High-Performance Graphics (2021)
N. Binder and T. Ritschel (Editors)

Directed Acyclic Graph Encoding for Compressed Shadow Maps

Leonardo Scandolo, and Elmar Eisemann †

Delft University of Technology

Abstract

Detailed shadows in large-scale environments are challenging. Our approach enables efficient detailed shadow computations for static environments at a low memory cost. It builds upon compressed precomputed multiresolution hierarchies but uses a directed acyclic graph *to encode its tree structure. Further, depth values are compressed and stored separately and we use a bit-plane encoding for the lower tree levels entries in order to further reduce memory requirements and increase locality. We achieve between 20% to 50% improved compression rates, while retaining high performance.*

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

1. Introduction

Shadow Maps (SM) [Wil78] enable real-time shadow computations but suffer from staircase artifacts due to insufficient SM resolutions. While a high SM resolution can mitigate this problem, artifact-free shadows require orders of magnitude larger resolutions than what is supported on consumer GPUs.

In static scenes with closed geometry and static light, it is possible to apply lossless compression schemes to SMs. Such compression still must allow efficient random access, since time budgets for shadow calculations are typically only a few milliseconds. A separate scheme is needed for shadows cast by dynamic objects, but these objects can still be shadowed by the compressed SMs. The main related approaches include voxelized shadows encoded as *directed acyclic graphs* (DAG) [SKOA14] and *multiresolution hierarchies* (MH) [SBE16a], which both lead to tremendous memory savings. We present a novel structure, combining these techniques, lowering memory requirements even further. We start with an MH encoding but propose to decouple the MH structure from the depth values it stores. The MH tree structure will be encoded as a DAG, which vastly reduces its memory footprint, especially at large resolutions. The depth values are compressed separately in a factorized representation. The resulting structure is 20% to 50% smaller than the current state-of-the-art [SBE16b], while maintaining fast random access.

† e-mail: {l.scandolo,e.eisemann}@tudelft.nl

2. Related Work

In this section, we will focus on works related to precomputed shadow generation algorithms. For a more general view on shadows, we refer to [ESAW11].

Arvo et al. [AH05] introduced compressed shadow maps, based on the concept of dual shadow maps [WE03], capturing the first and second surface visible from the light. A SM with depth values within these bounds results in the same shadows. Arvo et al. represent scanlines as linear segments, remaining within the dual SM bounds. While achieving good compression rates, the high amount of line segments that need to be traversed result in low performance. Similarly, Ritschel et al. [RGKM07] compress groups of similar equal-resolution shadow maps by representing pixels for the whole group as line segments.

Scandolo et al. [SBE16a] created a shadow map representation that respects the bounds of a dual shadow map using a scheme in two dimensions instead of one. They create a sparse SM hierarchy based on a quadtree, where depth values at higher levels represent large amounts of similar values in lower levels. This sparsity at low hierarchy levels leads to high compression rates. Merged multiresolution hierarchies [SBE16b] extend this approach by finding regions in the MH that admit a single representation that stays within the dual SM bounds. In practice, when two or more subtrees are compatible, one is modified to hold the single representation values, the others are eliminated and their parents point to the now shared representation. Hereby, memory requirements sink while the traversal method stays identical. This method has also been utilized for compression/rendering of 3D layered volumes [GRRP*20].

A very efficient tree structure are *directed acyclic graphs* (DAGs), which were initially geared towards compressing sparse binary voxel grids [KSA13]. They have proven useful in

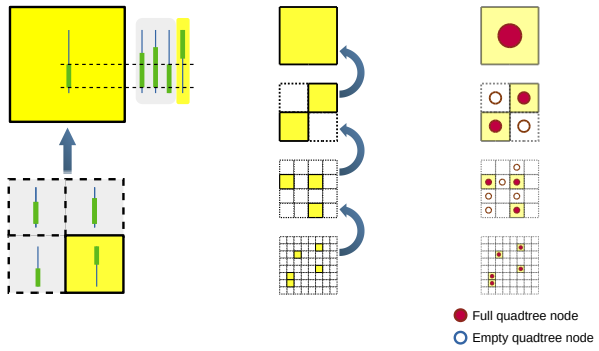

Figure 1: *The core step of MH construction replaces as many values in a higher resolution level by a single value at the next lower resolution level. This process is repeated hierarchically, and then a quadtree is constructed to represent this hierarchy.*

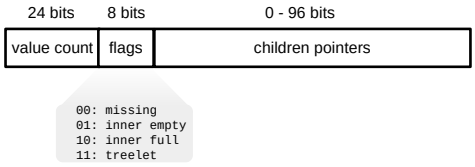

Figure 2: *Inner nodes can be between 8 and 40 bytes long, depending on the amount of children nodes present.*

many applications and recent years brought various improvements [VMG16, VMG17, KRB*16, vdLSE20]. DAGs compress binary *Sparse Voxel Octrees* (SVO) [LK10, CNLE09] by replacing redundant subtrees via a pointer to one common location. If one encodes shadows in a sparse voxel grid spanning the complete scene, the compression via a DAG can lead to good rates [SKOA14, KSA15]. Nevertheless, the original SVO is typically 16 to 18 levels deep, which reflects the effective depth precision and is lower than the typical 24 or 32 bits of a standard SM.

Recent work went beyond binary information in a DAG by decoupling the voxel geometry from the stored values. By keeping extra information to track DAG traversals, one can link a unique index to each region in space represented by a node [DKB*16, DSKA19, CBE20] These indices then allow access to a compressed list of values. Our work is inspired by these structures, but encodes depth values in an MH. We propose several modifications to accommodate the different characteristics of the structure and its values.

3. Our Method

We create a dual shadow map [WE03], encoding the first and second surface as seen from the light source. If the depth values in an SM remain within their respective intervals, shadows will remain correct, thus compression is lossless in the evaluation sense. Multiresolution hierarchies (MH) [SBE16b] encode sparse depth textures as a hierarchy, where each level halves the resolution along each axis. The core of the algorithm is shown in Fig. 1: a group of four pixels at a certain level and their intervals are analyzed to replace as many as possible by a single new interval on the next (quarter-resolution) level of the hierarchy. Only the children that cannot be represented by this interval are kept, the other intervals are represented implicitly by the parent. This process is repeated throughout all levels (bottom to top). The resulting structure is encoded as a sparse quadtree with intervals in some nodes and special empty nodes with no values to preserve connectivity.

Merged Multiresolution Hierarchies (MMH) additionally exploit the redundancy in the structure. A pair of subtrees in the hierarchy with the same topology and intersecting depth intervals can be replaced by a single subtree with values inside the corresponding intersected intervals. Similar to MMHs, we exploit redundancy in this tree of depth intervals; we compress the tree structure and the depth intervals separately, which proves very beneficial.

In the following, we will explain the basis of our tree encoding (Sec. 3.1), followed by an introduction of a special representation for the final levels of the resulting quadtree (Sec. 3.2), and an explanation of a separate encoding scheme for the depth values (Sec. 3.3). Finally, we will provide details on the structure traversal (Sec. 3.5), and how to reconstruct depth values from our compressed representation (Sec. 3.6).

3.1. Structure compression

Our method builds upon the encoding of attributes in a DAG representation [DKB*16]. We aim to separate the sparse quadtree structure from the stored values, which enables us to store the tree as a DAG with a single representation of every subtree topology. In contrast to MMHs, we are only concerned with the structure and not the values (which will be encoded separately), enabling much more merging options. Similar to attribute-encoding DAGs [DKB*16], we store the depth values according to a depth-first traversal of the tree and each inner node of the DAG stores the total amount of values in its corresponding subtree (see Fig. 2). During a tree traversal, we can then keep track of the index of the current visited node, allowing us to retrieve the corresponding value from the encoded depth-value list. As a final note, in contrast to typical DAG structures used for voxel encoding, we allow the presence of empty nodes, which do not contain values.

3.2. Treelets

Previous DAG implementations encode the last levels of the hierarchy as a flat bitmap to reduce pointer memory. While useful for sparse binary voxel data, MHs hierarchically encode one 32-bit value per location, so reverting to a dense representation would require storing many repeated values. Instead, we maintain the hierarchical structure via a binary tree representation.

In practice, we store subtrees of height ≤ 4 as a 4-level treelet (see Fig. 3). We assume a full tree, where a bit is set to one if a node contains a value and zero otherwise, stored in depth-first

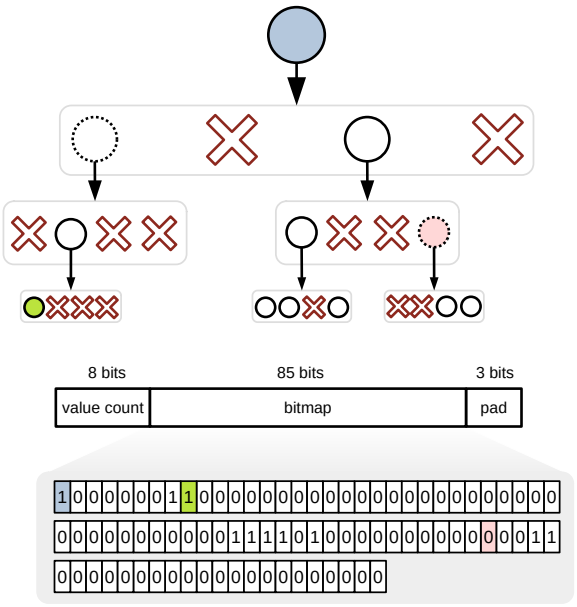

Figure 3: *A 4-level subtree is encoded as a 12 byte treelet in binary form with depth-first ordering. Empty inner nodes are drawn with a dotted line. Some nodes are highlighted in color to match their position in the binary representation.*

order for efficient traversal. A full quadtree of height n then fits into $\sum_{d=1}^{n} 4^{(d-1)}$ bits.

3.3. Depth value compression

Initially, we create the depth-first ordered depth-interval list. As elements close in the list mostly correspond to spatially-close locations, one can expect overlapping depth intervals. To this extent, we partition the list into chunks of 32 intervals and compress each chunk individually. We proceed in two steps; finding a common representative, and then a minimal bit representation.

For each chunk of 32 intervals, we initially want to find a depth value that is contained by as many of those intervals as possible. We can then store this representative value, along with a bitmask marking which intervals it satisfies. We consider all depth intervals as closed intervals, therefore one or more of the interval limits (min or max) must satisfy as many intervals as possible. A brute-force approach can test all such values against all intervals and keep the one that satisfies the maximum amount. A faster approach is to create a sorted list of all limits in increasing order. Starting from the lowest minimum, which we know satisfies one interval, we traverse the list while keeping a counter of satisfied intervals. While traversing the list, we increase the counter when we encounter a minimum limit, and decrease it when we encounter a maximum limit. The value encountered when the counter was maximized is selected as a representative.

For the remaining chunk intervals not satisfied by the chosen representative, we could proceed iteratively, choosing further representatives until all are covered. Nevertheless, this would quickly yield diminishing returns and also reduce evaluation and compression efficiency. Instead, we use a factorized representation. We encode a depth value for each remaining interval as $\mathbf{c_l} + \mathbf{c_f} * \mathbf{i_v}$, where c_l and c_f are constant factors per chunk, and i_v is different for each of the remaining intervals. The goal is to store c_l and c_f as 32-bit floating point values, and store the i_v factors as integers with as few bits as possible. To achieve this, we first inspect the bounds of the intervals and find the lowest bound v_l and highest bound v_h in the set. We assign v_l to c_l, and for increasing choices of bit count b for the interval factors, we define an appropriate c_f and test whether all intervals can be satisfied. For a choice of b, we set $c_f = (v_h - v_l)/(2^b - 1)$, the smallest factor such that the depth values can span the range of values from v_l to v_h with only b bits for each i_v. We then attempt to represent each of the intervals as $v_b = c_l + c_f * i_{vb}$, where i_{vb} has b bits. There may not exist such i_{vb} factor for all intervals, especially if very small intervals are present, so we repeat this process for increasing values of b. Once we find a suitable value for b, we store this choice in the chunk, as well as c_l, c_f. We store the list of b-bits i_v factors separately, and store the address to the start of the list in the chunk (see Fig. 4). To increase evaluation efficiency, we attempt this encoding only with bit counts that are multiples of 8, and when $b = 32$, we resort to simply storing the depth values directly.

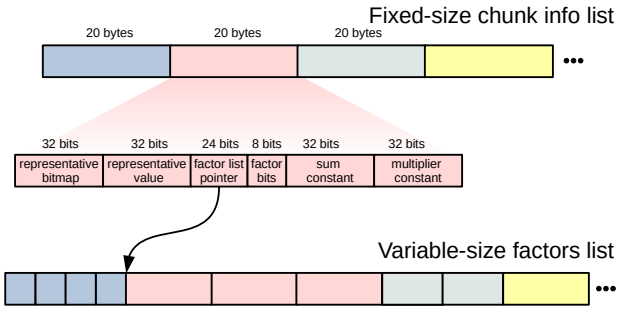

Figure 4: *Values are encoded in two lists, a fixed-sized element list containing information of each 32-element value chunk, and a variable-sized element list containing multipliers needed to reconstruct the original values.*

3.4. Encoding

The DAG structure contains two different types of members: inner nodes and treelets. Inner nodes consist of 3 parts: children flags, subtree value count, and child pointers (see Fig. 2). Child flags store 2 bits per child, totaling 8 bits and they encode the type of each child: non-existant, full inner node, empty inner node, or treelet. The subtree value count is stored in 24 bits to ensure a 32-bit alignment for nodes. This allows us to represent up to roughly 16 million values, which is enough for our tests, but can easily be extended by using additional bits. Finally, the children pointers are stored as 32-bit integers, which ensures a 32-bit alignment.

Treelets encode the value count as an 8-bit value, since our choice of height results in a maximum of 85 values. This is then

Algorithm 1 Treelet traversal algorithm pseudo-code

```
function GetValueIndexTreelet(coord,searchState)
    bitmap ← readBitmap(searchState.nodePtr)
    for i ∈ [0,4) do
        level ← 3-i                ▷ Check levels from bottom to top
        nodeBitIndex ← preorderIndex(coord, level)
        exists ← checkBit(bitmap, nodeBitIndex)
        if exists then
            count ← countSetBitsUpto(bitmap, nodeBitIndex)
            return searchState.valueCount + count + 1
        end if
    end for
    return searchState.valueLast
end function
```

followed by an 85-bit bitmap encoding the existing nodes, stored in depth-first order. This ordering respects the overall depth-value list indexing, and therefore allows us to compute the index by counting the amount of set-bits in the bitmap up to the corresponding index.

The depth value list is divided into two parts: a fixed-size chunk info list, and a variable-size list of multipliers (see Fig. 4). The chunk info list contains the representation bitmap for the chunk, as well as the representative value, c_l and c_f (defined in Sec. 3.3) as 32-bit floating point values. Lastly, the info list contains a pointer to the start of the multipliers, stored contiguously as a different list.

3.5. Structure traversal

Traversal of the structure is similar to a typical quadtree traversal, except that two value indices need to be maintained: the last seen value index v_{last}, and the current values count v_{count}. Both indices are initialized at 0, which corresponds to the value stored at the root node. Starting from the root node, we perform the same operations for all inner nodes encountered. We compute the child index from the sample coordinate, and inspect the corresponding flags stored in the node. If the child does not exist, we simply return the last seen value index. In any other case, we update the v_{count} by adding the value counts from existing children at lower indices. If the child is full, we copy v_{count} to v_{last}, which corresponds to the index of the current child. For empty or full inner nodes, we simply repeat the same steps until the child does not exist, or a treelet child is found.

Treelet traversal essentially inverts the traversal direction. Since we have immediate access to the complete structure in the form of a bitmap, we can check for the existence of a node at the last level of the structure, and if that is not the case, progressively check the parent node. Initially v_{count} points to the index of the first node stored in the treelet. Once a node is found, its value index will correspond to v_{count} plus the count of existing nodes before the found node in depth-first order. To efficiently compute this count, we store the treelet bitmap in depth-first order, which allows us retrieve the value simply by counting the set bits in the bitmap up to the node location. This ordering can be computed as:

$$n + \sum_{l=0}^{l_{max}} \left(\sum_{i=0}^{l} 4^i * l_{index} \right)$$

Algorithm 2 Value retrieval algorithm pseudo-code

```
function GetValue(index)
    ChunkIndex ← index / floatsPerChunk
    InChunkIndex ← index % floatsPerChunk
    bitmap ← readBitmap(ChunkIndex)
    exists ← isBitOn(bitmap, InChunkIndex)
    if !exists then
        return readBaseValue(ChunkIndex)
    end if
    factorIndex ← countSetBitsUpto(bitmap, InChunkIndex)
    factorBits ← readFactorBits(ChunkIndex)
    factorsPointer ← readFactorsPointer(ChunkIndex)
    i_v ← readFactor(factorsPointer, factorBits, factorIndex)
    if factorBits == 32 then
        return interpretAsFloat(i_v)
    else
        c_l, c_f ← readMultiplyAddFactors(ChunkIndx)
        return c_l + c_f * i_v
    end if
end function
```

where n is the level of the found node (starting from zero), l_{max} is the treelet maximum level, and l_{index} is the child index at level l in the path towards the found node. For our 4-level treelets, this resolves to $n + 21 * l_0 + 5 * l_1 + l_2$. If no node is found, we simply return the v_{last}. Alg. 1 shows pseudo-code for treelet traversal.

The evaluation code can be easily extended to handle PCF shadowing, similarly to the method employed for MMHs [SBE16b]. At the start of the pixel evaluation, we analyse the quadtree coordinates of the kernel samples, traverse the hierarchy down to the lowest common ancestor, and store its location. We then simply obtain the stored value for each sample iteratively, starting our traversal at the lowest ancestor, thereby reducing the total amount of traversal steps. As an added optimization, we cache the last node traversed to test whether the next sample evaluation can be started at that same location, should the cached node be an ancestor of the next sample.

3.6. Value retrieval

Once we obtain the value index from the DAG structure traversal, the procedure to retrieve the depth value is straightforward. If the bitmap value for the index in the corresponding chunk info value is set, we simply return the representative value. Otherwise, we read the per-chunk multiplier bit size, and retrieve the multiplier value using the pointer stored with the chunk info. We then compute the value with a multiply-add operation using the factors stored with the chunk info. In the special case of a 32-bits multiplier bit size, the multiplier is the actual floating-point value, so we simply reinterpret the index value as a 32-bit floating-point variable. Pseudo-code for the value retrieval algorithm is listed as Alg. 2.

3.7. Skip list

To accelerate tree traversal, we can create a compact node list densely representing a specific level near the root. This allows us to start traversal at that level, skipping levels above it, and in some

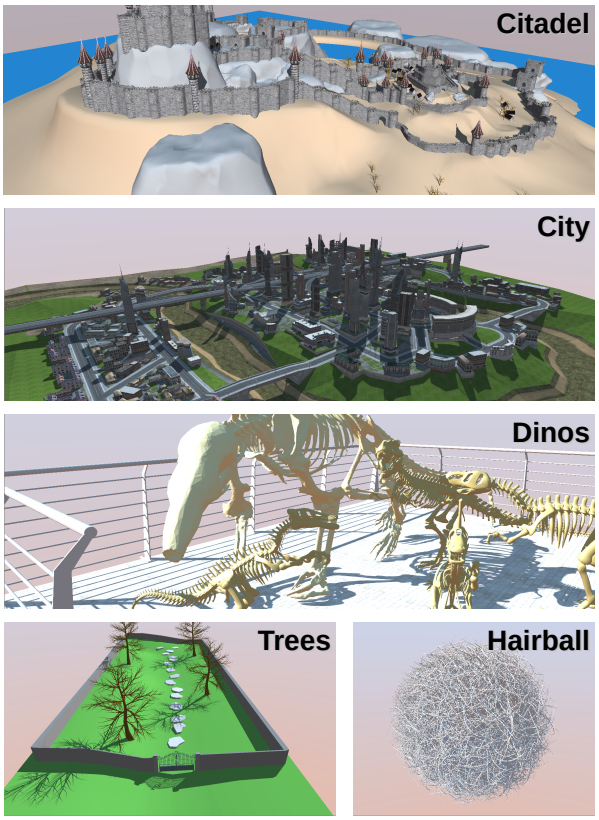

Figure 5: *Scenes used for our tests.*

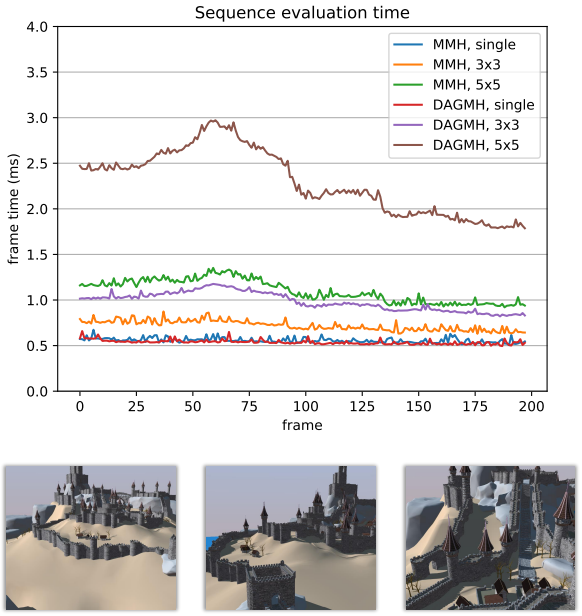

Figure 6: *Per-frame timing of shadow evaluation for our solution (DAG-MH) and MMHs at full HD in a flyby of the citadel scene at $64K^2$ resolution. Initial, mid, and end frame shown below.*

cases to return a value without any tree traversal steps. In practice, for larger resolution we create this skip list at level 6. Hereby, we maintain efficient traversal of the tree while keeping the total memory needed low, as this list typically represents a few KB, which is less than one percent of the total structure size.

4. Results

All results were obtained on an Intel i7-5820K CPU running Windows 10, 32GB of memory and an NVIDIA Titan V GPU with 12GB of video memory. Fig. 5 shows the test scenes, of which the Citadel and City scenes are examples of large scale urban environments, whereas the Trees and Dinos scenes are examples of smaller but geometrically complex scenes. The Hairball scene is a standard highly complex mesh, used as a stress test.

4.1. Compression

We compare the compression rate of our proposed data structure against a MMH [SBE16a] and DAG-based voxelized shadows [KSA15]. Table 1 shows that our method outperforms the competitors in all cases. In the best case, it uses as little as half the memory (Hairball scene). The citadel scene's $512K^2$ SM only requires around 31MB. Table 2 shows compression times for our solution and competing schemes for the citadel scene at different resolutions. Dual shadow map rendering times account for roughly 80% of the total creation times for our solution. The remaining time is evenly split between the initial MH creation, and the DAG and depth value encoding.

4.2. Evaluation efficiency

We also compare the evaluation efficiency of our solution against a standard MMH [SBE16b] using single value and hierarchical PCF-based approaches. We use a prerecorded path in the citadel scene with full HD screen resolution and a $64K^2$ shadow map resolution. Fig. 6 shows the timings. Our solution can compute shadows with a single sample in around 0.5 ms, 3x3 PCF in 1ms and 5x5 pcf in roughly 2.5ms. Single evaluation is roughly as fast as MMH evaluation, whereas 3x3 PCF and 5x5 PCF are roughly half as fast. This is the result of having to retrieve a value count for each non-traversed node, which results in more memory accesses per traversal step. Similar to other MH-based solutions, filtering is hierarchical, meaning that the kernel footprint can always roughly match the pixel footprint, resulting in antialiased shadows with small kernel sizes [SBE16a]. Nevertheless, accelerating filtered lookups remains an important avenue of future work. DAG-based voxelized shadow solutions [KSA15] can also be evaluated in under 1ms, even for large kernel sizes (9x9), although the filtering is not hierarchical, which can result in flickering for large pixel footprints.

4.3. Parameter analysis

Treelet size: Increasing the treelet depth saves memory that would otherwise be used to store node pointers in a standard tree representation. Nevertheless, this representation cannot capture similarities

Scene	Method	$4k^2$	$8k^2$	$16k^2$	$32k^2$	$64k^2$	$128k^2$	$256k^2$	$512k^2$
	DAG	0.737	1.79	3.88	8.25	17.00	34.4	69.6	139.7
Citadel	MMH	0.351	0.765	1.58	3.24	6.52	13.26	26.93	54.49
	DAG-MH	0.252	0.571	1.21	2.43	4.68	8.84	16.59	31.16
	DAG	0.929	2.01	3.90	8.10	16.73	33.72	67.89	135.9
City	MMH	0.411	0.845	1.71	3.44	6.99	13.90	27.90	55.54
	DAG-MH	0.329	0.663	1.29	2.47	4.70	8.83	16.73	31.45
	DAG	2.06	5.03	10.71	22.41	45.62	93.46	190.6	384.6
Trees	MMH	0.767	1.898	4.394	9.73	20.85	43.36	88.34	177.6
	DAG-MH	0.480	1.23	2.96	6.54	13.66	27.53	54.19	104.9
	DAG	1.82	3.67	7.29	15.21	31.93	67.24	139.1	281.4
Dinos	MMH	0.875	1.79	3.69	7.65	15.90	32.41	65.41	131.2
	DAG-MH	0.542	1.07	2.09	4.04	7.71	14.57	27.40	51.50
	DAG	21.21	40.69	71.52	133.60	262.50	532.23	1086.56	-
Hairball	MMH	5.56	12.37	27.28	58.53	122.46	251.10	-	-
	DAG-MH	3.73	8.66	18.01	35.33	67.32	127.09	239.9	-
	Uncompressed	64	256	1024	4096	16384	65536	262144	1048576

Table 1: *Memory requirements comparison in MB between our solution (DAG-MH), MMHs, and DAG encodings.*

Method	$4k^2$	$16k^2$	$64k^2$	$256k^2$
DAG	0.047	0.351	4.110	59.85
MH	0.251	0.774	5.488	59.30
MMH	0.257	1.465	10.67	84.42
DAG-MH	0.247	0.875	6.225	62.33

Table 2: *Compression time comparison in seconds for our solution (DAG-MH) and competing methods.*

	Treelet levels	2	3	4	5
	DAG structure size	0.756	0.696	0.622	0.788
	Inner nodes	52746	47086	26425	12452
$16k^2$	Nodes size	0.756	0.668	0.375	0.172
	Treelets	23	5683	21595	14666
	Treelets size	<0.001	0.027	0.247	0.615
	DAG structure size	2.475	2.399	2.261	3.493
	Inner nodes	177882	170726	124947	66475
$64k^2$	Nodes size	2.475	1.585	0.831	0.704
	Treelets	23	7179	47157	61164
	Treelets size	<0.001	0.814	1.430	2.789

Table 3: *DAG structure size and composition characteristics when using different treelet heights for the Citadel scene.*

inside different treelets, which is why DAG representations work better for very large trees. Furthermore, the treelet bitmap needs to be read during traversal, which can result in large amounts of memory reads when the treelet depth is high. Table 3 shows the results of compressing the Citadel scene using different treelet sizes. 4-level treelets result in the best compression rates, due to the best compromise between treelet and inner-node memory requirements.

Value chunk size: Table 4 showcases the size and composition of the depth representation for different chunk sizes. With larger chunk sizes, fewer values can be covered by the representative value overall, and more bits are needed to represent each remaining

	Values per chunk	16	32	64	128
	DAG values size	0.688	0.588	0.578	0.592
	Chunks map size	0.397	0.220	0.132	0.088
$16k^2$	Chunks factors size	0.291	0.368	0.446	0.504
	Avg. factor qty	57.6%	61.5%	64.6%	67.7%
	Avg. factor bits	10.95	13.09	15.33	16.71
	DAG values size	2.897	2.423	2.362	2.388
	Chunks map size	1.728	0.960	0.576	0.384
$64k^2$	Chunks factors size	0.169	1.463	1.786	2.004
	Avg. factor qty	52.6%	56.3%	59.5%	62.3%
	Avg. factor bits	10.87	12.89	15.13	16.48

Table 4: *Value compression rates and characteristics for different chunk sizes tested for the Citadel scene.*

value as a factor. Past 32, we see diminishing returns, and past 128 values, we see an increase in memory consumption as more bits are used for storing factors. Further, retrieving a value can require accessing the complete value bitmap, making smaller chunk sizes more attractive. A size of 32 achieves the best compromise between memory savings and access time.

5. Conclusion and future work

We have introduced a novel encoding for compressed SMs, relying on three key components: DAG encoding of the quadtree structure, use and encoding of treelets for lower tree levels, and a specialized depth-value compression scheme. The resulting structure reduces memory requirements up to 50% compared to state of the art solutions while retaining real-time evaluation rates. Interesting avenues of future work include using more compact DAG representations, such as symmetry-aware DAGs [VMG17], and considering lossy representations [vdLSE20].

Acknowledgments This work is partly supported by VIDI NextView, funded by NWO Vernieuwingsimpuls.

References

[AH05] ARVO J., HIRVIKORPI M.: Compressed shadow maps. *Vis. Comput. 21*, 3 (Apr. 2005), 125–138. 1

[CBE20] CAREIL V., BILLETER M., EISEMANN E.: Interactively modifying compressed sparse voxel representations. In *Computer Graphics Forum* (2020), vol. 39, Wiley Online Library, pp. 111–119. 2

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009), pp. 15–22. 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). 2

[DSKA19] DOLONIUS D., SINTORN E., KÄMPE V., ASSARSSON U.: Compressing color data for voxelized surface geometry. *IEEE TVCG 25*, 2 (2019), 1270–1282. 2

[ESAW11] EISEMANN E., SCHWARZ M., ASSARSSON U., WIMMER M.: *Real-Time Shadows*. A.K. Peters, 2011. 1

[GRRP*20] GRACIANO A., RUEDA-RUIZ A. J., POSPISIL A., BITTNER J., BENES B.: Quadstack: An efficient representation anddirect rendering of layered datasets. *IEEE TVCG* (2020). 1

[KRB*16] KÄMPE V., RASMUSON S., BILLETER M., SINTORN E., ASSARSSON U.: Exploiting coherence in time-varying voxel data. In *Proceedings of ACM i3D* (2016), pp. 15–21. 2

[KSA13] KÄMPE V., SINTORN E., ASSARSSON U.: High resolution sparse voxel dags. *ACM Transactions on Graphics 32*, 4 (2013). SIGGRAPH 2013. 1

[KSA15] KÄMPE V., SINTORN E., ASSARSSON U.: Fast, memory-efficient construction of voxelized shadows. I3D '15, ACM. 2, 5

[LK10] LAINE S., KARRAS T.: *Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation*. NVIDIA Technical Report NVR-2010-001, NVIDIA Corporation, Feb. 2010. 2

[RGKM07] RITSCHEL T., GROSCH T., KAUTZ J., MÜELLER S.: Interactive illumination with coherent shadow maps. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques* (2007), EGSR'07, Eurographics Association, pp. 61–72. 1

[SBE16a] SCANDOLO L., BAUSZAT P., EISEMANN E.: Compressed multiresolution hierarchies for high-quality precomputed shadows. *Computer Graphics Forum (Proc. EG) 35*, 2 (May 2016). 1, 5

[SBE16b] SCANDOLO L., BAUSZAT P., EISEMANN E.: Merged multiresolution hierarchies for shadow map compression. *Computer Graphics Forum 35*, 7 (2016), 383–390. 1, 2, 4, 5

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

[vdLSE20] VAN DER LAAN R., SCANDOLO L., EISEMANN E.: Lossy geometry compression for high resolution voxel scenes. *Proceedings of ACM i3D 3*, 1 (2020), 1–13. 2, 6

[VMG16] VILLANUEVA A. J., MARTON F., GOBBETTI E.: Ssvdags: symmetry-aware sparse voxel dags. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2016), pp. 7–14. 2

[VMG17] VILLANUEVA A. J., MARTON F., GOBBETTI E.: Symmetry-aware sparse voxel dags (ssvdags) for compression-domain tracing of high-resolution geometric scenes. *Journal of Computer Graphics Techniques Vol 6*, 2 (2017). 2, 6

[WE03] WEISKOPF D., ERTL T.: Shadow mapping based on dual depth layers. *Eurographics 2003 Short Papers* (2003). 1, 2

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