Pacific Graphics 2016
E. Grinspun, B. Bickel, and Y. Dobashi
(Guest Editors)

Volume 35 (2016), Number 7

Merged Multiresolution Hierarchies for Shadow Map Compression

Leonardo Scandolo, Pablo Bauszat, and Elmar Eisemann

Delft University of Technology, Netherlands

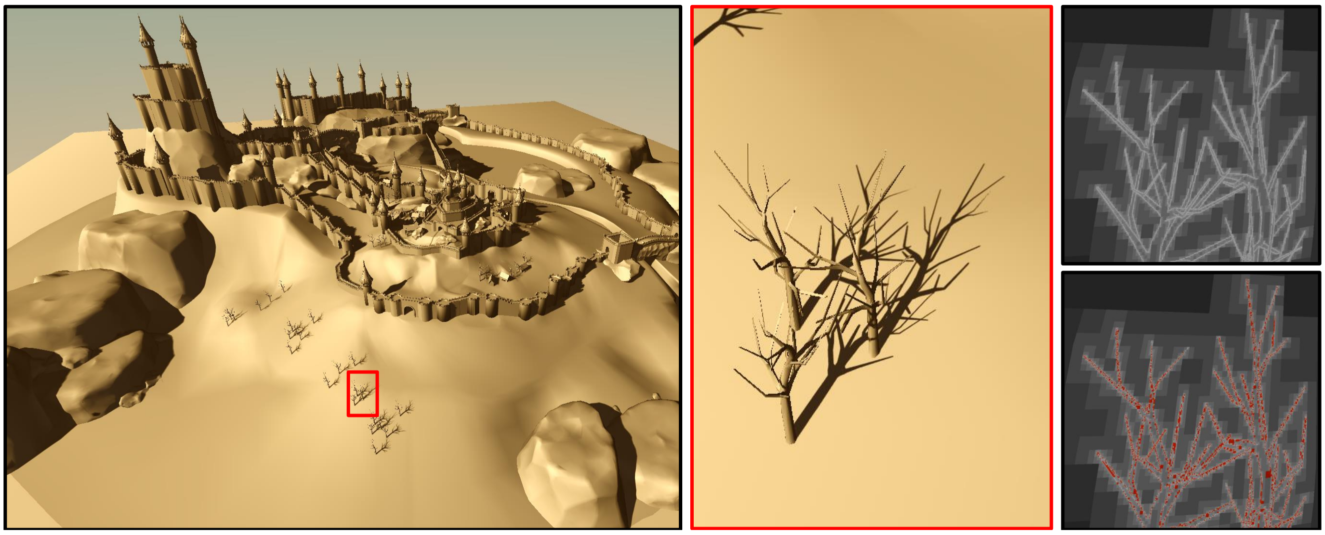


Figure 1: High-quality shadows generated using a single shadow map with a resolution of $256k \times 256k$. The shadow map is efficiently stored as a compressed merged multiresolution hierarchy. The top-right inset visualizes the hierarchy depth for traditional MH-based compression [SBE16]. Our representation (bottom-right) merges redundant subtrees (indicated in red) and reduces the required memory from 38.43 MB down to 23.77 MB (61.8%) compared to the previous approach while maintaining full run-time performance.

Abstract

Multiresolution Hierarchies (MH) and Directed Acyclic Graphs (DAG) are two recent approaches for the compression of high-resolution shadow information. In this paper, we introduce Merged Multiresolution Hierarchies (MMH), a novel data structure that unifies both concepts. An MMH leverages both hierarchical homogeneity exploited in MHs, as well as topological similarities exploited in DAG representations. We propose an efficient hash-based technique to quickly identify and remove redundant subtree instances in a modified relative MH representation. Our solution remains lossless and significantly improves the compression rate compared to both preceding shadow map compression algorithms, while retaining the full run-time performance of traditional MH representations.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture, I.4.2 [Computer Graphics]: Compression (Coding)—Exact coding

1. Introduction

Real-time high-quality shadows are still a challenge for today's rendering applications. A recent trend is to precompute and compress high-resolution shadow maps [Wil78]. This approach generates high-quality shadows for static environments and, in contrast to traditional techniques (e.g., light maps), allows for shadow casting on dynamic receivers. Only shadows from dynamic objects have to be handled with traditional shadow techniques. The precomputed shadow map is typically of very high resolution (up to

$1024k \times 1024k$ texels), making compression a necessity. These solutions should not alter the shadow test results and provide efficient run-time access.

Most shadow map compression techniques build upon a well-known observation: in a shadow map, any arbitrary depth value within the interval defined by the entry and exit point of an occluding object can be used for a conservative depth test if the object is watertight and never seen from the inside [Woo92]. Therefore, each texel in a shadow map corresponds to an interval of possible values which can represent it, a concept usually referred to as dual shadow maps [WE03]. By exploiting this property, specialized shadow map compression algorithms achieve significantly higher compression rates than traditional lossless image compression.

Currently, multiresolution hierarchies (MH) achieve the highest compression rates [SBE16] while supporting real-time shadow evaluation. Starting with a dual shadow map, the algorithm derives an extremely sparse hierarchical representation by exploiting the similarities of depth intervals in homogeneous regions. The sparse hierarchy is encoded in a compressed quadtree representation. As every level in the hierarchy encodes a complete shadow map, hierarchical filtering is naturally supported.

Directed Acyclic Graphs (DAG) [KSA13] are another compact representation for high-resolution shadow information. Instead of storing a shadow map, DAGs encode a voxelized representation of the depth test results for discrete points in the scene [SKOA14, KSA15]. The information is stored in a sparse voxel octree, and compression is further achieved by merging equal tree structures. Since each node in a DAG simply encodes the state for lit or shadowed, equal structures can directly be found by a simple dictionary search.

In this paper, we propose a novel data structure that fuses both concepts and exploits both homogeneity in depth intervals as well as topological similarities. Our method is based on the observation that traditional MH representations frequently exhibit similar hierarchical patterns around small structures and edges due to the regular and axis-aligned nature of the underlying quadtree. This results in a significant amount of almost identical, redundant subtrees in the quadtree representation, especially at the finer levels of the hierarchy (Fig. 4). We exploit the fact that each node in these subtrees does not represent a single value, but a depth interval defining a margin of tolerance. This allows us to merge subtrees by finding a common instance that is able to represent multiple occurrences of similar subtrees without introducing any loss of information. Given that each node represents a depth interval, subtree matching and merging becomes a much more complex task compared to the DAG approach. Therefore, we propose an efficient matching algorithm based on a hashing strategy. We show that our method significantly improves the compression rate by up to 40% compared to MH-based and DAG compression, while maintaining the full rendering performance of the traditional MH-based compression algorithm.

2. Related Work

Shadow maps have been around in computer graphics for decades and many techniques address their shortcomings (e.g., [WSP04, PWC*09, SFY13]). In this section, we primarily discuss previous work on image compression and specialized shadow compression algorithms. For a more general and detailed review of shadow algorithms, we refer the reader to the surveys of Eisemann et al. [ESAW11] and Woo et al. [WP12].

2.1. Texture compression

Image compression techniques are abundant; two widespread algorithms are the JPEG [PM92] and PNG [Bou97] standards. Traditional techniques often do not provide efficient random-access to the data, i. e., they require to decode larger parts or the whole image at once, which prohibits their use for real-time applications. In the context of real-time rendering, several GPU-supported compression formats that allow random-access queries exist, such as S3TC [INH99], ETC [SAM05], ASTC [NLP*12] and others. While these methods work well for texture and normal maps, they are lossy and encoding an image with these techniques results in a globally bounded per-texel error. In the context of shadow map compression, it is necessary to ensure that the result of the depth test is always correct in order to avoid artifacts and thus, lossy techniques are not suitable. Although lossless compression methods based on decomposition and block-based matching exist [HC07], they achieve very modest results compared to specialized shadow map compression methods that can exploit per-texel error margins.

2.2. Shadow map compression

In order to alleviate biasing problems in shadow mapping, Weiskopf and Ertl [WE03] introduced the concept of dual shadow maps. Since only points outside closed objects are ever tested during shadow mapping, any value inside the first closed object as seen from the light source for a given texel can be stored in a shadow map without affecting the result. Dual shadow maps capture this concept by representing per-texel allowable depth intervals as a pair of depth maps representing minimum and maximum valid depths.

Compressed Shadow Maps In their seminal work for shadow map compression, Arvo et al. [AH05] proposed the use of dual shadow maps to exploit the fact that consecutive texels in a scanline have very similar depth intervals in homogeneous shadow-map regions. Consequently, each scanline can be decomposed into a piecewise-linear function that respects the per-texel depth intervals. Only relatively few segments have to be stored for representing a scanline, leading to significant compression rates. However, the homogeneity is only exploited along a single dimension and filtering typically requires the traversal of several scanlines.

Multiresolution Hierarchies MH-based shadow map compression [SBE16] exploits local homogeneity in shadow maps by creating a sparse decomposition of a dual shadow map. The algorithm recursively groups four neighboring texels together in the same fashion as traditional mip-maps. However, each texel does not represent a single value but a depth interval instead, and the algorithm finds the largest subset of texels with intersecting intervals for each group. The subset is then represented at a lower resolution by the intersection interval of all texels in the subset and only the remaining texels which are not in the subset need to be stored explicitly. Initially, this procedure starts at the high-resolution dual

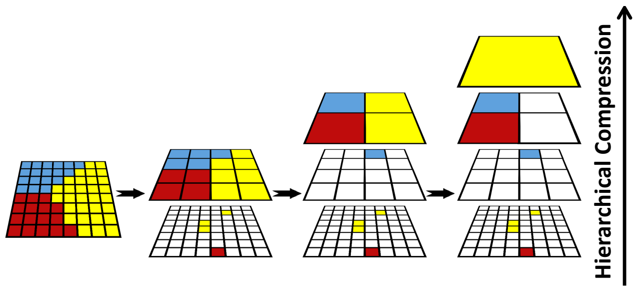


Figure 2: MH creation iteratively sparsifies each resolution level. Neighboring intersecting depth intervals, represented by equal-color texels here, are removed and represented at coarser levels in the hierarchy. Compression is *hierarchical*, i.e., information is moved up the hierarchy where it is represented more compactly.

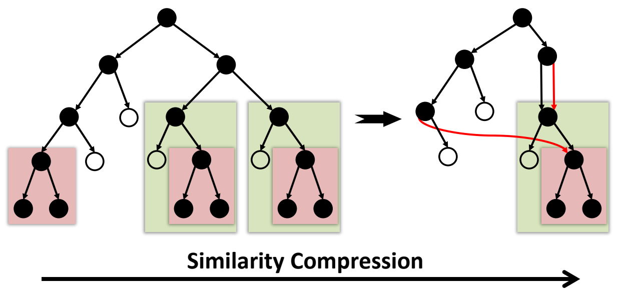


Figure 3: DAG compression identifies equivalent subtrees and redirects pointers in order to have a single representative for each subtree configuration. Compression is based on *similarity*, two equal subtrees anywhere in the hierarchy are merged.

shadow map and effectively sparsifies it in favor of a common lower-resolution representation, thus reducing the memory requirements. The sparsification is then recursively performed again at every decreasing resolution representation until a single texel level is reached (Fig. 2). This repeated sparsification process exploits neighboring homogeneity and per-texel allowable depth intervals to replace large homogeneous regions of the original high-resolution dual shadow map by a single representative.

Finally, each depth interval is replaced with a single representative value (usually the midpoint of the interval), and the structure is encoded in a sparse quadtree for fast run-time evaluation on the GPU. The resulting quadtree contains one node per remaining value in the sparse decomposition, and adds special empty inner nodes to preserve connectivity. At run-time, the value for a specific texel is retrieved by traversing the quadtree top-down, following the path defined by the location of the texel, and returning the value of the deepest non-empty node.

As is common practice, each inner node of the quadtree contains a single pointer to its first child node, and all sibling nodes are stored consecutively in memory. To further reduce memory requirements, the pointers are compressed using the method proposed by Lefebvre et al. [LH07]. Their approach is based on the insight that serializing the nodes using a depth-first order leads to small distances between nodes at lower levels of the hierarchy. Since these levels typically contain the larger number of nodes, pointers can be replaced with (positive) offsets in the serialized layout, which can be encoded using fewer bits. The growing offsets for the coarser levels of the hierarchy are alleviated by introducing an additional offset scale for each level and, if necessary, a padding in the memory layout. Each level l of the tree has an associated scaling factor $s(l)$, and the offset o of a node is multiplied by the scaling factor during evaluation to retrieve the actual address $(s(l) \times o)$ at which the children are located. Hereby, the final serialized quadtree uses only 16 bits per offset, which along with a 32-bit depth value, an 8-bit flag for the children, and 8 bits of padding leads to an inner node size of 64 bits. Since empty inner nodes do not store depth values, they are represented using 32 bits.

Directed Acyclic Graphs A DAG is a special type of sparse voxel octree [LK10] where matching subtrees are merged and only one instance of them is kept in the structure. The DAG method was first introduced for geometry [KSA13] and later extended to encode precomputed static, voxelized shadow information of a scene [SKOA14, KSA15]. During construction, a dictionary of existing subtrees in the structure is kept. Before adding a new subtree to a DAG, the dictionary is checked to see if a matching subtree is already present, and if so, the new subtree is replaced by a pointer to the existing one. This procedure exploits structural similarity and creates a very sparse representation for data sets that contain repeating patterns, such as typical computer-generated scenes (Fig. 3).

Despite their ability to greatly reduce memory requirements for 1-bit or 2-bit values, DAGs are not well suited for compressing more complex data, since equal subtrees become scarce. State-of-the-art approaches in this area [DKB*16] require heavy quantizing of data in order to obtain acceptable results, which is incompatible with the lossless constraint of shadow map compression. Hence, the DAG algorithm cannot be directly applied for merging subtrees of nodes representing depth intervals or full 32-bit precision values.

3. Merged Multiresolution Hierarchies

Our goal is to create a structure that can capture both the hierarchical similarities exploited by MH representations as well as the non-hierarchical similarities exploited by DAG representations. We propose to merge subtrees of an MH quadtree if we find that they (a) exhibit the same structure and (b) each corresponding pair of non-empty nodes have matching depth intervals. If both conditions are met, we can select one of the subtrees and replace its depth intervals for each node with the intersection of the corresponding nodes in the other trees. We then remove all other repeating occurrences and replace them with references to the modified subtree. This allows us to remove redundant subtrees from the hierarchy without violating the bounds of the dual shadow map.

In order to be able to merge similar structures at different depths, we change the MH representation to represent depth values relative to their parent node. For every node in the hierarchy, we subtract the mid-interval value of the parent from the interval bounds of the children. Since the mid-interval value will be stored in the final serialized quadtree, the original absolute value for a node is recovered by summing the depth values along a traversal path (Alg. 1). Changing the traditional MH representation from an absolute one

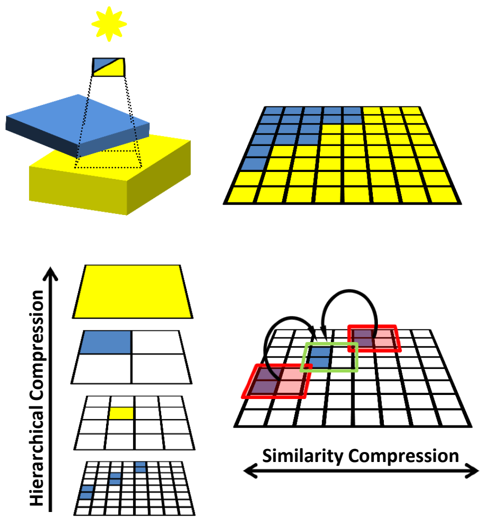


Figure 4: MMH captures both the hierarchical compression present in MH representations as well as the similarity compression present in DAGs.

to a relative one, does not increase evaluation times since all the ancestors of a node have to be visited anyways during the top-down tree traversal. Having subtree values relative to their parent nodes is beneficial, because it allows structurally similar subtrees at different levels of the hierarchy to be potentially merged.

A major task is to identify and match redundant subtrees in the hierarchy. Unfortunately, this task is not as straightforward as in the case of DAGs, where subtrees are only mergeable if they are identical. Subtrees in a MH representation can potentially have very different depth intervals, but are still mergeable as long as an intersection for each pair of corresponding nodes and their intervals exists. Hence, building a dictionary of MH subtrees is not enough to capture similarity; we need to compare each corresponding depth interval between subtrees to establish mergeability. Further, multiple combinations for merges exist frequently, rendering this task a combinatorial problem.

High resolution MHs may contain potentially millions of subtrees and comparing each possible pair is infeasible in practice. We restrict matching of subtrees to pairs of equal topology, which allows us to define clear partitions of subtrees. Although cases exist where a subtree is mergeable with another subtree of larger topology, these cases are unusual and would result in increased evaluation times for the replaced smaller topology branch. We propose a two-step approach to greatly reduce the number of tests and efficiently find small sets of similar subtrees.

The overview of our technique is as follows. We start by creating the high-resolution dual shadow map, which defines the per-texel depth intervals, via depth-peeling; this represents the highest resolution level of the initial MH. Then, we iteratively sparsify each level in a bottom-up manner and create a quadtree representation, as proposed by Scandolo et al. [SBE16]. The depth intervals for each node are kept as auxiliary information, and we convert them to a relative representation as explained earlier.

Algorithm 1 Pseudo-code for relative quadtree evaluation

```
function evaluateQuadtree(rootNode, coord) :
    value ← rootNode.value
    node ← rootNode
    while true:
        child ← getChildNode(node, coord)
        if isLeaf(child):
            return value + child.value
        if not isEmpty(child):
            value ← value + child.value
        node ← child
```

In the next step, we use a hash-based technique to partition the set of all inner nodes according to the topology of their descendants (Sec. 3.1). We then perform the pair-wise matching solely between elements in the hash collision lists, hereby enforcing comparisons only between trees of matching topology (not necessarily at the same depth in the hierarchy). Fig. 5 illustrates this procedure. To reduce the number of tests further, we restrict comparisons within each collision list to local areas by inserting elements following a spatial ordering. To accelerate the pair-wise subtree matching, we introduce an interval-based rejection test (Sec. 3.2). Finally, we discuss how to perform the final serialization step while taking into account the presence of merged subtrees (Sec. 3.3).

3.1. Hashing

Given an inner node (empty or full) in the initial MH quadtree structure, its child pointer always indicates the first existing child node in memory. All siblings of a child node lie consecutively in memory and the parent node contains a flag byte that is used to indicate every child's existence and type (full inner node, empty inner node, or leaf node). Since each group of 4 sibling nodes is packed consecutively in memory, the subtrees we merge have a 4-node pack at their root level (see Fig. 5). Hence, each subtree has exactly one parent node, whose child pointer indicates the first node in the 4-node root level. If two subtrees can be merged, we redirect the pointer in one of their parents to the other subtree.

Testing all subtree combinations is impractical due to the very large quantity of them present in an MH. We therefore assign a hash code for each subtree, which we store in their parent node, such that potential merge candidates collide when inserted in a hash table. We then only need to test node pairs in the same collision lists.

The hash encodes the topology of a subtree by involving the flags of all its nodes. Note that the parent node type does not have to match for two subtrees to be compatible, since the parent node is not part of the subtree. Further, we only match subtrees of the same height, and thus we keep a separate hash table for each subtree height to avoid collisions. We insert each node into its corresponding hash table, according to the hash code stored in it. This allows us to keep a reference to the subtree parent in order to redirect its child pointer if a merge candidate is found. Hence, after each node has been inserted into the hash table corresponding to its height,

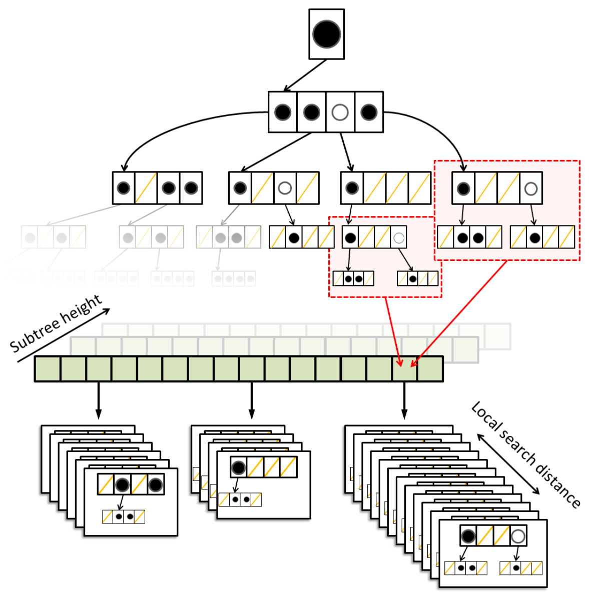


Figure 5: The matching procedure inserts subtrees into their corresponding hash table based on their height using a topology-based hash code. Subtree pairs in each hash table collision list within a certain distance are then tested to determine their mergeability.

each hash table entry contains a collision list representing all equal-topology subtrees (Fig. 5).

Since the hash value stored at each inner node represents its descendant subtree, it is convenient to produce the hash codes for the whole hierarchy in a single bottom-up sweep of the hierarchy. This allows us to easily track the height of the subtree, in order to insert it into the corresponding hash table. Further, we insert nodes into the hash table in a depth-first order to preserve a spatial locality inside each collision list. Our hash function, which fulfills these properties, is presented in Alg. 2.

3.2. Subtree Matching

So far, we have produced the collision lists of the hash tables. Now, the actual matching takes place and the hash tables are processed in the order of their height from large to small. Hereby, we make sure that larger subtree matching can remove larger parts of the tree before proceeding to test the smaller subtrees. The collision lists themselves can be treated in arbitrary order, as no elements of two separate collision lists will ever be tested. Therefore we can speed-up the subtree matching step by processing all collision lists of a single hash table in parallel, since there is no dependency between them. To further increase performance, we reduce the number of overall tests by a local matching and speed up the individual pair-wise tests by an early rejection test.

Local matching Testing all possible subtree pairs in a collision list would lead to a creation time that is quadratic with respect to the number of nodes. This would render the algorithm impractical, and thus we propose a local matching scheme to keep the run-time

Algorithm 2 Pseudo-code for bottom-up hash code construction

```
function bottomUpHashCreation() :
    for level from deepestLevel to rootLevel
        for every node in level
            updateHash(node)

function updateHash(node) :
    node.size ← typeSize(node.type)
    if isLeaf(node)
        node.hash ← 1
        return
    hash ← node.childFlags
    for each child in node.children
        node.hash ← node.hash + (child.size « childIndex)
        node.hash ← node.hash * (child.hash « childIndex)
        node.size ← node.size + child.size
    return hash
```

Algorithm 3 Pseudo-code for subtree matching

```
function matchPairs(hashTable, searchLimit) :
    for each collisionList in hashTable
        listSize ← collisionList.size
        for i from listSize - 1 to 0
            for j from i + 1 to min(i + searchLimit, listSize - 1)
                $n_1$ ← collisionList[i]; $n_2$ ← collisionList[j]
                if ($n_1$.lMax<$n_2$.lMin) || ($n_1$.hMin>$n_2$.hMax)
                    continue
                if offsetTooLarge($n_1$, $n_2$)
                    continue
                if fullTest($n_1$,$n_2$)
                    merge($n_1$,$n_2$)
                    break
```

bounded. Having used a depth-first insertion, the nodes in each collision list correspond to spatially-close neighborhoods. It is likely to find matches between elements close in the list as they tend to be located nearby similar features. In consequence, we limit the testing of a node to a certain distance within a collision list. In our experiments, a distance limit of 1000 entries provided a good trade-off between compression and construction time. Furthermore, increasing this limit further will eventually lead to testing subtrees that cannot be merged since their distance in the final structure will be too large to represent as an offset pointer. We discuss the influence of this parameter further in Sec. 4 and show that the usage of local matching decreases compression only slightly while reducing the run-time of the algorithm by several orders of magnitude.

Early Rejection When testing two subtrees for merge compatibility, all of their nodes and depth intervals need to be compared. While this process can become costly, specially for large trees, a single mismatch is enough to stop the comparison. Hence, we introduce a fast early rejection test.

For our rejection test, we aggregate information about the depth intervals of each subtree and store it in their parent node. This is done during the previously described bottom-up traversal that computes the hash codes of each subtree. We exploit the observation

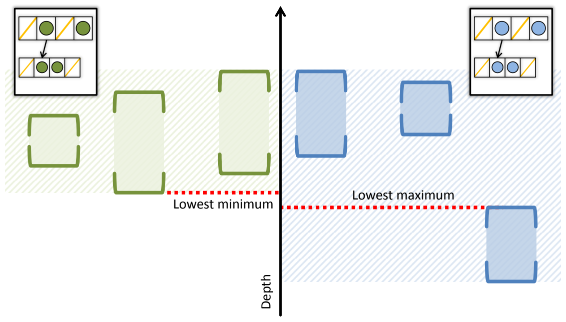


Figure 6: Compatibility testing of two trees with equal topology can be sped up by observing the aggregated statistics of their nodes' depth intervals. In this example, we see that the lowest minimum depth of all nodes in the left tree is higher than the lowest maximum depth of all nodes in the right tree. We can then conclude that the lowest interval in the right tree is guaranteed to not have an intersection with any intervals of the left tree.

that merging is only possible if the depth interval of all corresponding node pairs in the tested subtrees intersect (otherwise, there is no possible conservative depth value). In consequence, a conservative test checks if there is at least one node in one of the subtrees whose depth intervals fails to intersect the union of all intervals in the other subtree. In order to test this property, we keep track of the lowest and highest minimum and maximum depth of all node intervals for each subtree. If the highest minimum depth of a subtree is higher than the highest maximum of another subtree, at least one node exists in the first tree without an intersection with any node in the second one. The same is true if the lowest maximum depth of a subtree is lower than the lowest minimum of another subtree. Fig. 6 provides a graphical example of this test. In both cases, merging the two subtrees can be immediately rejected if the test fails.

Subtree Merging Once a pair of compatible subtrees has been found, we have to decide which one to remove and which one to keep. We apply a simple rule and always remove the one that appears earlier in the collision list. The reason is that the serialized quadtree uses positive offsets instead of full pointers, and removing the earlier subtree avoids a negative offset. The depth intervals of all nodes in the surviving subtree are modified to intersect the intervals of the nodes in the removed subtree. Hereby, we ensure that this subtree correctly represents both original subtrees. If the surviving subtree is later merged with another one, the intervals in its nodes will be the intersection of all the intervals of the corresponding nodes in the original trees. At this point, instead of actually removing the redundant subtree immediately from the hierarchy, we only mark its parent as merged and save a reference to the surviving subtree's root. All marked subtrees will later be skipped during the serialization step.

Alg. 3 summarizes the matching algorithm. Note that the outer loop over each element in the collision list proceeds in back-to-front order. This ensures that each subtree indexed by the outer loop is not already merged, thus allowing us to safely merge it with another subtree if we find a suitable match.

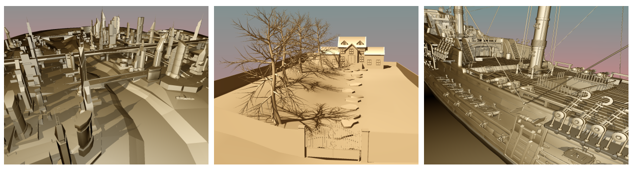


Figure 7: An overview of the tested CITYSCAPE, VILLA, and SHIP scene. The CLOSED CITY scene is shown in Fig. 1. Larger images can be found in the supplementary material.

3.3. Serialized Tree Creation

Once the subtree matching is completed and redundant subtrees are marked in the hierarchy, we perform the quadtree serialization. While the original serialization applied a single depth-first order traversal, our altered serialization requires two passes through the hierarchy. The reason is that to correctly encode the redirected child pointers, we need to know the final position of a subtree in the serialized representation. To this extent, we perform a first pass where we only compute the final position of all nodes in the serialized representation. In the second pass, we write the serialized structure and use the previously computed positions to assign the correct offset to the nodes that were marked as merged.

Offset scaling After the merging process, each inner node contains a pointer to a subtree, which might be shared by several nodes. These pointers will be represented in the form of 24-bits offsets into the serialized quadtree. The original MH quadtree used 16 bit offsets and added 8 bits of padding. In our case, merged subtrees may be very distant in the final serialized quadtree layout, and therefore we use the padding bits to increase the offset size to 24 bits.

Two nodes at different levels in the quadtree might share the same subtree, since subtrees are merged according to their topology and depth intervals, and not their position in the final structure (see Fig. 5). This means that during the quadtree traversal, the same subtree may be encountered at two different levels. Consequently, we need to ensure the offset scaling factors of those levels are the same. Fortunately, the quadtree depth-first order ensures that the scale factor will always be *one* for the lower levels; imagine a complete quadtree, then the first 12 levels ($4^{12} = 2^{24}$) contain only parent nodes whose offset to the children will fit into 24 bits. Therefore, we restrict merge operations to the lowest 12 levels to ensure that all merged nodes are at levels with scale *one*. Since the lowest tree levels typically contain most of the nodes, very little compression is lost by this restriction.

To ensure that all merged subtrees use unscaled offsets and fit in 24 bits, we determine whether there is a risk to exceed this bound during each merge test. We compute a conservative bound on the number of required bits for the resulting offset and forbid the merging operation if it is too large. This bound is the distance within the original (unmerged) quadtree as computed in [SBE16], since our merging algorithm can only cause a decrease in this offset.

4. Results

We implemented our approach using C++ with multi-threading support and CUDA. The unmerged parts of the MH construc-

Scene		$4K^2$	$16K^2$	$64K^2$	$256K^2$
Closed City	DAG	0.62	3.40	14.90	60.46
	MH	0.41	2.10	9.33	38.43
	Ours	**0.30**	**1.43**	**5.80**	**23.73**
Cityscape	DAG	0.94	3.94	16.38	63.34
	MH	0.60	2.73	11.53	47.07
	Ours	**0.40**	**1.67**	**6.83**	**27.71**
Villa	DAG	1.78	9.27	39.70	166.47
	MH	0.88	5.01	23.61	101.52
	Ours	**0.63**	**3.62**	**17.36**	**74.11**
Ship	DAG	2.01	8.66	35.57	153.67
	MH	1.11	5.64	24.57	102.36
	Ours	**0.82**	**3.83**	**16.34**	**67.64**

Table 1: Size comparison in MB for MH-based compression [SBE16], DAG-based compression [KSA13], and MMH-based compression for our four test scenes and varying shadow map resolutions.

Scene		$4K^2$	$16K^2$	$64K^2$	$256K^2$
Closed City	DAG	0.098	0.53	4.88	65.23
	MH	0.27	1.24	5.80	56.17
	Ours	0.37	1.59	8.72	72.54
Ship	DAG	0.12	0.67	6.16	78.74
	MH	0.31	1.59	7.08	69.30
	Ours	0.54	3.1	15.6	109.9

Table 2: Creation time comparison in seconds of the MH-based compression [SBE16], DAG-based compression [KSA15], and MMH-based compression for at varying resolutions for the Closed City and Ship test scenes.

tion and the baseline comparison are implemented according to [SBE16] and perform most computations on the GPU using NVidia CUDA. All statistics were captured on a Windows 7 PC with an Intel i7-5820K CPU, 16GB of system memory, and an NVidia Titan X GPU. We evaluate our method for four test scenes (Fig. 1 and Fig. 7). The Closed City (613K triangles) and the Cityscape (11K triangles) scene are typical examples of open-world games or architectural scenes. The Villa (89K triangles) scene and the Ship (810K triangles) scene exhibit small geometric details making them hard cases for typical shadowing algorithms.

In Table 1, we compare the memory footprint of traditional MH compression, DAG compression and our MMH compression method. Our solution reduces the memory footprint by up to 60% compared to DAG compression and up to 40% compared to MH-based approaches. Averaged over all scenes and resolutions, our approach achieves 31.3% higher compression rates than traditional MH-based compression.

Table 2 shows a comparison of creation times for our method and the competing approaches. It can be seen that our approach adds an overhead to the construction time that is between 30% and 150% at higher resolutions in comparison to MH-based and DAG-based compression. However, precomputing and compressing a static shadow map only needs to be done once and construc-

Scene	Resolution	Interval mismatch	Offset too large	Full test rejection
Closed City	$16K^2$	67.2 %	0.00 %	32.63%
	$64K^2$	64.8 %	0.28 %	34.76 %
	$256K^2$	63.5 %	11.9 %	24.39 %
Cityscape	$16K^2$	64.88 %	0.00 %	34.90 %
	$64K^2$	62.15 %	1.49 %	36.21 %
	$256K^2$	61.94 %	13.51 %	24.44 %

Table 3: Early rejection statistics showing the effectiveness of each individual test during the examination of node pairs. Note that the rejection criteria are given in the order that they are performed in the implementation, e.g., a pair rejected for interval mismatch will never be tested for offset incompatibility.

tion time is often not considered as significant as memory reduction in applications typically employing this technique.

Large shadow maps cannot be rendered in a single render call, but are instead created using tiled rendering. In our implementation, we use a tile size of $4k \times 4k$ leading to around 1 GB of required GPU memory during construction. CPU memory requirements are smaller, since the CPU only performs the merging operations on the already compressed MH structure tiles. For the scene in Fig. 1, the CPU memory usage never exceeded 600 MB during construction.

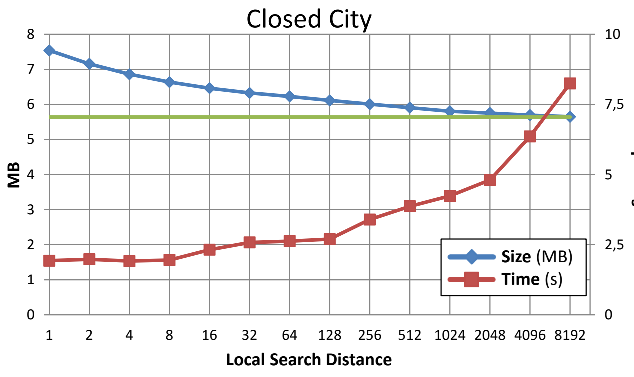
For all scenes, the evaluation time for a full HD image at 256k resolution is below 1 ms for single queries, and between 1 ms and 2.5 ms using a 3x3 hierarchical PCF filter kernel. We include details of the evaluation of run-time performance as supplementary material, since the measured difference to the original MH-based compression method was insignificant in all our experiments. A more detailed comparison of evaluation times between MH-based and DAG compression was presented by Scandolo et al. [SBE16].

We also evaluated the influence of the local search distance parameter (Fig. 8). It can be seen that our standard choice of 1000 leads to a good trade-off between the achieved compression and construction time. Beyond this point the cost of comparing nodes dominates the total time, and the diminishing returns in achievable compression do not seem to justify the strong increase in construction time. Furthermore, local search distance is eventually limited by the maximum offset representable in the final quadtree structure.

Finally, Table 3 shows statistics for the early rejection test during the individual pair-wise subtree matching. It can be seen that the interval-based rejection test quickly eliminates most (avg. 64%) of the comparisons and only one quarter to one third of the subtree pairs perform a full node-by-node comparison. Further, it can be seen that only a small amount of merges are rejected due to the 24-bit offset limit for our choice of local search distance.

5. Conclusion

We introduced the concept of *Merged Multiresolution Hierarchies* for compact representation of extremely high-resolution shadow maps. Our method provides a novel way of fusing the existing MH-based and DAG compression algorithms, which allows simultaneous exploitation of hierarchical and topological similarities leading to significantly higher compression rates. The memory foot-



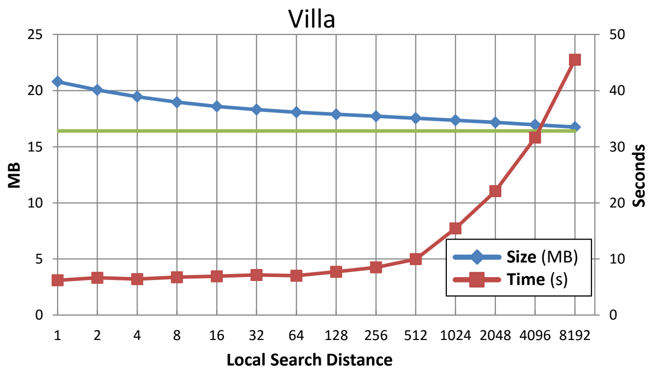


Figure 8: Compressed size vs. subtree merging time for varying local search distances in the CLOSED CITY and VILLA scenes at 64k resolution. The graph shows the diminishing returns of larger distances during the pair matching of subtrees. The green line shows the minimum attainable size, resulting from an infinite search distance. A reasonable trade-off between final size and creation time can be achieved using search distances between 500 and 1000.

print of our compressed representation is up to 40% smaller compared to previous techniques, while maintaining full run-time performance. We proposed an efficient construction scheme based on a hash-based strategy addressing the issue of raised complexity due to more sophisticated subtree merging constraints. Our approach can be used to speed-up shadow computations in real-time graphics applications and enables the use of highly detailed and complex static shadow casters.

6. Acknowledgements

The DRAGON model is available at the *Stanford 3D scanning repository*[†]. The SHIP scene was modeled by Greg Zaal and Chris Kuhn (*blendswap.com*[‡]). The CLOSED CITY and VILLA scenes, and the DAG compression code were kindly provided by Erik Sintorn. This work was partially funded by the EU FP7-323567 project Harvest4D and the Intel VCI at Saarland University.

References

[AH05] ARVO J., HIRVIKORPI M.: Compressed shadow maps. *Vis. Comput. 21*, 3 (Apr. 2005), 125–138. 2

[Bou97] BOUTELL T.: Png (portable network graphics) specification version 1.0. 2

[DKB*16] DADO B., KOL T. R., BAUSZAT P., THIERY J.-M., EISEMANN E.: Geometry and attribute compression for voxel scenes. *Computer Graphics Forum (Proc. Eurographics) 35*, 2 (may 2016). 3

[ESAW11] EISEMANN E., SCHWARZ M., ASSARSSON U., WIMMER M.: *Real-Time Shadows*. A.K. Peters, 2011. 2

[HC07] HUANG Y.-H., CHUNG K.-L.: Texture- and multiple-template-based algorithm for lossless compression of error-diffused images. *Trans. Img. Proc. 16*, 5 (May 2007), 1258–1268. 2

[INH99] IOURCHA K., NAYAK K., HONG Z.: System and method for fixed-rate block-based image compression with inferred pixel values, Sept. 21 1999. US Patent 5,956,431. 2

[KSA13] KÄMPE V., SINTORN E., ASSARSSON U.: High resolution sparse voxel dags. *ACM Transactions on Graphics 32*, 4 (2013). SIGGRAPH 2013. 2, 3, 7

[KSA15] KÄMPE V., SINTORN E., ASSARSSON U.: Fast, memory-efficient construction of voxelized shadows. I3D '15, ACM. 2, 3, 7

[LH07] LEFEBVRE S., HOPPE H.: Compressed random-access trees for spatially coherent data. EGSR'07, pp. 339–349. 3

[LK10] LAINE S., KARRAS T.: *Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation*. NVIDIA Technical Report NVR-2010-001, NVIDIA Corporation, Feb. 2010. 3

[NLP*12] NYSTAD J., LASSEN A., POMIANOWSKI A., ELLIS S., OLSON T.: Adaptive scalable texture compression. In *HPG '12* (2012), Eurographics Association, pp. 105–114. 2

[PM92] PENNEBAKER W. B., MITCHELL J. L.: *JPEG Still Image Data Compression Standard*, 1st ed. Kluwer Academic Publishers, Norwell, MA, USA, 1992. 2

[PWC*09] PAN M., WANG R., CHEN W., ZHOU K., BAO H.: Fast, sub-pixel antialiased shadow maps. *Computer Graphics Forum 28*, 7 (2009), 1927–1934. 2

[SAM05] STRÖM J., AKENINE-MÖLLER T.: ipackman: High-quality, low-complexity texture compression for mobile phones. HWWS '05, ACM, pp. 63–70. 2

[SBE16] SCANDOLO L., BAUSZAT P., EISEMANN E.: Compressed multiresolution hierarchies for high-quality precomputed shadows. *Computer Graphics Forum (Proc. EG) 35*, 2 (May 2016). 1, 2, 4, 6, 7

[SFY13] SHEN L., FENG J., YANG B.: Exponential soft shadow mapping. *Computer Graphics Forum 32*, 4 (2013), 107–116. 2

[SKOA14] SINTORN E., KÄMPE V., OLSSON O., ASSARSSON U.: Compact precomputed voxelized shadows. *ACM Trans. Graph. 33*, 4 (July 2014), 150:1–150:8. 2, 3

[WE03] WEISKOPF D., ERTL T.: Shadow mapping based on dual depth layers. *Eurographics 2003 Short Papers* (2003). 2

[Wil78] WILLIAMS L.: Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph. 12*, 3 (Aug. 1978), 270–274. 1

[Woo92] WOO A.: The shadow depth map revisited. In *Graphics Gems III*, Kirk D., (Ed.). Academic Press, 1992, pp. 338–342. 2

[WP12] WOO A., POULIN P.: *Shadow Algorithms Data Miner*. A K Peter/CRC Press, June 2012. 2

[WSP04] WIMMER M., SCHERZER D., PURGATHOFER W.: Light space perspective shadow maps. EGSR'04, pp. 143–151. 2

† http://www.graphics.stanford.edu/data/3Dscanrep
‡ http://www.blendswap.com