# Fast, Memory-Efficient Construction of Voxelized Shadows

Viktor Kämpe, Erik Sintorn, Dan Dolonius, and Ulf Assarsson

**Abstract**—We present a fast and memory efficient algorithm for generating Compact Precomputed Voxelized Shadows. By performing much of the common sub-tree merging before identical nodes are ever created, we improve construction times by several orders of magnitude for large data structures, and require much less working memory. To further improve performance, we suggest two new algorithms with which the remaining common sub-trees can be merged. We also propose a new set of rules for resolving undefined regions, which significantly reduces the final memory footprint of the already heavily compressed data structure. Additionally, we examine the feasibility of using CPVS for many local lights and present two improvements to the original algorithm that allow us to handle hundreds of lights with high-quality, filtered shadows at real-time frame rates.

**Index Terms**—Shadow, voxel, directed acyclic graph, real-time

## 1 Introduction

The current de-facto standard algorithm for rendering shadows from distant light sources (e.g., the sun) in large open scenes is the *Cascaded Shadow Maps* (CSM) [1], [2] approach. The idea is to split the current camera-view frustum into several regions, or *cascades*, and to render a traditional shadow map [3] for each. Rendering, and performing look-ups in, a shadow map is extremely fast on current graphics hardware, and the CSM approach helps significantly in reducing under-sampling artifacts that occur when a shadow map is sampled at a frequency that is lower than the screen-sampling frequency. On the other hand, the algorithm will, by design, sample the shadow-casting *geometry* in distant regions very sparsely, which also leads to geometric aliasing artifacts.

Recently, a different approach, called *Compact Precomputed Voxelized Shadows* (CPVS), has been suggested [4]. Here, a shadow map is rendered at a resolution that is high enough to avoid geometric aliasing. This shadow map is then converted into a *Directed Acyclic Graph* (DAG) that contains the voxelized, binary shadow information for any point in the scene. The compact DAG is generated by merging common sub-trees of an intermediate *Sparse Voxel Octree* (SVO) representation. The DAG representation can be two orders of magnitude smaller than the corresponding shadow map at high resolutions. When the scene can be described entirely by closed geometry, compression rates increase to three orders of magnitude. The method can only be used to cast shadows from static geometry, as the compression is done in a pre-compute pass, but dynamic geometry can receive shadows, and high-quality filtered look-ups are evaluated at a cost that is much lower than what would be required to render and evaluate a CSM. Shadows from dynamic geometry can then easily be supported using, for instance, CSM, at an overall much lower cost than using CSM for the full scene.

However, usability of the method is highly limited by the time taken to generate the CPVS. We present a number of elegant, non-intuitive, modifications to the algorithm to improve its performance. We show that much of the common sub-tree merging can be performed during node-insertion, before identical nodes are even created. For scenes consisting of closed geometry, where large regions inside objects need no shadow classification and can be considered undefined, we will show (in Section 6) that this results in a performance increase of approximately 150× for the actual DAG construction and a performance increase of approximately 15× for the full construction. We show that for such scenes, the DAG obtained from our improved node-insertion algorithm is already orders of magnitude smaller than the corresponding shadow map and can be used without further compression if fast build times are a priority. Additionally, we show that if a slight increase in final DAG size is acceptable, or if the scene geometry contains no self-intersections, the rigorous identification of undefined regions suggested in the paper by Sintorn et al. [4] can be replaced by a much simpler algorithm that can be an order of magnitude faster at lower resolutions.

Otherwise, the DAG can be further compressed and we suggest two novel algorithms for improving the performance of this step. For scenes with no closed geometry, our new algorithms improve performance by approximately a factor 2.5×. Besides making CPVS an even more attractive alternative to shadow maps or light maps, these performance improvements open up for the possibility of creating the data structure during level load, or even distributing the generation over a large number of frames for a slowly moving dynamic light (e.g., the sun).

Another contribution is a new set of rules for deciding on how to resolve undefined regions. Our new method will set undefined voxels to lit or shadowed in such a way that the number of unique nodes are kept locally minimal. This

• *The authors are with the Department of Computer Science, Chalmers University, Goteborg 41296, Sweden.*
*E-mail: {kampe, erik.sintorn, dolonius, uffe}@chalmers.se.*

*Manuscript received 14 Sept. 2015; revised 27 Jan. 2016; accepted 16 Feb. 2016. Date of publication 9 Mar. 2016; date of current version 7 Sept. 2016.*
*Recommended for acceptance by P. Sander and J. Keyser.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/TVCG.2016.2539955*

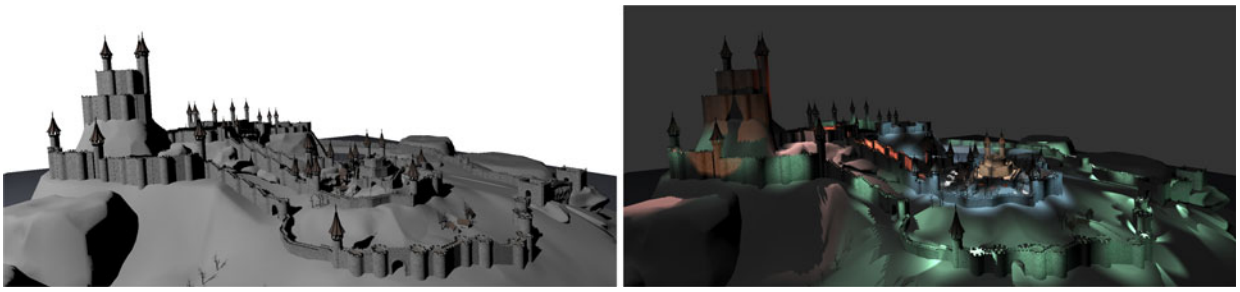


Fig. 1. The left image shows a scene lit by the sun with precomputed voxelized shadows of resolution $262{,}144^3$. Our novel algorithm generates this shadow information in 32 seconds and compresses it to 48 MB (versus 100 MB for the previous CPVS method). To the right is the same scene lit by 165 spotlights with precomputed shadows, each with a resolution of $8{,}192^3$. The average build time for these CPVSs is 114 ms, and the average size is 0.5 MB (versus 128 MB for a 16-bit shadow map). Evaluating shadows for all lights at 1,920 $\times$ 1,080 takes 3.2 ms.

results in an up to $3\times$ reduction in size of the final data structures for the tested scenes.

Additionally, we have explored how the algorithm performs for small, local lights. Such lights will not require extreme resolutions, but we will show that the CPVS can still offer data structures that are one to two orders of magnitude smaller than the corresponding shadow map at resolutions of e.g., $8{,}192^3$, making them an affordable alternative to shadow maps also in scenarios where we have many bounded lights (see Fig. 1). In Section 5, we will show that with two small but important improvements to the algorithm, they can be combined with a simple light-culling technique to provide shadows from hundreds of lights with high-quality filtering at real-time framerates.

## 2 Previous Work

Rendering shadows in real time has been a hot research topic for nearly four decades, and a complete overview is out of scope for this article. Instead, we refer the reader to the book by Eisemann et al. [5]. In this section, we will briefly overview recent work that is closely related to our topic.

*Precomputed and compressed shadows.* Evaluating visibility between a view sample (of a pixel) and, e.g., a light-source is often among the most time-consuming parts of generating an image in real time, and it is common practice to pre-compute as much of this work as possible (see the survey by Ramamoorthi [6]). Specifically, if the light and shadow-casting geometry can be considered static, the shadow information can be precomputed and stored in a *light map* and then be queried with a simple texture lookup while shading the view sample. Even though a number of lossy compression schemes have been suggested for this type of data (see, e.g., the works of Rasmusson et al. [7] for a survey of hardware accelerated light-map compression, or Lefebvre and Hoppe [8] for a well performing hierarchical compression scheme), the memory footprint can easily become unreasonable if high resolutions are desired. These methods can also only support static shadow *receivers* and require a unique UV-parameterization for all objects. The memory requirements are even more unsustainable if many lights are to be considered.

Therefore, it can be preferable to pre-compute and store a representation of the shadow-casting geometry (e.g., a shadow map) instead. For distant lights in a large open scene, this can be as simple as rendering a large shadow map for all static geometry and using that instead of real-time methods for distant geometry (see e.g., the presentation by Schultz [9]). Since this information will usually be very memory expensive, it is desirable to compress it, if this can be done without introducing artifacts or too expensive decoding. The methods suggested by Arvo and Hirvikorpi [10] and by Sintorn et al. [4] both achieve high compression rates while allowing for fast filtered shadow lookups. This paper is an extended version of the paper by Kämpe et al. [11].

*Rendering with many lights.* The problem of rendering scenes with many light sources in real time has received much attention lately, both by researchers and by the industry. A common scenario in real-time applications is that there are many (hundreds or thousands) of lights in the scene, but each light has a bounded influence region. To achieve real-time frame rates in such scenes, the lights must be culled efficiently. Examples of such techniques include *Tiled Shading* [12], *Forward+* [13] and *Clustered Shading* [14]. These techniques do not explicitly take shadowing into account, however. In a recent paper by Olsson et al. [15], real-time shadows for hundreds of lights are shown to be feasible by carefully rendering only those parts of the shadow maps that are required and only at a resolution that gives an approximate one-to-one mapping between view samples and shadow-map samples. This latter restriction means that the shadow-casting geometry might be gravely undersampled, but the method shows promising results and is currently the best candidate in a setting where all geometry is dynamic. This method would be a good compliment to our algorithm, to handle shadows cast from dynamic objects, while avoiding the large workload of the static shadow casters.

## 3 Construction

In this section, we will explain how we build a partially compressed DAG from a set of depth maps. We will quickly review how SVO construction has been done in previous work [4], explain how many of the identical nodes can be culled in the insertion pass (Sections 3.1 and 3.2), discuss approaches to finding regions where shadow information can be considered undefined (Section 3.3), and finally, explain our new approach to deciding how nodes intersecting these regions can be resolved (as either completely shadowed or completely lit) for an improved memory footprint (Section 3.4).

The CPVS stores binary visibility information for every cell in a grid that is a discretization of the light's *Normalized*

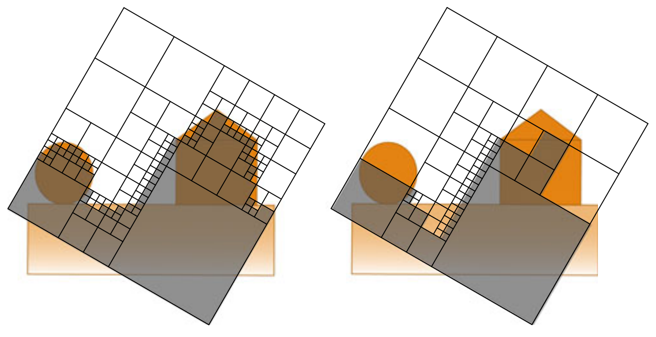


Fig. 2. With the closed object optimization, the finest resolution is mainly constructed along the mid-air shadow boundary, but the majority of nodes along the boundary become identical.

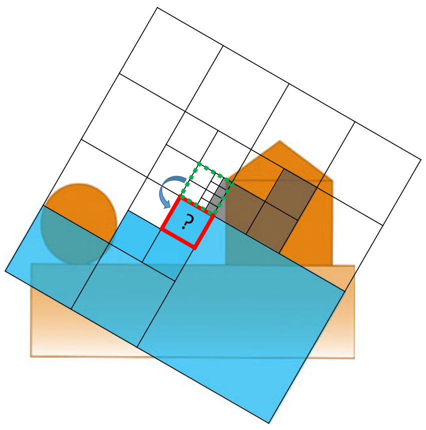


Fig. 3. For each volume to be constructed (blue) that does not contain shadow casting surfaces, we determine if we can reuse the representation of the volume closer to the light, thereby culling construction work.

*Device Coordinates* (NDC). Storing the shadows as a dense grid, or even an SVO, would consume a prohibitive amount of memory at high resolutions. A DAG, on the other hand, exploits the numerous similarities within this grid and achieves much more efficient storage while maintaining fast traversal of the data structure.

The DAG can be constructed top-down with visibility information provided by a depth map of corresponding resolution [4]. During construction, a voxel at the finest resolution is classified as lit or shadowed by comparing its depth against the depth map. To determine if a larger volume, for higher levels, is fully lit or shadowed, the z-bounds of the volume is tested against a min-max hierarchy of the depth map. Starting with the root volume, each of its eight subvolumes is constructed recursively. The recursion ends at the finest resolution, or when reaching a fully lit or shadowed subvolume. During construction, nodes are inserted in a DAG, such that a node describes its homogeneous subvolumes in a bit mask and each non-homogeneous subvolume by a reference to another node.

When it is possible to determine that no shadow queries will be made in a volume, e.g., because it is inside a closed object, the shadow value of the volume can be chosen freely. This allows formation of larger homogeneous regions, which can decrease the memory consumption considerably.

After the top-down construction and insertion of nodes into the DAG, identical nodes are merged to obtain the globally minimal number of nodes. This compression is the subject of Section 4.

## 3.1 Workload

Typically, large volumes of the scene will be homogeneously lit or shadowed and result in early termination in the DAG. These regions require very little memory and are fast to construct. At the boundaries between lit and shadowed space, however, we want the accuracy of the finest voxel resolution. The shadow boundaries will therefore dominate the memory consumption as well as the construction time.

The shadow boundaries consist of the shadow-casting surfaces themselves and the boundaries in mid air between shadow casters (aligned with the light direction). Closed objects enable a relaxation of the boundaries around the shadow-casting surfaces (see the sparser voxel representation of the roof in Figs. 2b versus 2a) and allow early termination of the DAG (and its construction), which saves both memory and construction time.

We still need to resolve the shadow along the mid-air boundaries to the finest resolution. Fortunately, a cross section of a mid-air boundary is identical for all depths between the shadow-casting surfaces (see Fig. 2), which results in very few unique nodes in the final DAG.

With the mid-air boundaries not contributing to the final node count, the final memory consumption scales as if we voxelized only the shadow-casting surfaces. With the closed object optimization, the final memory consumption of the DAG instead scales as if we only voxelized the silhouettes of the shadow-casting surfaces, i.e., as a one-dimensional curve instead of a two-dimensional surface. However, in the original algorithm, it is only the *final* memory consumption that scales as the silhouettes. During construction, the number of nodes to insert into the DAG is still proportional to the number of non-unique voxels needed to represent the two-dimensional mid-air boundary. In the next section, we will explain how we cull identical nodes *before* they are inserted into the DAG, thereby making the construction time proportional to the one-dimensional silhouettes, as well.

## 3.2 Culling Construction of Identical Nodes

We start by requiring the recursive top-down construction of the DAG to happen in Z-order, i.e., we complete processing of volumes closer to the light before we continue to those farther away. For each volume we process during construction, we first determine if it is homogeneously lit or shadowed by testing its bounds against the min-max depth hierarchy. When the volume is non-homogeneous, we would normally construct a new node (describing the volume) and insert it into the DAG. Before we construct a new node, we first test if the non-homogeneous volume is identical to the adjacent volume closer to the light (which is already represented in the DAG) (see Fig. 3). When they are identical, we just use the same node reference as the adjacent volume and terminate the recursion. This culls both construction and insertion of many nodes, just as homogeneous volumes do. When the volume is neither homogeneous nor identical to the adjacent volume, we need to recursively construct a new node and insert it into the DAG.

Two volumes, adjacent in Z-order, are identical when neither of them contain shadow-casting surfaces. To test if a volume is identical to the adjacent one, we compare the maximum depth of the volume against the depth of the next shadow-casting surface. During construction, we maintain a hierarchy of depths to the next shadow-casting surface, which we update for each completed node. Along with the depth, we keep the reference to the node we will re-use (the last completed node for each entry).

For each new node we construct, the depth of the next shadow-casting surface is calculated by the recursive construction. During the recursion, for homogeneous sub-volumes, we obtain this depth from the min-max hierarchy, and at the finest level, we obtain this depth from the depth map.

*Size of DAG after insertion.* As previously stated, our main incentive for culling identical nodes during construction is to increase performance of the algorithm. However, since the key is to perform much of the compression in the insertion step, we also greatly reduce the amount of working memory required during construction. We have measured the size of the DAG immediately after insertion as compared to after the compression steps (see Section 6), and we find that (in the tested scene), if we use the closed object optimization, the compression step (described in Section 4) reduces the DAG by approximately a factor of two. While this is certainly worth the effort if memory footprint is of highest importance, we *can* choose to output the DAG immediately after insertion, skipping the compression step, if build performance is more important. The data structure is still several orders of magnitude smaller than the corresponding shadow map.

## 3.3 Finding Undefined Regions

When the scene consists of closed objects, regions inside closed geometry will never be queried and may, therefore, hold an undefined value. This allows us to store a much less detailed data structure, as described above (see Fig. 2). We first establish, for each shadow-map texel, the *enter-depth* (the depth at which a light ray first enters a shadow caster), and the *exit-depth* (the depth at which the light ray first exits the closed geometry, which may be composed by several intersecting closed objects) and store the results in two maps. How to generate these maps will be discussed in this section. Nodes that intersect only one of these maps, or that lie entirely between the enter and exit depths, are called undefined and will be resolved as homogeneously lit or shadowed, as described in the next section.

The *enter-depth* map is rendered as an ordinary shadow map. Finding the *exit depth* is not as trivial, and requires us to consider all fragments that fall within a pixel. In the paper by Sintorn et al. [4], this is achieved by rendering each *layer* of the geometry in front to back order (using depth-peeling [16]), increasing a counter every time the closest fragment is front facing and decreasing it when it is back facing. Whenever the counter returns to zero, the fragment's depth is the exit depth.

Depth peeling has the drawback of requiring several render passes, which can be expensive for detailed geometry. The problem is similar to that of *Order Independent Transparency* (OIT), and several papers exist that attempt to improve performance by rendering the geometry in a single pass but building a list of fragments per pixel (an A-Buffer) instead (e.g., [17]). We have experimented with such solutions, but in our experiments, the improved depth-peeling algorithm explained below has had consistently better performance. One reason for this is that an A-buffer approach will have to store *all* fragments in a pixel, and then resolve the exit depth in a second pass, whereas a depth-peeling algorithm will process the layers in order and can stop rendering to a pixel as soon as the exit depth is found.

*Finding the exit depth.* To find the exit depth, we start with the enter depth as an input texture, and render front- and back-facing triangles to two separate buffers, storing the closest fragment that lies beyond the enter depth. We then compare, for each pixel, the depth of the front- and back-facing fragment, and if the back-facing fragment's depth is closer, we have clearly found the exit depth. Otherwise, we iteratively find the next front- and back-facing layers, using the results from the previous pass as input textures, until we have found the exit depth for all pixels. This algorithm has two advantages. First, the counter of the number of times we enter and exit a closed object becomes implicit. Second, for the same number of processed triangles and fragments, we advance two layers instead of one.

*Assuming no self-intersections.* We additionally note that if we can assume that the input geometry has no self-intersections (and no objects are inside other objects), as well as being closed, the exit depth can be found very easily by simply rendering the back-facing triangles with depth testing enabled. Modeling all objects as not self-intersecting can be very difficult, but most 3D modeling packages have tools for creating *boolean unions* of closed objects automatically.

The same approach to finding the exit depth *can* be used even without new restrictions to the input geometry. If two objects do intersect, the exit depth will not be the optimal depth, but it will always lie behind the enter depth and in front of the true exit depth, so there will be no artifacts in the visible shadows. Interestingly, we show (in Section 6) that for our tested scenes, the final size of the compressed data structure does not increase significantly when using this simplistic algorithm. This is because the vast majority of nodes of the DAG lies in the bottom layers, and even with this less exact method, sufficiently large undefined regions are found to remove most nodes in these layers.

## 3.4 Resolving Undefined Regions

We exploit undefined regions to reduce the overall memory consumption. As Sintorn et al. [4], we resolve undefined regions to form homogeneous regions, wherever possible, by making nodes containing lit and undefined regions fully lit, and nodes containing shadowed and undefined regions fully shadowed. When a node contains both lit and shadowed regions, it is not possible to form a homogeneous region, and Sintorn et al. [4] resolve undefined regions to shadowed, but admit that this heuristic might miss compression opportunities.

We propose a new way of resolving undefined regions, and we will show (in Section 6) that it has a significant positive impact on the final memory performance. This new method locally minimizes the number of unique visibility

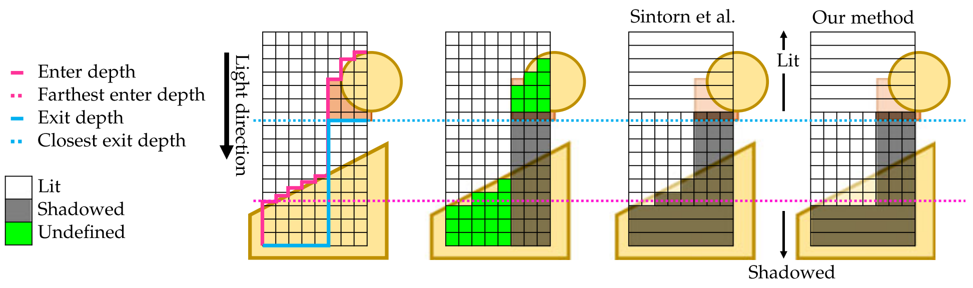


Fig. 4. At the finest level, we classify the cells as either lit, shadowed or undefined from the enter and exit-depth maps (two left-most figures). Our method of resolving undefined regions results in fewer unique visibility masks (two right-most figures).

masks between the near and the far plane of the light frustum. It does not resolve the undefined regions in a globally optimal way, but will locally minimize the number of unique nodes in any $8 \times 8$ texel tile, and is very fast to execute.

Prior to constructing the DAG, we consider each $8 \times 8$ texel tile of the depth map. In each tile, we have homogeneously lit slices from the near plane until the closest *exit depth*, since no cell has to be shadowed in this depth range (see Fig. 4). After the farthest *enter depth*, we will have homogeneously shadowed regions, since there will be no lit cells after this depth. For depths between the fully lit and fully shadowed regions, we need to store a minimal number of visibility masks that describe the visibility transitions.

At the closest *exit depth*, the slice cannot be set to fully lit and we need a visibility mask. We create a visibility mask that has shadow in as many cells as possible by setting all bits corresponding to cells that are beyond their corresponding *enter depth*. This visibility mask can then be re-used for all depths until the closest *exit depth* of the remaining cells. At the end of this depth range, we need another transition to a new visibility mask (with more shadowed cells). We repeat this process of forming visibility masks until we reach the fully shadowed region or the far plane.

We compute all visibility masks for each $8 \times 8$ texel tile upfront. For each visibility mask, we also keep the depth to which the mask can be re-used. For non-closed geometry, we use the *enter depth* also as the *exit depth*, but otherwise follow the same procedure. Since each block is computed separately, this can be performed in parallel on the GPU. Besides reducing the computation times, this method also reduces the amount of memory transferred to the host.

After this step, the DAG is constructed as described above, except that we now never have to query the finest level of the min-max depth hierarchy, and instead query the visibility masks of the tile.

## 4 Compression

After inserting all nodes (with culling of identical nodes) as described above, we have a partially reduced DAG that will contain no redundant sub-trees describing the mid-air boundaries between lit and shadowed space. The next step is to reduce this DAG to an *optimal* DAG, which contains no redundant sub-trees at all. In the next section, we will first review how this is achieved in previous work ([4], [11], [18]), and then, in Section 4.2, we will suggest an alternative algorithm that achieves the same result much more efficiently.

When building large CPVS data structures, construction must often be split into parts, as there is insufficient GPU memory to hold the original shadow map and perhaps even insufficient CPU memory to hold the intermediate partially reduced DAGs. In such cases, a number of optimal sub-DAGs will be created (one for each node at some level of the hierarchy), and in a final step, these sub-DAGs will be combined into one large optimal DAG. In Section 4.3, we will describe a simple modification to the original algorithm which improves the speed of this final step.

### 4.1 Recap of Bottom-Up Compression

In previous work, compressing an SVO (or a partially reduced DAG) to an optimal DAG has been done with a straightforward bottom-up algorithm, as illustrated in Fig. 5. Starting at the leaf level (where nodes contain only a bitmask representing $8 \times 8 \times 1$ voxels), all nodes in this level are sorted. In a sorted list, identical nodes are easily identified as they lie at consecutive indices, and so all but

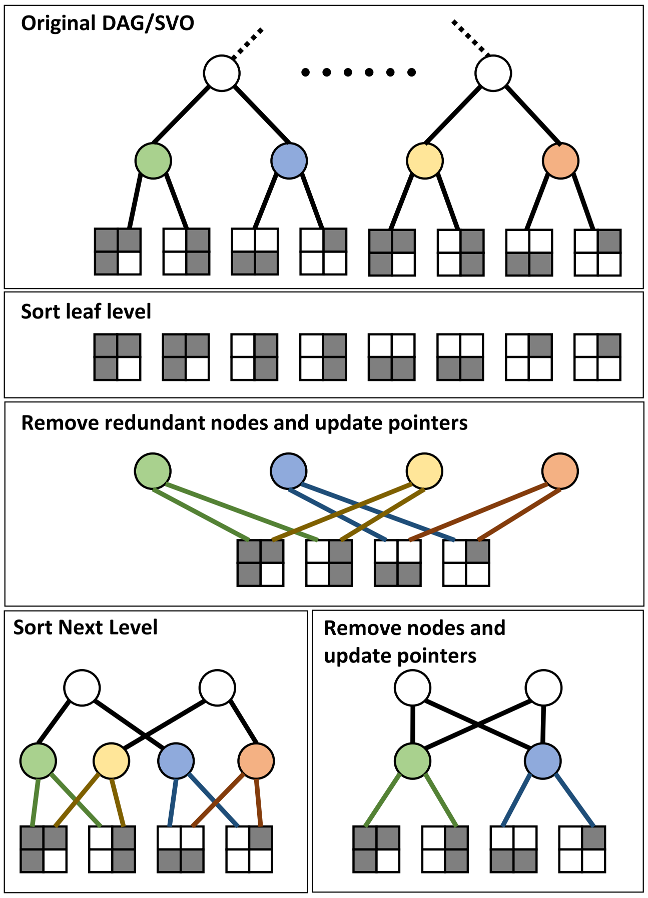

Original DAG/SVO
Sort leaf level
Remove redundant nodes and update pointers
Sort Next Level
Remove nodes and update pointers

Fig. 5. The bottom-up compression algorithm used in previous work.

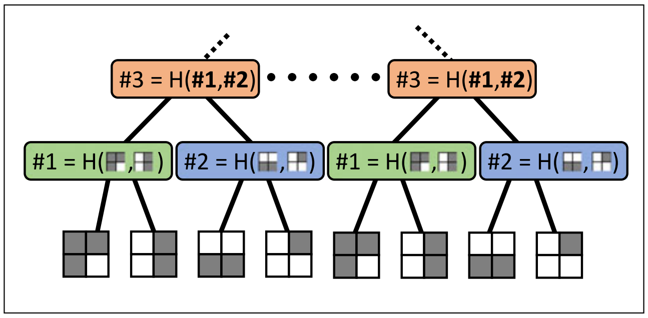


Fig. 6. By calculating hashes, bottom up, from the contents of the child nodes, nodes that are roots of identical sub-trees will have identical hashes.

one in a set of identical nodes can be removed, and the parent nodes pointing to the removed nodes are updated to point at the one remaining. The exact ordering used for sorting is not important, and the leaf nodes are simply sorted by treating the bitmask as an integer. Having processed the leaf level, the next level can be processed similarly. The nodes are sorted, this time using the child pointers as the sorting key, and then redundant nodes are removed and parent pointers are updated. The process is repeated for each level to achieve an optimal DAG.

One problem with this algorithm is that the comparison operator for internal nodes is potentially very expensive. With eight child pointers, we may have to perform eight integer comparisons to find out whether one node should precede another. The same problem exists when deciding whether two nodes are equal.

## 4.2 Top-Down Compression

Our initial approach to reducing the cost of comparing internal nodes was to generate a hash value based on the child pointers as soon as the level below had been processed. That way, identical nodes will have the same hash value, and as the actual ordering of nodes is not important, sorting can be done by simply comparing hash values. However, if we instead generate the hash values for the lowest internal node level, $L$, from the *contents* of the leaf nodes (i.e., the bitmask), we can immediately identify identical nodes in that level. If we then generate the hash values at level $(L-1)$ from the hash values at level $L$, and so on, we can immediately identify the roots of identical sub-trees *at any level* without first compressing the lower levels. This is illustrated in Fig. 6 for a binary tree.

Having calculated all hash values, which is quickly done in a bottom up sweep over all nodes, we can now find the optimal DAG using a top down algorithm. We first sort all nodes in the level below the root node using the hash values as key. Next, redundant nodes are removed and parent pointers are updated. Finally, the next level is generated by concatenating all child nodes of the *remaining* nodes, and the process is repeated for that level. Thus, the main advantage of the new method is that nodes in lower levels can be found to be part of redundant sub-trees early on and will not be processed at all. For instance, in Fig. 5, only half of the leaf nodes would have to be sorted and compacted.

When using a simple comparison of hash values to determine if subgraphs are identical, it is important that no hash collisions occur. A hash collision occurs if two root nodes of non-identical subgraphs coincidentally obtain identical hash values, and would lead to one of these subgraphs being incorrectly replaced by the other. The effect would be that, in the corresponding subvolume of the scene, the light-visibility information would be a copy of a completely unrelated subvolume, which is, of course, unacceptable. However, we can assert that the probability of hazardous hash collisions becomes negligible. Assuming 64-bit hash values and that the hash function is ideal in its distribution of hash values (which it of course is not in practice), the probability, $p_n$, of a hash collision in a level of $n$ unique nodes is roughly $\frac{n^2}{2^{64}}$. In our worst case (for $256K$-res.), the total probability, $P$, of a hash collision is $0.0002$. While we have observed no collisions when using 64-bit hash values in our experiments, the probability of a collision might be uncomfortably high for even higher resolutions and more complex scenes. In such cases, we recommend simply using a 96-bit hash value which would reduce the probability of collision to about $10^{-13}$ in the above example.

This new algorithm can be very beneficial to performance when we build large data structures and there is much redundancy (as shown in Section 6). However, our implementation is not quite as fast as the bottom-up approach when there is little redundancy in the input DAG, as it introduces the extra work of generating layer $L$ from the remaining nodes at level $L-1$. For the same reason, we have found that the new algorithm does not improve performance for the final compression pass, where several sub-DAGs are compressed into one large DAG, and therefore we use the bottom-up approach for this step. Note however, that the bottom-up approach *can* make use of the generated hash values to improve performance of sorting the nodes.

## 4.3 Merging Sub-DAGs

Having a sorting key (the hash value) that is identical for roots of identical sub-trees, allows us to additionally suggest a simple optimization to the final compression pass. In previous work, merging sub-DAGs is done by concatenating all levels of all sub-DAGs and then processing them bottom-up, in turn sorting each layer, removing redundant nodes and updating the parent's pointers. We note that each layer in the input sub-DAGs is already sorted, as a result of being compressed. Therefore, instead of re-sorting the entire concatenated layers of the final DAG, we can simply merge the layers of two sub-DAGs in a single sweep over the nodes.

To produce a final compressed DAG from $N$ input sub-DAGs, we process the layers one by one in a bottom up order as before. At each layer, $i$, we first merge each pair of sub-DAG layers, $((L_{i,0}, L_{i,1}), \ldots, (L_{i,N-1}, L_{i,N}))$ to produce $\frac{N}{2}$ intermediate sub-layers. We proceed recursively, merging layers, pairwise, until the final layer is produced. After this improved sorting step, we remove redundant nodes and update parent pointers as before.

Apart from consistently outperforming the previous algorithm, the new algorithm can be beneficial in that it divides the final compression into smaller chunks of work. This will allow for starting the final compression in a separate thread as soon as two sub-DAGs have been generated, which could hide much of the cost on multi-core hardware. Perhaps more importantly, it allows for the work to be divided over several frames in a realtime application, which would facilitate, for instance, the updating of shadows for a slowly moving lightsource (e.g., the sun) in the background.

## 5 Many Local Lights

We have explored the viability of using Compact Precomputed Voxelized Shadows in scenes containing many local, bounded lights, rather than a single distant light. Specifically, we have modified the ClosedCity scene to contain hundreds of spotlights with far attenuation and cut-off, to evaluate whether such a scene can be efficiently lit using CPVS, both in terms of memory and rendering performance. The main differences from previous use cases, apart from having to handle many lights efficiently, are that these lights will have a perspective projection with a large field-of-view and that the extreme resolutions used for distant lights are not required, or even desirable, in this setting.

While Sintorn et al. [4] show that compression rates increase significantly with increasing shadow resolution, they still achieve compression rates of about $10\times$, for non-closed geometry, and around $100\times$ for closed geometry, when the original shadow maps are as small as $4{,}096 \times 4{,}096$. With the added compression that our improved algorithm obtains, and since they can now be built in a reasonable time, it is possible to use precomputed, high-resolution, CPVSs for hundreds of lights while staying well within a reasonable memory budget. We will show (in Section 6) that these data structures can then be efficiently queried with large PCF filters to achieve high-quality shadows in real-time framerates.

*Many lights.* In any performance-critical application where many bounded lights are used, it is important to perform culling to avoid testing all lights against all pixels. We have chosen to implement the *Tiled Shading* approach [12], where a light is assigned to a list per screen-space tile, if the light's bounding volume intersects the tile's (three-dimensional) bounding volume. This approach can cull many more lights, but to further improve culling before the actual shadow-lookups are made, it would probably be beneficial to employ the *Clustered Shading* approach [14]. The choice of light-culling technique is orthogonal to our method.

When light culling has been performed, we simply start one thread per pixel (in CUDA) and loop through the list of assigned lights. Each entry in this list contains the light's model-view-projection matrix and a pointer to the appropriate CPVS. The filtered visibility value is then calculated and stored in a list for each pixel.

*Perspective lights.* When we have few discrete depth values in a CPVS (e.g., 4,096), we have to distribute them carefully. With large field-of-view point lights, a plain discretization of the lights NDC coordinates will result in poor depth precision close to the far plane. Our solution is similar to that of Olsson et al. [14], but while their goal is to achieve as cubical voxels as possible, our goal is to distribute $N$ depth values between the near and far plane to get a constant ratio between a voxel's height and depth. Therefore, we calculate a voxel's depth value, $\mathbf{z}$, from the lights view space as:

$$\mathbf{z} = \left\lfloor N \frac{\log \frac{\mathbf{z}_{\text{vs}}}{\text{near}}}{\log \frac{\text{far}}{\text{near}}} \right\rfloor. \quad (1)$$

Another issue with a high field-of-view is that the amount of biasing required is highly dependent on where within the frustum the view sample lies. As in the paper by Sintorn et al. [4], we bias the lookup point by moving one half filter width in the direction of the normal. In their implementation, however, this distance was roughly estimated while rendering the G-Buffer, using derivatives of the light's NDC coordinates. This approach is not directly available to us, as we must calculate a bias per light. Instead, the view sample's normal is sent along to the look-up kernel and transformed, for each light, by the light's model-view-projection matrix. The biasing is then performed in integer coordinates *after* the voxel coordinates have been calculated. This approach allow us to use a minimal bias at any position in the frustum.

## 6 Results

Unless stated otherwise, measurements were performed on a desktop computer with an Intel Core i7 3,930K CPU, 32 GB DDR3 1,600 MHz RAM, and an NVIDIA GTX 980 GPU connected via PCI Express 2.0 x16.

### 6.1 Construction

The construction is partially done on the GPU and partially on the CPU and consists of the following steps:

- *Render maps*: Render depth maps (OpenGL).
- *Precompute*: Compute the min-max hierarchy and visibility masks (CUDA).
- *Transfer*: Transfer the min-max hierarchy and visibility masks to host (PCIe).
- *Insert*: Insert nodes into sub-DAGs (CPU).
- *Compress subdags*: Compress sub-DAGs (CPU).
- *Compress final*: Compress final DAG (CPU).

We render the *enter-depth* map and the *exit-depth* map in OpenGL and use them to compute the visibility masks (with corresponding re-use depth) and the min-max hierarchy in CUDA. We only compute the min-max hierarchy down to an entry per $8 \times 8$ texel tile, as finer resolutions are not needed after the construction of visibility masks. The visibility masks and min-max hierarchy are then transferred to the host, and we perform insertion of nodes, compression of sub-DAGs and final compression on the CPU. Since rendering depth maps of the full resolution is infeasible, we perform construction for one 4K $\times$ 4K texel region at a time.

*Overall construction performance.* In Table 1, we show construction times for a single large directional light in three scenes. The scenes and lights are the same as those used in the measurements by Sintorn et al. [4]. The Necropolis scene consists of non-closed geometry, while the other two are entirely built from closed geometry (ClosedCity and FractalLandscape).

At moderate resolutions, construction times are fast enough to be performed during, for instance, level-load in a video-game. At higher resolutions, at least for closed geometry, construction times are still fast enough that, for

TABLE 1
Total CPVS Construction Times for Three Scenes with and without (★) Detection of Undefined Region Inside Closed Geometry

| **Resolution:** | **4K$^3$** | **16K$^3$** | **64K$^3$** | **256K$^3$** |
|---|---|---|---|---|
| Necropolis ★ | 336 ms | 4.75 s | 25 s | 312 s |
| ClosedCity ★ | 207 ms | 3.02 s | 40.7 s | 579 s |
| ClosedCity | 40 ms | 328 ms | 2.6 s | 31.8 s |
| FractalL. ★ | 200 ms | 2.9 s | 42 s | 589 s |
| FractalL. | 26 ms | 219 ms | 2.5 s | 36 s |
| Sintorn et al. [4] | 2.0 s | 18 s | 256 s | 5,520 s |

*Top-down compression and merge-sort final compression are used for non-closed objects. The last row contains the construction times presented by Sintorn et al. [4] for FractalLandscape.*

TABLE 2
Construction Times for ClosedCity (Our Implementation) with and without Culling Construction of Identical Nodes

| **Resolution:** | | **4K$^3$** | **16K$^3$** | **64K$^3$** | **256K$^3$** |
|---|---|---|---|---|---|
| | | No Culling | | | |
| not | total | 304 ms | 4.1 s | 61 s | 918 s |
| closed | DAG | 293 ms | 4 s | 58.7 s | 876 s |
| closed | total | 173 ms | 2.1 s | 32.1 s | 503 s |
| | DAG | 140 ms | 1.9 s | 29 s | 470 s |
| | | Culling | | | |
| not | total | 207 ms | 3.02 s | 40.7 s | 579 s |
| closed | DAG | 197 ms | 2.8 s | 38.6 s | 543 s |
| closed | total | 40 ms | 328 ms | 2.6 s | 31.8 s |
| | DAG | 17 ms | 128 ms | 665 ms | 3.0 s |

*Timing is reported both for the total construction and the time for creating the DAG (inserting and finding identical nodes). For non-closed objects we utilize the top-down compression and the merge-sort final compression.*

instance, a lighting artist could be expected to await their completion after pressing a button.

Our construction times are more than two orders of magnitude faster than those reported by Sintorn et al. [4] (see Table 1), but they also state that the construction speed was not their primary concern.

*Culling insertion of identical nodes.* In Table 2, we report the timings when building the ClosedCity scene, both as closed and non-closed geometry and with and without taking advantage of our improved culling algorithm. In these experiments, for closed geometry, we use depth peeling to find undefined regions (see Section 3.3), and we only employ our top-down compression (Section 4.2) and merge-sort final compression (Section 4.3) for non-closed geometry.

For non-closed geometry, we build the data structure from a simple shadow map, and the construction time is dominated by inserting nodes and compressing the DAG on the CPU. When we employ culling of already constructed nodes, processing time is almost halved as we insert much fewer nodes into the the intermediate DAG, which also improves the speed of compression. For closed geometry, the intermediate data structure does not have to finely represent the shadow-casting surfaces but, unless we employ culling, it still contains very many redundant nodes that describe the transition from shadowed to visible regions in open space. Therefore, construction time is much faster than for non-closed geometry but is still dominated by insertion and compression. When using closed geometry *and* culling, the total work required for constructing the DAG is small compared to the time taken to generate the enter- and exit-depth maps.

*Improved compression.* Table 3 shows a break-down of the timings for individual parts of our algorithm when constructing shadow information for the ClosedCity scene. For non-closed geometry, we also present the timings of our top-down algorithm for compressing sub-DAGs and our merge-sort variant of final compression.

The top-down algorithm requires a large amount of redundancy to be present in the sub-DAG to outperform the bottom-up approach, and so for closed geometry (where only the silhouettes are finely described by unique nodes), we actually observe very little or no gain in performance with this algorithm. For non-closed geometry, however, we see a speed up ranging from $\times 1.3$ to $\times 2.5$. Since sub-DAG compression is also the most expensive part of our algorithm in this case, the total compression time is almost halved when rendering high-resolution shadow information.

At moderate resolutions, the merge-sort final compression algorithm also performs much better than the previous, but in these experiments, the time taken for final compression is insignificant. Using the final merge-sort algorithm for closed objects is also faster in itself but would require us to generate hash values resulting in a net loss of performance.

## 6.2 Memory Consumption

We have measured the final memory consumption for the two scenes of closed geometry, with and without the new method of resolving undefined regions. The new method compresses the final memory consumption of the CPVSs by an additional 1.4–3.0× (see Table 4).

In Table 5, we show the resulting final DAG sizes if processing is stopped after the insertion step (as discussed in Section 3.2), along with the shorter total construction times.

TABLE 3
Breakdown of Construction Timings for ClosedCity

| **Resolution:** | **4K$^3$** | **16K$^3$** | **64K$^3$** | **256K$^3$** |
|---|---|---|---|---|
| | | **non-closed** | | |
| render maps | 2.1 ms | 18.5 ms | 273 ms | 4.2 s |
| precompute | 1.8 ms | 29.2 ms | 474 ms | 7.41 s |
| transfer | 5.6 ms | 90.5 ms | 1.4 s | 23.4 s |
| insert | 92.9 ms | 1.36 s | 20.6 s | 325 s |
| compress subdags | 142 ms | 2.07 s | 29.7 s | 445 s |
| *(w/ top-down opt.)* | *105 ms* | *1.30 s* | *15.1 s* | *182 s* |
| compress final | – | 372 ms | 3.88 s | 36.5 s |
| *(w/ merge-sort opt.)* | – | *214 ms* | *2.88 s* | *35.9 s* |
| **Total** | **245 ms** | **3.95 s** | **56.4 s** | **842 s** |
| **(w/ opt.)** | **207 ms** | **3.02 s** | **40.8 s** | **579 s** |
| | | **closed** | | |
| render maps | 22.3 ms | 170 ms | 1.52 s | 20.4 s |
| precompute | 0.91 ms | 14.9 ms | 235 ms | 3.73 s |
| transfer | 1.2 ms | 18.3 ms | 280 ms | 4.52 s |
| insert | 8.79 ms | 48.8 ms | 264 ms | 1.24 s |
| compress subdags | 7.34 ms | 44.1 ms | 217 ms | 934 ms |
| compress final | – | 32.2 ms | 173 ms | 852 ms |
| **Total** | **40.5 ms** | **328 ms** | **2.69 s** | **31.7 s** |

TABLE 4
Resulting Memory Consumption with the Closed Object Optimization, Comparing the New Method with Sintorn et al. [2014]

| Res. | ClosedCity [MB] new | ClosedCity [MB] old | ClosedCity ratio | FractalL. [MB] new | FractalL. [MB] old | FractalL. ratio |
|---|---|---|---|---|---|---|
| $4K^3$ | 0.83 | 1.18 | 1.43 | 0.44 | 0.76 | 1.72 |
| $8K^3$ | 1.89 | 2.82 | 1.49 | 0.84 | 1.59 | 1.90 |
| $16K^3$ | 3.96 | 6.25 | 1.58 | 1.60 | 3.34 | 2.08 |
| $32K^3$ | 7.70 | 12.87 | 1.67 | 3.05 | 6.98 | 2.29 |
| $64K^3$ | 14.28 | 25.43 | 1.78 | 5.76 | 14.36 | 2.49 |
| $128K^3$ | 26.15 | 49.72 | 1.90 | 10.85 | 29.30 | 2.70 |
| $256K^3$ | 48.04 | 100.05 | 2.08 | 20.35 | 60.99 | 3.00 |

We show this both for when the DAG is built with optimal undefined regions and when no self-intersections is falsely assumed (as described in Section 3.3). In our example scene, using less than optimal undefined regions will not affect the final DAG size significantly and the total processing time is roughly halved. At a resolution of $16K^3$, additionally ignoring the compression steps reduces the processing time to 61 percent, at the cost of a 62 percent larger final DAG. Note that this is still only 1.1 percent of what the corresponding shadow map would cost.

### 6.3 Rendering with Many Lights

In order to test the viability of using CPVSs in a scene with many bounded lights, we have modified the ClosedCity scene (used for measurements by Sintorn et al. [4]) to contain 165 spotlights, each of which lights a small portion of the scene. For each spotlight, we built a CPVS at resolution $8{,}192^3$, which results in sufficiently sharp shadows for all lights, with a $9 \times 9$ PCF filter. All measurements in this section were performed on a desktop computer with an Intel Core i5 2,500K CPU, 16 GB DDR3 1,333 MHz RAM, and an NVIDIA GTX Titan GPU connected via PCI Express 2.0 x16. For these experiments, the exact depth-peeling algorithm for finding undefined regions was used. Neither the top-down compression, nor the merge-sort final compression algorithms were used.

Fig. 7 shows how the construction times and final data-structure sizes are distributed over the different lights. The

TABLE 5
Final DAG Size and Build Time for ClosedCity When Stopping After Insertion, Partial Compression or Final Compression Steps, and with or without Full Depth-Peeling for Undefined Region Identification

| | $16K^3$(size/time) | | $64K^3$(size/time) | |
|---|---|---|---|---|
| **With depth peeling** | | | | |
| insertion | 6.3 MB | 252 ms | 30.4 MB | 2.30 s |
| partial compress | 4.4 MB | 296 ms | 19.5 MB | 2.51 s |
| final compression | 3.9 MB | 328 ms | 14.2 MB | 2.69 s |
| **Without depth peeling** | | | | |
| insertion | 6.5 MB | 125 ms | 31.4 MB | 1.2 s |
| partial compress | 4.5 MB | 169 ms | 20.1 MB | 1.47 s |
| final compression | 4.0 MB | 202 ms | 14.7 MB | 1.64 s |

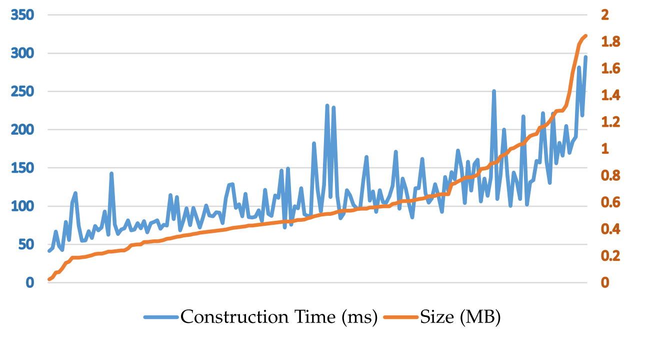


Fig. 7. Distribution of final data structure sizes and construction times for the 165 CPVSs in ClosedCity.

sizes of the data structures vary between 0.3 and 1.7 MB, for a total of 96 MB. The build times are mostly dependent on the depth complexity and the amount of geometry that intersects the lights' frustums. The total build time is 19 seconds.

Fig. 8 shows timings of different parts of the algorithm, along with a curve showing the average number of lights per pixel for each frame. The sequence was rendered at a resolution of $1{,}920 \times 1{,}080$. The first two steps are generating the bounding boxes for each $8 \times 8$ screen space tile and then intersecting the lights' bounding volumes with these to produce a light list per tile. This is done in two CUDA passes and takes fairly constant time. The next step, calculating shadows, is highly dependent on how many lights are overlapping the tiles in the current frame. The final step, shading, is a full-screen fragment-shader pass, where each pixel loops through the light list of the tile it resides in and accumulates the contribution of each affecting light.

Each shadow lookup returns a filtered visibility using the equivalent of a $9 \times 9$ PCF filter. We replace the top six levels of each CPVS with a small grid of pointers, which costs an aditional 128 kB per light, for a small performance improvement. At worst, the time for calculating shadows for all pixels is 9 ms.

## 7 Conclusion and Future Work

We have presented an algorithm for generating Compact Precomputed Voxelized Shadows, which improves the construction speed of up to two orders of magnitude and increases the compression by up to 3×. This makes construction much more feasible to perform in runtime, e.g., during level load or amortized over several frames. We

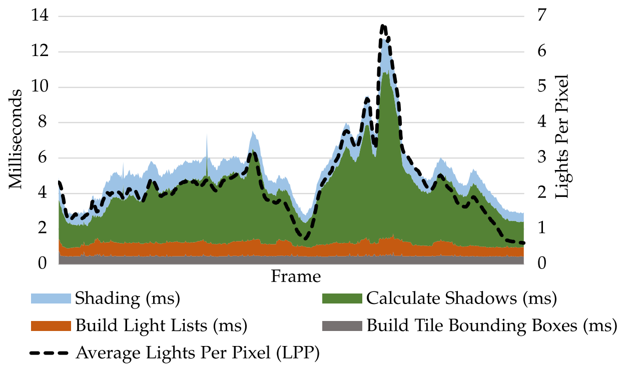


Fig. 8. The measured performance in a flythrough animation of the ClosedCity scene. The dashed line is the average number of lights that are assigned to each tile in the frame.

show that CPVSs for hundreds of spotlights, at a resolution of $8K^3$, can be constructed at around 100 ms and 0.5 MB per light. The memory consumption is about 100 times lower than a corresponding shadow map. We also suggest a novel transform from the light's NDC into voxel space, to maintain high depth precision when the light's frustum is not near-orthographic.

We have presented a top-down compression algorithm which was shown to be much more efficient in cases where there is much redundancy left in the DAG after the insertion step. This algorithm should also be very beneficial when voxelizing surfaces [18].

We have shown that, for closed objects, at the cost of a small increase in memory footprint, the performance cost of identifying undefined regions can be significantly reduced. Thus, a natural next step towards reducing construction times further would be to move the insert and compression passes to the GPU, which would also completely remove the expensive PCI bus transfers.

## ACKNOWLEDGMENTS

The NECROPOLIS scene is distributed with the Unreal Development Kit by Epic Games. The FRACTALLANDSCAPE scene was generated using the software World Machine. The NVIDIA GTX Titan GPU used for this research was donated by NVIDIA. This work was supported by the Swedish Foundation for Strategic Research under Grant RIT10-0033 and the Swedish Research Council under Grant 2014-4559.

## REFERENCES

[1] W. Engel. (2006). Cascaded shadow maps [Online]. Available: http://books.google.se/books?id=isu_QgAACAAJ

[2] F. Zhang, H. Sun, L. Xu, and L. K. Lun, "Parallel-split shadow maps for large-scale virtual environments," in *Proc. Int. Conf. Virtual Reality Continuum Appl.*, 2006, pp. 311–318. [Online]. Available: http://doi.acm.org/10.1145/1128923.1128975

[3] L. Williams, "Casting curved shadows on curved surfaces," *SIGGRAPH Comput. Graph.*, vol. 12, pp. 270–274, Aug. 1978. [Online]. Available: http://doi.acm.org/10.1145/965139.807402

[4] E. Sintorn, V. Kämpe, O. Olsson, and U. Assarsson, "Compact precomputed voxelized shadows," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 150:1–150:8, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2601097.2601221

[5] E. Eisemann, M. Schwarz, U. Assarsson, and M. Wimmer. (2011). *Real-Time Shadows*. A. K. Peters. [Online]. Available: http://www.cg.tuwien.ac.at/research/publications/2011/EISEMANN-2011-RTS/

[6] R. Ramamoorthi, "Precomputation-based rendering," *Found. Trends. Comput. Graph. Vis.*, vol. 3, no. 4, pp. 281–369, Apr. 2009. [Online]. Available: http://dx.doi.org/10.1561/0600000021

[7] J. Rasmusson, J. Ström, P. Wennersten, M. Doggett, and T. Akenine-Möller, "Texture compression of light maps using smooth profile functions," in *Proc. Int. Conf. High Perform. Graph.*, 2010, pp. 143–152. [Online]. Available: http://dl.acm.org/citation.cfm?id=1921479.1921501

[8] S. Lefebvre and H. Hoppe, "Compressed random-access trees for spatially coherent data," in *Proc. 18th Eurograph. Conf. Rendering Tech.*, 2007, pp. 339–349. [Online]. Available: http://dx.doi.org/10.2312/EGWR/EGSR07/339-349

[9] N. Schulz. (2014). The rendering technology of ryse [Online]. Available: http://www.crytek.com/download/2014_03_25_CRYENGINE_GDC_Schultz.pdf

[10] J. Arvo and M. Hirvikorpi, "Compressed shadow maps," *Vis. Comput.*, vol. 21, no. 3, pp. 125–138, Apr. 2005. [Online]. Available: http://dx.doi.org/10.1007/s00371-004-0276-9

[11] V. Kämpe, E. Sintorn, and U. Assarsson, "Fast, memory-efficient construction of voxelized shadows," in *Proc. 19th ACM SIGGRAPH Symp. Interactive 3D Graph. Games*. 2015, pp. 25–30. [Online]. Available: http://dl.acm.org/citation.cfm?id=2699284

[12] O. Olsson and U. Assarsson, "Tiled shading," *J. Graph., GPU, Game Tools*, vol. 15, no. 4, pp. 235–251, 2011. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/2151237X.2011.621761

[13] T. Harada, J. McKee, and J. C. Yang, "Forward+: A step toward film-style shading in real time," in *GPU Pro 4*, W. Engel, Ed. Boca Raton, FL, USA: CRC Press, 2013, pp. 115–135.

[14] O. Olsson, M. Billeter, and U. Assarsson, "Clustered deferred and forward shading," in *Proc. 4th ACM SIGGRAPH/Eurograph. Conf. High-Performance Graph.*, 2012, pp. 87–96.

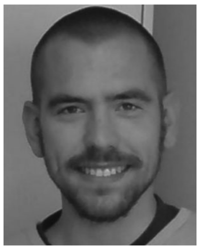
[15] O. Olsson, E. Sintorn, V. Kämpe, M. Billeter, and U. Assarsson, "Efficient virtual shadow maps for many lights," in *Proc. 18th Meeting ACM SIGGRAPH Symp. Interactive 3D Graph. Games*. 2014, pp. 87–96.

[16] C. Everitt, "Interactive order-independent transparency," Tech. Rep., NVIDIA Corporation, 2001.

[17] A. L. Sylvain Lefebvre, and Samuel Hornus, "Per-pixel lists for single pass a-buffer," in *GPUPro 5*, W. Engel, Ed. Boca Raton, FL, USA: CRC Press, 2014.

[18] V. Kämpe, E. Sintorn, and U. Assarsson, "High resolution sparse voxel dags," *ACM Trans. Graph.*, vol. 32, no. 4, 2013, Jul. 2013, Art. no. 101.

**Viktor Kämpe** is currently working toward the PhD degree in computer graphics at the Department of Computer Science and Engineering, Chalmers University of Technology. His research interests include geometric primitives and real-time shadows.



**Erik Sintorn** received the PhD degree from Chalmers University of Technology, in 2013, where he now works as a postdoc in the Computer Graphics Research Group, Department of Computer Science and Engineering. His research interest include real-time shadows, transparency and global illumination.

**Dan Dolonius** received the MSc degree in applied mathematics. He is currently working toward the PhD degree in computer graphics at the Chalmers University of Technology. He has worked at Autodesk with their render engines and applications. His research interests include real-time rendering, compression, and GPU algorithms.

**Ulf Assarsson** is a professor in computer graphics, at the Department of Computer Science and Engineering, Chalmers University of Technology. His main research interests include real-time rendering, global illumination, many lights, GPU-Ray Tracing, and hard and soft shadows. He is co-author of the book *Real-Time Shadows*.