

# Compact Precomputed Voxelized Shadows

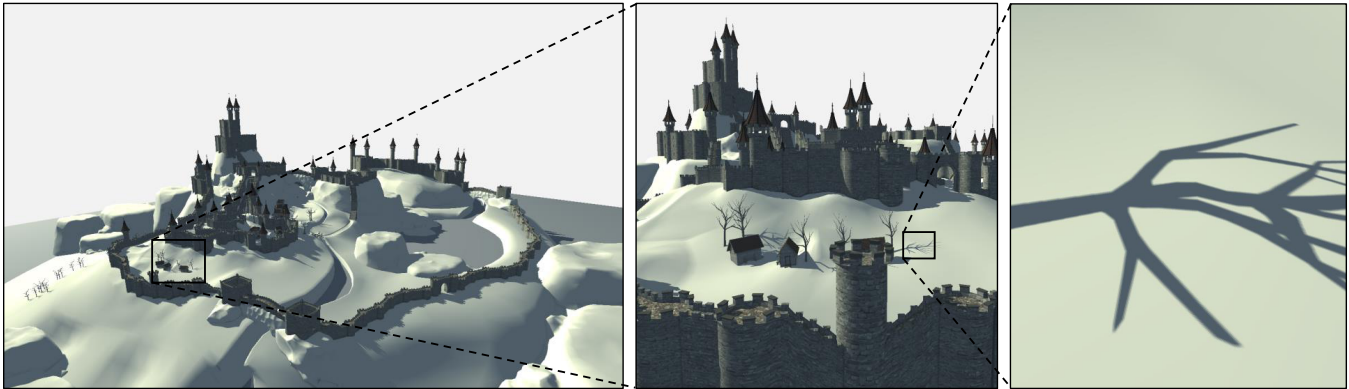
Erik Sintorn\*

Viktor Kämpe\*

Ola Olsson\*

Ulf Assarsson\*

Chalmers University of Technology



**Figure 1:** An example of using our algorithm to evaluate precomputed shadows from the sun when viewing the scene at varying scales. Our compact data structure occupies 100MB of graphics memory and is equivalent to a  $256k \times 256k$  (i.e.  $262144^2$ ) shadow map. With a filter size of  $9 \times 9$  taps, shadow evaluation is done in  $< 1ms$  at 1080p resolution.

## Abstract

Producing high-quality shadows in large environments is an important and challenging problem for real-time applications such as games. We propose a novel data structure for precomputed shadows, which enables high-quality filtered shadows to be reconstructed for any point in the scene. We convert a high-resolution shadow map to a sparse voxel octree, where each node encodes light visibility for the corresponding voxel, and compress this tree by merging common subtrees. The resulting data structure can be many orders of magnitude smaller than the corresponding shadow map. We also show that it can be efficiently evaluated in real time with large filter kernels.

**CR Categories:** I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing and texture;

**Keywords:** shadows, real-time, precomputed

## 1 Introduction

With state of the art real-time shadowing algorithms, it is still difficult to generate high-quality shadows in large open scenes. When a user needs to simultaneously see detailed shadows of nearby objects and alias-free shadows in the far distance, the traditional shadow mapping algorithm [Williams 1978] algorithm breaks down. A common remedy is to use Cascaded Shadow Maps (CSMs), where the view frustum is partitioned and one shadow map is rendered for each partition [Engel 2006; Zhang et al. 2006; Lloyd et al. 2006]. This provides a roughly uniform shadow-map resolution for all view samples. However, even with aggressive culling techniques, this can require re-rendering parts of the scene several times, which affects performance. Additionally, the resolution in distant regions can, even if it matches the screen sample frequency well, be insufficient to capture complex shadow casters without aliasing.

Most virtual scenes in real-time applications contain a large portion of static geometry, and it is not uncommon to have one or more static key lights (e.g. the sun). Under these circumstances, it can be desirable to use precomputed shadows, which can be both faster to evaluate and can provide higher quality shadows than what is possible with fully dynamic techniques. A dynamic technique, e.g. CSMs, can then be used for the dynamic geometry, which represents just a fraction of the triangles and fill-rate requirements. This generally results in both higher, and more stable, performance and improved quality.

Consequently, this is very common in practice and is supported in some form by all major game engines. One common approach is to store pre-calculated visibility information in texture maps (often called *light maps*), which can be immediately queried during shading. Light maps provide light-source visibility information during shading at almost no cost but are quite limited in some respects. First, they can only be used to evaluate shadows on the surface of static geometry, which is problematic as dynamic and volumetric receivers must use some other technique to be shadowed by the static environment. Secondly, for the static geometry, a unique UV parametrisation must be created, which can be difficult and cumbersome. Thirdly, even with lossy image-compression techniques, these maps can require a considerable amount of memory if high resolutions are required.

The goal of this paper is to design a data structure that provides precomputed shadow information from static geometry and a static light, and which enables high-quality filtered shadows to be reconstructed for any point in the scene. Thus, both static and dynamic geometry can receive shadows from the static environment, and a separate real-time technique only needs to support shadows from dynamic geometry. We achieve this by voxelizing shadow information for the entire space to an octree and then compressing this tree by merging common subtrees. Our suggested data structure is extremely compact but can still be used to obtain high-quality results very quickly while shading. Our data structure provides equivalent information to that of an extremely high-resolution shadow map at a fraction of the memory cost. We go on to show that for closed geometry, or when the scene is not to be used with dynamic objects (e.g.

\*{erik.sintorn|kampe|olaolss|uffe}@chalmers.se

for an architectural walk-through), we can reduce the size further, to the point of requiring three orders of magnitude less memory than a 16-bit shadow map for the scene and resolution shown in Figure 1.

By storing light visibility, as opposed to depth values, high-quality filters using large filter kernels can be evaluated very efficiently, enabling shadows for a full HD resolution with a speed and quality that is much higher than what is obtainable with CSM. In addition, we demonstrate two practical methods for filtering shadows in the distance, which both removes aliasing and improves performance.

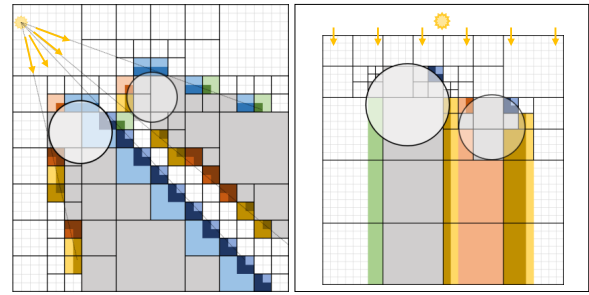
## 2 Previous Work

**Real-time shadows.** There exists a very large body of work concerning the generation of shadows in real time. For a detailed survey, we refer the reader to either of two excellent recent books: Eisemann et al. [2011] and Woo and Poulin [2012]. Most current real-time applications that require shadows cast from point lights onto a large virtual world use a variant of the *Cascaded Shadow Maps* (CSM) technique [Engel 2006; Lloyd et al. 2006; Zhang et al. 2006]. The idea is to partition the view frustum and render a *shadow map* [Williams 1978] per partition. This reduces the under- and over-sampling problems that would be caused by using a single shadow map. However, to completely hide artifacts caused by aliasing and visible transitions between partitions, a large number of high-resolution shadow maps are still required. A more general approach to achieving sufficient shadow-map resolution is to estimate the required resolution for tiles of the shadow map in a first camera pass and then render the shadow map at varying resolutions for different tiles [Lefohn et al. 2007; Giegl and Wimmer 2007]. These techniques are still too expensive for most real-time applications, and none of the techniques mentioned so far alleviate the aliasing problems that occur when distant shadow casters are too complex to be accurately represented by a low-resolution shadow map.

Whichever discrete shadow-map method is used, the signal must be reconstructed at the sample point using some form of filtering. Since the depth values stored in a shadow map cannot be directly pre-filtered, this can become very expensive. In order to afford large filter kernels, several alternative methods have been suggested, such as *Variance Shadow Maps* (VSMs) [Donnelly and Lauritzen 2006], *convolution shadow maps* [Annen et al. 2007], and *Exponential Shadow Maps* (ESMs) [Annen et al. 2008]. These methods store an approximation of the visibility function rather than a single depth, and can be pre-filtered. Due to their approximate nature, these algorithms all have problematic failure cases.

There are also several real-time alias-free algorithms based either on *shadow volumes* [Crow 1977; Heidmann 1991; Sintorn et al. 2011], or on *irregular rasterization* [Johnson et al. 2005; Aila and Laine 2004; Sintorn et al. 2008], but these methods are currently not fast enough to be used for complex scenes in real time.

**Precomputed shadows.** To improve rendering quality and performance, many real-time applications employ some form of *pre-computed radiance transfer* technique. We refer the reader to the survey by Ramamoorthi [2009] for an overview of such methods. Precomputed point light visibility can be stored in a *light map*, for fast, pre-filtered lookups. Rasmusson et al. [2010] present a hardware-accelerated compression scheme that specifically targets light maps and provide an overview of standard techniques. Lefebvre and Hoppe [2007] suggest using a hierarchical representation of spatially coherent data (such as light maps) to improve compression rates at the cost of more expensive lookups. However, even when employing lossy compression schemes, capturing high-frequency changes in visibility can still require too much memory for light maps to be feasible for large scenes. Binary visibility over a surface



**Figure 2:** Overview of our compression method. Left: Solid voxelization of shadowed space, identical (colored) nodes are merged. Right: Voxelization in light-space provides much better compression.

can also be encoded using *distance fields*. Such data can often be very well reconstructed, as suggested by Green [2007], from low-resolution maps, but the method fails to capture thin or complex shadows. In addition, these techniques all require an unambiguous UV parametrization for all geometry, and can only store visibility information exactly at the considered (static) surfaces.

**Shadow map compression.** A number of algorithms exist that attempt to compress and decompress depth buffers in real time, as discussed by Hasselgren and Akenine-Möller [2006]. These methods reduce bandwidth requirements but do not achieve sufficient compression ratios to be useful for storing extremely high-resolution shadow maps. The method by Arvo and Hirvikorpi [2005], however, reaches high compression ratios (e.g. 50:1 compared to a 16-bit shadow map for a simple scene). In their method, each scan line is compressed to a set of line segments that lie between the first two closest surfaces as seen from the light. These line segments can be stored in a hierarchical fashion, or the shader can perform a binary search during lookup. However, their method does not directly allow for dynamic shadow receivers, and should perform poorly for thin, non-flat shadow casters. Their suggested method helps to increase performance and quality of PCF lookups, as they can use analytic filtering in the horizontal direction, but for an  $N \times N$  filter, they will need to perform  $N$  hierarchical searches in the map.

## 3 Overview

Our goal is to design a data structure that represents the same information as an extremely high-resolution shadow map, but within a reasonable memory footprint. If the data structure is to be attractive for real-time applications, it is important that evaluating visibility at any point is not much more expensive than when using a shadow map. Inspired by the high compression rates obtained for voxelized geometry in the work by Kämpe et al. [2013], we take a novel and perhaps somewhat unintuitive approach to compressing the original shadow map.

The basic idea behind our algorithm is to create a solid voxelization of shadowed space, stored as an octree, and then compress this by merging common subtrees to create a Directed Acyclic Graph (DAG). Looking at Figure 2, it is clear that the octree will need to store leaf nodes only along the surfaces that enclose the shadowed space (we will call this the *shadow boundary*), and so its size will be largely dependent on the area of that surface.

To improve the effectiveness of common-subtree merging, we perform the voxelization in projected light space (see Figure 2). With a voxelization in this space, a node at level  $L$  with coordinates  $(x, y, z)$  will be identical to the node at coordinates  $(x, y, z + 1)$  unless a

shadow-casting surface intersects that node. This property has no impact on the size of the uncompressed octree, but it means that we will suddenly have an abundance of identical nodes at all levels and the common-subtree merging will be a lot more effective. In fact, the size of the compressed octree will now, at worst, be proportional to the area of the (closest) shadow-casting surfaces (see Figure 2). Additionally, common subtrees of these surfaces will be merged, which leads to an extremely high degree of compression compared to storing the octree itself and, as will be shown in Section 7, compared to the original shadow map.

## 4 Data Structure and Evaluation

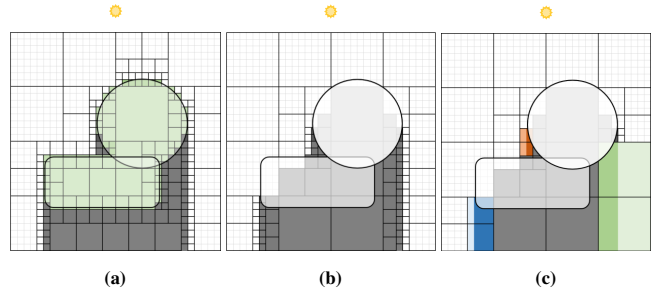
As in the paper by Kämpe et al. [2013], the octree is represented via a DAG and each node will consist of a childmask and a set of pointers. Identical subtrees are easily shared between parents by simply pointing to the same node. Kämpe et al. [2013] voxelize surfaces, and a node will either intersect the surface or not. The childmask therefore needs a single bit per subnode. Our data structure, however, represents a voxelized volume, and thus our childmasks must be able to identify a node as 1) intersecting the shadow boundary, 2) lying entirely outside shadowed space or 3) lying entirely inside shadowed space. Therefore, our childmasks are stored as 16-bit words instead of 8-bit (i.e. 2 bits per child).

The final three levels of our DAG (representing subvolumes of  $4 \times 4 \times 4$  voxels) will not be able to benefit from common-subtree merging. The smallest possible such subtree would consist of a parent node with a 16-bit childmask and a 32-bit pointer, and a child with an 8-bit childmask. Therefore, it is efficient to merge the three lowest levels and let our leaves be stored as  $4 \times 4 \times 4$ -bit grids (64-bit leafmasks).

To evaluate whether a point,  $\mathbf{p}$ , is in shadow or not, we fetch the childmask of the root node and check the status in the childmask for the subnode that contains  $\mathbf{p}$ . If this node is marked as entirely inside or outside the shadow region, we are done. If it intersects the shadow boundary, the appropriate pointer is fetched and the process is repeated for the subnode. If traversal reaches a leaf node, a direct lookup is performed in the leafmask.

Thus, for a DAG with resolution  $2^n \times 2^n \times 2^n$ , we will need to perform  $2(n-2) + 1$  fetches for any sample that needs to traverse to a leaf node. However, many samples can terminate traversal early, and for those remaining the cache-hit ratio is very high. Still, a single lookup in our data structure is certainly more expensive than a single shadow-map lookup. We will show, however, that filtered lookups can be extremely efficient, even to the point of outperforming traditional shadow maps.

**Closed object optimization.** If we know that some or all of the shadow-casting objects are closed, and that the user will never view the scene from inside of these objects we can optimize our data structure to increase compression rates by yet an order of magnitude. The region of space that lies inside of one or more such closed objects is considered *undefined* (see Figure 3a), and nodes that lie entirely in undefined space can be set to either visible or not visible, depending on which choice gives a smaller DAG (see Figure 3b). The effect of this optimization is quite dramatic as it means that most shadow-casting surfaces, that previously had to be defined at the finest level in our data structure, will now only need fine-level detail around their silhouettes (see Figure 3c). In the paper by Arvo and Hirvikorpi [2005], the region between the two closest shadow-casting surfaces is considered undefined and our approach is similar, but allows for even better compression and for shadows cast on dynamic objects.



**Figure 3:** For closed objects, a) voxels inside objects are considered undefined (green) b) tree is simplified and c) common subtrees are merged (identical nodes are shown in color).

## 5 Filtering

A shadow map is a discrete image of depths, and so aliasing and other errors are easily introduced in several different ways. In this section, we will discuss how our algorithm handles such errors, using the terminology suggested by Eisemann et al. [2011].

**Initial sampling errors** are introduced while creating the shadow map. *Undersampling* occurs when the shadow map is being rendered at a frequency lower than the final screen-sampling frequency, and results in a visible magnification of the discrete shadow map. This is the error that can be alleviated using techniques such as CSM [Engel 2006]. *Initial sample aliasing* is introduced when the shadow map is rendered at an insufficient resolution, with respect to the geometry. By super sampling the shadow map, and pre-filtering the result, techniques such as ESM [Annen et al. 2008] can reduce these problems. With our solution, initial sampling errors can be avoided simply because our compressed data structure can accommodate very high resolutions. If the resolution is sufficient for shadows at close range, it will also be sufficient for distant objects.

**Resampling errors** occur when the shadow-map signal is incorrectly reconstructed during lookups. If a view sample is considered a small surface patch, the visibility lookup should return a filtered visibility, based on that patch's projection on the shadow map. Since shadow maps cannot be pre-filtered, the common solution is to blur shadow edges using Percentage Closer Filtering (PCF) [Bunel and Pellacini 2004], which can become very expensive with many samples. Alternative methods that allow for pre-filtering (e.g. VSMs [Donnelly and Lauritzen 2006]) can be much faster for large filter kernels.

With our data structure, PCF is a very attractive option since we store the binary visibility of all voxels with at most a single bit per voxel. Therefore, when we have fetched the leafmask containing the visibility of our voxel, we already have a large chunk of nearby visibility samples at hand. Since the information stored is the actual visibility (not a depth), determining the percentage of visible samples becomes a simple bit-masking operation (see Section 6.2).

To further exploit this, we note that, if all PCF samples are taken at the same depth, we can ensure that we fetch even more valid visibility samples in a single leafmask, by rearranging the lowest levels of our hierarchy so that the leafmasks encode  $8 \times 8 \times 1$  instead of  $4 \times 4 \times 4$  voxels. The level above the leaf nodes will then encode  $1 \times 1 \times 8$  nodes, and higher levels will have the same node layout as before. We will show in Section 6.2 that this alternative node layout leads to extremely efficient PCF lookups.

To antialias shadow borders in the foreground, it is common practice

(and usually sufficient) to use a constant-size filter kernel for all lookups. However, if a single shadow map is used for a large scene, a view sample in the far distance can actually require an extremely large filter-kernel to avoid aliasing. We suggest two different methods to deal with this situation. Both methods will transform the pixel of the currently considered view sample to projected light space, to determine the approximate area in voxels. If the area is larger than the size of the PCF filter, that filter alone will not suffice, and we calculate at which level,  $L$ , in the tree the resolution is more appropriate.

**Multi-Resolution Antialiasing (MRAA).** The first solution is to simply generate several data structures, one for each power-of-two resolution, and store them side-by-side. For each lookup, we then decide which resolution is most appropriate (based on  $L$ ) and traverse the corresponding data structure. This method allows for very high-quality antialiasing for filter kernels of any size, but requires roughly twice the amount of memory compared to storing only the full resolution DAG.

**Voxel Occupancy Antialiasing (VOAA).** If the extra memory cost is not acceptable, an alternative method is to store, for each internal node, the proportion of contained voxels that lie in shadow (its *occupancy*). This can be done after compression, and the information is stored in the unused upper part of the word containing the childmask, so that there is no extra memory cost associated with this method. When traversing the data structure, we can now stop traversal when we have reached a node at level  $L$  and return the occupancy for that node as our visibility value. Note that this is an approximation and not equivalent to a standard filtered lookup, as the filter in this case is a volume, not an area. To avoid visible level-of-detail transitions in animation, the occupancy value can be a linear interpolation between the occupancy at level  $L$  and  $L - 1$ .

## 6 Implementation

In this section, we will discuss particulars about our implementation for building, and rendering with, the suggested data structure. We will then discuss a few optimizations to our traversal algorithm.

### 6.1 Building the Data Structure

We generate our data structure from a shadow map, as this leads to a straightforward top-down algorithm, but any method that can determine the visibility of leaf voxels could be used. We start from a shadow map that matches the  $(x, y)$  resolution of the desired voxelization. In our implementation, the  $z$  (depth) resolution of our data structure is set to equal  $x$  and  $y$ , but the layout could be chosen arbitrarily. We then build a min-max hierarchy of this shadow map, halving the resolution at each step. To build the data structure, we then start at the root node (which contains the entire light frustum), and test all of its children against the second level (of resolution  $2 \times 2$ ) of the min-max hierarchy. If the Axis Aligned Bounding Box (AABB) of the subnode is found to lie entirely beyond the maximum of the depths in the shadow map, the whole node is in shadow and the childmask of the root node is updated to reflect this. Conversely, if the entire AABB of the subnode lies closer to the light than the min depth found in the hierarchy, the node is marked as visible. If neither condition is true, the node is marked as intersected and the process is repeated recursively for this subnode, until the maximum level of the data structure is reached. At the final level, the voxels are tested against the original shadow map and visibility is recorded in the leafmask. We now have a Sparse Voxel Octree (SVO), which we proceed to reduce to an optimal DAG exactly as described by Kämpe et al. [2013]. At high resolutions we generate and compress

the shadow map in tiles, producing a set of sub-DAGs from each tile. These are then combined and compressed into the final, optimal, DAG.

**Closed objects optimization.** Closed objects are inserted into the tree using a slightly modified algorithm. As explained in Section 4, any node that lies entirely inside one or more closed objects can be considered undefined. Any node that lies entirely outside all closed objects must have correct visibility. Following a ray from the light, any intersection with a polygon will mean that we either enter or exit a closed object. In our implementation we identify all nodes that lie between the first intersection, and the intersection where the ray again enters empty space, as undefined. There is very little to be gained from also identifying subsequent undefined regions (as such regions are surrounded by shadowed space), and it would make the algorithm much more complex.

To achieve this, we generate (in addition to the shadow map), a map containing the depth at which the first undefined region *ends*. To handle intersecting closed objects, we extract the layers of the scene via depth peeling [Everitt 2001] and maintain a per-pixel counter (increased for front-facing, decreased for back-facing polygons), to find the first depth at which the counter returns to zero.

A min-max hierarchy is built for this map as well, and the generation of the initial SVO is changed so that a node that intersects neither map, but lies within the undefined region, is marked as undefined. A node that intersects *either* of the two maps is traversed further. Whenever all subnodes of a node have been evaluated, the parent node will decide the final status of its undefined children. If a node contains only shadowed and undefined subnodes, the entire node is set to shadowed. If a node contains only visible and undefined subnodes, it is set to visible. If the node contains subnodes of all three types (shadowed, visible and undefined), the undefined nodes are set to shadowed and the node is marked as intersected. This last rule is required because the final data structure cannot contain undefined nodes. There could be a potential gain in choosing undefined nodes so that the whole node will match an already existing node, but we have not found an efficient way of performing such a search. After this step, we have an SVO which is compressed exactly as before.

If the data structure is not going to be used in combination with dynamic geometry, the same optimization can be utilized for non-closed objects. Arvo and Hirvikorpi [2005] noted that in such cases, the space between the first and second shadow-casting layers can be considered undefined (as it will not be queried).

### 6.2 Traversing the Data Structure

We have evaluated our data structure in a deferred-shading context. In a first pass from the camera, position, normal and material properties are stored in a G-Buffer. Our shadow-evaluation code is implemented as a separate pass in CUDA, where a thread is started per view sample. The kernel evaluates light visibility for that view sample and stores the result in an auxiliary buffer to be used for shading in a final pass. Note that there is nothing in the traversal code that could not just as well have been implemented in a fragment shader, and so our data structure works in a forward shading context as well.

To evaluate the visibility of a view sample, we first calculate the discrete coordinate  $(0, 0, 0) \leq \mathbf{P} < (S, S, S)$  (where  $S$  is the resolution of the data structure) of the voxel it resides in. We obtain  $\mathbf{P}$  by simply scaling the *normalized device coordinates* of the sample. The components of  $\mathbf{P}$ , then, will be  $(L - 1)$ -bit wide integers, where  $L$  is the number of levels in the data structure. To find out which subnode to traverse into at some level  $i$ , we extract the  $(L - 2 - i)$ th

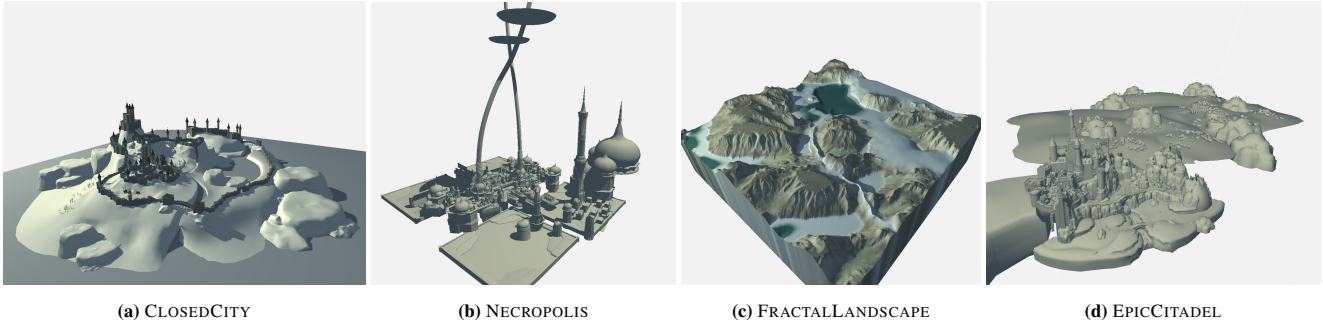


Figure 4: The scenes used to test compression and performance of our data structure.

bit of each component and concatenate them to a 3-bit *child index* that identifies the subnode (see Figure 5).

Next, we read the childmask of the root node (at level  $i = 0$ ) and look at the bits corresponding to this child index. If the node is marked as completely shadowed or visible, we return visibility 0 or 1 respectively. If the node is marked as intersected, we proceed recursively until we have reached level  $L - 3$ , the level above the leaf nodes. Here, we use the three last bits of  $z$  as our child index (due to the flat leaf-node layout explained above). At the leaf node level, the last three bits of  $x$  and  $y$  are concatenated to form the index of the correct bit of the leafmask.

$$\mathbf{P} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{matrix} \overbrace{\begin{bmatrix} 000110100110 \\ 010100111010 \\ 100101101011 \end{bmatrix}}^{L-1} \\ \text{Level } 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \end{matrix} \begin{matrix} \text{child index at level 7} \\ \text{child index at level 9} \\ \text{leaf index} \end{matrix}$$

Figure 5: The discrete position defines the traversal. Each bit defines a half space for the corresponding axis and level. The integer formed by concatenating bit  $k$  of each component is the child index to traverse at the corresponding level. The last three levels are encoded differently for fast PCF lookups.

**Filtered lookups.** As described in Section 5, the final three levels are stored as  $8 \times 8$ -bit grids containing the visibility of voxels at the same depth. Thus, with filter sizes of  $9 \times 9$  or less, we will at most need to visit 4 such leaf nodes, and we know beforehand which these nodes are. To obtain a filtered shadow value, we need to calculate, for each of these nodes, what proportion of the contained voxels lies inside of the filter and are in shadow. For each leaf node, we generate a bitmask that clears all bits that lie outside the filter (see Figure 6). After ANDing the childmask with this bitmask, we count the remaining set bits, and the sum of set bits for all four nodes are divided by the total number of voxels within the filter ( $9 \times 9$ , in this case), to obtain the filtered visibility.

**Traversal order.** With filtered lookups, we will visit several leaf nodes, and we know the path to each before we start traversal. Usually, the beginning of these paths will be identical, and we can easily establish at which levels they diverge. Therefore, we can significantly reduce the time taken in traversal by comparing the paths of two consecutive leaf nodes and backtracking only to the level at which the paths diverge.

If we do this, the order in which we visit the leaf nodes is also important. We can reduce the number of iterations in the traversal loop by choosing to always proceed to the leaf node that requires

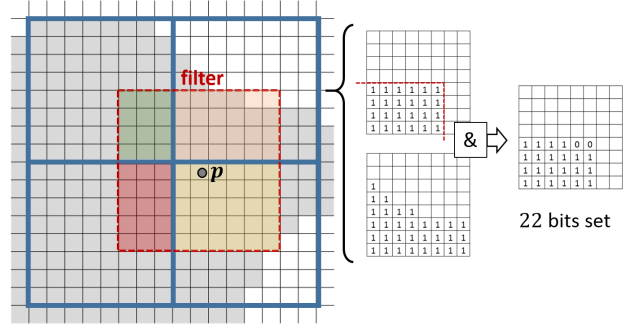


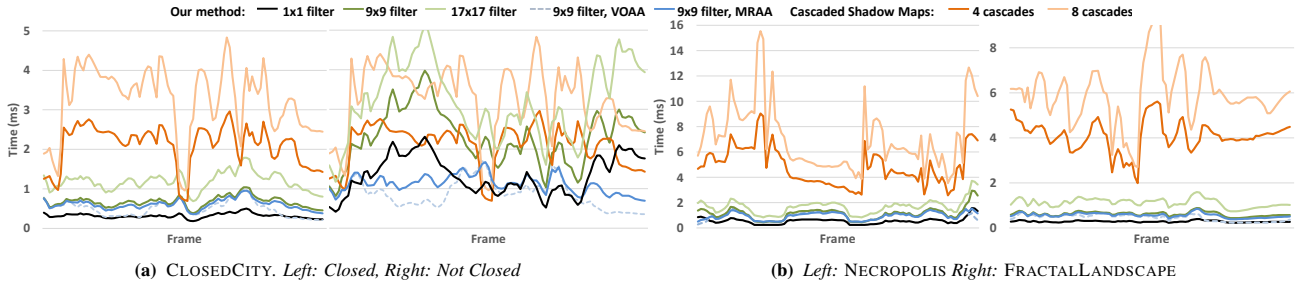
Figure 6: To obtain a filtered shadow value, we calculate a bitmask that corresponds to the voxels covered by the filter, for each node. This mask is ANDed with the leafmask. We then count the number of set bits in the result.

backtracking the smallest number of levels. Looking at Figure 7, we can see that moving from one leaf-node to another will entail backtracking to one of two different levels,  $L_1$  or  $L_2$  (with a  $9 \times 9$  filter or smaller). Whichever order we choose, we will backtrack to one of these levels once, and twice to the other. Therefore, with a fixed starting point, we will simply choose the one of two predetermined traversal orders that lets us backtrack to the higher level only once. Which traversal order to choose depends only on which of  $L_1$  and  $L_2$  are highest.

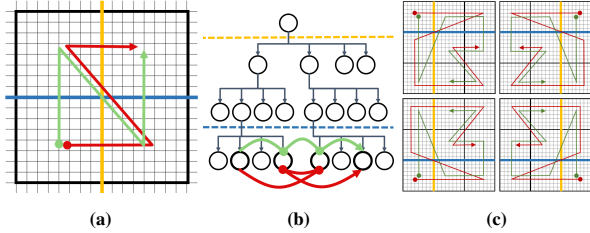
For larger filter kernels, there are more permutations of possible orders. Our optimized traversal implementation currently supports filter sizes up to  $17 \times 17$ , which will have to visit at most 9 leaf nodes. Of these nodes, some will have the same parent and should be evaluated consecutively (the order is unimportant). Figure 7c shows the four different ways that we can encounter these nodes and eight different traversal orders, one of which will be optimal. For filters of this size, we use a small precomputed lookup table to choose the right traversal order. The optimized CUDA kernel we use for traversal of our data structure is available as supplementary material.

**Optimizations.** We do not need a full stack in order to perform this backtracking. Since we know that we will only need to return to one of two possible levels, these are the only levels for which the state is stored.

The final optimization to our traversal code is to simply store the top part of our DAG in a grid structure, so that we can immediately fetch the offset to the sub-DAG we need to traverse. In our implementation, we have used a grid of size  $128 \times 128 \times 128$  throughout, effectively



**Figure 8:** Performance for our method with different sizes of filters and different antialiasing techniques for distant shadows, and CSM, with different numbers of cascades. Lookup cost for CSMs is in the range of 0.2ms-0.4ms and viewport culling cost is in the range of 0.1ms-0.35ms.



**Figure 7:** a) Two possible traversal orders for up to  $9 \times 9$  filters, one of which will be optimal. b) Examples of how these could access a hierarchy. In this case, the green traversal order is more efficient. c) The eight traversal orders we choose from for a  $17 \times 17$  filter.

replacing the top 8 levels of our DAG with an 8MB grid.

## 7 Results

In this section, we will report the results of using our data structure to compress shadow maps and to use them for evaluating shadows in real time. All performance results presented were measured on a PC equipped with an Intel Core i7-3930K CPU and an Nvidia TITAN GPU. The original size of all rendered images is  $1920 \times 1080$ .

**Scenes.** We have used four different scenes for our comparisons in this section (see Figure 4). The NECROPOLIS (2.1M triangles) and EPICCITADEL (373k triangles) scenes are from the Epic UDK. The CLOSED CITY scene (613k triangles) is a mock-up game scene made by us, where all objects are closed. The FRACTALLANDSCAPE scene (2.1M triangles) is a randomly generated fractal landscape and consists entirely of closed geometry. We have created fly-throughs for three of these scenes, which are used to measure performance for varying views. For the FRACTALLANDSCAPE scene, we have added a dynamic object that receives shadows from the static geometry. The scenes and fly-through paths are demonstrated in the accompanying video.

**Compression.** The final sizes of the compressed data for the various scenes are presented in Figure 9. The CLOSED CITY and FRACTALLANDSCAPE scenes both contain only closed geometry, and we have built them both with the closed-geometry optimization and without. Using the optimization, the compression factor is more than three orders of magnitude, for both scenes, when compared to a 16-bit shadow map at  $256k \times 256k$ . Without the optimization, compression ratios are still very good.

The NECROPOLIS and EPICCITADEL scenes are not closed, and so we could not use our closed-object optimization for these builds.

Nevertheless, both scenes compress to approximately 2% of a 16-bit shadow map of the same resolution ( $128k \times 128k$ ), making such extreme resolutions a possibility at a reasonable memory budget. NECROPOLIS is also built using the static-geometry optimization described in Section 6.1, and then compresses to approximately 0.5%.

**Build Times.** Building the data structure is currently a fairly time-consuming process. Our implementation runs on the CPU and only performs very simple parallelization of the workload, so running times could certainly be improved. As can be seen from the table below, the time taken to build our data structure scales approximately with the resolution of the shadow map from which it is built.

FRACTALLANDSCAPE, closed.						
1K <sup>3</sup>	4K <sup>3</sup>	16K <sup>3</sup>	32K <sup>3</sup>	64K <sup>3</sup>	128K <sup>3</sup>	256K <sup>3</sup>
0.5s	2s	18s	1m8s	4m16s	17m35s	1h32m

	Resolution:	1024 <sup>3</sup>	4096 <sup>3</sup>	16K <sup>3</sup>	32K <sup>3</sup>	128K <sup>3</sup>	256K <sup>3</sup>
CLOSED CITY	16-bit SM	2 MB	34 MB	537 MB	2 GB	34 GB	137 GB
	Closed	202 KB	1 MB	6 MB	13 MB	50 MB	100 MB
	Compression	9.61%	3.53%	1.16%	0.60%	0.14%	0.07%
	Not Closed	489 KB	4 MB	36 MB	96 MB	639 MB	1638 MB
FRACTALL.	16-bit SM	2 MB	34 MB	537 MB	2 GB	34 GB	137 GB
	Closed	164 KB	759 KB	3 MB	7 MB	29 MB	61 MB
	Compression	7.81%	2.26%	0.62%	0.33%	0.09%	0.04%
	Not Closed	490 KB	5 MB	50 MB	146 MB	1155 MB	-
NECROPOLIS	16-bit SM	2 MB	34 MB	537 MB	2 GB	34 GB	-
	Not closed	539 KB	5 MB	37 MB	100 MB	657 MB	-
	Compression	25.72%	13.66%	6.94%	4.66%	1.91%	-
	Static geometry	444 KB	3 MB	14 MB	31 MB	144 MB	-
CITADEL	16-bit SM	2 MB	34 MB	537 MB	2 GB	34 GB	-
	Not closed	585 KB	6 MB	44 MB	115 MB	727 MB	-
	Compression	27.90%	16.76%	8.19%	5.37%	2.11%	-

**Figure 9:** Compression results.

**Cascaded shadow maps.** For comparison we implemented the CSM algorithm, based on the description by Zhang et al. [2005]. View frustum split distances are a mix between linear and logarithmic positioning and are tweaked for each scene. A bounding-volume hierarchy is created for the scene, and the geometry is culled for each partition. The culling is implemented on the GPU using CUDA and directly produces drawing commands in an OpenGL buffer. To draw the batches into the cascades, we use layered rendering and a single draw call (glMultiDrawElementsIndirect), utilizing a geome-

try shader to route each batch to the appropriate cascade layer. To improve performance, we detect empty cascades, i.e. those that do not contain any shadow *receivers*, by performing an extra culling pass before culling shadow casters. Only hardware-supported  $2 \times 2$  taps PCF is used to filter the shadows. Each cascade layer has a resolution of  $2048 \times 2048$  and uses 16 bits of depth precision. As seen in Figure 8, the time taken to execute culling and shading is small, and the majority of the time using CSMs is spent rasterizing triangles. This demonstrates that using CSMs for dynamic geometry while using our solution for static geometry can provide a high-performance solution, by reducing the number of triangles rasterized.

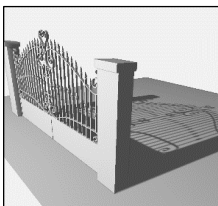
**Evaluation performance.** The time taken to evaluate shadows for each frame of the fly-throughs is presented in Figure 8a and Figure 8b, along with the time taken for our implementation of CSMs using four and eight cascades. The time measured for CSM is the sum of the time taken to render the shadow maps and the time taken to perform lookups, since each shadow map has to be re-rendered with every new viewpoint.

In most scenes, we outperform CSMs by a substantial margin for all variations of the algorithm, and also provide much higher visual quality. The closed version of CLOSED-CITY conforms to this, but in the non-closed version, performance is much closer. The reason is that the viewpoint is quite far from the scene, which means that memory access patterns are incoherent near the leaves. The antialiasing schemes presented in Section 5 improve this situation by terminating higher in the tree. This behaviour is mostly absent from the scene built with the closed geometry optimization, as traversal of unshadowed surfaces generally can terminate quite high in the tree, which shows that this is not only a storage optimization but also improves run-time performance considerably.

The results confirm that our algorithm handles large filter sizes very well, showing at worst a factor 2.5 increase in shading time when going from a single sample to  $9 \times 9$  taps.  $17 \times 17$  taps costs at worst 4.5 times a single tap.

We have measured the performance improvement obtained using our traversal order optimization by comparing to the time taken to traverse in a fixed order (still only backtracking to the closest common level between leaf nodes). The optimization reduces the worst case running time by 7.5% and 10.8% for  $9 \times 9$  and  $17 \times 17$  filters respectively, in the CLOSED-CITY scene (built with the closed objects optimization). In NECROPOLIS the corresponding numbers are 14.2% and 26.6%.

**Filtering performance.** We have compared the time taken to perform shadow lookups with large filter sizes between our algorithm, standard shadow maps, and VSMS. The resolution of the shadow data structures is  $2048 \times 2048$  in all cases. The results are shown together with the scene and view in Figure 10.



Filter Size	Blur	Evaluate	Sum
<b>Filter Size 9x9</b>			
Shadow Maps	-	0.52ms	0.52ms
VSM	0.62ms	0.19ms	0.81ms
Ours	-	0.43ms	0.43ms
<b>Filter Size 17x17</b>			
Shadow Maps	-	1.9ms	1.9ms
VSM	0.82ms	0.19ms	1.01ms
Ours	-	0.76ms	0.76ms

Figure 10: Lookup performance with large filter sizes.

For the shadow-map algorithm, we utilized hardware-accelerated PCF filtering to take  $2 \times 2$  samples in each texture lookup. We used a 16-bit shadow map, so the memory cost is 8MB. For VSMS, we

rendered a standard shadow map which was then blurred in two passes with a separable box filter. The VSM is stored in a two-channel 32-bit buffer, so the memory cost is 32MB, making our method interesting even for small static maps (at 981Kb, for this scene, when using neither of the space-saving optimizations). While the lookup in a VSM is very fast, the map must also be blurred each frame if used in a CSM setting, and the blurring stage is quite costly and scales poorly with higher resolutions.

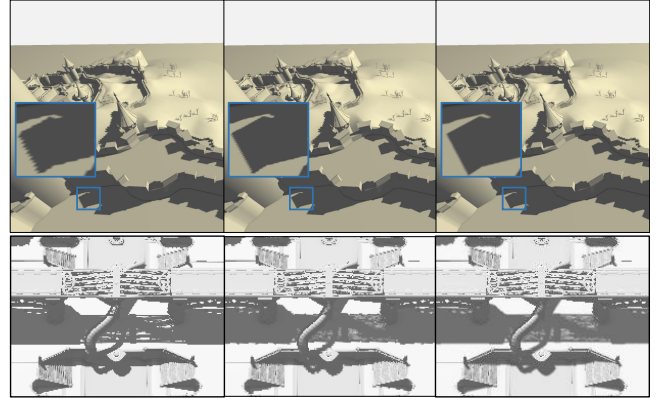


Figure 11: Top: Filtering quality. Left to right are CSM (4 layers,  $2 \times 2$ , 1.8ms), CSM (8 layers,  $2 \times 2$ , 2.4ms) and OURS ( $9 \times 9$  filter, 0.47ms). Bottom: Distant shadow antialiasing. Looking at a small region of an image of NECROPOLIS (see also the supplementary video). Left to right:  $9 \times 9$  filter, VOAA, MRAA.

**Image quality.** The top row of Figure 11 shows our basic  $9 \times 9$  filter compared to the fast  $2 \times 2$  shadow-map filtering supported by hardware. The much higher shadow resolution and larger filter sizes achievable with our method lead to much higher quality shadow borders. The bottom row shows the shadow cast from a complex structure in the NECROPOLIS scene. The camera is very far away, and the figure shows only a small region of the image. Here, a  $9 \times 9$  filter is clearly insufficient. The VOAA method manages to antialias the cast shadows quite well but transitions are noticeable under animation. The MRAA method works extremely well, with barely noticeable transitions in level.

## 8 Limitations and Future Work

Our technique stores visibility information at a quality equivalent to a shadow map, given that the resolution in  $z$  (depth) is sufficient. Therefore, self-shadowing artifacts for hard shadows can be avoided using similar biasing techniques as are used for shadow mapping. For efficient lookups with larger filter sizes, however, our method tests all samples against the same depth, which is more restricting than using individual shadow-map taps. Our solution is to always bias the shadow-lookup position one half filter size in the direction of the surface normal, which, with very large filter sizes, can lead to a visible offset of the shadow. More sophisticated biasing methods, as well as developing a blocker-search method to estimate penumbra widths, are interesting areas of future research.

As our data structure has uniform resolution in *light space*, a perspective light will have non-uniform resolution in *world space*. Thus, visibility close to the light might be stored at unnecessarily high resolutions. A future improvement would be to allow for an increasing  $(x, y)$  resolution at increasing distance from the light.

Our antialiasing methods for distant shadows require a large bias (in world space) for very distant shadows and could potentially

lead to light leakage for very thin shadow casters, although we have not observed any. This problem is very similar to what would occur with CSMs. With the VOAA approach it is also possible to see the transitions from one level-of-detail to another, as we move away from the shadow, which can probably be avoided by linearly interpolating between levels.

We would like to explore using our data structure for ray tracing of volumetric shadows for single scattering and for compressing shadows from transparent shadow casters. We would also like to optimize our construction methods.

## Acknowledgements

The EPICCITADEL and NECROPOLIS scenes are both distributed with the Unreal Development Kit by Epic Games. The FRACTAL-LANDSCAPE scene is generated using the software World Machine. This work was supported by the Swedish Foundation for Strategic Research under Grant RIT10-0033. The TITAN GPU used for this research was donated by the NVIDIA Corporation.

## References

- AILA, T., AND LAINE, S. 2004. Alias-free shadow maps. In *Proc. EG Symposium on Rendering 2004*, EGSR'04, 161–166.
- ANNEN, T., MERTENS, T., BEKAERT, P., SEIDEL, H.-P., AND KAUTZ, J. 2007. Convolution shadow maps. In *Proc. EG Symposium on Rendering 2007*, EGSR'07, 51–60.
- ANNEN, T., MERTENS, T., SEIDEL, H.-P., FLERACKERS, E., AND KAUTZ, J. 2008. Exponential shadow maps. In *Proc. of Graph. Interface 2008*, Canadian Information Proc. Soc., GI '08, 155–161.
- ARVO, J., AND HIRVIKORPI, M. 2005. Compressed shadow maps. *Vis. Comput.* 21, 3 (Apr.), 125–138.
- BUNNEL, M., AND PELLACINI, F. 2004. Shadow map antialiasing. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, R. Fernando, Ed. Pearson Higher Education.
- CROW, F. C. 1977. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph.* 11 (July), 242–248.
- DONNELLY, W., AND LAURITZEN, A. 2006. Variance shadow maps. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, ACM, I3D '06, 161–165.
- EISEMANN, E., SCHWARZ, M., ASSARSSON, U., AND WIMMER, M. 2011. *Real-Time Shadows*. A.K. Peters.
- ENGEL, W. 2006. Cascaded shadow maps. In *ShaderX5: Advanced Rendering Techniques*, T. Forsyth, Ed., Shaderx series. Charles River Media, Inc.
- EVERITT, C., 2001. Interactive order-independent transparency. Published online at [http://www.nvidia.com/object/Interactive\\_Order\\_Transparency.html](http://www.nvidia.com/object/Interactive_Order_Transparency.html).
- GIEGL, M., AND WIMMER, M. 2007. Fitted virtual shadow maps. In *Proceedings of Graphics Interface 2007*, ACM, GI '07, 159–168.
- GREEN, C. 2007. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 Courses*, ACM, SIGGRAPH '07, 9–18.
- HASSELGREN, J., AND AKENINE-MÖLLER, T. 2006. Efficient depth buffer compression. In *Proc. 21st ACM SIGGRAPH/EG Symp. on Graphics Hardware*, ACM, GH '06, 103–110.
- HEIDMANN, T. 1991. Real shadows, real time. *Iris Universe* 18, 28–31. Silicon Graphics, Inc.
- JOHNSON, G. S., LEE, J., BURNS, C. A., AND MARK, W. R. 2005. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph.* 24, 4 (Oct.), 1462–1482.
- KÄMPE, V., SINTORN, E., AND ASSARSSON, U. 2013. High resolution sparse voxel dags. *ACM Trans. Graph.* 32, 4 (July), 101:1–101:13.
- LEFEBVRE, S., AND HOPPE, H. 2007. Compressed random-access trees for spatially coherent data. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, Eurographics Association, EGSR'07, 339–349.
- LEFOHN, A. E., SENGUPTA, S., AND OWENS, J. D. 2007. Resolution-matched shadow maps. *ACM Trans. Graph.* 26, 4 (Oct.).
- LLOYD, D. B., TUFT, D., YOON, S.-E., AND MANOCHA, D. 2006. Warping and partitioning for low error shadow maps. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques*, Eurographics Association, EGSR'06, 215–226.
- RAMAMOORTHY, R. 2009. Precomputation-based rendering. *Found. Trends. Comput. Graph. Vis.* 3, 4 (Apr.), 281–369.
- RASMUSSEN, J., STRÖM, J., WENNERSTEN, P., DOGGETT, M., AND AKENINE-MÖLLER, T. 2010. Texture compression of light maps using smooth profile functions. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, 143–152.
- SINTORN, E., EISEMANN, E., AND ASSARSSON, U. 2008. Sample based visibility for soft shadows using alias-free shadow maps. In *Proc. of 19th EG Conf. on Rendering*, Eurographics Association, EGSR'08, 1285–1292.
- SINTORN, E., OLSSON, O., AND ASSARSSON, U. 2011. An efficient alias-free shadow algorithm for opaque and transparent objects using per-triangle shadow volumes. *ACM Trans. Graph.* 30, 6 (Dec.), 153:1–153:10.
- WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.* 12 (August), 270–274.
- WOO, A., AND POULIN, P. 2012. *Shadow Algorithms Data Miner*. Taylor & Francis.
- ZHANG, F., SUN, H., AND NYMAN, O. 2005. Parallel-split shadow maps on programmable GPUs. In *GPU Gems 3*, Addison-Wesley, H. Nguyen, Ed.
- ZHANG, F., SUN, H., XU, L., AND LUN, L. K. 2006. Parallel-split shadow maps for large-scale virtual environments. In *Proc. Virtual Reality Continuum and Its Applications*, ACM, VRCIA '06, 311–318.