Efficient Virtual Shadow Maps for Many Lights

Ola Olsson* Erik Sintorn* Viktor Kämpe* Markus Billeter* Ulf Assarsson*

Chalmers University of Technology

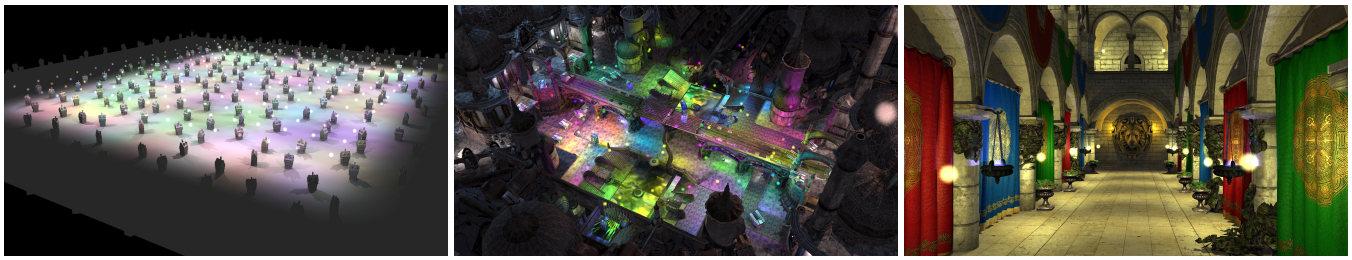


Figure 1: *Scenes rendered with many lights casting shadows at 1920×1080 resolution on an NVIDIA Geforce Titan. From the left:* HOUSES *with 1.01M triangles and 256 lights (23ms),* NECROPOLIS *with 2.58M triangles and 356 lights (34ms),* CRYSPONZA *with* 302*K triangles and 65 lights (16ms).*

Abstract

Recently, several algorithms have been introduced that enable real-time performance for many lights in applications such as games. In this paper, we explore the use of hardware-supported virtual cube-map shadows to efficiently implement high-quality shadows from hundreds of light sources in real time and within a bounded memory footprint. In addition, we explore the utility of ray tracing for shadows from many lights and present a hybrid algorithm combining ray tracing with cube maps to exploit their respective strengths. Our solution supports real-time performance with hundreds of lights in fully dynamic high-detail scenes.

CR Categories: I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Display Algorithms

Keywords: real-time,shadows,virtual,cube map

1 Introduction

In recent years, several techniques have been presented and refined that enable real-time performance for applications such as games using hundreds to many thousands of lights. These techniques work by binning lights into tiles of various dimensionality [Olsson and Assarsson 2011; Harada 2012; Olsson et al. 2012]. Many simultaneous lights enable both a higher degree of visual quality and greater artistic freedom, and these techniques are therefore directly applicable in the games industry [Swoboda 2009; Ferrier and Coffin 2011; Persson and Olsson 2013].

However, this body of previous work on real-time many-light algorithms has studied almost exclusively lights that do not cast shadows. While such lights enable impressive dynamic effects and more detailed lighting environments, they are not sufficient to capture the details in geometry, but tend to yield a flat look. Moreover, neglecting shadowing makes them more difficult to use, as light may leak through walls and similar occluding geometry, if care is not taken when placing the lights. For dynamic effects in interactive environments, controlling this behaviour is even more problematic. Shadowing is also highly important if we wish to employ the lights to visualize the result of some light-transport simulation, for example as done in *Instant Radiosity* [Keller 1997].

This paper aims to compute shadows for use in real-time applications supporting several tens to hundreds of shadow-casting lights. The shadows are of high and uniform quality, while staying within a bounded memory footprint.

As a starting point, we use *Clustered Deferred Shading* [Olsson et al. 2012], as this algorithm offers the highest light-culling efficiency among current real-time many-light algorithms and the most robust shading performance. This provides a good starting point when adding shadows, as the number of lights that require shadow computations is already close to the minimal set. Moreover, clustered shading provides true 3D bounds around the samples in the frame buffer and therefore can be viewed as a fast voxelization of the visible geometry. Thus, as we will see, clusters provide opportunities for efficient culling of shadow casters and allocation of shadow resolution.

1.1 Contributions

We contribute an efficient culling scheme, based on clusters, which is used to render shadow-casting geometry to many cube shadow maps. We demonstrate that this can enable real-time rendering performance using shadow maps for hundreds of lights, in dynamic scenes of high complexity.

We also contribute a method for quickly determining the required resolution of the shadow maps. This is used to show how hardware-supported virtual shadow maps may be efficiently implemented. To this end, we also introduce a very efficient way to determine the parts of the virtual shadow map that need physical backing. We demonstrate that these methods enable the memory requirements to stay within a limited range, while enabling uniform shadow quality.

Additionally, we explore the performance of ray tracing for many lights. We demonstrate that a hybrid approach, combining ray tracing and cube maps, offers high efficiency, in many cases better than

*e-mail:ola.olsson|erik.sintorn|kampe|billeter|uffe@chalmers.se

using either shadow maps or ray tracing individually.

We also contribute implementation details of the discussed methods, showing that shadow maps indeed can be made to scale to many lights. Thus, this paper provides an important benchmark for other research into real-time shadow algorithms for many lights.

2 Previous Work

Real Time Many Light Shading *Tiled Shading* is a recent technique that supports many thousands of lights in real-time applications [Swoboda 2009; Olsson and Assarsson 2011; Harada 2012]. In this technique, lights are binned into 2D screen-space tiles that can then be queried for shading. This is a very efficient and simple process, but the 2D nature of the algorithm creates a strong view dependence, resulting in poor worst case performance and unpredictable frame times.

Clustered Shading extends the technique by considering 3D bins instead, which improves efficiency and robustness [Olsson et al. 2012]. The clusters provide a three-dimensional subdivision of the view frustum and, thus, sample groupings with predictable bounds. This provides a basic building block for many of the new techniques described in this paper. See Section 3.1, for a more detailed overview.

Shadow Algorithms Studies on shadowing techniques generally present results using a single light source, usually with long or infinite range. Consequently, it is unclear how these techniques scale to many light sources, whereof a large proportion cover only a few samples. For a general review of shadow algorithms, see Eisemann et. al. [2011].

Virtual Shadow Maps Software-based virtual shadow maps have been explored in several publications to achieve high quality shadows in bounded memory [Fernando et al. 2001; Lefohn et al. 2007]. Recently, API and hardware extensions have been introduced that makes it possible to support virtual textures much more conveniently and with performance equalling that of traditional textures [Sellers et al. 2013].

Many light shadows There does exist a corpus of work in the field of real-time global illumination, which explores using many light sources with shadow casting, for example *Imperfect Shadow Maps* [Ritschel et al. 2008], and *Many-LODs* [Hollander et al. 2011]. However, these techniques generally assume that a large number of lights affect each sample to conceal approximation artifacts. In other words, these approaches are unable to produce accurate shadows for samples lit by only a few lights.

Ray Traced Shadows Recently, Harada et. al. [2013] described ray traced lights in conjunction with Tiled Forward Shading. They demonstrate that it can be feasible to ray trace shadows for many lights but do not report any analysis or comparison to other techniques.

3 Basic Algorithm

Our basic algorithm is shown below. The algorithm is constructed from clustered deferred shading, with shadow maps added. Steps that are inherited from ordinary clustered deferred shading are shown in gray.

1. Render scene to G-Buffers.
2. Cluster assignment – calculating the cluster keys of each view sample.
3. Find unique clusters – finding the compact list of unique cluster keys.
4. Assign lights to clusters. – creating a list of influencing lights for each cluster.
5. Select shadow map resolution for each light.
6. Allocate shadow maps.
7. Cull shadow casting geometry for each light.
8. Rasterize shadow maps.
9. Shade samples.

3.1 Clustered Shading Overview

In clustered shading the view volume is subdivided into a grid of self-similar sub-volumes (clusters), by starting from a regular 2D grid in screen space, e.g. using tiles of 32×32 pixels, and splitting exponentially along the depth direction. Next, all visible geometry samples are used to determine which of the clusters contain visible geometry. Once the set of occupied clusters has been found, the algorithm assigns lights to these, by intersecting the light volumes with the bounding box of each cluster. This yields a list of cluster/light pairs, associating each cluster with all lights that may affect a sample within (see Figure 2). Finally, each visible sample is shaded by looking up the lights for the cluster it is within and summing their contributions.

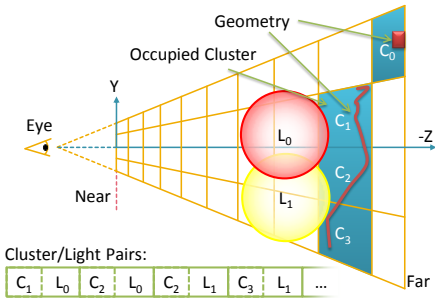


Figure 2: *Illustration of the depth subdivisions into clusters and light assignment. Clusters containing some geometry are shown in blue.*

The key pieces of information this process yields are a set of occupied clusters with associated bounding volumes (that approximate the visible geometry), and the near-minimal set of lights for each cluster. Intuitively, this information should be possible to exploit for efficient shadow computations, and this is exactly what we aim to do in the following sections.

3.2 Shadow Map Resolution Selection

One way to calculate the required resolution for each shadow map is to use the screen-space coverage of the light-bounding sphere. However, this produces vast overestimates whenever the camera is near, or within, the light volume. To calculate a more precisely matching resolution, one might follow the approach in *Resolution Matched Shadow Maps* (RMSM) [Lefohn et al. 2007], using shadow-map space derivatives for each view sample. However, applying this naïvely would be expensive, as the calculations must be repeated for each sample/light pair, and would require derivatives to be stored

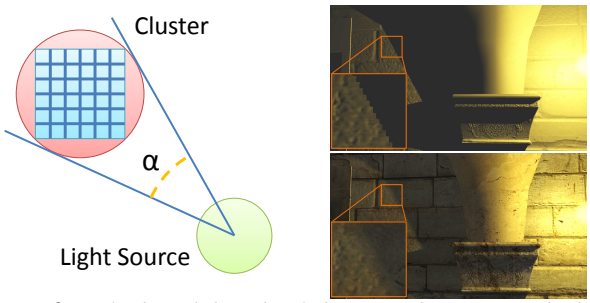


Figure 3: *Left, the solid angle of cluster, with respect to the light source, α, subtended by the cluster, illustrated in 2D. Right, example of undersampling due to an oblique surface violating assumptions in Equation 1, shown with and without textures and PCF.*

in the G-Buffer. Our goal is not to attempt alias-free shadows, but to quickly estimate a reasonable match. Therefore, we base our calculations on the bounding boxes of the clusters, which are typically several orders of magnitude fewer than the samples.

$$R = \sqrt{\frac{S/(\alpha/4\pi)}{6}} \tag{1}$$

The required resolution (R) for each cluster is estimated as the number of pixels covered by the cluster in screen space (S), divided by the proportion of the unit sphere subtended by the solid angle of the cluster bounding sphere (α), and distributed over the six cube faces (see Figure 3 and Equation 1).

This calculation is making several simplifying assumptions. The most significant is that we assume that the distribution of the samples is the same in shadow-map space as in screen space. This leads to an underestimate of the required resolution when the light is at an oblique angle to the surface (see Figure 3). A more detailed calculation might reduce these errors, but we opted to use this simple metric, which works well for the majority of cases.

For each cluster/light pair, we evaluate Equation 1 and retain the maximum R for each light as the shadow map resolution, i.e. a cube map with faces of resolution $R \times R$.

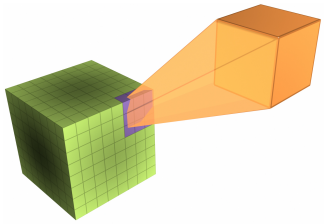


Figure 4: *The projected footprint (purple) of an AABB of either a batch or a cluster (orange), projected onto the cube map (green). The tiles on the cube map represent either virtual texture pages or projection map bits, depending on application.*

3.3 Shadow Map Allocation

Using the resolutions computed in the previous step, we can allocate one *virtual* cube shadow map for each light requiring a non-zero resolution. This does not allocate any actual physical memory backing the texture, just the virtual range.

In virtual textures, the pages are laid out as tiles of a certain size (e.g. 256×128 texels), covering the texture. Before we can render into the shadow map we must *commit* physical memory for those pages that will be sampled during shading. This can be established by projecting each sample onto the cube map, and record the requested page. To implement this efficiently, we again use the cluster bounds as proxy for the view samples, and project these onto the cube maps, (see Figure 4). The affected tiles are recorded in the *virtual-page mask*.

3.4 Culling Shadow-Casting Geometry

When managing many lights, culling efficiency is an important problem. The basic operation we wish to perform is to gather the minimal set of triangles that need to be rendered into each cube shadow map. This can be achieved by querying an acceleration structure with the bounding sphere defined by the light position and range. Real-time applications typically support this kind of query against a scene graph, or similar, for view frustum and shadow-map culling.

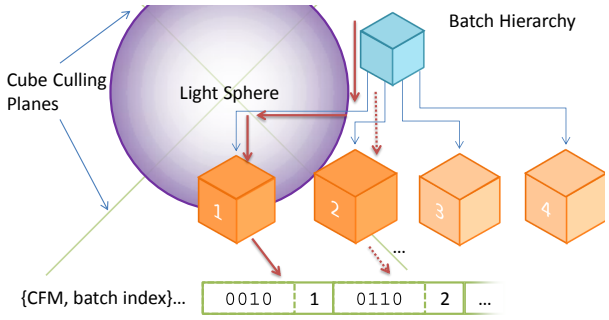


Figure 5: *Illustration of batch hierarchy traversal. The AABBs of batches* 1 *and* 2 *intersect the light sphere, and are tested against the culling planes, which determine the cube faces the batch must be rendered to.*

We make use of a *bounding volume hierarchy* (BVH), storing groups of triangles called *Batches* at the leaves. Each batch is represented by an *axis aligned bounding box* (AABB), which is updated at run time, and has a fixed maximum size. This allows us to explore which granularity offers the best performance for our use case. The hierarchy is queried for each light, producing a list of batch and light index pairs, identifying the batches to be drawn into each shadow map. For each pair, we record the result of culling for each cube face, as this information is needed later when rendering. The result is a bit mask with six bits that we call the *cube-face mask* (CFM), see Figure 5.

4 Algorithm Extensions

4.1 Projection Maps

Efficient culling also ought to avoid drawing geometry into unsampled regions of the shadow map. In other words, we require something that identifies where shadow receivers are located. This is similar in spirit to *projection maps*, which are used to guide photon distribution in photon maps, and we adopt this name.

Fortunately, this is almost exactly the same problem as establishing the needed pages for virtual textures (Section 3.3), and we reuse the method of projecting AABBs onto the cube faces. To represent the shadow receivers, each cube face stores a 32×32 bit mask (in contrast to page masks, which vary with resolution), and we rasterize the cluster bounds into this mask as before.

We then perform the same projection for each batch AABB that was found during the culling, to produce a mask for each shadow caster. If the logical intersection between these two masks is zero for any cube face, we do not need to draw the batch into this cube face. In addition to testing the mask, we also compute the maximum depth for each cube face and compare these to the minimum depth of each batch. This enables discarding shadow casters that lie behind any visible shadow receiver. For each batch, we update the cube-face mask to prune non-shadowing batches.

4.2 Non-uniform Light Sizes

The resolution selection presented in Section 3.2 uses the maximum sample density required by a cluster affected by a light. If the light is large and the view contains samples requiring very different densities, this can be a large over-estimate. This happens when a large light affects both some, relatively few, samples nearby the viewer but also a large portion of the visible scene further away (see Figure 6). The nearby samples dictate the resolution of the shadow map, which then must be used by all samples.

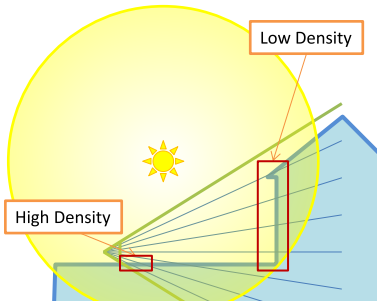


Figure 6: *Illustration of light requiring different sample densities within the view frustum. The nearby, high density, clusters dictate the resolution for the entire light.*

If there are only uniformly sized lights and we are comfortable with clamping the maximum allowed resolution, then this is not a significant problem. However, as our results show, if we have a scene with both large and small lights, then this can come to dominate the memory allocation requirements (e.g.NECROPOLIS, see Figure 12).

To eliminate this issue, we allow each light to allocate a number of shadow maps. We use a fixed number, as this allows fast and simple implementation, in our tests ranging from 1 to 16 shadow maps per light. To allocate the shadow maps, we add a step where we build a histogram over the resolutions requested by the clusters affecting each light. The maximum value within each histogram bucket is then used to allocate a distinct shadow map. When the shadow-map index is established, we replace the light index in the cluster light list with this index. Then, culling and drawing can remain the same, except that we sometimes must take care to separate the light index from the shadow-map index.

4.3 Level of Detail

For high-resolution shadow maps that are used for many view samples, we expect that rasterizing triangles is efficient, producing many samples for each triangle. However, low-resolution shadow maps sample the shadow-casting geometry sparsely, generating few samples per triangle. To maintain efficiency in these cases, some form of *Level of Detail* (LOD) is required.

In the limit, a light might only affect a single visible sample. Thus, it is clear that no amount of polygon-based LOD will suffice by itself. Consequently, we explore the use of ray tracing, which can random access geometry efficiently. To decide when ray tracing should be used, we simply use a threshold (in our tests we used 96 texels as the limit) on the resolution of the shadow map, which is tested after the resolution has been calculated. Those shadow maps that are below the threshold are not further processed and are replaced by directly ray tracing the shadows in a separate shading pass. We refer to this as the hybrid algorithm. Additionally, we evaluate using ray tracing for all shadows to determine the cross-over point in efficiency versus shadow maps.

Since we aim to use the ray tracing for LOD purposes, we chose to use a voxel representation, which has an inherent polygon-agnostic LOD and enables a much smaller memory footprint than would be possible using triangles. We use the technique described by Kämpe et. al. [2013], which offers performance comparable to state of the art polygon ray tracers and a very compact representation.

One difficulty with ray tracing is that building efficient acceleration structures is still a relatively slow process, at best offering interactive performance, and dynamically updating the structure is both costly and complex to implement [Karras and Aila 2013]. We therefore use a static acceleration structure, enabling correct occlusion from the static scene geometry, which often has the highest visual importance. As we aim to use the ray tracing for lights far away (and therefore low resolution), we consider this a practical use case to evaluate. For highly dynamic scenes, our results that use ray tracing are not directly applicable. Nevertheless, by using a high-performance accelerations structure, we aim to explore the upper bound for potential ray tracing performance.

To explore the use of polygon-based LOD, we evaluate a low-polygon version of the HOUSES scene (see Section 6). This is done in lieu of a full blown LOD system to attempt to establish an upper bound for shadow-mapping performance when LOD is used.

4.4 Explicit Cluster Bounds

As clusters are defined by a location in a regular grid within the view frustum, there is an associated bounding volume that is implied by this location. Computing explicit bounds, i.e. tightly fitting the samples within the cluster, was found by Olsson et al. [2012] to improve light-culling efficiency, but it also incurred too much overhead to be worthwhile. When introducing shadows and virtual shadow map allocation, there is more to gain from tighter bounds. We therefore present a novel design that computes approximate explicit bounds with very little overhead on modern GPUs.

We store one additional 32-bit integer for each cluster, which is logically divided into three 10-bit fields. Each of these represent the range of possible positions within the implicit AABB. With this scheme, the explicit bounding box can be established with just a single 32-bit `atomicOr` reduction for each sample. By using the bits to represent a number line, we can only represent as many discrete positions as there are bits. Thus, 10 bits for each axis enables down to a 1000-fold reduction in volume.

To reconstruct the bounding box, we make use of intrinsic bit-wise functions to count zeros from both directions in each 10-bit field. These bit positions are then used to scale and bias the implicit AABB in each axis direction.

5 Implementation

We implemented the algorithm and variants above using OpenGL and CUDA. All computationally intense stages are implemented on the GPU, and in general, we attempt to minimize stalls and GPU to CPU memory transfers. However, draw calls and rendering state

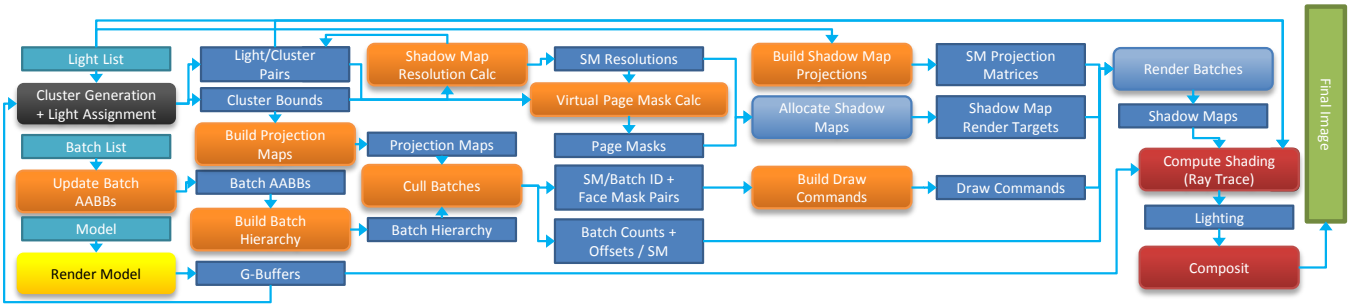


Figure 7: *Stages (rounded) and data (square) in the algorithm implementation. Stage colors correspond to those used in Figure 8. All computationally demanding stages are executed on the GPU, with sequencing and command issue performed by the CPU.*

changes are still necessary to invoke from the CPU, and thus, we must transfer some key information from the GPU. The system is illustrated in Figure 7.

5.1 Shadow Map Resolution Selection

The implementation of shadow-map resolution selection is a set of CUDA kernels, launched with one thread per cluster/light pair. These kernels compute the resolution, cube-face mask, virtual-page mask, and also the projection map, for each shadow map. To reduce the final histograms and bit masks, we use atomic operations, which provide adequate performance for current GPUs. The resulting array of shadow-map resolutions and the array of virtual-page masks are transferred to the CPU using an asynchronous copy.

5.2 Culling Shadow-Casting Geometry

In the implementation, we perform culling before allocating shadow maps, as this allows better asynchronous overlap, reducing stalls, and also minimizes transitions between CUDA and OpenGL operation.

5.2.1 Batch Hierarchy Construction

Each batch is a range of triangle indices and an AABB. A batch is constructed such that all the vertices share the transformation matrix[1] and are located close together, to ensure coherency under animation. At run time, we re-calculate each batch AABB from the vertices every frame to support animation. The resulting list is sorted along the Morton curve, and we then build an implicit left balanced 32-way BVH by recursively grouping 32 consecutive AABBs into a parent node. This is the same type of hierarchy that was used for hierarchical light assignment in clustered shading, and has been shown to perform well for many light sources [Olsson et al. 2012].

The batches are created off-line, using a bottom-up agglomerative tree-construction algorithm over the scene triangles, similar to that described by Walter et. al. [2008]. Unlike them, who use the surface area as the *dissimilarity function*, we use the length of the diagonal of the new cluster, as this produces more localized clusters (by considering all three dimensions). After tree construction, we create the batches by gathering leaves in sub-trees below some predefined size, e.g. 128 triangles (we tested several sizes, as reported below). The batches are stored in a flat array and loaded at run time.

5.2.2 Hierarchy Traversal

To gather the batches for each shadow map, we launch a kernel with a CUDA *block* for each shadow map. The reason for using blocks is that a modern GPU is not fully utilized when launching just a warp per light (as would be natural with our 32-way trees). The block uses a cooperative depth-first stack to utilize all warps within the block. We run this kernel in two passes to first count the number of batches for each shadow map and allocate storage, and then to output the array of batch indices. In between, we also perform a prefix sum to calculate the offsets of the batches belonging to each shadow map in the result array. We also output the cube-face mask for each batch. This mask is the bitwise **and** between the cube-face mask of the shadow map and the batch. The counts and offsets are copied back to the CPU asynchronously at this stage, as they are needed to issue drawing commands.

To further prune the list of batches, we launch another kernel that calculates the projection-map overlap for each batch in the output array and updates the cube-face mask.

The final step in the culling process is to generate a list of draw commands for OpenGL to render. We use the OpenGL 4.3 *multi-draw indirect* feature (`glMultiDrawElementsIndirect`), which allows the construction of draw commands on the GPU. We map a buffer from OpenGL to CUDA and launch a kernel where each thread transforms a batch index and cube-face mask output by the culling into a drawing command. The vertex count and offset is provided by the batch definition, and the instance count is the number of set bits in the cube-face mask.

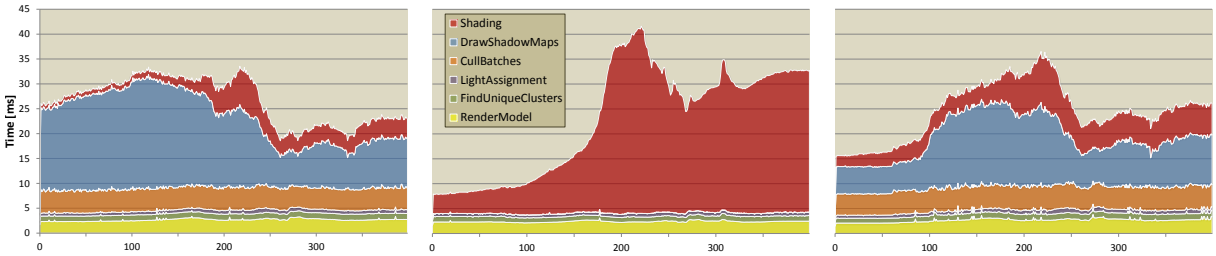
5.3 Shadow Map Allocation

To implement the virtual shadow maps, we make use of the OpenGL 4.4 ARB extension for sparse textures (`ARB_sparse_texture`). The extension enables vendor-specific page sizes which can be queried. Textures with sparse storage must be aligned to page boundaries. On our target hardware, the page size is 256×128 texels for 16-bit depth textures (64kb), which means that our square cube-map faces must be aligned to the larger value. For our implementation, the practical page granularity is therefore 256×256 texels, and this also limits the maximum resolution of our shadow maps to $8\text{K} \times 8\text{K}$ texels, as we use up to 32×32 bits in the virtual-page masks.

Thus, for each non-zero value in the array of shadow map resolutions, we round the requested resolution up to the next page boundary and then use this value to allocate a texture with virtual storage specified. Next, we iterate the virtual-page mask for each face and commit physical pages. If the requested resolution is small, in our implementation below 64×64 texels, we use an ordinary physical cube map instead.

In practice, allocating textures is a slow operation in OpenGL, and we instead pre-allocate a pool of cube textures. We create enough textures of each resolution to match the peak demands of our application. Since the textures are virtual (or small), the memory demands

[1] We only implement support for a single transform per vertex, but this is trivially extended to more general transformations, e.g. skinning.



(a) Shadow Maps (PMCD-EB). (b) Ray Tracing. (c) Hybrid (PMCD-EB).

Figure 8: *Timings from the* NECROPOLIS *scene animation. The performance is broken down into the principal stages of the algorithms. Note that for (b) and (c), the ray tracing time forms part of the shading.*

of this pool is small. At run time, we pick a cube map of the correct resolution from this pool and proceed as before.

5.3.1 Workarounds

Unfortunately, committing physical storage is very slow on current drivers[2]. As a fall back, we therefore implemented an additional pool of physical textures, and pick the next free one of the closest matching resolution. For the physical pool, we cannot allocate all the needed resolutions up-front, as the memory requirements are prohibitive, e.g. a single 8K cube map requires 750Mb of memory (this, in fact, being the raison d'être for the virtual shadow maps). Consequently, this method will suffer from very poor and varying shadow quality but enables us to measure the performance of all the other parts of the algorithm.

On game consoles, where the developers are able to directly manage resources, the straightforward implementation might be expected to work well. Also, extensions such as the explicit page-pool management proposed by AMD (`AMD_texture_tile_pool`) [Sellers et al. 2013] indicate that the page-allocation performance problem is possible to address. For our purposes, going yet further and allowing pages to be managed fully on the GPU, for example using some manner of *indirect* call, similar to that used for draw commands, would seem ideal.

5.4 Rasterizing Shadow Caster Geometry

With the set up work done previously, the actual drawing is straightforward. For each shadow map, we invoke `glMultiDrawElementsIndirect` once, using the count and offset shipped back to the CPU during the culling. To route the batches to the needed cube map faces, we use layered rendering and a geometry shader. The geometry shader uses the instance index and the cube-face mask (which we supply as a per-instance vertex attribute) to compute the correct layer.

The sparse textures, when used as a frame buffer target, quietly drop any fragments that end up in uncommitted areas. This matches our expectations well, as such areas will not be used for shadow look ups. Compared to previous work on software virtual shadow maps, this is an enormous advantage, as we sidestep the issues of fine-grained binning, clipping and copying and also do not have to allocate temporary rendering buffers.

We did not implement support for different materials (e.g. to support alpha masking). To do so, one draw call per shadow material type would be needed instead.

[2]The NVIDIA beta driver version 327.24 was used in our measurements.

6 Results and Discussion

All experiments were conducted on an NVIDIA GTX Titan GPU and an Intel Core i7-3930K CPU. We used three scenes (see Figure 1). HOUSES is designed to be used to illustrate the scaling in a scene where all lights have a similar size and uniform distribution. NECROPOLIS is derived from the Unreal SDK, with some lights moved slightly and all ranges doubled. We added several animated cannons shooting lights across the main central area, and a number of moving objects. The scene contains 275 static lights and peaks at 376 lights. CRYSPONZA is derived from the Crytek version of the Sponza atrium scene, with 65 light sources added. Each scene has a camera animation, which is used in performance graphs (see the supplementary video).

We evaluate several algorithm variants with different modifications: Shadow maps with projection map culling (*PMC*), and with added depth culling (*PMCD*); with or without explicit bounds (*EB*); only using cluster face mask culling (*CFM*); Ray Tracing; and Hybrid, which uses PMCD-EB. Unless otherwise indicated, four cube shadow maps per light is used.

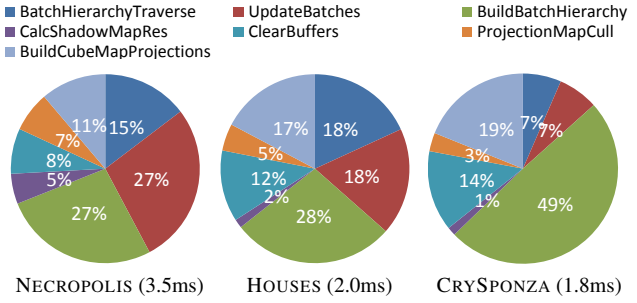
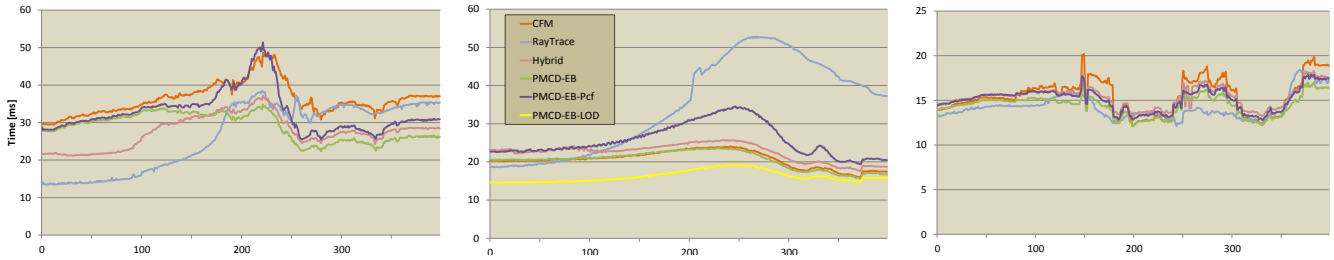


Figure 9: *Timing breakdown of the steps involved in culling batches. The displayed percentage represents the maximum time for each of the steps over the entire animation.*

As noted in Section 5.3.1, current API and driver performance for committing physical memory is very poor. All performance measurements are therefore reported using the fall-back implementation, which uses a pool of physical pre-allocated shadow maps. We performed the same measurements on the full implementation to ensure that they produce representative figures. The pool will run out of high-resolution shadow maps at times, which results in too low sample density and affects the shadow map rendering times. These variations are within 100% of the reported figures and do not affect the peak times reported. It was found that other factors such as re-binding render targets had greater performance impact.



(a) NECROPOLIS. (b) HOUSES. (c) SPONZA.

Figure 10: *Wall-to-wall frame times from the scene animations, for different algorithm variations.*

All reported figures are using a batch size of up to 128 triangles. We evaluated several other batch sizes and found that performance was similar in the range 32 to 512 triangles per batch, but was significantly worse for larger batches. This is expected, as larger batches lead to more triangles being drawn, and rasterization is already a larger cost than culling in the algorithm (see Figure 8(a)).

Performance We report the wall-to-wall frame times for our main algorithm variants in Figure 10. These are the times between consecutive frames and thus include all rendering activity needed to produce each frame. From these results, it is clear that virtual shadow maps with projection-map culling offer robust and scalable performance and that real-time performance with many lights and dynamic scenes is achievable.

As expected, ray tracing offers better scaling when the shadows require fewer samples, with consistently better performance in the first part of the zooming animations in NECROPOLIS and HOUSES (Figure 10). When the lights require more samples, shadow maps generally win, and also provide better quality (as we are ray tracing a fairly coarse voxel representation).

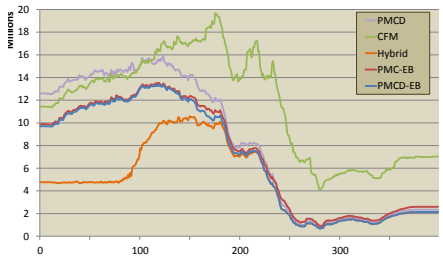


Figure 11: *Triangles drawn each frame in the* NECROPOLIS *animation with different culling methods. The naïve method, that is, not using the information about clusters to improve culling, is not included in the graph to improve presentation. It renders between 40 and 126 million triangles per frame and never less than six times the number of PMCD.*

The hybrid method is able to make use of this advantage and provides substantially better performance early in the NECROPOLIS animation (Figure 8(c)). However, it fails to improve worst-case performance because there are always a few small lights visible, and our implementation runs a separate full-screen pass in CUDA to shade these. Thus, efficiency in these cases is low, and we would likely see better results if the ray tracing better integrated with the other shading. An improved selection criterion, based on the estimated cost of the methods rather than just shadow-map resolution, could also improve performance. For example, the LOD version of the HOUSES scene (Figure 10(b)) highlights that the cost of shadow mapping is correlated to the number of polygons rendered. The LOD version also demonstrates that there exists a potential for performance improvements using traditional polygon LOD, as an alternative or in addition to ray tracing.

Shadow filtering, in our implementation a simple nine-tap *Percentage-Closer filter* (PCF), has a quite high proportion of the total cost, especially in the scenes with relatively many lights affecting each sample (Figure 10). Thus, techniques to reduce this cost, by restricting filtering or using pre-filtering, could be a useful addition.

Culling Efficiency Culling efficiency is greatly improved by our new methods exploiting information about shadow receivers inherent in the cluster, as shown in Figure 11. Compared to naïvely culling using the light sphere and drawing to all six cube faces, our method is at least six times more efficient.

When adding the max depth culling for each cube face, the additional improvement is not as significant. This is not unexpected as the single depth is a very coarse representation, most lights are relatively short range, and the scene is mostly open with little occlusion. Towards the end of the animation, where the camera is inside a building, the proportion that is culled by the depth increases somewhat. The cost of adding this test is very small (see Figure 9: 'ProjectionMapCull').

Memory Usage As expected, using only a single shadow map per light has very high worst case for NECROPOLIS (Figure 12:'PMCD-EB-1SM'). With four shadow maps per light, we get a better correspondence between lighting computations (i.e., the number of light/sample pairs shaded) and number of shadow maps texels allocated. This indicates that peak shadow map usage is correlated to the density of lights in the scene, which is a very useful property when budgeting rendering resources. The largest number of shadow-map texels per lighting computation occurs when shadow maps are low resolution, early in the animation, and does not coincide with peak memory usage. We tested up to 16 shadow maps per light, and above eight, the number of texels rises again.

Explicit bounds The explicit bounds provide improved efficiency for both the number of shadow-map texels allocated and number of triangles drawn by $8 - 35\%$ over the NECROPOLIS animation. The greatest improvement is seen near the start of the animation, where many clusters are far away and thus have large implicit bounds in view space (Figure 11).

Quality As seen in Figure 3, there exist sampling artifacts due to our choice of resolution calculations. However, as we recalculate the required resolutions continuously and select the maximum for each shadow map, we expect these errors to be stable and consistent. In

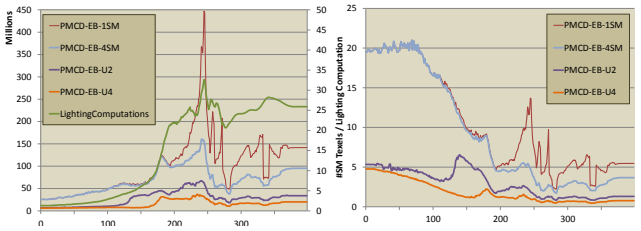


Figure 12: *Allocated shadow-map texels for various scenarios over the* NECROPOLIS *animation. Shows the performance with a varying number of shadow maps per light, the effect of the global undersampling parameter (u2|u4 suffix), and also plots the number of Lighting Computations for each frame (secondary axis).*

the supplementary video, it is difficult to notice any artifacts caused by switching between shadow-map resolutions.

We also added a global parameter controlling undersampling to enable trading visual quality for lower memory usage (see Figure 12). This enables a lower peak memory demand with uniform reduction in quality. For a visual comparison, see the supplementary video.

7 Conclusion

We presented several new ways of exploiting the information inherent in the clusters provided by clustered shading, which enable very efficient and effective culling of shadow casting geometry. With these extensions, we have demonstrated that using hardware-supported virtual cube shadow maps is a viable method for achieving high-quality real-time shadows, scaling to hundreds of lights.

In addition, we show that memory requirements when using virtual cube shadow maps as described in this paper remains proportional to the number of shaded samples. This is again enabled by utilizing clusters to quickly determine both the resolution and coverage of the shadow maps.

We also demonstrate that using ray tracing can be more efficient than shadow maps for shadows with few samples and that a hybrid method building on the strength of both is a promising possibility.

The implementation of `ARB_sparse_texture` used in our evaluation does not offer real-time performance. However, we expect that future revisions, perhaps combined with new extensions, will make this possible. In addition, on platforms with more direct control over resources, such as game consoles, this problem should be greatly mitigated.

8 Future Work

In the future, we would like to explore more aggressive culling schemes, for example using better max-depth culling. We also would like to explore other light distributions, which might be supported by pre-defined masks, yielding high flexibility in distribution.

Acknowledgements

The Geforce GTX Titan used for this research was donated by the NVIDIA Corporation. We also want to acknowledge the anonymous reviewers for their valuable comments, and Jeff Bolz, Piers Daniell and Carsten Roche of NVIDIA for driver support. This research was supported by the Swedish Foundation for Strategic Research under grant RIT10-0033.

References

EISEMANN, E., SCHWARZ, M., ASSARSSON, U., AND WIMMER, M. 2011. *Real-Time Shadows*. A.K. Peters.

FERNANDO, R., FERNANDEZ, S., BALA, K., AND GREENBERG, D. P. 2001. Adaptive shadow maps. In *Proc.*, SIGGRAPH '01, 387–390.

FERRIER, A., AND COFFIN, C. 2011. Deferred shading techniques using frostbite in "battlefield 3" and "need for speed the run". In *Talks*, SIGGRAPH '11, 33:1–33:1.

HARADA, T., MCKEE, J., AND YANG, J. C. 2013. Forward+: A step toward film-style shading in real time. In *GPU Pro 4: Advanced Rendering Techniques*, W. Engel, Ed. 115–134.

HARADA, T. 2012. A 2.5D culling for forward+. In *SIGGRAPH Asia 2012 Technical Briefs*, SA '12, 18:1–18:4.

HOLLANDER, M., RITSCHEL, T., EISEMANN, E., AND BOUBEKEUR, T. 2011. ManyLoDs: parallel many-view level-of-detail selection for real-time global illumination. *Computer Graphics Forum 30*, 4, 1233–1240.

KARRAS, T., AND AILA, T. 2013. Fast parallel construction of high-quality bounding volume hierarchies. In *Proc.*, HPG '13, 89–99.

KELLER, A. 1997. Instant radiosity. In *Proc.*, SIGGRAPH '97, 49–56.

KÄMPE, V., SINTORN, E., AND ASSARSSON, U. 2013. High resolution sparse voxel dags. *ACM Trans. Graph. 32*, 4. SIGGRAPH 2013.

LEFOHN, A. E., SENGUPTA, S., AND OWENS, J. D. 2007. Resolution-matched shadow maps. *ACM Trans. Graph. 26*, 4 (Oct.).

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 *Proc.*, EGGH-HPG'12, 87–96.

PERSSON, E., AND OLSSON, O. 2013. Practical clustered deferred and forward shading. In *Courses: Advances in Real-Time Rendering in Games*, SIGGRAPH '13, 23:1–23:88.

RITSCHEL, T., GROSCH, T., KIM, M. H., SEIDEL, H.-P., DACHSBACHER, C., AND KAUTZ, J. 2008. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph. 27*, 5 (Dec.), 129:1–129:8.

SELLERS, G., OBERT, J., COZZI, P., RING, K., PERSSON, E., DE VAHL, J., AND VAN WAVEREN, J. M. P. 2013. Rendering massive virtual worlds. In *Courses*, SIGGRAPH '13, 23:1–23:88.

SWOBODA, M., 2009. Deferred lighting and post processing on playstation 3. Game Developer Conference.

WALTER, B., BALA, K., KULKARNI, M., AND PINGALI, K. 2008. Fast agglomerative clustering for rendering. In *IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008*, 81–86.