High Performance Graphics (2012), pp. 1–10
C. Dachsbacher, J. Munkberg, and J. Pantaleoni (Editors)

Clustered Deferred and Forward Shading

Ola Olsson, Markus Billeter, and Ulf Assarsson

Chalmers University of Technology

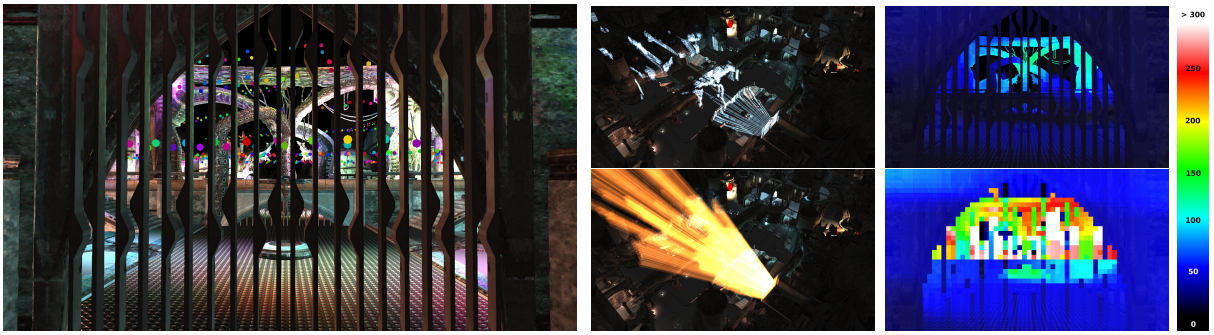


Figure 1: *Clustered Shading groups samples from a view (left image) into clusters (show in blue in the top center image). For shading, each cluster is assigned lights that affect the cluster. Since the clusters are small in comparison to volumes created by e.g. screen space tiling (shown in red in the bottom center image), the number of lighting computations per pixel is kept low (top right image) when compared to Tiled Shading (bottom right image). The colors indicate the number of lighting computations per pixel, ranging from less than* 50 *for blue pixels, to in excess of 300 for white pixels. The scene contains around 2400 light sources, and is rendered in* 17*ms by our method (*2.3*ms for clustering,* 1.5*ms for light assignment and* 5.6 *ms for shading; remaining frame time is dominated by rendering to G-buffers and, here, visualizing light sources with* glutSolidSphere*()), compared to* 26*ms for the Tiled Shading implementation (*1.0*ms for light assignment and* 17.7*ms for shading).*

Abstract

This paper presents and investigates Clustered Shading for deferred and forward rendering. In Clustered Shading, view samples with similar properties (e.g. 3D-position and/or normal) are grouped into clusters. This is comparable to tiled shading, where view samples are grouped into tiles based on 2D-position only. We show that Clustered Shading creates a better mapping of light sources to view samples than tiled shading, resulting in a significant reduction of lighting computations during shading. Additionally, Clustered Shading enables using normal information to perform per-cluster back-face culling of lights, again reducing the number of lighting computations. We also show that Clustered Shading not only outperforms tiled shading in many scenes, but also exhibits better worst case behaviour under tricky conditions (e.g. when looking at high-frequency geometry with large discontinuities in depth). Additionally, Clustered Shading enables real-time scenes with two to three orders of magnitudes more lights than previously feasible (up to around one million light sources).

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

1. Introduction

In recent years, *Tiled Shading* [OA11] in various forms has been gathering increased attention in the games development community. The most popular form is *Tiled Deferred Shading*, which has been implemented on both the Sony PlayStation 3 and Microsoft XBox 360 console, as well

as PC [BE08, And09, Swo09, Lau10, Cof11]. More recently *Tiled Forward Shading*, has also gained attention [McK12].

Tiled deferred shading removes the bandwidth bottleneck from deferred shading, instead making the technique compute bound. This enables efficient usage of devices with a high compute-to-bandwidth ratio, such as modern consoles and GPUs. Modern high-end games are using tiled deferred shading to allow for thousands of lights, which are required to push the limits of visual fidelity [FC11]. With large numbers of lights, GI effects can be produced that affect dynamic as well as static geometry.

Tiled shading groups samples in rectangular screen-space tiles, using the min and max depth within each tile to define sub frustums. Thus, tiles which contain depth values that are close together, e.g. from a single surface, will be represented with small bounding volumes. However, for tiles where one or more depth discontinuities occur, the depth bounds of the tile must encompass all the empty space between the sample groups (illustrated in Figure 1). This reduces light culling efficiency, in the worst case degenerating to a pure 2D test. This results in a strong dependency between view and performance, which highly is undesirable in real-time applications, as it becomes difficult to guarantee consistent rendering performance at all times.

We introduce *Clustered Shading*, where we explore higher dimensional tiles, which we collectively call *clusters*. Each cluster has a fixed maximum 3D extent, which means that there is no degenerate case depending on the view. Each sample can at worst be over-represented by a fixed volume, and empty space is ignored.

We show how clustered shading can be implemented efficiently on the GPU, supporting both deferred and forward shading implementations. Our implementation shows much less view-dependent performance, and is much faster for some cases that are challenging for tiled shading. We also extend beyond 3D clusters and also use normal information. This is used to implement light *back-face culling* on a per-cluster basis, discarding lights that affect no samples. To robustly support large numbers of lights, we also implement a hierarchical light assignment approach, which is shown to enable real-time performance for up to 1M lights.

2. Previous Work

Deferred shading was first introduced in a hardware design in 1988 [DWS*88], with a more general purpose method using full screen *Geometry Buffers* (G-Buffers) following in 1990 [ST90]. Deferred shading decouples geometry and light processing, making it relatively simple to manage large numbers of light sources. It has become mainstream only in recent years, as hardware has become more powerful and raising the bar on visual fidelity requires more and more lights.

Tiled shading is a relatively recent development that builds upon deferred shading. Aimed primarily at addressing the memory bandwidth bottleneck in deferred shading, it has been implemented in many modern computer games. Since game consoles are highly bandwidth constrained devices, tiled deferred shading has quickly become an important algorithm for high-profile games [BE08, And09, Swo09, Lau10, Cof11, McK12]. The trend for computational power to increase faster than memory bandwidth is also present in consumer GPUs. Tiled shading has been shown to scale well with successive GPU generations [OA11].

2.1. Cluster Determination

To enable efficient processing of clusters, we need some way of determining what clusters are present in a given frame. In a deferred shading setting, this requires analysis of the whole frame buffer, which must be done efficiently on the GPU to minimize data transfers and synchronization. Determining a grouping of samples that goes beyond simple 2D tiling is a fairly common problem in GPU rendering algorithms.

Resolution Matched Shadow Maps (RMSM), must determine which shadow pages are used by the view samples [LSO07]. The method achieves this by first exploiting screen space coherency to reduce duplicate requests from adjacent pixels in screen space. Globally unique requests are then determined by sorting and compacting the remaining requests.

Garanzha et al. [GL10] present a similar technique that they call *Compress-Sort-Decompress* (CSD). Their goal is to find 3D (or 5D) clusters in a frame buffer, which are used to form ray packets. The main differences are that Garanzha et al. treat the frame buffer as a 1D sequence and use *run length encoding* (RLE) to reduce duplicates before sorting. They expand the result after the sorting.

The approaches in both RMSMs and CSD rely on the presence of coherency between *adjacent* input elements, in 2D and 1D respectively. In many cases, this is a reasonable assumption. However, techniques such as *multi sampling anti aliasing* (MSAA) with alpha-to-coverage, or stochastic transparency [ESSL10], invalidate this assumption. Coherency is still present in the frame buffer, but not between adjacent samples. For scenes with low coherence between adjacent samples, both of these methods degenerate to sorting the entire frame buffer.

Virtual textures face a very similar problem as RMSMs, having to determine the used pages in a virtual texture. Mayer [May10] surveys several techniques for solving this problem, all of which are very similar to the above methods. Hollemeersch et al. [HPLdW10] describe a different solution, which instead directly sets a flag in the virtual page table, to indicate that a page is needed. Next the page table is compacted, producing the unique pages needed.

Flagging and compacting page tables does not need to use adjacency to reduce work. All samples requiring the same

page will set the same flag, regardless of their position in the frame buffer, eliminating duplicate requests. This method should therefore be more robust with respect to incoherent frame buffers. However, as direct indexing is used, there must be relatively few possible indices (in this case pages).

Liktor and Dashsbacher [LD12] determine and allocate unique shading samples using a related technique. However, because of the high number of unique shading sample identifiers, a direct mapping is not feasible. Also, as they need to allocate space for the samples during the process, they use a more compact hash table to track which samples exist. Space for the samples is allocated in a continuous array, further complicating the process.

3. Clustered Deferred Shading Algorithm

Our algorithm consists of the following basic steps, each of which will be described in more detail in the following sections.

1. Render scene to G-Buffers.
2. Cluster assignment.
3. Find unique clusters.
4. Assign lights to clusters.
5. Shade samples.

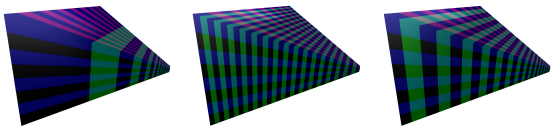
The first step, rendering the model to populate the G-Buffers, does not differ from traditional deferred shading or from tiled deferred shading. The second step computes for each pixel which cluster it belongs to according to its position (and possibly normal). In the third step, we reduce this into a list of unique clusters. The fourth step, assigning lights to clusters, consists of efficiently finding which lights influence which of the unique clusters and produce a list of lights for each cluster. Finally, for each sample, these light lists are accessed to compute the sample's shading.

3.1. Cluster Assignment

The goal of the cluster assignment is to compute an integer *cluster key* for a given view sample from the information available in the G-Buffers. We make use of the position and, optionally, the normal.

There is a potentially limitless number of ways to group view samples. Fundamentally, we desire samples that are close to each other to be grouped together, as they are likely to be affected by the same set of lights. There are many dynamic clustering algorithms available, e.g. k-means clustering, but none of these perform well enough on the millions of samples required to be of interest. Consequently, we employ a regular subdivision, or quantization, of the sample positions, as this is both fast and provides predictable cluster sizes.

The way in which we chose to quantize positions is important in several ways. We desire the clusters to be small, such



(a) Uniform NDC. (b) Uni. view space. (c) Exp. view space.

Figure 2: *Depth subdivision schemes: (a), uniform subdivision of normalized device coordinates; (b), uniform subdivision in view space; and (c), exponential spacing in view space.*

that as few lights as possible affect each, but, conversely, they should contain as many samples as possible to keep the light assignment and shading efficient. We also desire the number of bits required to encode the cluster key to be small and predictable.

A common method is to simply use a world space (virtual) uniform grid [GL10]. This method provides quick cluster key computation, and all clusters are the same size. However, selecting the proper grid cell size requires manual tweaking for each scene, and, depending on scene size, may require a very large number of bits to represent the key. Furthermore, as the grid is viewed under projection, far-away clusters become small on screen. Thus, in large scenes, it is possible to encounter views where many of the clusters are pixel sized, causing poor performance.

We therefore explored an alternative approach, based on the observation that we are only interested in points within the view frustum. Starting with the uniform screen space tiling used in tiled deferred shading, we extend this by also subdividing along the z-axis in view space (or normalized device coordinates), in a manner similar to [HM08]. Viewed in world space, this produces small sub-frustums partitioning the view frustum, see Figure 2.

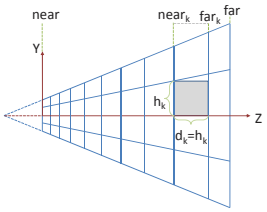


Figure 3: *Exponential spacing in view space. For a given partition k, the near and far planes, as well as cell height and depth are shown.*

The simplest way to perform the z subdivision is to partition the depth range in normalized device coordinates into a set of uniform segments. However, because of the non-linear nature of normalized device coordinates, such a quantization leads to very uneven cluster dimensions. Clusters close to the near plane become very thin, whereas those towards the far

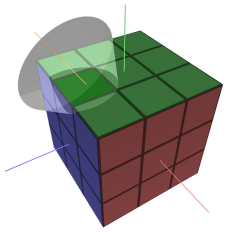


Figure 4: *Quantization of normal directions on the unit cube, using 3×3 subdivisions on each face, and the reconstructed normal cone for one subdivision.*

plane become very long (Figure 2(a)). Uniform subdivision in view space produces the opposite artifact, where clusters near the view point are long and narrow, and those far away are wide and flat (Figure 2(b)).

We therefore choose to perform the subdivision in view space, by spacing the divisions exponentially to achieve self-similar subdivisions [LTYM06], such that the clusters become as cubical as possible (Figures 2(c) and 3).

In Figure 3, we illustrate the subdivisions of a frustum. The number of subdivisions in the Y direction (S_y) is given in screen space (e.g. to form tiles of 32×32 pixels). The near plane for a division k, $near_k$, can be calculated from

$$near_k = near_{k-1} + h_{k-1}.$$

For the first subdivision, $near_0 = near$, i.e. the near viewing plane. For a given field of view of 2θ, we find that

$$h_0 = \frac{2\, near \tan\theta}{S_y}.$$

It follows that $near_k$ can be computed using the following expression:

$$near_k = near \left(1 + \frac{2\tan\theta}{S_y}\right)^k. \tag{1}$$

Solving Equation 1 for k, we find that

$$k = \left\lfloor \frac{\log\left(-z_{vs}/near\right)}{\log\left(1 + \frac{2\tan\theta}{S_y}\right)} \right\rfloor. \tag{2}$$

Using Equation 2, we can now compute the cluster key tuple (i, j, k) from screen-space coordinates (x_{ss}, y_{ss}) and the view-space depth z_{vs}. Coordinates (i, j) are the screen space tile indices, i.e. for tile size (t_x, t_y), $(i, j) = (\lfloor x_{ss}/t_x \rfloor, \lfloor y_{ss}/t_y \rfloor)$.

Using our more dynamic definition of a cluster opens up for the ability to use attributes other than the position to define the cluster key. We extend the cluster key with a number of bits that encode a quantized normal direction (Figure 6). We quantize normals by cube face and a discrete 2D grid over each face, as illustrated in Figure 4. Clustering on normals improves culling of lights (see Section 3.3).

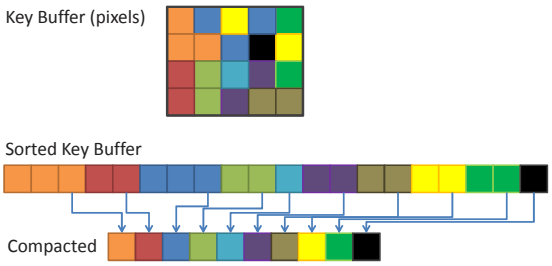


Figure 5: *Sorting and compacting the key buffer to find unique clusters. The cluster keys in the key buffer are sorted and then compacted, to find the list of unique clusters. The sorting is, for instance, based on the view sample's depth and normal direction.*

3.2. Finding Unique Clusters

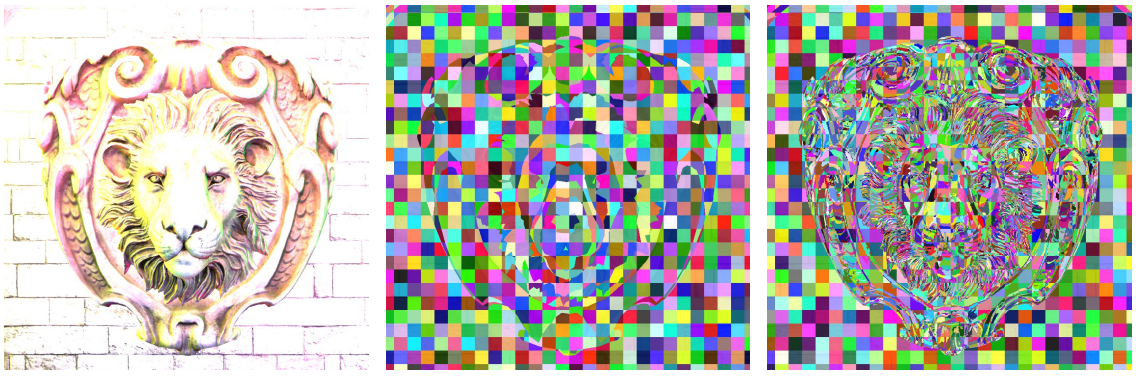
We will here present two different options that we use for identifying unique clusters: with sorting and with page tables.

The perhaps most obvious method to find the unique clusters in parallel is to simply sort the cluster keys, and then perform a compaction step that removes any with an identical neighbour (see Figure 5). Both sorting and compaction are relatively efficient and readily available GPU building blocks [HB10, BOA09]. However, despite steady progress, sorting remains an expensive operation, and we therefore explore better performing alternatives.

As noted in Section 2, methods that rely on *adjacent* screen-space coherency are not robust, especially with respect to stochastic frame buffers. We therefore focus on techniques that do not suffer from this weakness.

3.2.0.1. Local Sorting In our first technique, we sort samples in each screen space tile locally. This allows us to perform the sorting operation in on-chip shared memory, and use local (and therefore smaller) indices to link back to the source pixel. We extract unique clusters from each tile using a parallel compaction. From this, we get the globally unique list of clusters. During the compaction, we also compute and store a link from each sample to its associated cluster.

3.2.0.2. Page Tables The second technique is similar to the page table approach used by virtual textures (Section 2). However, as the range of possible cluster keys is very large, we cannot use a *direct* mapping between cluster key and physical storage location for the cluster data; it simply would typically not fit into GPU memory. Instead we use a *virtual* mapping, and allocate physical pages where any actual keys needs storage. Lefohn et.al. [LSK*06] provide details on software GPU implementation of virtual address translation. We exploit the fact that all physical pages are allocated in a compact range, and we can therefore compact that range to find the unique clusters.



(a) Reference view (b) Clustering based on 3D position only (c) Using 3D position and sample normal

Figure 6: *Results of different clustering methods. (a) The rendered and lit reference view is shown to the right. (b) The center image shows the results of clustering on position only. (Each cluster is assigned a random color.) (c) Clustering based on position and normals is shown to the right . In both cases, flat regions produce a clustering very similar to screen space tiling.*

Whether using sorting or page tables, the cluster key defines implicit 3D bounds and, optionally, an implicit normal cone. However, as the actual view-sample positions and normals typically have tighter bounds, we also evaluate explicit 3D bounds and normal cones. We compute the explicit bounds by performing a reduction over the samples in each cluster (e.g., we perform a min-max reduction to find the AABB enclosing each cluster). The results of the reduction are stored separately in memory.

When using page tables, the reduction is difficult to implement efficiently. Because of the many-to-one mapping from view samples to cluster data, we would need to make use of atomic operations, and get a high rate of collisions. We deemed this to be impractically expensive. We therefore only implement explicit bounds for the first technique based on sorting (after the local sort, information about which samples belong to a given cluster is readily available).

3.3. Light Assignment

The goal of the light assignment stage is to calculate the list of lights influencing each cluster. Previous designs for tiled deferred shading implementations have by and large utilized a brute force approach to finding the intersection between lights and tiles. That is, light-cluster overlaps were found by, for each tile, iterating over all lights in the scene and testing bounding volumes. This is tolerable for reasonably low numbers of lights and clusters.

To robustly support large numbers of lights and a dynamically varying number of clusters, we use a fully hierarchical approach based on a spatial tree over the lights. Each frame, we construct a *bounding volume hierarchy* (BVH) by first sorting the lights according to the Z-order (Morton Code) based on the discretized centre position of each light. We derive the discretization from a dynamically computed bounding volume around all lights.

The leaves of the search tree we get directly from the sorted data. Next, 32 consecutive leaves are grouped into a bounding volume (AABB) to form the first level above the leaves. The next level is constructed by again combining 32 consecutive elements. We continue until a single root element remains.

For each cluster, we traverse this BVH using depth-first traversal. At each level, the bounding box of the cluster (either explicitly computed from the cluster's contents or implicitly derived from the cluster's key) is tested against the bounding volumes of the child nodes. For the leaf nodes, the sphere bounding the light source is used; other nodes store an AABB enclosing the node. The branching factor of 32 allows efficient SIMD-traversal on the GPU and keeps the search tree relatively shallow (up to 5 levels), which is used to avoid expensive recursion (the branching factor should be adjusted depending on the GPU used, the factor of 32 is convenient on current NVIDIA GPUs).

If a normal cone is available for a cluster, we use this cone to further reject lights that will not affect any samples in the cluster. This happens if ω, the angle between the incoming light direction from the centre of the cluster AABB (d_i) and the normal cone axis (a), is greater than $\pi/2 + \alpha + \delta$. The angle α is the normal-cone half angle, and δ is the half angle of the cone from the light enclosing the cluster AABB (see Figure 7).

3.4. Shading

Shading differs from Tiled Shading only in how we look up the cluster for the view sample in question. For Tiled Shading, a simple 2D lookup, based on the screen-space coordi-

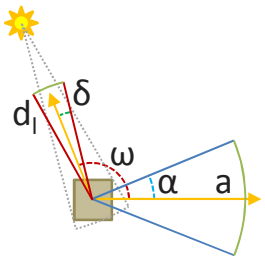


Figure 7: *Back-face culling of lights against clusters. A normal cone (blue), with the opening angle α is derived from or stored with the cluster. The normals of the samples contained in this cluster are all within this cone. The cone originating at the light source and enclosing the cluster (dashed grey – geometrically equivalent to the red cone) gives the angle δ. If the angle between the incoming light and the axis of the normal cone (ω) is greater than $\pi/2+\alpha+\delta$, the light faces the back of all samples in the cluster, and can therefore be ignored.*

nates, is sufficient to retrieve light-list offset and count. However, for clustered approaches, there no longer exists a direct mapping between the cluster key and the index into the list of unique clusters.

In the sorting approach, we explicitly store this index for each pixel. This is achieved by tracking references back to the originating pixel, and, when the unique cluster list is established, storing the index to the correct pixel in a full screen buffer.

When using page tables, after the unique clusters are found, we store the cluster index back to the physical memory location used to store the cluster key earlier (using the same page table as before). This means that a virtual lookup for the cluster key will yield the cluster index. Thus, each sample can look up the cluster index using the cluster key computed earlier (or re-computed).

4. Implementation and Evaluation

We implemented several variants of the new algorithm using OpenGL and CUDA. The variants are as follows (suffixes used are documented in Table 1):

- *ClusteredDeferred[Nk][En][Eb][Pt]* – clustered deferred shading.
- *ClusteredForward* – clustered forward shading. Clustered forward shading requires a pre-z pass to prime the depth buffer, which is used for clustering. Currently only implemented with page tables.

Additionally, we implemented the following methods for comparison, as described in [OA11]:

- *Deferred*, traditional deferred shading, with stencil optimization. This means that light assignment will be exact per sample using a stencil test [AA03].
- *TiledDeferred*, standard tiled deferred shading.
- *TiledDeferredEn*, tiled deferred shading with explicit normal cones computed per tile.
- *TiledForward*, standard tiled forward shading, with a depth pre-pass, to enable min-max culling of lights.

Table 1: *Suffixes identifying variations of the clustered methods.*

Suffix	*Meaning*
Nk[X]	Clustering based on normal, using $X \times X$ subdivisions to a cube face.
En	Explicit normals cones are derived and used.
Eb	Explicit Bounds (3D AABB) are derived and used.
Pt	Page Tables are used to find the unique clusters

4.1. Cluster Key Packing

For maximum performance when using sorting or page tables, we wish to pack the cluster key into as few bits as possible. We allocate 8 bits to each i and j components, which identify the screen-space tile the cluster belongs to. This allows up to 8192×8192 size render targets (assuming screen-space tile size of 32×32 pixels). The depth index k is determined from settings for the near and far planes and Equation 2. In our scenes, we found 10 bits to be sufficient. This leaves up to 6 bits for the optional normal clustering. Using 6 bits, we can for instance support a resolution up to 3×3 subdivisions on each cube face ($3 \times 3 \times 6 = 54$ and $\lceil \log_2 54 \rceil = 6$). For more restricted environments, the data could be packed more aggressively, saving both time and space.

4.2. Tile Sorting

To the cluster key (between 10 and 16 bits wide) we attach an additional 10 bits of meta-data, which identifies the sample's original position relative to its tile. We then perform a tile-local sort of the cluster keys and the associated meta-data. The sort only considers the up-to 16 bits of the cluster key; the meta-data is used as a link back to the original sample after sorting. In each tile, we count the number of unique cluster keys. Using a prefix operation over the counts from each tile, we find the total number of unique cluster keys and assign each cluster a unique ID in the range $[0...\text{numClusters})$. We write the unique ID back to each pixel that is a member of the cluster. The unique ID also serves as an offset in memory to where the cluster's data is stored.

Bounding volumes (AABB and normal cone) can be reconstructed from the cluster keys, in which case each cluster

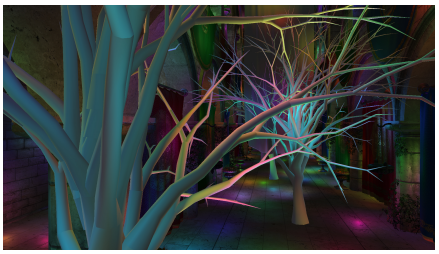


Figure 8: *A view of the Crytek Sponza scene, with 10k lights randomly placed. The tree branches cause discontinuities in the depth buffer, making it more challenging for tiled deferred shading.*

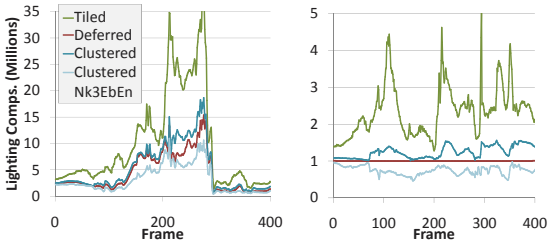


Figure 9: *(left) Millions of lighting computations performed along a fly-through of the Necropolis scene. (right) Same data, normalized to the Deferred method.*

only needs to store its cluster key. For explicit bounding volumes, we additionally store the AABB and/or normal cone. The explicit bounding volumes are computed using a reduction operation: for instance, AABBs can be found using a min- and a max-reduction operation on the sample positions. The meta-data from the locally sorted cluster keys gives us information on which samples belong to a given cluster.

4.3. Page Tables

We implemented a single level page table using a two pass approach. First the required pages are flagged in the table. Then, the physical pages are allocated using a parallel prefix sum, and finally the keys are stored into the physical pages. Performing the physical page allocation on the fly in a single pass was more than 2 times slower, but could still be viable on hardware with faster atomic operations.

4.4. Light Assignment

As described in Section 3.3, we construct a search tree over the lights each frame. Construction relies on efficient sorting functions; here we use the sorting function provided by Thrust [HB10]. To construct the upper levels of the tree, we launch a CUDA warp (32 threads) for each node to be constructed. The warp performs an in-warp parallel reduction over the children's bounding volumes.

For traversal, we again take advantage of the 32-wide fanout of the search tree. For each cluster we allocated a warp that traverses the tree in depth-first order. Each thread in the warp tests the 32 bounding volumes of the children in parallel. By providing unrolled implementations for trees of depth up to 5, we can avoid expensive recursion in CUDA. With a depth of 5, we can support up to 32 million lights, which we deemed to be sufficient (it is trivial to expand this).

5. Results and Discussion

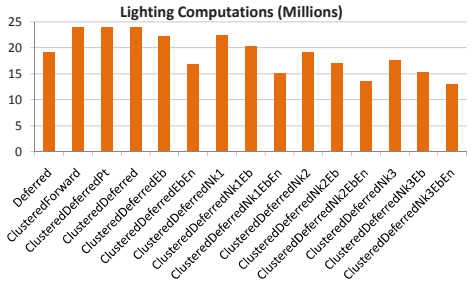
We measured performance for the algorithm and variants described in the previous section, and measurements are performed on an NVIDIA GTX 480 GPU, unless otherwise indicated. We used the set of scenes listed below.

- *Necropolis*. Scene from the Unreal Development Kit [Epi11] (Figure 1). The scene contains 653 lights, with bounded ranges. The majority are spot lights. However we treat all lights as point lights (this is a limitations in our implementation). The scene contains around 2M triangles and is normal mapped. We created a camera animation covering the length of the map (see supplemental video). To bring the number of lights up further, we added several cannon towers to the scene which shoot out colourful spheres, bringing the total number of lights up to around 2500 during the animation.
- *Sponza*. We used the version of sponza made available by Crytek [Cry10] (Figure 8). To make the scene more challenging, with more discontinuities, we injected a set of bare trees. We generated 10k random lights within the scene AABB.

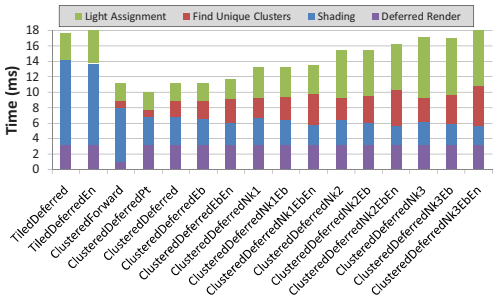
5.1. Performance Analysis

The main advantage of clustered shading over tiled shading is the reduced view dependence. By avoiding empty space, efficiency should be similar to that of deferred shading with stencil optimization and less variable than tiled shading. This is shown in Figures 9 and 10(a), which both adopt the lighting computations metric from [OA11].

Since clustering and light assignment introduce overheads, it is expected that tiled shading performs better when there are fewer lights, or few discontinuities. Clustered shading is still expected to have less view-dependent variability in frame times. Figure 11 confirms that this is the case for the necropolis scene, which has relatively few discontinuities and lights. Even the most complex clustered algorithm tested (ClusteredNk3EbEn), offers worst case performance comparable to tiled deferred. This is also the case for the more challenging scene shown in Figure 10(b), with many discontinuities and lights, indicating greater robustness for clustered shading. We also see that the best performing clustered variant (ClusteredDeferredPt) is around 50% faster in the worst case on the necropolis animation.



(a) Efficiency, millions of lighting computations.



(b) Performance, milliseconds for important stages.

Figure 10: *Performance measured for the tested algorithms for the view of the crytek sponza scene shown in Figure 8. Tiled variants have been excluded from (a), as they make comparison difficult. They perform around 90 million lighting computations. For the same reason, Deferred and TiledForward have been excluded from (b). Deferred takes a total of* 97.1 *ms, and TiledForward* 23.6 *ms to render.*

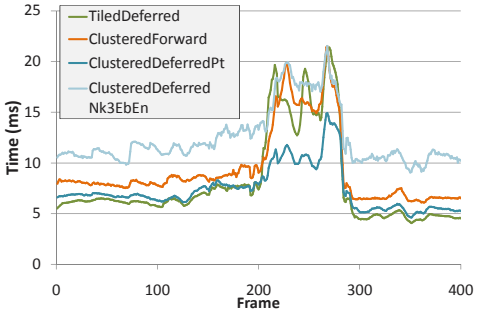


Figure 11: *Run time performance of some the algorithm variants over the Necropolis scene animation.*

ClusteredForward offers very competitive performance, similar to TiledDeferred in the Necropolis animation sequence (Figure 11). This is interesting as, by using forward shading, this variant inherently support MSAA, custom material shaders, and sidesteps the issue of G-Buffer storage. This is remarkable since TiledForward performs significantly worse than TiledDeferred (which is why TiledForward was excluded from Figure 11).

Table 2: *Light assignment performance scaling with an increasing number of randomly distributed lights.*

#lights	Clustered Light Assignment Time	Tiled Light Assignment Time
32	0.71 ms	0.24 ms
1024	0.73 ms	0.51 ms
32768	1.42 ms	9.31 ms
1048576	5.73 ms	341.56 ms

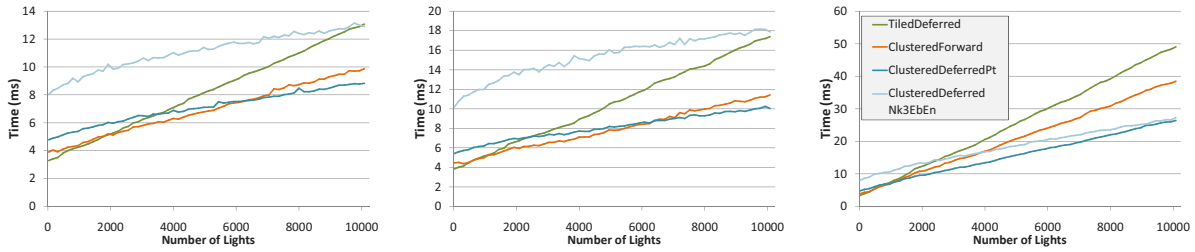
Run-time performance is influenced by many factors, including the number of lights, light density, the level of discontinuities, algorithm complexity, and various implementation details. In Figure 12, we explore the first three of these options. While the crossover point between tiled and clustered implementations is at most around $2k$ lights, the most important conclusion is that clustered shading is very competitive even for cases with very few lights.

Using normal cones and explicit bounds improves efficiency and *shading time* in all methods tested (Figures 9 and 10). However, as other stages become slower, this does not translate into faster rendering overall. Even the relatively modest overhead of adding normal cone construction to tiled deferred (TiledDeferredEn) is too large to offer any net benefit. This affirms that the major performance gain comes from the move beyond 2D tiles. To make these more advanced clusterings attractive, either faster methods for light assignment and clustering must be found, or the shading cost must increase.

As our clustered shading implementation uses a light hierarchy for light assignment, it should scale well with increasing numbers of lights. Table 2 shows this, where we compare the hierarchical light assignment against the brute-force approach used by the tiled implementation. For small numbers of lights, various overheads dominate the assignment time, making the clustered variant slightly more expensive. At 1M lights, our clustered-shading implementation runs at over 35 fps, where the lights are uniformly distributed and up to 100 lights ($\sim$ 45 on average) end up influencing each cluster.

6. Conclusion and Future Work

In this paper, we have presented and evaluated *Clustered Shading*. In clustered shading, we group similar view samples according to their position and, optionally, normal into clusters. We then determine what light sources potentially influence what clusters. Compared to tiled shading, clusters generally are smaller, and therefore will be affected by fewer light sources. The optional per-cluster normal-information allows us to cull back-facing light sources against clustering, further reducing the number of light sources affecting each



(a) Sponza, no trees. (b) Sponza, with trees. (c) Sponza, no trees, 2x light radius.

Figure 12: *Crossover points for various algorithms and numbers of lights for the view of Sponza seen in Figure 8. Note that (a) and (c) use the same view, but without trees, and therefore contain fewer discontinuities.*

cluster. We have shown that efficiency is indeed superior, and that performance is more robust with respect to changing viewing conditions. Our implementation shows that both clustered deferred and forward shading offer real-time performance and can scale up to 1M lights. In addition, overhead for the clustering is low, making it competitive even for few lights.

In the future, we would like to explore approximative lighting, where a heuristic is used to determine if all view samples in a cluster are affected approximately equally by a certain light. If so, the lighting for that light source is evaluated once and re-used for all samples in the cluster. In some initial tests, we have observed an up to around 20% reduction in lighting computations, at very little computational cost. (However, this produced some subtle visual discrepancies, which we have been unable to work around at this point.)

We believe that it is possible to produce high quality approximations. These approximations may require additional per-cluster data, such as average shininess for specular computations. A better heuristic for determining when approximation is possible would also have to be developed.

It would also be interesting to investigate how clustered shading interacts with more complex shading, e.g. switching due to type of material. Since clustered shading has a much smaller shading cost than tiled shading, we expect better scaling with shader complexity.

References

[AA03] ARVO J., AILA T.: Optimized shadow mapping using the stencil buffer. *journal of graphics, gpu, and game tools 8*, 3 (2003), 23–32. 6

[And09] ANDERSSON J.: Parallel graphics in frostbite - current & future. SIGGRAPH Course: Beyond Programmable Shading, 2009. URL: http://s09.idav.ucdavis.edu/talks/04-JAndersson-ParallelFrostbite-Siggraph09.pdf. 2

[BE08] BALESTRA C., ENGSTAD P.-K.: The technology of uncharted: Drake's fortune. Game Developer Conference, 2008. URL: http://www.naughtydog.com/docs/Naughty-Dog-GDC08-UNCHARTED-Tech.pdf. 2

[BOA09] BILLETER M., OLSSON O., ASSARSSON U.: Efficient stream compaction on wide simd many-core architectures. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 159–166. doi:http://doi.acm.org/10.1145/1572769.1572795. 4

[Cof11] COFFIN C.: Spu-based deferred shading in battlefield 3 for playstation 3. GDC 2011, 2011. URL: http://www.slideshare.net/DICEStudio/spubased-deferred-shading-in-battlefield.-3-for-playstation-3. 2

[Cry10] Cryengine3 | crytek | sponza model, 2010. URL: http://www.crytek.com/cryengine/cryengine3/downloads. 7

[DWS*88] DEERING M., WINNER S., SCHEDIWY B., DUFFY C., HUNT N.: The triangle processor and normal vector shader: a vlsi system for high performance graphics. *SIGGRAPH Comput. Graph. 22*, 4 (1988), 21–30. doi:http://doi.acm.org/10.1145/378456.378468. 2

[Epi11] EPIC GAMES: Unreal development kit, 2011. URL: http://www.udk.com/. 7

[ESSL10] ENDERTON E., SINTORN E., SHIRLEY P., LUEBKE D.: Stochastic transparency. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), ACM, pp. 157–164. doi:http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1730804.1730830. 2

[FC11] FERRIER A., COFFIN C.: Deferred shading techniques using frostbite in "battlefield 3" and "need for speed the run". In *ACM SIGGRAPH 2011 Talks* (New York, NY, USA, 2011), SIGGRAPH '11, ACM, pp. 33:1–33:1. doi:10.1145/2037826.2037869. 2

[GL10] GARANZHA K., LOOP C.: Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Computer Graphics Forum 29*, 2 (2010), 289–298. doi:10.1111/j.1467-8659.2009.01598.x. 2, 3

[HB10] HOBEROCK J., BELL N.: Thrust: A parallel template library, 2010. Version 1.3.0. URL: http://www.meganewtons.com/. 4, 7

[HM08] HUNT W., MARK W. R.: Ray-specialized acceleration structures for ray tracing. In *IEEE/EG Symposium on Interactive Ray Tracing 2008* (Aug 2008), IEEE/EG, pp. 3–10. 3

[HPLdW10] HOLLEMEERSCH C.-F., PIETERS B., LAMBERT P., DE WALLE R. V.: Accelerating virtual texturing using cuda. In *GPU Pro*, Engel W., (Ed.). A K Peters, 2010, pp. 623–642. 2

[Lau10] LAURITZEN A.: Deferred rendering for current and future rendering pipelines. SIGGRAPH Course: Beyond Programmable Shading, 2010. URL: http://bps10.idav.ucdavis.edu/talks/12-lauritzen_DeferredShading_BPS_SIGGRAPH2010.pdf. 2

[LD12] LIKTOR G., DACHSBACHER C.: Decoupled deferred shading for hardware rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2012), I3D '12, ACM, pp. 143–150. doi:10.1145/2159616.2159640. 3

[LSK*06] LEFOHN A. E., SENGUPTA S., KNISS J., STRZODKA R., OWENS J. D.: Glift: Generic, efficient, random-access gpu data structures. *ACM Trans. Graph. 25*, 1 (Jan. 2006), 60–99. URL: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1122501.1122505, doi:10.1145/1122501.1122505. 4

[LSO07] LEFOHN A. E., SENGUPTA S., OWENS J. D.: Resolution-matched shadow maps. *ACM Trans. Graph. 26*, 4 (2007), 20. doi:http://doi.acm.org/10.1145/1289603.1289611. 2

[LTYM06] LLOYD D. B., TUFT D., YOON S.-E., MANOCHA D.: Warping and partitioning for low error shadow maps. In *Proceedings of the Eurographics Workshop/Symposium on Rendering, EGSR* (June 2006), Eurographics Association, pp. 215–226. 4

[May10] MAYER A. J.: *Virtual Texturing*. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Oct. 2010. URL: http://www.cg.tuwien.ac.at/research/publications/2010/Mayer-2010-VT/. 2

[McK12] MCKEE J.: Technology behind amd's "leo demo". Game Developers Conference, 2012. URL: http://developer.amd.com/gpu_assets/AMD_Demos_LeoDemoGDC2012.ppsx. 2

[OA11] OLSSON O., ASSARSSON U.: Tiled shading. *Journal of Graphics, GPU, and Game Tools 15*, 4 (2011), 235–251. doi:10.1080/2151237X.2011.621761. 1, 2, 6, 7

[ST90] SAITO T., TAKAHASHI T.: Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph. 24*, 4 (1990), 197–206. doi:http://doi.acm.org/10.1145/97880.97901. 2

[Swo09] SWOBODA M.: Deferred lighting and post processing on playstation 3. Game Developer Conference, 2009. URL: http://www.technology.scee.net/files/presentations/gdc2009/DeferredLightingandPostProcessingonPS3.ppt. 2