree build is parametrised by a maximum number of photons per leaf $N_{l,max}$, and a maximum tree depth d_{max}. Note that d_{max} must not exceed the Morton code resolution in bits per dimension i, otherwise photons that belong to different leaves will map to the same Morton code, and consequently be assigned to the same leaf, thus corrupting the octree.

Kontkanen's algorithm uses a queue of size $N_{l,max} + 1$ that traverses the leaf file in a single pass. Figure 5 illustrates a simplified construction for a 2D quadtree, with the same principles applying to the octree. With the photons sorted in Morton order, they already have the ordering of an octree traversal imposed on them. A leaf is generated when the photon at the queue tail lies outside the leaf's bounding box. In this case, all photons inside the bounding box are dequeued and assigned to the leaf; **Morton ordering guarantees no further photons within this node will be encountered**. On the other hand, if the photon at the queue tail also lies in the bounding box (implying the same for all queued photons), an internal node is generated and subdivided into 8 child nodes (one for each suboctant) as the leaf capacity is exceeded by definition of the queue length. The construction then continues recursively in

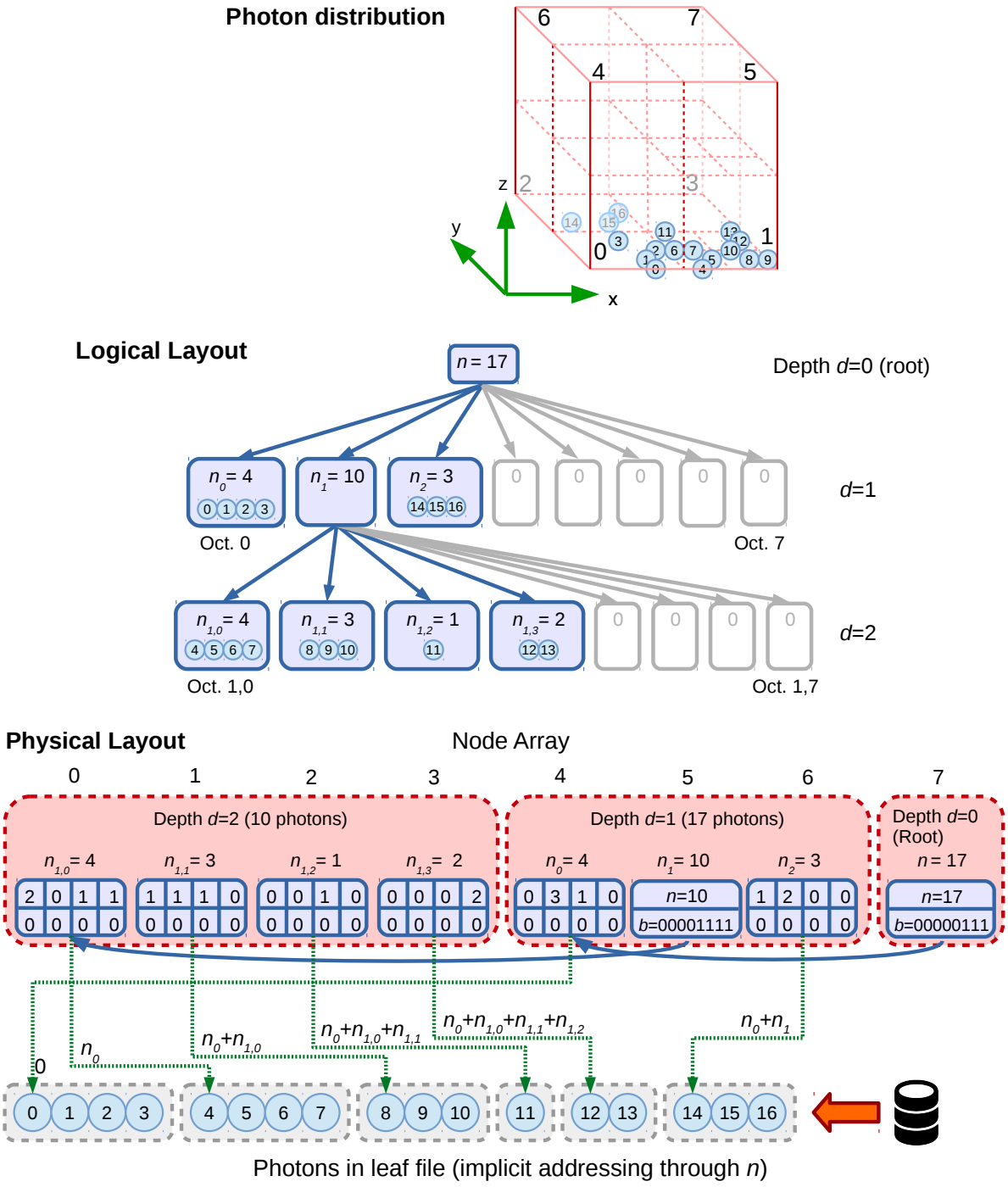

Figure 4: *Logical and corresponding physical layout an ooC octree with a leaf capacity of $N_{l,max} = 4$ for a simplified photon distribution in the $Z = 0$ plane. Note the photons are numbered in Morton order, and the octants are similarly numbered as indicated for the top level bounding box (red solid lines) in the distribution. The photon count n_i corresponds to octant i and may be doubly indexed for suboctants (e.g. for the subdivided octant 1). Only non-empty nodes are stored (faded in the logical representation). The physical representation indicates the presence of a child node for octant i if bit b_i of the branch bitmask b is set. Internal nodes physically index their first child node in Morton order (blue arrows), which is immediately followed by its siblings. Photons physically reside in the leaf file in Morton order and are indexed by accumulating the photon counters n_i during traversal (green arrows). Note that the physical layout reverses the linear ordering of the nodes, i.e. from maximum depth to the root.*

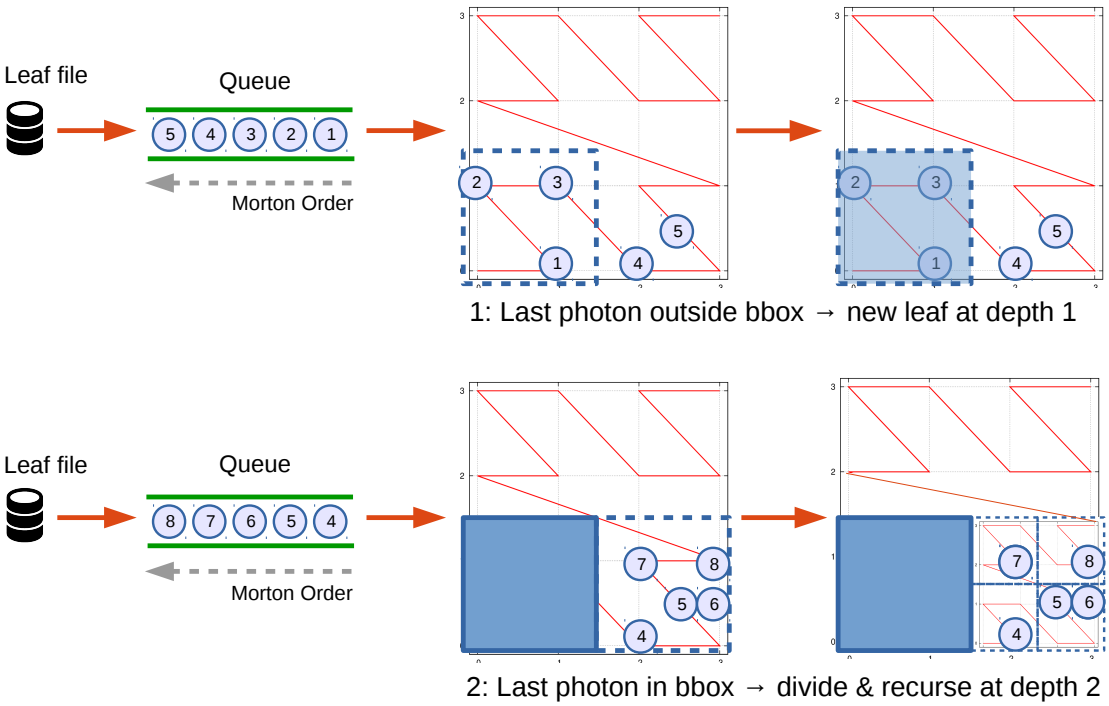

Figure 5: *Single-pass ooC octree construction, illustrated for a 2D quadtree for clarity. The current node's bounding box is superimposed in blue dashed lines, with the origin in the lower left. Photons presorted in Morton order (corresponding to the red Z-curve in the quadtree) are read from the leaf file into a queue of length $N_{l,max}+1$, where the leaf capacity $N_{l,max}=4$ in this example. In step 1, the last photon in the queue lies outside the current bounding box, and a new leaf is generated containing all photons in the bounding box, dequeueing these in the process. In step 2, the construction proceeds to the next sibling node in Morton order, and the queue is refilled. In this case, the last photon lies within the bounding box and therefore exceeds the leaf capacity; consequently, a new internal node is generated and subdivided, with the construction recursing in the subquadrant child nodes.*

the child nodes in Morton order after replenishing the queue.

The maximum leaf count $N_{l,max}$ is subject to the limits of the node data structure but also provides a trade-off between the number of nodes traversed during a lookup (lowering $N_{l,max}$ results in a deeper tree as more internal nodes need to be subdivided) and the cost of traversing all photons in a leaf. **A tree can overflow when a node at depth d_{max} needs to be subdivided. This can theoretically happen with extremely dense photon maps, and would require the photons to be merged and stored in a leaf at this depth. This issue has not yet been encountered, and no code is currently in place to deal with it; octree construction simply fails. This is left as future work.**

The construction algorithm also builds the bookkeeping information (i.e. photon counters, see section 3.3) embedded in each node for implicit addressing during octree lookups. The octree nodes are linearly stored in the node array, with parent nodes being appended to their children in the node array. The bottom-up construction therefore results in a post-order traversal of the tree nodes. Consequently, the nodes are stored reversed, and **the octree root is infact at the end of the node array**.

The octree nodes and primaries are saved to the main photon map file which is loaded in-core for lookups, and therefore separate from the out-of-core leaf file. The octree nodes specifically serve to index the photons and are maintained in-core to facilitate photon retrieval.

4 Nearest Neighbour Search

In order for ***rtrace*** to evaluate the irradiance or contributions from the photon map for a given sensor position, we locate the k_p closest photons using a nearest neighbour (NN) search in the octree.

A first implementation using a progressively refined NN search proposed by Connor and Kumar [CK10] yielded poor performance and was significantly slower than the iC k-d tree, which uses Bentley's original tree traversal algorithm [Ben75]. Connor and Kumar's method operates exclusively in Morton space and traverses the leaf file directly. Its access patterns tend to be highly irregular as they can cover very large intervals of the Z-curve, resulting in poor cache performance and excessive disk activity. While tests with disk-based data using the operating system's memory mapping mechanism are documented in [CK10], the authors make no explicit claims of their method's fitness for out-of-core applications.

Having abandoned Connor and Kumar's method, a simpler traversal similar to that detailed in [BSC15] was implemented, using the in-core octree nodes to index the photons in the leaf file.

During the tree traversal, the photons in the leaf file are indexed implicitly via the photon counters n in the internal octree nodes (see section 3.3). These counters are accumulated for pruned subtrees during the traversal (i.e. those whose bounding boxes lie outside the maximum search radius corresponding to the distance to the furthest found photon). The accumulated photon counters yield a photon's index once a leaf is reached; this index is then simply the position of the photon's data record in the leaf file. This process is summarised in figure 6.

The NN search passes the found photons for evaluation by the standard photon density estimate [Sch15]. With the iC photon map, these are returned as in-core references (pointers). In an out-of-core context, returning external references as indices into the leaf file is undesirable as this may incur additional disk activity if the corresponding pages are no longer cached when the photons are evaluated. The ooC implementation therefore redundantly stores the found photons in-core in a search buffer, whence they are immediately accessible without additional I/O.

5 Photon Cache

Photons are loaded on demand from the leaf file during photon map lookups. The goal of minimising disk access and I/O latency is critical for the performance of any out-of-core implementation. Towards this end, the ooC photon map employs a customised photon cache which retains a subset of the photons in-core. The cache is transparent when accessing photons in the octree, and implemented in similarly general fashion to the latter in the sense that it is agnostic to the type of data it contains; like the octree, it refers to the photons simply as records of a given size.

Figure 7 shows the internal layout of the cache. It maintains a lookup table referring to fixed-size blocks of consecutive photons read from the leaf file residing in core memory. These blocks are referred to as *pages* in operating system terminology (where caching plays a central role), while the lookup table is referred to as a *pagetable*. The pagetable is accessed using standard open hashing techniques.

The page table entries simultaneously form nodes in a doubly linked list indicating the page access sequence, with the most most recently used (**MRU**) page at the head, and the least recently used (**LRU**) page at the tail. As hashing is an entropic process, the ordering of the pagetable entries does not generally coincide with the access sequence, necessitating the linked list.

5.1 Parametrisation

The cache is dimensioned by the number of records per page n_r, the number of pages n_c currently in the page table (and thus residing in core memory), and the page table capacity N_c. The user can adjust the cache size to the available physical memory via the **-ac** and **-aC** options to ***rtrace***.

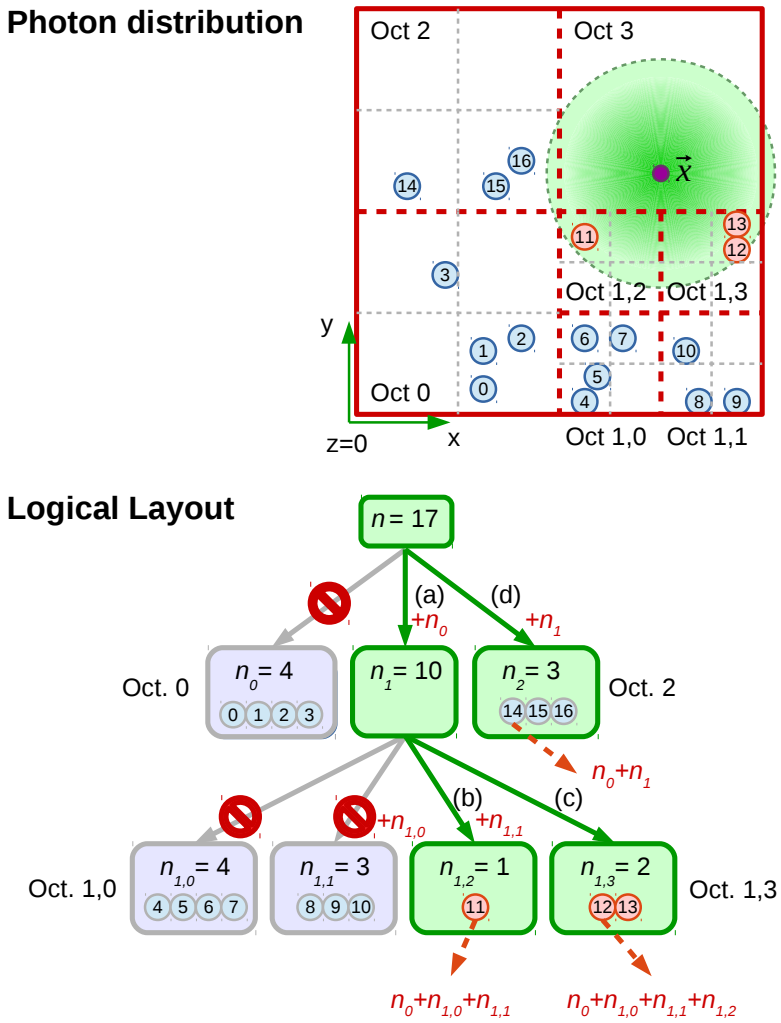

Figure 6: *Nearest neighbour search in a simplified ooC octree with photons lying in the $z = 0$ plane. The search radius (green) overlaps octants 1…3 in the photon distribution, and encloses photons (red) in suboctants 1,2 and 1,3 of octant 1. Nodes are traversed in Morton order (left to right in the logical representation) and depth first, as indicated by the green path annotated (a)–(d). The search prunes all nodes whose octants lie outside the current bounding box and adds their photon counts n_i (red offsets along traversal path) to an accumulated index to the first photon in the octant under inspection (red arrows at leaves). Octants in a leaf which lie outside the bounding box are similarly handled. Note that the empty octant 3 is absent and implicitly pruned. Photons in an overlapping leaf octant are linearly tested for inclusion in the search radius in Morton order. The photons in octant 2 fail this test, even though the octant is still traversed.*

The **-ac** option sets the number of photons per page n_r expressed as a multiple of the number of photons k_p for NN lookups in the photon map. This is considered more convenient to the user, as the page size depends on k_p. This is because NN lookups tend towards local access patterns within the same page when the octree traversal reaches the leaves. By enlarging n_r, the user can reduce the frequency of on-demand loading during the NN search at the expense of longer loading times per page for large values of k_p. Scalability tests with the ooC photon map cache revealed that the pagesize can have a significant effect on performance [SGW16], as evidenced by the benchmark results in figure 8. Acceptable values for **-ac** lie in the range 4.0–8.0.

The **-aC** option sets the overall size of the cache in number of photons. From this, the pagetable capacity is derived by dividing by the pagesize n_r. Note that this is internally rounded to the nearest

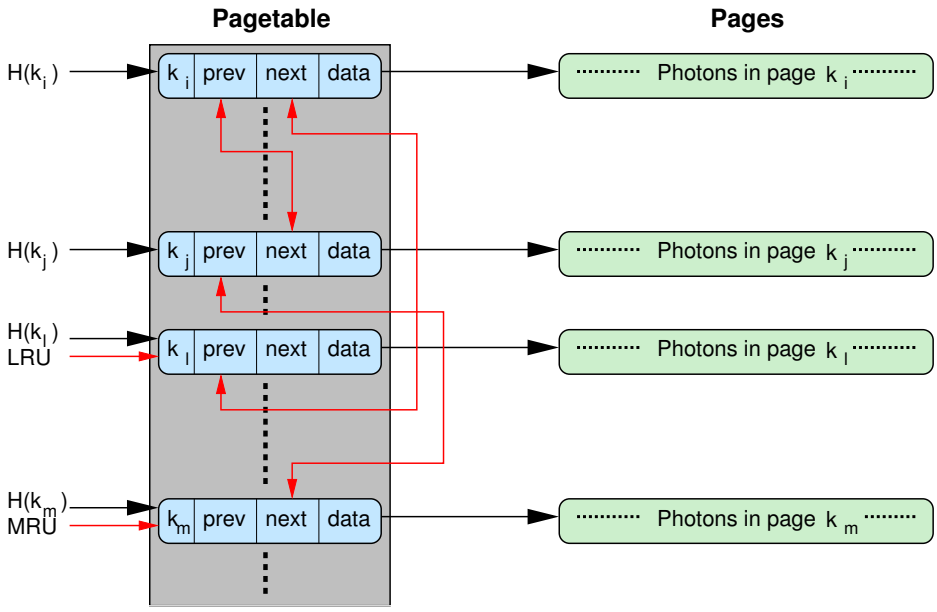

Figure 7: *Internal layout of ooC photon cache. The pagetable contains an entry for each page residing in core memory, with a* ***data*** *field pointing to the photons in that page. Pagetable entries are accessed by a photon's key k via a hashing function $H(k)$. The most recently used page is identified by* ***MRU****, which is the head of a doubly linked list embedded in the page table entries via the* ***prev*** *and* ***next*** *pointers. The tail of this list,* ***LRU****, points to the least recently used page. This list orders the resident pages according to their access sequence (red arrows) and does not generally coincide with the order of the pagetable entries; in this example, page keys k_m and k_l are the most resp. least recently used.*

prime number for improved hashing, and the cache size is therefore only approximate. This parameter has a surprisingly minor effect on performance compared to the pagesize, and good results can already be achieved with only 10000 cached photons (see figure 8). Obviously this parameter should not exceed the amount of physical memory available, otherwise the cache itself is paged by the operating system and thus forfeited.

5.2 The Hashing Function

In the absence of hashing collisions (see next section), pagetable lookups via a good hashing function are O(1) and thus occur in constant time. This is clearly optimal for a cache whose performance is critical to the entire system. Hashing is a well-studied and fundamental field in computer science, and numerous variants have been proposed.

The hashing function $H(k)$ takes a page key k to generate a corresponding pagetable index. The key k for a given record index i (i.e. a photon's offset in the leaf file in the context of the ooC photon map) is the base address of the page containing that photon. The cache uses the simple hashing function:

$$H(k) = km \bmod N_c, \quad k = \left\lfloor \frac{i}{n_r} \right\rfloor, \tag{3}$$

where m is a large prime number to decorellate H for successive values of k. Ideally, H should map uniformly to the entire page table.

The left plot in figure 9 is exemplary for a poor hashing function, as it does not allocate pages uniformly, but clusters them instead, leading to frequent hashing collisions (see next section). The right plot exhibits a near-optimal, even distribution using the hashing function in equation 3.

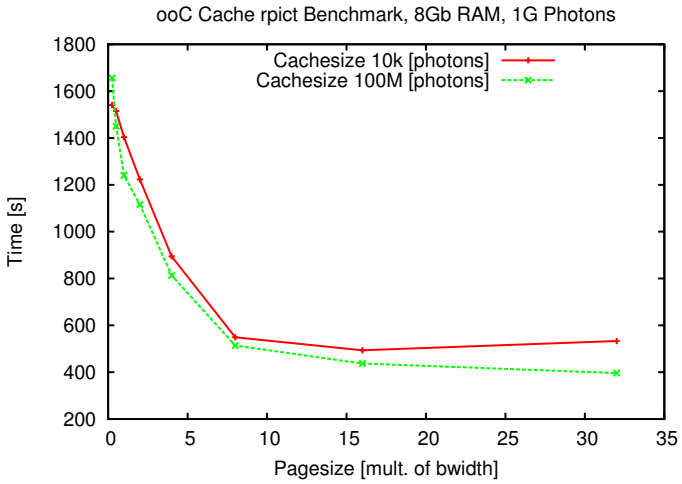

Figure 8: *Effect of ooC cache pagesize (as multiple of photons per NN lookup) and overall cache size in photons.*

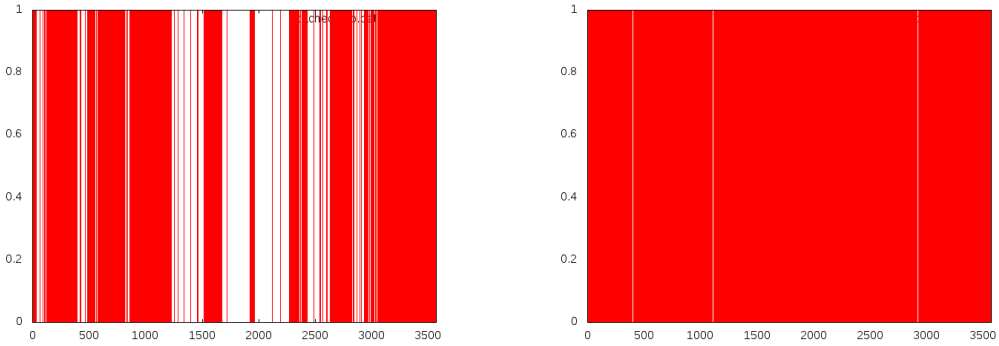

Figure 9: *Cache pagetable population plots (0: vacant, 1: occupied) with different hashing functions. The hashing function on the left is suboptimal as it does not allocate pages uniformly, while the function on the right is near-optimal.*

For best hashing performance, N_c should also be prime. The ooC photon map implementation rounds this parameter to the nearest prime, if necessary.

5.3 Hash Collisions

The hashing function is surjective, i.e. a many-to-one mapping, meaning that multiple keys can map to the same pagetable index. When a key is mapped to a pagetable entry already occupied by another page, a *hash collision* occurs. A poorly chosen hashing function can dramatically degrade performance due to frequent hash collisions; a good hashing function minimises hash collisions by using sufficiently large prime numbers for entropy.

Collisions can be further reduced by providing adequate free entries in the pagetable. A maximum load factor $\alpha < 1$ defines the maximum allowable fraction of occupied pages n_c/N_c. The load factor governs the tradeoff between pagetable capacity and hash collision frequency, since the latter increases as the table fills up. By using a maximum load factor, the table is never completely filled to reduce the likelihood of collisions, as it guarantees a constant number of empty slots for new pages when others are removed.

When a collision does occur, a linear probe for the next pagetable entry containing k (when a page

is accessed) or for the next free pagetable entry (when a new page is loaded) is made:

$$H'(k) = (H(k) + 1) \bmod N_c. \tag{4}$$

With a well-chosen α, collisions are typically resolved in a few iterations of equation 4. An acceptable compromise between pagetable capacity and collision frequency has been achieved for $\alpha = 0.75$, which is borne out by the author's experience and cited in the literature.

5.4 Page Replacement Policy

The cache is initially empty and progressively filled with pages once records residing in disjunct pages are accessed. When a page is accessed, its corresponding pagetable entry is located via hashing, possibly after iteratively resolving hash collisions as described above. If an entry containing the key was found, the page already resides in-core (i.e. is cached, termed a *cache hit*), and the desired record within that page can be returned. The page table entry is placed at the head of the access sequence list and referred to as the new **MRU** page. This is analogous to removing an arbitrary item from a stack and placing it on top.

If an empty pagetable entry was found, the accessed page does not reside in the cache (termed a *cache miss*) and needs to be loaded from the leaf file. If the pagetable load factor is below α, a new pagetable entry is created at the found position. The page data is allocated, loaded from the leaf file, and linked to the pagetable entry.

As the pagetable fills due to cache misses, the load factor approaches α. Once this limit is reached, the page table is considered full in order to prevent aggravating hash collisions. This implies a page must be evicted (removed from core memory and the page table) before a new page can be loaded whenever a cache miss occurs.

Pages are evicted using a standard LRU (least recently used) page replacement policy [TB14] based on the access sequence list embedded in the pagetable. The **LRU** page is removed from the page table, but, for reasons of efficiency, its allocated data block is reused to avoid memory fragmentation and redundant allocation overhead. The page is loaded as a new page as described above and becomes the new **MRU** page. Freeing the evicted pagetable entry guarantees a constant load factor.

5.5 Pagetable Defragementation

An evicted page leaves behind an empty pagetable entry which disrupts the probing sequence in equation 4 to resolve hash collisions; the subsequent pagetable entries in the sequence will no longer be found.

Consequently, a page eviction is automatically followed by a defragmentation that closes the resultant gap in the pagetable by iteratively shifting the next entry down. The sequence ends when the next empty entry is found, marking the end of a hash collision probe sequence.

Alternative methods exist that mark deleted pagetable entries for reuse for newly loaded pages with "tombstones". These markers serve to indicate that a hash collision probe sequence should continue. While simpler than defragmentation, this can quickly case the pagetable to become polluted with tombstones. This condition leads to increasingly longer hash collision probe sequences, which in turn dramatically degrades performance to the point where the cache spends more time with bookkeeping than actual paging. Again, the remedy is to regularly clean up the pagetable, which essentially incurs more overhead over time than immediate defragmentation after deletion. Tests conducted with the two methods revealed that defragmentation is the preferred approach in terms of performance, and benefits code transparency.

5.6 Optimisations

The cache is optimised to bypass page table lookups (including hashing and potential collision resolution) and all associated MRU bookkeeping if the same page is accessed in repetition; this occurred with over 90% of cache lookups during beta testing and benchmarking [SGW16].

When running ***rtrace*** with the **-n** option, parallel lookups contend for simultaneous access to the octree leaf file, which would further aggravate I/O latency. For this reason, each process maintains a private cache instance. Note that option **-aC** then refers to the number of cached photons per process, and thus the effective cache size scales with the number of processes.

6 Software Architecture

Figure 10 provides an overview of the RADIANCE photon map's software architecture after integration of the ooC photon map alongside the existing iC one. The blue blocks correspond to software modules built from C source files with the same name. Red arrows between the modules correspond to calls between modules. The (major) routines exposed by a module are summarised in red text inside the corresponding block.

Three major scopes are identified in figure 10: the basic photon mapping modules (photon map base) and the iC and ooC backends they interface to. In the following sections we detail the functionality of each module and its major routines.

6.1 Compilation

The iC and ooC data structures cannot coexist in the current implementation and are linked to the photon map base via interface routines at compile time. This linkage is subject to the **PMAP_OOC** macro; if defined, the ooC photon map is built, otherwise the iC photon map is built by default.

The ooC photon map can be built with the following command in the **ray**/**src**/**rt** directory of the RADIANCE source tree:

```
rmake OPT+=-DPMAP_OOC
```

6.2 Photon Map Base

6.2.1 mkpmap

This is the main module for building the photon map, and calls **distribPhotons()** or **distribPhotonContrib()**, followed by **savePMaps()**, in the **pmap** module to build and save photon maps.

6.2.2 rtmain/rvmain/rpmain/rcmain

These are the main rendering modules in stock RADIANCE system. These call **loadPmaps()** in the **pmap** module and trigger photon map density estimates via the ambient calculation after rays intersect a diffuse object.

6.2.3 pmap

The main photon mapping module that groups basic functions accessed by **main()** in both the **mkpmap** and **rtmain** (etc.) modules. These functions include photon distribution (**distribPhotons()**), aggregate loading and saving of photon maps (**loadPmaps()**, **savePmaps()**), and density estimates (**photonDensity()**). The latter uses **findPhotons()** in the **pmapdata** module for NN lookups.

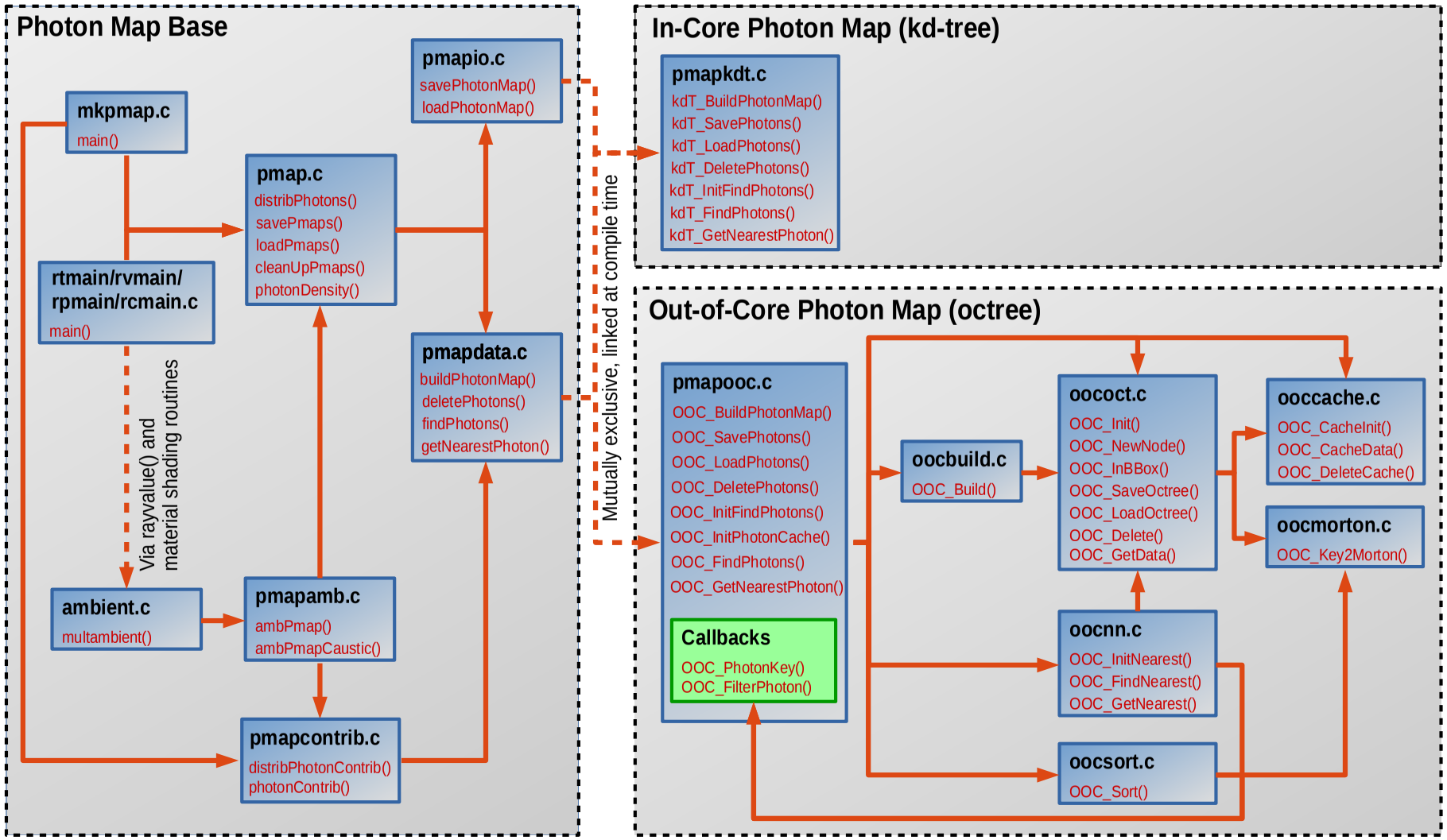

Figure 10: RADIANCE *photon map software architecture, comprised of the base photon mapping code and mutually exclusive iC and ooC data structures that are linked at compile time. Arrows represent the call graph between the modules.*

6.2.4 ambient

The ambient calculation module in the RADIANCE code that performs stratified Monte Carlo raytracing in **multambient()** to integrate the diffusely reflected illuminance. It is called via the various material shading routines. In photon mapping mode, this calls the density estimation wrappers in **ambPmap()** and **ambPmapCaustic()**.

6.2.5 pmapamb

Density estimation wrapper routines **ambPmap()** and **ambPmapCaustic()** called by **multambient()**. These serve as interface to RADIANCE's ambient calculation and call **photonDensity()** and **photonContrib()** in **pmap** and **pmapcontrib**, respectively.

6.2.6 pmapcontrib

Module specific to contribution photon map used by ***rcontrib***. Contains dedicated photon distribution and density estimation routines **distribPhotonContrib()** and **photonContrib()** for contribution photons.

6.2.7 pmapio

Module for photon map file operations. Individual photon maps are loaded/saved by **loadPhotonMap()** and **savePhotonMap()** using RADIANCE's portable I/O routines in the **portio** module. These routines save the general photon map metadata which is not specific to the underlying data structure; specialised routines for the latter are then called as **kdT_SavePhotons()** or **OOC_SavePhotons()** in **pmapkdt** and **pmapooc**, respectively, depending on the definition of the **PMAP_OOC** macro.

6.3 In-Core Photon Map

6.3.1 pmapkdt

All functionality specific to the k-d tree used in the iC photon map is encapsulated in the **pmapkdt** module, and is linked to the interface routines in **pmapio** and **pmapdata** at compile time when **PMAP_OOC** is undefined. The routines are not detailed here further as the focus is on the ooC photon map.

6.4 Out-of-Core Photon Map

6.4.1 pmapooc

This is the main module that exposes the ooC photon map to the base modules via interface routines which are linked at compile time when **PMAP_OOC** is defined. The majority of the routines in this module are simply wrappers which call the routines specific to the ooC octree detailed below. A special case are two functions acting as callback for the octree, which has no notion of the photon type it stores. **OOC_PhotonKey()** retrieves a given photon's key (3D position) when sorting photons in Morton order. **OOC_FilterPhoton()** tests a given photon for acceptance or rejection during NN search based on its normal and, in case of a contribution photon, its emitting light source.

6.4.2 oocsort

This module performs the external photon sort based on Morton codes to generate the octree leaf file, as detailed in section 3.1. The routine **OOC_Sort()** is called by **OOC_BuildPhotonMap()** in **pmapooc** and uses the **OOC_PhotonKey()** callback to obtain the photons' positions. These then serve as keys from which the Morton Codes are generated with **OOC_Key2Morton()** in the **oocmorton** module.

6.4.3 oocbuild

This module implements the octree construction algorithm detailed in section 3.4. It provides the **OOC_Build()** routine called by **OOC_BuildPhotonMap()** in the **pmapooc** module to generate the octree nodes from the leaf file after photon sorting. It uses several routines and macros defined in the **oococt** module, notably **OOC_NewNode()** to allocate a new octree node in the node array, and **OOC_InBBox()** to test photons for inclusion in the current bounding box.

6.4.4 oococt

This module provides the fundamental definitions and routines for the octree outlined in section 3.3. The data structures for an octree node **OOC_Node** and the octree container **OOC_Octree** are shown in listing 1. The space occupied by each field in **OOC_Node** is specified via bit fields. Internal nodes and leaves share the space allocated to a node by defining both as members **node** and **leaf** of a union. A **type** bit arbitrates how the fields are interpreted during access: **type**=0 identifies an interior node, while **type**=1 identifies a leaf . Both members occupy 9 bytes, which is padded to 12 bytes when compiling on the x86_64 platform.

The implementation is not specific to the photon map, and can maintain arbitrary records of a given size. It is therefore agnostic to the contents of the actual records, and requires a callback to access the keys within them. This callback function **key** is passed along with the record size in bytes **recSize**, the photon distribution's bounding box **bound** and its origin **org** to **OOC_Init()**. The photon key access function **OOC_PhotonKey()** in **pmapooc** acts as callback for the octree.

Besides node allocation and bounding box tests, the **oococt** module also provides the octree-specific I/O routines **OOC_SaveOctree()** and **OOC_LoadOctree()**, which are called by **pmapooc**'s wrapper routines **OOC_SavePhotons()** and **OOC_LoadPhotons()**. The octree-specific routines append or continue reading the octree node array after the generic metadata written or read by the generic routines **savePhotonMap()** and **loadPhotonMap()** in **pmapio**. Note the leaf file is saved separately in the **oocsort** module during photon sorting, but loaded in the **oococt** module.

OOC_GetData() is the fundamental routine to retrieve photons from the octree, which is used by the NN search module **oocnn** during its octree traversal. **OOC_GetData()**, in turn, transparently retrieves the photons via the cache by calling **OOC_CacheData()** in the **ooccache** module.

6.4.5 oocnn

This module implements the NN search for photon map lookups in the octree, as detailed in section 4. The routine **OOC_FindNearest()** is called by the wrapper **OOC_FindPhotons()** in the **pmapooc** module; the latter also lazily initialises the photon cache in preparation for octree traversal by calling **OOC_CacheInit()** in the **ooccache** module. **OOC_FindNearest()** is passed the **OOC_FilterPhoton()** callback in **pmapooc** to test photons for acceptance based on their normal and – in case of contribution photons – their emitting light source.

The **oocnn** module also defines a priority queue **OOC_SearchQueue** and **OOC_SearchQueueNode** which contains the k_p closest photons found by **OOC_FindNearest()**. Photons are sorted within the queue as a binary tree based on distance, with the furthest photon at the root. The photons are copied into a buffer contained in the priority queue (rather than their references) so their later retrieval will not potentially trigger cache misses and degrade performance if their containing pages have been evicted. The priority queue refers to the photons via their buffer indices; if the queue needs to be resorted, only the indices to the buffered photons are swapped, not the actual photons themselves.

Once the NN search returns, the photons are evaluated by **photonDensity()** in **pmap** (or **photonContrib()** in **pmapcontrib**) and retrieved via **OOC_GetNearestPhoton()** in **pmapooc**. The latter is a

```
/* Octree index & data counter types */
typedef uint32_t            OOC_NodeIdx;
typedef uint32_t            OOC_DataIdx;
typedef uint8_t             OOC_OctCnt;

/* Octree node field sizes for above */
#define OOC_NODEIDX_BITS    31
#define OOC_DATAIDX_BITS    32


/* Octree node */
typedef struct {
   union {
      struct {
         /* Interior node (node.type = 0) with:
            node.kid    = index to 1st child node in octree array (a.k.a
                          "Number One Son"), immediately followed by its
                          nonzero siblings as indicated by branch
            node.num    = num records stored in this subtree,
            node.branch = branch bitmask indicating nonzero kid nodes */
         char           type   : 1;
         OOC_NodeIdx    kid    : OOC_NODEIDX_BITS;
         OOC_DataIdx    num    : OOC_DATAIDX_BITS;
         uint8_t        branch;
      } node;

      struct {
         /* Leaf node (leaf.type = node.type = 1 with:
            leaf.num [k] = num records stored in octant k */
         char           type   : 1;
         OOC_OctCnt     num    [8];
      } leaf;
   };
} OOC_Node;


/* Top level octree container */
typedef struct {
   FVECT          org, bound;     /* Cube origin (min. XYZ), size, and
                                     resulting bounds (max. XYZ) */
   RREAL          size,
                  *(*key)(const void*); /* Callback for data rec coords */
   RREAL          mortonScale;    /* Scale factor for generating Morton
                                     codes from coords */
   OOC_Node       *nodes;         /* Pointer to node array */
                                  /* ************************************
                                   * **** NODES ARE STORED IN POSTFIX ****
                                   * **** ORDER, I.E. ROOT IS LAST!   ****
                                   * ************************************/
   OOC_DataIdx    numData;        /* Num records in leaves */
   OOC_NodeIdx    maxNodes,       /* Num currently allocated nodes */
                  numNodes;       /* Num used nodes (<= maxNodes) */
   unsigned       recSize,        /* Size of data record in bytes */
                  leafMax,        /* Max #records per leaf (for build) */
                  maxDepth;       /* Max subdivision depth (for build) */
   FILE           *leafFile;      /* File containing data in leaves */
   OOC_Cache      *cache;         /* I/O cache for leafFile */
} OOC_Octree;
```

Listing 1: Definitions for octree node **OOC_Node** and octree container **OOC_Octree** in the **oococt** module

wrapper for **OOC_GetNearest()** in **oocnn**, which knows how to access individual photons buffered in the search queue.

6.4.6 ooccache

This module implements the photon cache detailed in section 5. It provides routines to initialise the cache (**OOC_CacheInit()**, called by **pmapooc**), and the main cache access routine **OOC_CacheData()**. The latter is used by **oococt** to access the octree leaves.

6.4.7 oocmorton

This module implements the 3D key to Morton code mapping detailed in section 3.2 via the **OOC_Key2Morton()** function. This function is fundamental to the entire ooC photon map and used extensively. Since Morton codes are always generated on the fly, this code is highly optimised for performance; the bitwise interleave uses an O(1) bitmask method [BLD13] instead of iteration. This code is inlined per key dimension as macro in **OOC_BitInterleave()**.

7 Acknowledgements

This research was supported by the Swiss National Science Foundation as part of the project "Simulation-based assessment of daylight redirecting components for energy savings in office buildings" (#147053).

The author would like to thank his colleagues Lars Grobe, Andreas Noback, and Carsten Bauer for alpha testing of the ooC code on Linux and MacOS platforms. Particular thanks also go to Daniel Plörer, who extensively beta tested the code with DRC case studies during the final phases of our project.

Finally, the author thanks Greg Ward for the initial discussion about the ooC data structure during his first visit to our lab. He takes the credit for the octree with implicit addressing. In addition, we'd all like to thank him for his tireless efforts in still developing and supporting RADIANCE after a quarter century!

References

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[BLD13] Jeroen Baert, Ares Lagae, and Philip Dutré. Out-of-core construction of sparse voxel octrees. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 27–32, New York, NY, USA, 2013. ACM.

[BSC15] Jens Behley, Volker Steinhage, and Armin B. Cremers. Efficient radius neighbor search in three-dimensional point clouds. In *IEEE International Conference on Robotics and Automation, ICRA 2015, Seattle, WA, USA, 26-30 May, 2015*, pages 3625–3630, 2015.

[CK10] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbour graphs for point clouds. *IEEE Transactions on Visualisation and Computer Graphics*, 16(4), 2010.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[KTO11] Janne Kontkanen, Eric Tabellion, and Ryan S. Overbeck. Coherent out-of-core point-based global illumination. *Computer Graphics Forum*, 30(4):1353–1360, 2011.

[Mea80] Donald Meagher. Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer. Technical Report IPL-TR-80-111, Image Processing Laboratory, Renssalaer Polytechnic Institute, October 1980.

[SBGW15] Roland Schregle, Carsten Bauer, Lars O. Grobe, and Stephen Wittkopf. EvalDRC: A tool for annual characterisation of daylight redirecting components with photon mapping. In *Proceedings of CISBAT*, 2015. http://dx.doi.org/10.13140/RG.2.1.2145.8647.

[Sch15] Roland Schregle. Development and integration of the radiance photon map extension. Technical report, Lucerne University of Applied Sciences and Arts, February 2015. http://doi.org/10.13140/2.1.3332.9449.

[SGW16] Roland Schregle, Lars O. Grobe, and Stephen Wittkopf. An out-of-core photon mapping approach to daylight coefficients. *Journal of Building Performance Simulation*, 2016. http://dx.doi.org/10.1080/19401493.2016.1177116.

[SH10] Mir Hadi Seyedafsari and Iraj Hasanzadeh. Optimal external merge sorting algorithm with smart block merging. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 4(2):237–240, 2010.

[TB14] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014.

Appendix

A Development Timeline

February 2014: initial discussion with Greg Ward during his visit to Lucerne (see sketch in figure 11).

December 2014: analysis of current photon map software architecture and necessary changes.

Januar–April 2015: research on Morton codes, data structures for point clouds, and NN search.

May–September 2015: ooC software development, integration with current photon map software.

October–December 2015: working prototype, alpha testing by Carsten Bauer.

January–May 2016: beta testing by Daniel Plörer using annual simulations of DRC case studies.

June 2016: released as part of official RADIANCE software distribution.

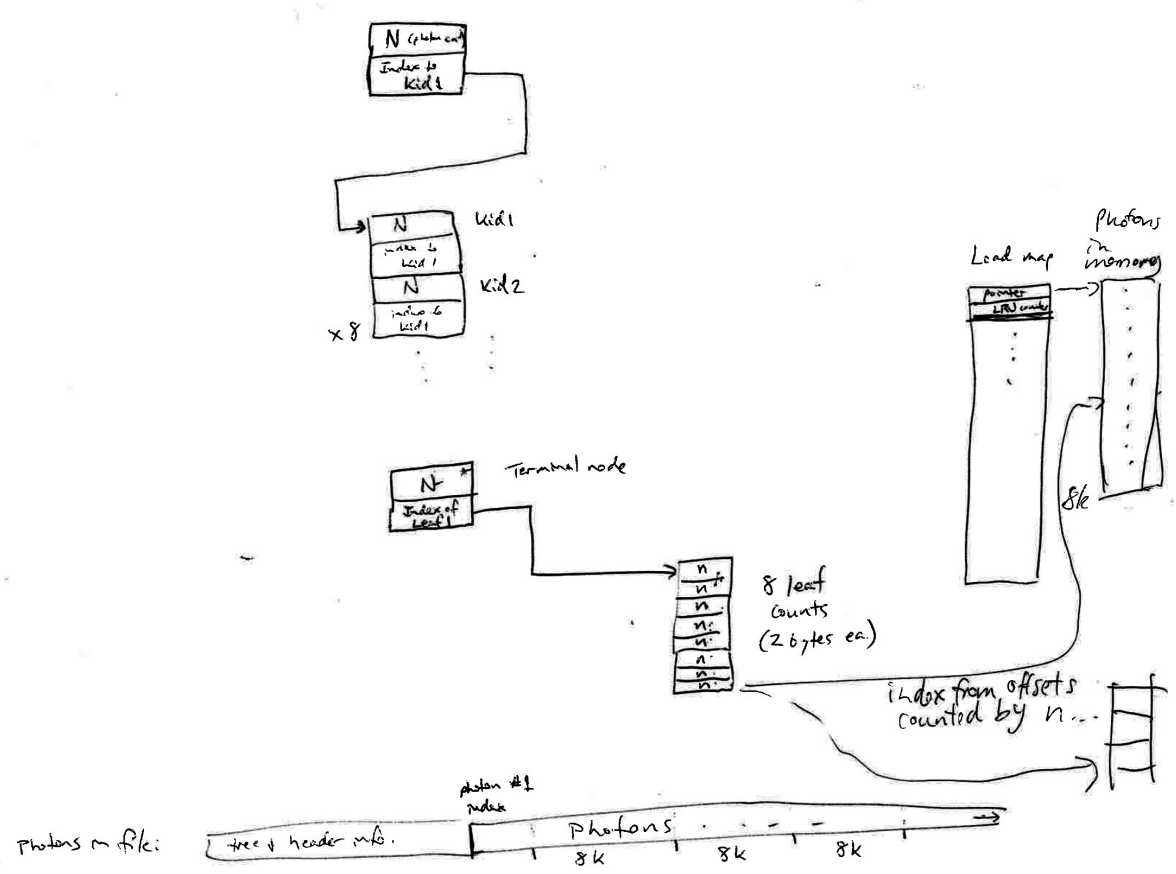

Figure 11: *Initial sketch of ooC data structure by Greg Ward, February 2014.*