Eurographics Symposium on Rendering 2011
Ravi Ramamoorthi and Erik Reinhard
(Guest Editors)

Volume 30 (2011), Number 4

Coherent Out-of-Core Point-Based Global Illumination

Janne Kontkanen† and Eric Tabellion‡ and Ryan S. Overbeck§

DreamWorks Animation

Figure 1: *A cityscape from the DreamWorks Animation movie "Kung Fu Panda 2". The out-of-core method described in our paper shaded the global illumination for the whole frame in 6 minutes 23 seconds. It used 128 million points, and it took an additional 4 minutes 18 seconds to build the out-of-core octree, resulting in 27 million octree nodes. The total amount of out-of-core data was 6.8 GB while the in-core point and node caches' memory usage did not exceed 2 GB. The cache hit rate was 99%. In Section 7, we demonstrate usage of up to 1.7 billion points (88 GB of data) within a 2 GB memory cap.*

Abstract

We describe a new technique for coherent out-of-core point-based global illumination and ambient occlusion. Point-based global illumination (PBGI) is used in production to render tremendously complex scenes, so in-core storage of point and octree data structures quickly becomes a problem. However, a simple out-of-core extension of a classical top-down octree building algorithm would be extremely inefficient due to large amount of I/O required. Our method extends previous PBGI algorithms with an out-of-core technique that uses minimal I/O and stores data on disk compactly and in coherent chunks for later access during shading. Using properties of a space-filling Z-curve, we are able to preprocess the data in two passes: an external 1D-sort and an octree construction pass.

1. Introduction

Point-based global illumination (PBGI) [Chr08] is an efficient method for computing smooth interreflections which are vital for photo-realistic rendering. PBGI is an alternative to ray tracing based approaches, such as Monte Carlo path tracing [Kaj86] or irradiance caching [WH92, TL04], and has been utilized in several DreamWorks Animation feature films, including "*Shrek Forever After*", "*Megamind*", and many others.

In its most basic form, PBGI is a level-of-detail (LOD) algorithm for simplifying both geometry and shading. As a preprocess, the geometry is finely sampled using a large number of points, which are shaded using direct illumination and a view-independent diffuse shading model. This point-based representation is then aggregated into a hierarchy, usually an octree, where each coarser level accumulates the geometry and outgoing radiance from the finer levels. At shade time, the octree is traversed to integrate the incoming radiance at each shade point.

However the point-set and the octree can become large, making it difficult to fit inside the resident memory of an affordable workstation or render farm server. The quality of the final shading is linked to the resolution of the point samples, thus production scenes require dense point-sets. As an example, rendering Figure 5c uses 927 million points, which results in an octree with 203 million nodes. The points themselves took 27.6 GB of disk-space and the octree an additional 22.7 GB.

† janne.kontkanen@gmail.com
‡ et@pdi.com
§ roverbeck@gmail.com

In order to meet the ever-growing demand for higher visual complexity, we need to constrain the memory used by the algorithm. However, a simple out-of-core modification of a classical top-down octree building approach would be incredibly inefficient due to large amount of I/O. Similarly, despite considerable amount of research in different types of octree build algorithms, no method has been published that is directly applicable to PBGI. In this paper we describe an efficient technique for out-of-core PBGI, which operates within a user-specified memory cap. Using this algorithm, the size of the point-set is only limited by hard drive's capacity.

Our primary contribution is an out-of-core octree build algorithm that requires minimal amount of I/O and stores both the points and the octree nodes coherently on disk. We first sort the points along a Z-curve, based on their Morton code [Mor66]. Utilizing the well-known property that the Z-curve exactly follows the depth first traversal, we build the PBGI octree in a single streaming pass over the sorted points. This algorithm builds upon an N-body simulation method developed by Salmon and Warren [SW97] and is described in detail in Section 4. As will be discussed in Section 4.2, we also introduce a technique to organize the octree nodes into coherent chunks, which can improve the final shading performance by more than 4×.

During final shading, we shade each point by performing a depth-first traversal of the octree, choosing the suitable LOD. We load the octree nodes on-demand to minimize memory use, and utilize a two-level LRU cache to scale to multiple processors. The portions of the octree not required for shading are never loaded into memory. The final shading stage is described in Section 5. Both octree building and final shading operate within a fixed memory cap. The result is a complete system for efficiently computing PBGI on huge point-sets.

2. Previous Work

Global Illumination: In the general sense, most global illumination (GI) algorithms are based on either Monte Carlo sampling using ray tracing or the finite element framework using radiosity. For a description of different global illumination techniques see the book by Dutré et al. [DBB02]. Irradiance caching [WH92, TL04] has been widely used in production rendering, especially before the introduction of PBGI. Some more recent related global illumination algorithms include multidimensional lightcuts [WABG06], and meshless hierarchical radiosity [LZT*08].

Monte Carlo sampling using ray tracing is generally considered the most robust approach to GI. Error appears in the form of noise, which is predictably reduced by simply adding more samples. Unfortunately, it can require a prohibitively large number of samples to reduce the noise to acceptable levels, and in animated sequences, the noise appears as a distracting buzzing.

We chose to extend the PBGI algorithm for several reasons. First, the pre-integration of lighting into clusters tends to give smoother results than ray tracing based approaches, given similar render times [KFC*10]. Additionally, PBGI's intrinsic LOD helps with the high geometric complexity of production rendering (see Figures 5 and 6 for example). For some applications, where accuracy is more important than smoothness, ray tracing may be preferable because PBGI does not converge to a physically accurate result, and its convergence is less predictable.

Point-Based Global Illumination: Point-based representations have a long history in graphics, first as a geometry for surface representation and primary visibility and more recently for global illumination [Bun05, Chr08, REG*09]. For an overview on the topic see Gross and Pfister [GP07].

Our approach is an efficient, parallel, and out-of-core implementation of Christensen's method [Chr08]. Similar to Christensen, our system creates a hierarchical point-based representation of the scene and stores it in an octree. Christensen mentions using an out-of-core representation of the octree for rendering, but does not describe construction or usage of the data structure.

GPU and Out-of-Core Global Illumination: Since Monte Carlo methods are common for computing GI, many techniques focus on moving raytracing out-of-core [BBS*09, PKGH97, CLF*03, KMKY10, YM06]. There are also a large number of methods to implement ray tracing on the GPU for faster convergence [PBD*10, GL10]. However, as far as we are aware, the only published system to do both is the *PantaRay*-system [PFHA10].

PantaRay is an out-of-core ray tracing engine for precomputing directional visibility. Similar to our system, they also use a two-level caching scheme to reduce mutex locking. Their system uses ray tracing with binary bounding volume hierarchies, and requires multiple GPUs to be efficient. Meanwhile, our system uses PBGI with octrees, and efficiently runs on the CPU.

Octrees and Morton Codes: We find that octrees are ideal for efficient out-of-core construction, storage, and traversal. Significant research has been published on out-of-core octrees (and quadtrees) in graphics and elsewhere. Several other works leverage Morton order for efficient traversal, in-core construction, and out-of-core storage. Tu et al. [TLO03] developed a general purpose library for manipulating large out-of-core octrees. Schaefer and Warren [SW03] and Cignoni et al. [CMRS03] both use octrees for large out-of-core mesh simplification, but both require the full octree to be stored in memory. Lauterbach et al. [LGS*09] use Morton codes to build general bounding volume hierarchies in parallel on the GPU.

Despite the long history of octrees and Morton codes, none of these construction algorithms suit our needs. We require adaptive, out-of-core octrees that can handle completely irregular point sets and explicitly store the hierarchical PBGI data. Some other approaches construct the oc-

tree top-down [LGS*09] which is not suitable for out-of-core construction as it requires many passes over the full point-set. Other approaches are non-adaptive, where the octree depth is determined by the number of bits used in the Morton code [ZGHG10]. These methods succomb to the "teapot in a stadium" problem, which would not be suitable for PBGI where the point densities vary dramatically. In our approach the number of bits in the Morton code only determines the maximum possible depth, but this depth is not usually reached during the build nor in the final octree. Still other approaches target different problem domains with unique construction and storage requirements. For example, Shaffer and Samet [SS87] operate on image pixel data and construct a quadtree by inserting pixels one-at-a-time in scan-line order. Without a scan-line order, the memory used by the algorithm can grow as large as the number of points. This would not work for PBGI data since the points are completely irregular and memory use would thus be effectively unbounded.

We have found that our octree construction algorithm is most similar to the method described by Salmon and Warren [SW97], who developed their method for general N-body simulation. We give a more thorough description of the build algorithm and handle the specifics of PBGI. We also introduce a chunking extension (see Section 4.2), which significantly speeds up the out-of-core octree traversal. Moreover, we offer an efficient and parallel algorithm for traversing the out-of-core data at shade time (see Section 5).

3. Background

This section overviews two key concepts that are necessary for understanding our algorithm: the PBGI system, and Morton codes.

3.1. PBGI

The full PBGI algorithm is composed of three stages: point generation, octree generation, and final shading.

Point Generation: We generate point samples of the scene roughly proportional to the size of a micropolygon tessellation of the scene and diffusely shade them in a single streaming out-of-core pass.

Octree Generation: An octree is constructed over the points. Each octree node describes the light reflected from the corresponding cluster of points to any direction. To compactly store the directional variation, spherical harmonics (SH) are used.

Final Shading: The octree is used to integrate global illumination. At each shading point, a set of nodes defining a suitable LOD is selected by recursively traversing down the octree until a refinement criteria is satisfied. Similarly to multi-dimensional lightcuts [WABG06], we call this set of nodes a *cut*. Then the radiance stored in these nodes is accumulated using a cube rasterizer.

Our work focuses on the out-of-core implementation of octree generation and final shading in Sections 4 and 5 respectively. We refer the reader to Christensen [Chr08] and Křivánek et al. [KFC*10] for more information on any of the above stages.

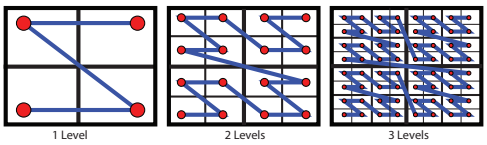


Figure 2: *Three levels of the Z-curve. The curve is in blue, and the red points correspond to the Morton code integer values along the curve. The black grid shows the quadtree structure for this Z-curve. Note the Z-shape hierarchy: one level is a single Z, two levels is four Zs arranged in a Z, and three levels give 16 Zs at the finest resolution.*

3.2. Morton Codes

In order to convert a 3D point into its corresponding Morton code, we first convert the floating point 3-vector into an integer 3-vector, mapping the scene's bounding box to a 21-bit integer range on each axis. We then interleave the bits of the X, Y, and Z coordinates into a single 64-bit value. For example, given the 2-bit integer coordinates (X_1X_0, Y_1Y_0, Z_1Z_0), the 6-bit Morton code would be $Z_1Y_1X_1Z_0Y_0X_0$.

The curve transcribed by connecting points in Morton order is referred to as a Z-curve. Figure 2 shows three iterations of a Z-curve in 2D and its associated quadtree. As can be seen in the figure, the Z-curve provides a natural 1D ordering of the 2D quadtree nodes. While other space-filling curves also exhibit this characteristic (such as the Peano-Hilbert curve [Hil91]), the Z-curve is both efficient and easy to implement.

Our build algorithm is based on the fact that the Morton order directly maps to the hierarchical structure of an octree [Mor66]. If a set of points are sorted according to their Morton order, it becomes possible to build an octree and the cluster data in a single streaming pass. This compares favorably with typical tree construction algorithms that require repeated iteration over the point-set.

4. Out-of-Core Octree Generation

The input to the out-of-core building algorithm is an unorganized point cloud and its bounding box that is computed during the point generation stage. The output consists of two out-of-core files: one for the octree nodes and another for the points.

Before constructing the octree, we sort the points according to their Morton code. Out-of-core sorting is a well-studied area and we chose to use the N-way merge algorithm [Knu98], which proceeds by sorting the input in blocks that fit in memory, and performs a final merge pass over all the blocks. Once the points have been sorted, the remaining task is to construct the octree and compute the clusters.

We first describe a basic implementation of the algorithm in Section 4.1, and then extend it in Section 4.2. The extension provides a more coherent ordering that significantly improves rendering performance.

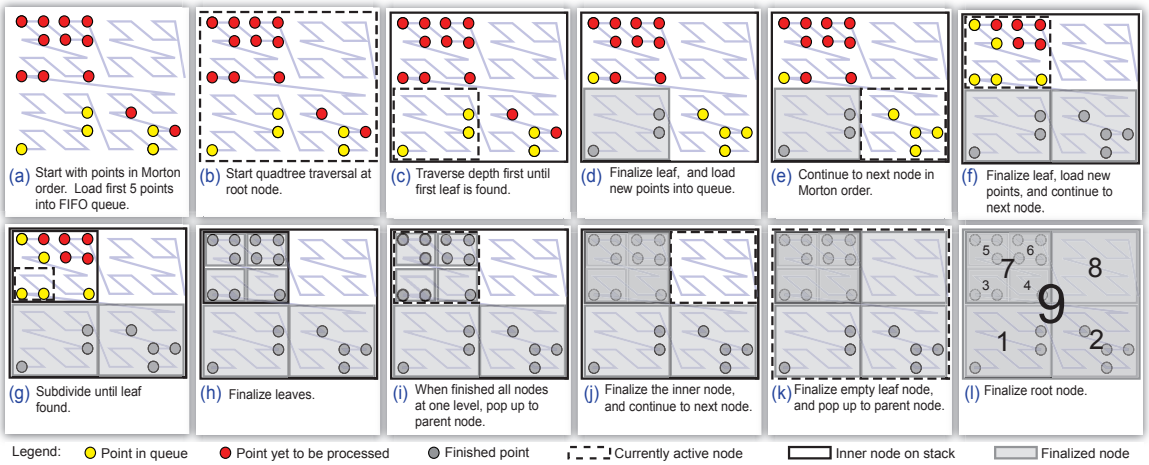


Figure 3: *A walkthrough of the octree construction algorithm. For simplicity, we illustrate in 2D with a quadtree. The algorithm proceeds from (a) to (l), using $leaf\_max = 4$. Note how the algorithm streams through the points along the Z-curve, keeping no more than $leaf\_max + 1 = 5$ points in memory, while simultaneously traversing and constructing the quadtree along a coarser Z-curve that adapts to the local point density. The quadtree nodes in (l) are indexed according to the order in which they are finalized and written to file.*

4.1. Basic Octree Construction

We construct the octree by performing a recursive depth-first traversal of the implicit structure of the tree, as illustrated in Figure 3. We explicitly construct the nodes in post-order, i.e. just before the traversal unwinds the stack to move one level up in the tree. Intuitively, the tree is built bottom-up, and each node is constructed after all its child nodes have been constructed. The octree nodes are written into one large file. We include pseudo-code for the build algorithm in the auxiliary material.

The cluster data (see [Chr08]) associated with each node has to be computed before writing the node to the file. We call the process of computing these attributes *finalizing* the node.

We store no more than *leaf_max* points in each leaf node. We use $leaf\_max = 16$ for all of our scenes, which seems to work well in practice. At any time, our algorithm needs only $leaf\_max + 1$ points in memory, and we use a FIFO queue to store these points.

Procedure: The following refers to Figure 3 to describe the basic algorithm in 2D. Extension to 3D is straightforward. Figure 3a shows a set of points and the Z-curve connecting them in light blue. For simplicity, we use $leaf\_max = 4$ in this example.

The algorithm starts by loading the first $leaf\_max + 1$ points into the FIFO queue (the yellow points in Figure 3a), and then it initiates a depth-first traversal at the root node of the quadtree (the dashed square in Figure 3b). Since the points are in Morton order, we need to inspect nodes at the lower left corner of the quadtree first.

We recursively subdivide the quadtree until we find a node that does not contain all of the points in the queue (Figure 3c). As we subdivide, the parent nodes are pushed onto the stack (the solid black square in Figure 3c). Note that it is sufficient to only test the last point in the queue to see whether all points fit in the node. If the last point is inside the node, then all points before it must also be inside the node. As we delve deeper in the quadtree, we always continue into the bottom left node, as this is the first point along the Z-curve.

When we find a node that does not hold the last point in the queue, that node becomes a leaf node (the dashed square in Figure 3c). We then finalize the node and extract the points from the queue that are in the node's bounds. We write the node to the file, storing the index of the first point and the number of points in the node. Once the leaf node is finalized, its points will not be revisited anymore (the gray square and the gray points in Figure 3d). Finally, more points are loaded to refill the queue (the yellow points in Figure 3d).

We continue the traversal to the sibling that is next in Morton order (the dashed square in Figure 3e). If this node holds $\leq leaf\_max$ points, we make it a leaf and continue to the next sibling. Otherwise, we recursively subdivide until we find a node that holds $\leq leaf\_max$ points. In Figure 3e, the first sibling has four points which is $\leq leaf\_max$ points so it is made into a leaf node. The next sibling in Morton order (Figure 3f) has $> leaf\_max$ points, so it is subdivided further (Figure 3g).

When we complete a full Z-shape at any level in the quadtree, we finalize the inner node at that level (Figure 3i). At this point, we compute the cluster attributes for the inner node from the attributes of its children (Figure 3j), and write out the inner node. Once an inner node is finalized, its direct children are freed from memory.

The algorithm continues until all points are exhausted and all nodes have been finalized and written to disk. The num-

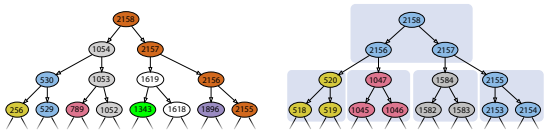


Figure 4: *Chunking illustrated with a simulated binary tree. Both images show a top portion of a larger tree (2158 nodes). The color indicates the page in which the node resides, and the number indicates the location of the node in the file.* ***Left:*** *Nodes are ordered without chunking which makes the page assignment appear almost random.* ***Right:*** *Chunking is used to improve the ordering. The light blue boxes indicate the chunks. Here, chunk_levels* $= 2$ *and the page size is 64 nodes.*

bers in Figure 3l show the order in which the nodes are finalized and written to disk. This number is used to uniquely identify the nodes.

This algorithm provides an efficient way to create out-of-core files for the octree nodes. As shown in Figure 3l, the nodes are stored in the file according to a depth-first traversal. Both node and point files are implicitly partitioned into constant sized pages. During shading the frequently used pages are cached in memory for quick access (see Section 5 for details).

4.2. Octree Construction with Chunking

In practice, the depth-first order is relatively incoherent except for a few levels close to the leaves of the tree. In a large tree, the child nodes end up quite far away from their parents. During shading, the child nodes are generally accessed soon after their parent, so it is better to store them near each other to increase the probability of them being in the same page.

We achieve a more coherent ordering of the nodes with an extension to the basic algorithm which we call *chunking*. Chunking is performed by delaying the output of the nodes, so that chunks of nodes are written at once, guaranteeing that the nodes within each chunk are placed next to each other. Figure 4 illustrates chunking in a binary tree. Note that in an octree, depth-first ordering is even worse, since only one of the eight children of each inner node immediately precedes the parent and the remaining seven are scattered across the file.

We implement chunking by augmenting the algorithm with *write-out queues* that are created and flushed periodically during the build. The size of the chunks is controlled by a single variable *chunk_levels*, that determines the maximum depth of each subtree. We found that *chunk_levels* $= 3$ gives good cache hit rates. This will create a queue for any inner node at every third level in the tree. Each queue gathers all the nodes for the subtree below it. Once the traversal returns to the inner node that created the queue, the queue gets written out and deallocated. For full implementation, see the pseudo-code in the auxiliary material and Section 7 for performance comparisons with different chunk sizes.

5. Out-of-Core Shading

Once the octree is built, we use it to efficiently compute approximations to various global illumination integrals. Some examples are one-bounce diffuse interreflections, ambient occlusion or high dynamic-range environment-map lighting.

As explained in Section 3.1, the shading is based on a recursive octree cut-picking algorithm, which starts from the root and traverses down the tree, until a refinement criteria is satisfied (see [Chr08] section 3.3 for details).

The PBGI shading system sees the octree nodes and points as two potentially gigantic files. As mentioned earlier, both files are implicitly decomposed into pages, which are loaded lazily, one page at a time and stored in an LRU cache. The cache's capacity is determined by the total memory budget allocated.

In our parallel implementation, we use one process per core. We decompose the image into small rectangular regions and store them in a shared work queue. Processes ready to work acquire the next region to be shaded from the shared work queue. During shading, all the processes traverse the octree and access the point and node caches simultaneously. Both caches are stored in shared memory and are protected by a mutex.

Since the caches are accessed frequently, mutex contention can become a problem. Similarly to the PantaRay-system [PFHA10], we utilize process-local caches on top of the shared caches. When a process requests a page, it first looks into its local cache. If the page is not found, the shared cache is queried. The shared cache either directly provides the page or loads it first from disk. This is effectively a two-level LRU cache using multi-level inclusion [HP03]. We use reference counting to deallocate pages that fall out from all the caches.

Since our cut picking algorithm accesses more node than point data, we allocate 3/4 of the memory budget for the node cache, and 1/4 for the point cache. We devote the first half of the memory budget to the shared cache, an the other half is divided equally amongst all the local caches. Node pages are 240 kB and contain 2048 nodes each. Point pages are 514 kB and contain 65K points each. With bigger page sizes the system cannot access the relevant portion of the data with a fine enough granularity and loads more data than needed. Conversely, smaller page sizes stress the caches, as they need to manage many more entries. We tuned our page sizes experimentally to provide a good balance.

6. Implementation Details

Of the various stages of the octree build algorithm, only the in-core portion of sorting is executed in parallel. This is because the rest of the algorithm is heavily I/O bound making it unlikely that parallelism would provide any benefit.

To maximize the overall I/O efficiency, we compressed the point and node data structures. We use (16-bit) half-floats to store the point normals and colors as well as the node

cluster's SH coefficients. Normals and color value ranges are well suited to this floating point representation, but the SH coefficients need to be re-normalized prior to compression to prevent the coefficients from falling out of range [KKZ08]. The normalization factor we use is an upper bound of the cluster projected area for any direction. We use the area of a disk with radius equal to half of the node's voxel diagonal. This factor can be easily stored in a lookup table, indexed by the level in the octree at which the cluster resides, therefore we do not need to store an additional normalization coefficient per node. This form of compression maintains enough accuracy at all levels of the octree and has not introduced noticeable artifacts.

As mentioned in Section 3.1, we use 64-bit Morton codes. The Morton code bit length determines the maximum octree depth, and a 64-bit code is able to encode 21 levels, which provides a resolution that has been sufficient in practice. Note that increasing the code size would not affect I/O efficiency, since the Morton codes are only explictly stored in-core and are re-computed on-demand. During final shading they are not needed anymore.

Our node data structure is 120 bytes and our point data structure is 32 bytes. All point and node indices are stored using 32-bit unsigned integers, which allows for 4.2 billion points and nodes. Increasing the bit length of the indices would produce larger node files and thus impact the I/O efficiency of our system. If we encounter scenes that require more than 4.2 billion nodes, we plan to use multiple levels of octrees (octrees of octrees) with independent 32-bit indices.

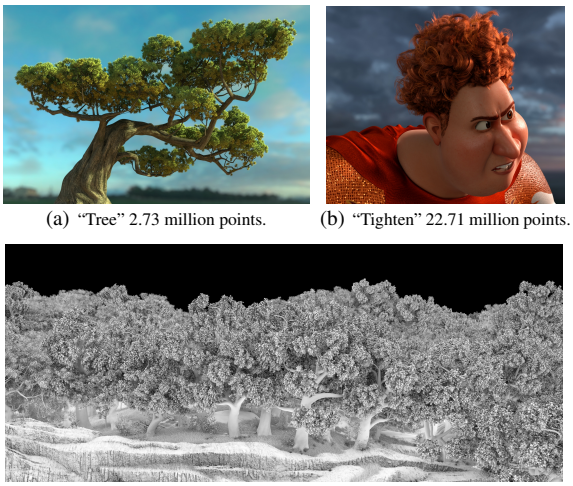
7. Results

The out-of-core PBGI system described in this paper is being used in the production of several feature films at DreamWorks Animation. All of the images in this paper were rendered on a HP Z800 workstation with a dual quad-core Intel Xeon CPU X5570 running at 2.93 GHz with 12 GB of memory. Each image was rendered at 1920×816 resolution and shaded with 8 cores.

We tested our algorithm on the scenes shown in Figures 1, 5, and 6, which vary in complexity from 2.73 million to 1.7 billion points. We compute diffuse interreflections for all scenes except for Figure 5c which uses ambient occlusion. Shade times for ambient occlusion are faster than for diffuse interreflections because we cull away portions of the scene that are far from the shade point.

Performance: The performance in various production scenes is shown in Table 1. The time spent shading images with GI only ("Shading time") accounts for the time to compute irradiance using our out-of-core PBGI system and excludes other aspects of shading. This computation is performed once per micropolygon in the image, as indicated by the number of ("Shading calls"). The cost of a single call is directly proportional to the ("Avg. cut size").

As can be seen, the overhead of our out-of-core algorithm during shading is small. The I/O time spent during shading ("Shading I/O time") varies roughly between 0.01% and 7%. Moreover, when our out-of-core algorithm does not need to access the entire scene, it can be even faster than a fully in-core implementation. The "Cache I/O ratio" is the fraction of the out-of-core data that is paged in-core during shading. When this fraction is small, our out-of-core system reduces I/O significantly compared to an in-core solution that must load the entire data-set. Even when the percentage is high and we must access most of the scene data, we found that our algorithm is often still faster than our previous in-core system. Rendering the scenes "Tree" and "Tighten" is $1.85\times$ and $1.23\times$ faster respectively, compared to our in-core system. We believe this speed benefit is largely due to the improved memory coherence from our chunking extension.



(a) "Tree" 2.73 million points. (b) "Tighten" 22.71 million points.

(c) "Forest" 927.07 million points.

Figure 5: *Some of the scenes used to study our algorithm's performance in Table 1. (a) and (b) were rendered with diffuse interreflections while (c) uses ambient occlusion. (b) is from the DreamWorks Animation movie "Megamind", and (c) is from the DreamWorks Animation movie "Shrek Forever After". In all these scenes (a-c), GI is computed on every leaf and hair, requiring a large number of points.*

The memory budget devoted to the caches can be set surprisingly low, even in highly complex scenes, without seriously degrading performance. The lower bound for a reasonable memory budget is determined by the size of the full cut of the octree required to shade a single pixel. If the cut does not fit in-core, the cache starts thrashing at every pixel, and shading slows down excessively. An indicator for an acceptable memory budget is the "Avg. traversal memory" listed in Table 1. It measures the average amount of memory needed to shade a single pixel. As shown in the table, these figures are remarkably small, and vary between 20 and 111 MB.

Coherent Chunking: All statistics listed in Table 1 use the enhanced octree construction algorithm with three levels of chunking, as described in Section 4.2. We also compared shading times using octrees built with various levels

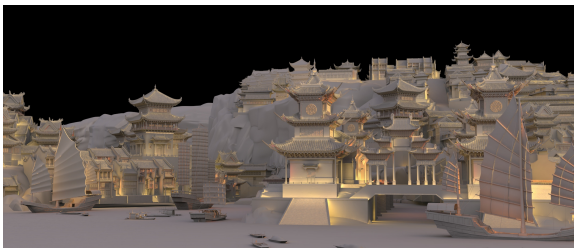
	Tree	Tighten	City	City static	Forest
Point count (millions)	2.73	22.71	128.10	1736.74	927.07
Node count (millions)	0.65	5.24	27.45	328.77	203.73
Shading calls (millions)	1.24	1.61	3.21	4.39	18.81
Point sort time (s)	0.82	7.70	198.95	3094.74	1618.25
Octree build time (s)	0.92	7.79	59.81	1268.51	603.97
Total build time (s)	1.73	15.49	258.75	4363.25	2222.12
Shading time (s)	116	173	383	907	1459
Shading I/O time (%)	0.01%	0.03%	1.93%	2.11%	7.00%
Out-of-core data (MB)	158	1,292	7,051	90,625	51,607
Cache peak RAM (MB)	157	951	2,076	2,074	3,226
Cache hit ratio	99.99%	99.99%	99.97%	99.96%	99.97%
Cache I/O ratio	99%	74%	60%	11%	48.50%
Avg. cut size	1055	1230	1378	1902	857
Avg. nodes traversed per cut	1797	2122	2559	3487	1906
Avg. traversal memory (MB)	20.40	50.95	80.67	111.46	47.70

Table 1: *Statistics for the scenes shown in Figures 5a (Tree), 5b (Tighten), 6a (City), 6b (City static) and 5c (Forest). This table lists the performance breakdown of our algorithm, followed by scene complexity statistics and cache memory utilization, hit ratio, I/O ratio (the percentage of data loaded from disk as compared to the total data size), the average cut size (the number of clusters or points on the cut), the number of octree nodes traversed to reach the cut and the average memory needed to load all node and point pages to shade a single pixel.*

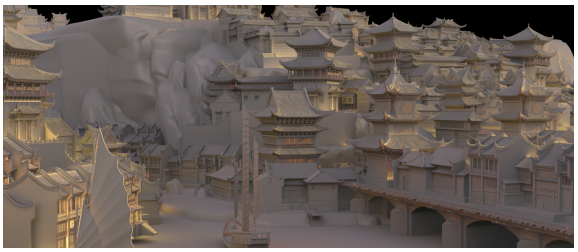
of chunking. In the "Tighten" scene, the shading time with three levels of chunking was 1.8× faster than without chunking. In both cases, the build took 15 seconds. In the "City" scene, the speedup during final shading due to chunking was 4.4×, and the build time was 4 minutes 18 seconds with chunking and 4 minutes 10 seconds without. In both scenes, the bulk of the speedup is due to highly reduced I/O during final shading. Using only two levels of chunking gives slightly less benefit than three, and using more than three levels does not noticeably improve performance. The build with chunking for all the scenes required at most 0.3 MB of extra in-core memory during the build, devoted to nodes residing in the node write-out queues.

Scalability: We measured scalability close to 7× on 8 cores for smaller scenes, but slightly less when the size of the dataset and the amount of I/O increases. For the "Tree" scene, the time is 7.2× faster, for "Tighten" 6.4×, for "City" 6.3× and for "Forest" 6.7× faster when compared to shading with a single core.

Limitation: As stated in Section 5, we use a two-level LRU cache with multi-level inclusion in order to reduce the expense of mutex locks. While a two-level cache is a common configuration in hardware, we found that using a separate 1^{st}-level cache per processor core is a relatively new area that introduces some problems. Specifically, over-subscription to one local cache can force the eviction of other processes' entries from the 2^{nd}-level cache, breaking the multi-level inclusion property. This may lead to situations where a page that already resides in one of the local caches is re-loaded because it cannot be found in the shared cache. We are currently exploring multiple avenues to address this issue.



(a) "City" 128.1 million points



(b) "City static" 1.737 billion points

Figure 6: *(a) The cityscape's GI-only render (irradiance) using a view-dependent point-set. See Figure 1 for the full composite. (b) Another view using a large view-independent point-set. The points were generated finely enough to allow rendering multiple views.*

8. Conclusion

We have introduced an out-of-core octree construction algorithm for PBGI that can operate within a small memory cap, and only requires a single streaming pass over the points after they have been sorted in Morton order. We introduce a novel extension to this algorithm which stores the octree data in coherent chunks, and we demonstrate that it improves PBGI shading time by a factor of up to 4.4×. We introduce a coherent and parallel shading algorithm that only loads the parts of the octree that are required during the final shading. All these pieces come together to provide a coherent, parallel out-of-core system for rendering high quality global illumination for extremely complex scenes. We demonstrate our system on real production scenes of up to 1.7 billion points, which we believe is unprecedented geometric complexity. As far as we are aware, the largest previously published number of primitives with indirect illumination or ambient occlusion was about 1 billion micropolygons [PFHA10].

Our octree construction algorithm builds upon the work of Salmon and Warren [SW97], who applied their method to N-body simulations. Our chunking extension for coherent data storage and our parallel out-of-core shading algorithm should also benefit this more general field.

References

[BBS*09] BUDGE B., BERNARDIN T., STUART J., SENGUPTA S., JOY K., OWENS J.: Out-of-core Data Management for Path Tracing on Hybrid Resources. *Computer Graphics Forum (EUROGRAPHICS '09) 28*, 2 (2009), 385–396.

[Bun05] BUNNELL M.: *GPU Gems 2*. 2005, ch. 14: Dynamic Ambient Occlusion and Indirect Lighting, pp. 223–233.

[Chr08] CHRISTENSEN P. H.: *Point-Based Approximate Color Bleeding*. Tech. Rep. 08-01, Pixar, 2008.

[CLF*03] CHRISTENSEN P. H., LAUR D. M., FONG J., WOOTEN W. L., BATALI D.: Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes. *Computer Graphics Forum (EUROGRAPHICS '03) 22*, 3 (2003), 543–552.

[CMRS03] CIGNONI P., MONTANI C., ROCCHINI C., SCOPIGNO R.: External Memory Management and Simplification of Huge Meshes. *IEEE TVCG 2003 9*, 4 (2003), 525–537.

[DBB02] DUTRE P., BALA K., BEKAERT P.: *Advanced Global Illumination*. A. K. Peters, Ltd., 2002.

[GL10] GARANZHA K., LOOP C.: Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Computer Graphics Forum (EUROGRAPHICS '10) 29*, 2 (2010), 289–298.

[GP07] GROSS M., PFISTER H.: *Point-Based Graphics*. Morgan Kaufmann Publishers Inc., 2007.

[Hil91] HILBERT D.: Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen 38* (1891), 459–460.

[HP03] HENNESSY J. L., PATTERSON D. A.: *Computer Architecture: A Quantitative Approach*, 3 ed. Morgan Kaufmann Publishers Inc., 2003.

[Kaj86] KAJIYA J. T.: The Rendering Equation. In *ACM SIGGRAPH '86* (1986), pp. 143–150.

[KFC*10] KŘIVÁNEK J., FAJARDO M., CHRISTENSEN P. H., TABELLION E., BUNNELL M., LARSSON D., KAPLANYAN A.: Global Illumination Across Industries. In *ACM SIGGRAPH 2010 classes* (2010).

[KKZ08] KO J., KO M., ZWICKER M.: Practical Methods for a PRT-based Shader Using Spherical Harmonics. In *ShaderX^6: Advanced Rendering Techniques*. Charles River Media, 2008.

[KMKY10] KIM T.-J., MOON B., KIM D., YOON S.-E.: RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies. *IEEE TVCG 2010 16*, 2 (2010), 273–286.

[Knu98] KNUTH D. E.: *The Art of Computer Programming: Sorting and Searching*, 2 ed., vol. 3. Addison Wesley Longman Publishing, 1998.

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH Construction on GPUs. *Computer Graphics Forum (EUROGRAPHICS '09) 28*, 2 (2009), 375–384.

[LZT*08] LEHTINEN J., ZWICKER M., TURQUIN E., KONTKANEN J., DURAND F., SILLION F., AILA T.: A Meshless Hierarchical Representation for Light Transport. *ACM TOG (SIGGRAPH '08) 27*, 3 (2008), Article 37.

[Mor66] MORTON G. M.: *A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*. Tech. rep., IBM Ltd., 1966.

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: Optix: a general purpose ray tracing engine. *ACM TOG (SIGGRAPH '10) 29* (2010), 66:1–66:13.

[PFHA10] PANTALEONI J., FASCIONE L., HALL M., AILA T.: PantaRay: Fast Ray-traced Occlusion Caching of Massive Scenes. *ACM TOG (SIGGRAPH '10) 29*, 3 (2010), Article 37.

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering Complex Scenes with Memory-Coherent Ray Tracing. In *ACM SIGGRAPH '97* (1997), pp. 101–108.

[REG*09] RITSCHEL T., ENGELHARDT T., GROSCH T., SEIDEL H.-P., KAUTZ J., DACHSBACHER C.: Micro-rendering for scalable, parallel final gathering. *ACM TOG (SIGGRAPH Asia '09) 28* (2009), 132:1–132:8.

[SS87] SHAFFER C. A., SAMET H.: Optimal quadtree construction algorithms. *Computer Vision, Graphics, and Image Processing 37*, 3 (1987), 402 – 419.

[SW97] SALMON J., WARREN M. S.: Parallel, Out-of-core Methods for N-body Simulation. In *SIAM Parallel Processing for Scientific Computing* (1997).

[SW03] SCHAEFER S., WARREN J.: Adaptive Vertex Clustering Using Octrees. In *SIAM Geometric Design and Computing* (2003).

[TL04] TABELLION E., LAMORLETTE A.: An Approximate Global Illumination System for Computer Generated Films. *ACM TOG (SIGGRAPH 2004) 23*, 3 (2004), 469–476.

[TLO03] TU T., LOPEZ J., O'HALLARON D.: *The Etree Library: A System for Manipulating Large Octrees on Disk*. Tech. Rep. CMU-CS-03-174, Carnegie Mellon School of Computer Science, 2003.

[WABG06] WALTER B., ARBREE A., BALA K., GREENBERG D. P.: Multidimensional Lightcuts. *ACM TOG (SIGGRAPH 2006) 25*, 3 (2006), 1081–1088.

[WH92] WARD G. J., HECKBERT P. S.: Irradiance Gradients. In *EGSR '92* (1992), pp. 85–98.

[YM06] YOON S.-E., MANOCHA D.: Cache-Efficient Layouts of Bounding Volume Hierarchies. *Computer Graphics Forum (EUROGRAPHICS '06) 25*, 3 (2006), 507–516.

[ZGHG10] ZHOU K., GONG M., HUANG X., GUO B.: Data-parallel octrees for surface reconstruction. *IEEE TVCG 2010 17*, 5 (2010), 669–681.