Fast, Memory-Efficient Construction of Voxelized Shadows

Viktor Kämpe Erik Sintorn Ulf Assarsson

Chalmers University of Technology

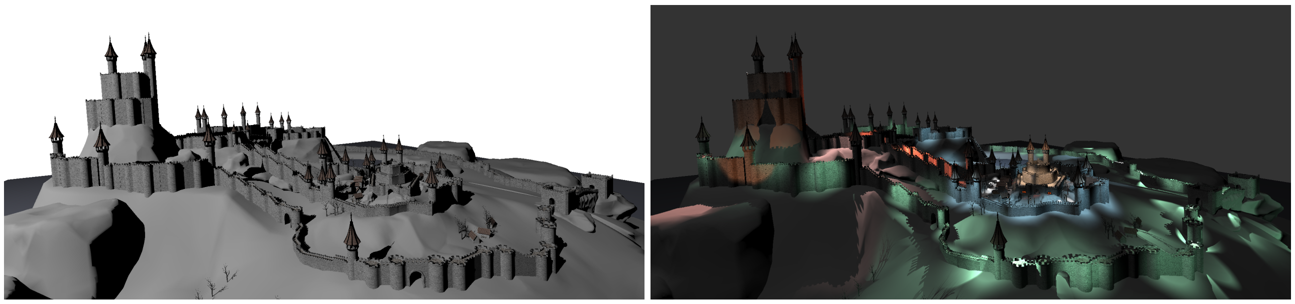

Figure 1: *The left image shows a scene lit by the sun with precomputed voxelized shadows of resolution* 262144^3*. Our novel algorithm generates this shadow information in 38 seconds and compresses it to 48MB (vs. 100MB for the previous CPVS method). To the right is the same scene lit by 165 spotlights with precomputed shadows, each with a resolution of* 8192^3*. The average build time for these CPVSs is 114ms, and the average size is 0.5MB (vs. 128MB for a 16-bit shadow map). Evaluating shadows for all lights at 1920×1080 takes 3.2ms.*

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. 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.

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

Keywords: 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) [Engel 2006; Zhang et al. 2006] approach. The idea is to split the current camera-view frustum into several regions, or *cascades*, and to render a traditional shadow map [Williams 1978] 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 [Sintorn et al. 2014]. 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 an elegant, non-intuitive, modification to the algorithm to improve its performance. We show that much of the common sub-tree merging can be performed 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 5) that this results in a performance increase of approximately 200× for the actual DAG construction and a performance increase of approximately 20× for the full construction. For scenes with no closed geometry, we still achieve an improved performance of approximately 2×. 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).

Our second 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 results in an up to 3× 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. 8192^3, making them an affordable alternative to shadow maps also in scenarios where we have many bounded lights (see Figure 1). In Section 4, 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. [2011]. 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 [2009]). 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. [2010] for a survey of hardware accelerated light-map compression, or Lefebvre and Hoppe [2007] 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-parametrization 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 [2014]). 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 [2005] and by Sintorn et al. [2014] both achieve high compression rates while allowing for fast filtered shadow lookups.

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* [Olsson and Assarsson 2011], *Forward+* [Harada et al. 2013] and *Clustered Shading* [Olsson et al. 2012]. These techniques do not explicitly take shadowing into account, however. In a recent paper by Olsson et al. [2014], 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 briefly describe how the voxelized shadows are computed. In Section 3.2, we will explain how we improve the construction speed, and in Section 3.4, we will describe how we improve the memory performance of the voxelized shadows. The CPVS stores binary visibility information for every cell in a grid that is a discretization of the light's *Normalized 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 [Sintorn et al. 2014]. 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 is achieved by, for each level in bottom-up order, sorting the nodes, removing all but one of identical nodes, and updating the references of their parents to reference the unique nodes.

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 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

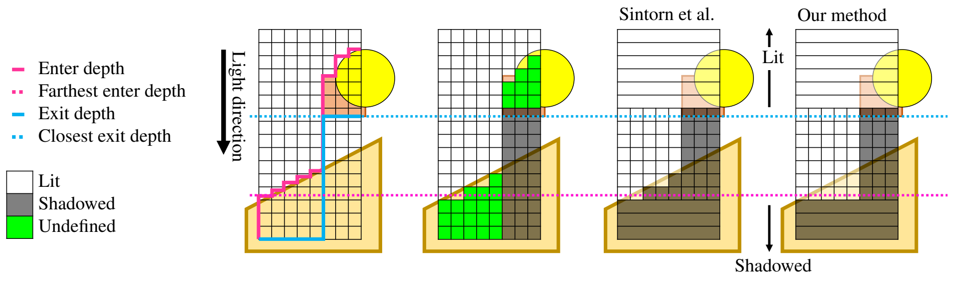

Figure 2: *At the finest level, we classify the cells as either lit, shadowed or undefined from the enter and exit-depth maps (two left-most figures). Our method of resolving undefined regions results in fewer unique visibility masks (two right-most figures).*

boundary is identical for all depths between the shadow-casting surfaces, 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). 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 subvolumes, we obtain this depth from the min-max hierarchy, and at the finest level, we obtain this depth from the depth map.

3.3 Finding Undefined Regions

Regions inside closed geometry will never be queried and can, therefore, hold an undefined value (shadowed or lit). We detect such regions using two depth maps (see Figure 2). The first depth map, the *enter-depth* map, is the depth at which light rays first enter shadow casters, and we render it as an ordinary shadow map. The second depth map, the *exit-depth* map, is the depth at which the light rays first exit the closed geometry and reach the outside again. All volumes that are between these two depth maps have undefined shadow values, since they are inside a union of closed objects. To find this *exit depth*, we iteratively depth peel front-facing and back-facing triangles separately (i.e, to separate buffers), starting at the *enter depth*. After each peel, we test whether we have exited the closed geometry, which is equivalent to the next back-facing surface being closer than the next front-facing surface. Peeling the front- and back-facing triangles separately has two advantages. First, the counter of the number of times we enter and exit a closed object becomes implicit. Secondly, for the same number of processed triangles and fragments, we advance two layers, instead of one.

3.4 Resolving Undefined Regions

We exploit undefined regions to reduce the overall memory consumption. As Sintorn et al. [2014], 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. [2014] 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 5) that it has a significant positive impact on the final memory performance. This new method locally minimizes the number of unique visibility 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 8×8 texel tile, and is very fast to execute.

Prior to constructing the DAG, we consider each 8×8 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 Figure 2). 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

		Resolution	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$	$128K^3$	$256K^3$
No culling	non closed	total	434ms	1.7s	6.5s	24.1s	92.7s	357.9s	1398.8s
		insert/merge	417ms	1.7s	6.4s	23.5s	90.3s	348.6s	1362.4s
	closed	total	235ms	695ms	2.5s	9.1s	35.8s	139.5s	550.5s
		insert/merge	146ms	580ms	2.3s	8.4s	33.2s	130.1s	515.2s
Culling	non closed	total	300ms	1.2s	4.5s	16.3s	61.5s	235.4s	906.4s
		insert/merge	283ms	1.2s	4.4s	15.7s	59.2s	226.1s	870.1s
	closed	total	113ms	185ms	377ms	1.1s	3.3s	10.9s	38.1s
		insert/merge	25ms	72ms	151ms	347ms	763ms	1.6s	3.3s

Table 1: *Construction times for* CLOSEDCITY *(our implementation) with and without culling construction of identical nodes.*

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 8×8 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 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. [2014] 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 4096×4096. 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 5) 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 [Olsson and Assarsson 2011], 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 [Olsson et al. 2012]. 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. 4096), 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. [2012], 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:

$$\mathbf{z} = \left\lfloor N \frac{\log \frac{\mathbf{z}_{\mathrm{vs}}}{\mathrm{near}}}{\log \frac{\mathrm{far}}{\mathrm{near}}} \right\rfloor \tag{1}$$

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. [2014], 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.

5 Results

We have performed all measurements on a desktop computer with an Intel Core i5 2500K CPU, 16 GB DDR3 1333MHz RAM, and an NVIDIA GTX Titan GPU connected via PCI Express 2.0 x16.

5.1 Construction

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

- Render depth maps (OpenGL).
- Compute the min-max hierarchy and visibility masks (CUDA).

- Transfer the min-max hierarchy and visibility masks to host (PCIe).
- Insert and merge nodes in the 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 up to an entry per 8×8 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 construction, insertion, and merging of DAG-nodes on the CPU. Since rendering depth maps of the full resolution is infeasible, we perform construction for one 8K×8K texel region at a time.

	ClosedCity			FractalL.		
Res.	[MB]		ratio	[MB]		ratio
	new	old		new	old	
$4K^3$	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

Table 2: *Resulting memory consumption with the closed object optimization, comparing the new method with Sintorn et al. [2014].*

We have measured 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. [2014]. One consists of non-closed geometry (NECROPOLIS), while the other two are entirely built from closed geometry (CLOSEDCITY and FRACTAL-LANDSCAPE). All scenes are built with resolutions from $4K^3$ to $256K^3$. Without the re-use of already constructed nodes, the construction time is dominated by the insertion and merging of nodes (see Table 1). For non-closed construction, and when re-using nodes, the time for insertion and merging is approximately halved, since we now only have to process nodes along the shadow-casting surfaces. For closed construction, re-using nodes saves the same amount of processing as it does for non-closed construction (in absolute terms), but as we do not process the shadow-casting surfaces, the insertion and merging work is decreased by up to two orders of magnitude.

For non-closed construction, the *total* construction time is still dominated by the insertion and merging of nodes (see Table 3). For closed construction, however, the time for insertion and merging only amounts for less than a fourth of the total construction time at $4K^3$ resolution. This proportion decreases further with increasing resolutions, as the amount of voxels around the silhouette grows by 2× per increase in resolution (while the rest of the construction grows by 4×). A further breakdown of the construction time shows that the dominating step is the detection of undefined regions (the depth peeling of the *exit-depth* map).

Our construction times are more than two orders of magnitudes faster than those reported by Sintorn et al. [2014] (see Table 4), but they also state that the construction speed was not their primary concern.

5.2 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 2).

Resolution	$4K^3$	$16K^3$	$64K^3$	$256K^3$
non-closed	300ms	4.5s	61.5s	906.4s
Enter depth map	4ms	20ms	291ms	4.5s
Visibility masks	2ms	29ms	482ms	7.8s
Min-max	1ms	1ms	7ms	119ms
Transfer	7ms	90ms	1.4s	22.6s
Insert nodes	107ms	1.7s	22.1s	339.8s
Merge nodes	176ms	2.7s	37.2s	530.4s
closed	113ms	377ms	3.3s	38.1s
Enter depth map	5ms	19ms	271ms	4.4s
Exit depth map	78ms	176ms	1.8s	23.0s
Visibility masks	1ms	12ms	181ms	2.9s
Min-max	1ms	1ms	7ms	111ms
Transfer	2ms	16ms	245ms	3.9s
Insert nodes	10ms	63ms	312ms	1.4s
Merge nodes	15ms	88ms	451ms	2.0s

Table 3: *Breakdown of construction timings for* CLOSEDCITY. *The most time consuming parts are highlighted in red.*

5.3 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. [2014]) to contain 165 spot-lights, each of which lights a small portion of the scene. For each spotlight, we built a CPVS at resolution 8192^3, which results in sufficiently sharp shadows for all lights, with a 9×9 PCF filter. Figure 3 shows how the construction times and final data-structure sizes are distributed over the different lights. The sizes of the data structures vary between 0.3 and 1.7 MB, for a total of 96MB. 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.

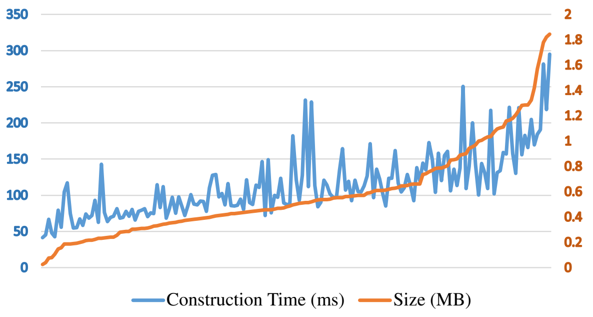

Figure 3: *Distribution of final data structure sizes and construction times for the 165 CPVSs in* CLOSEDCITY.

Figure 4 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 1920×1080. The first two steps are generating the bounding boxes for each 8×8 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

Resolution		$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$	$128K^3$	$256K^3$
NECROPOLIS	non-closed	209ms	720ms	2.6s	9.6s	34.5s	126.6s	470.4s
CLOSEDCITY	non-closed	300ms	1.2s	4.5s	16.3s	61.5s	235.4s	906.4s
	closed	113ms	185ms	377ms	1.1s	3.3s	10.9s	38.1s
FRACTALLANDSCAPE	non-closed	264ms	1.0s	4.0s	15.2s	57.6s	220.7s	874.2s
	closed	56ms	91ms	192ms	570ms	2.0s	7.1s	25.3s
Sintorn et al. [2014]	closed	2.0s	18.0s		68.0s	256.0s	1055.0s	5520.0s

Table 4: *Total CPVS construction times for three scenes with and without detection of undefined region inside closed geometry. The last row contains the construction times of Sintorn et al. [2014].*

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 9×9 PCF filter. We replace the top six levels of each CPVS with a small grid of pointers, which costs an aditional 128kB per light, for a small performance improvement. At worst, the time for calculating shadows for all pixels is 9ms.

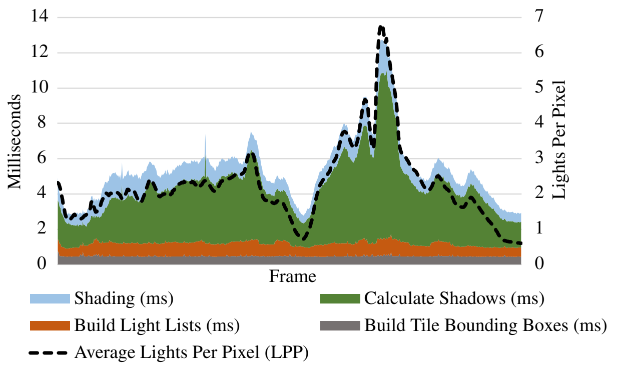

Figure 4: *The measured performance in a flythrough animation of the* CLOSEDCITY *scene. The dashed line is the average number of lights that are assigned to each tile in the frame.*

6 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 show that CPVSs for hundreds of spotlights, at a resolution of $8K^3$, can be constructed at around 100ms and 0.5MB 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.

For closed geometry, the construction bottleneck is no longer the insertion and merging of nodes, but the identification of undefined regions. In the future, we would like to explore alternative methods of identifying these undefined regions.

Acknowledgements

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.

References

ARVO, J., AND HIRVIKORPI, M. 2005. Compressed shadow maps. *Vis. Comput. 21*, 3 (Apr.), 125–138.

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.

HARADA, T., MCKEE, J., AND YANG, J. C. 2013. Forward+: A step toward film-style shading in real time. In *GPU Pro 4*, W. Engel, Ed. CRC Press, 115–135.

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.

OLSSON, O., AND ASSARSSON, U. 2011. Tiled shading. *Journal of Graphics, GPU, and Game Tools 15*, 4, 235–251.

OLSSON, O., BILLETER, M., AND ASSARSSON, U. 2012. Clustered deferred and forward shading. In *HPG '12: Proceedings of the Conference on High Performance Graphics 2012*.

OLSSON, O., SINTORN, E., KÄMPE, V., BILLETER, M., AND ASSARSSON, U. 2014. Efficient virtual shadow maps for many lights. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM.

RAMAMOORTHI, R. 2009. Precomputation-based rendering. *Found. Trends. Comput. Graph. Vis. 3*, 4 (Apr.), 281–369.

RASMUSSON, 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.

SCHULZ, N., 2014. The rendering technology of ryse.

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

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph. 12* (August), 270–274.

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.