Tiled Light Trees

Yuriy O'Donnell*, Electronic Arts
Matthäus G. Chajdas†, AMD

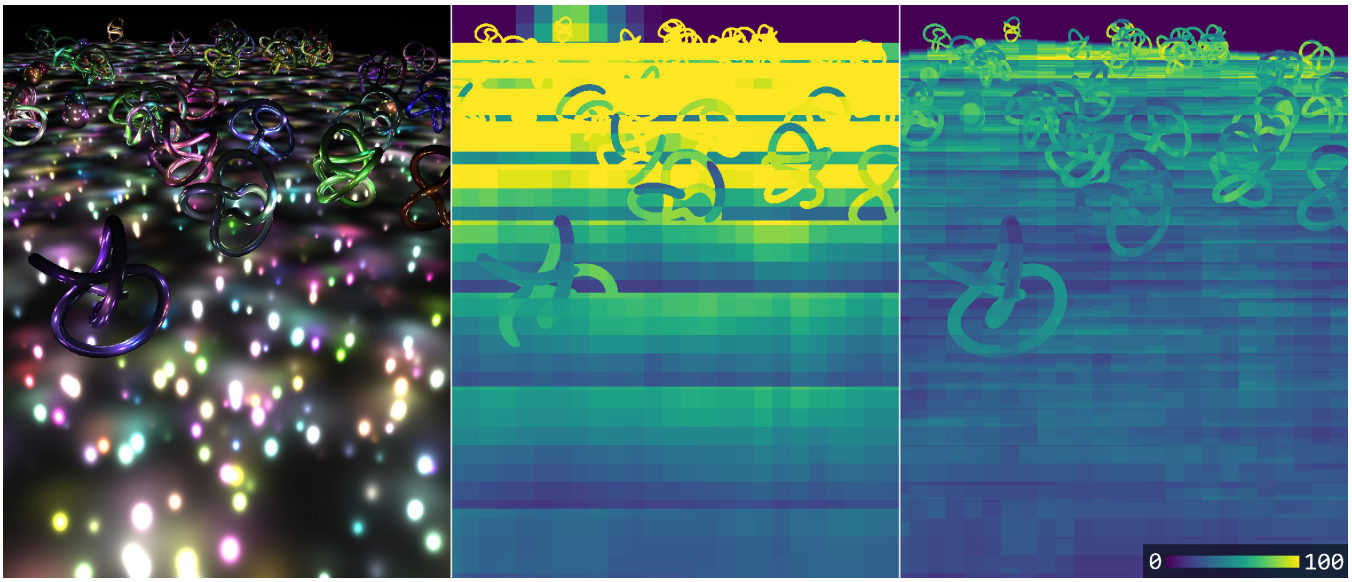


Figure 1: *We present a new algorithm to handle many light sources in real-time applications. Our algorithm is built upon "light trees", which allow the GPU to efficiently handle a wide range of different light distributions. From left to right, we show the shaded scenes, the number of light sources evaluated per tile using "practical clustered shading" and using our algorithm on the right. Compared to the current state of the art, our algorithm significantly reduces the number of light sources that have to be processed, in particular in distant areas of the scene.*

Abstract

Handling many light sources in real-time is still one of the big challenges in real-time graphics [Sousa and Geffroy 2016; Garawany 2016]. Even the most recent approaches like practical clustered shading still have various problem cases with low performance. Especially in scenes with high depth variance, existing algorithms cannot adapt to the distribution of light sources properly and end up evaluating many lights that don't contribute to the final image.

We present a new approach, "tiled light trees" – a hierarchical acceleration structure that adapts to the light source distribution. Our approach improves on the worst case performance of existing solutions. Due to traversal overhead, the proposed algorithm can be sometimes slower than clustered shading. To handle those situations optimally, we propose a hybrid approach which combines the strengths of light trees with clustered shading, outperforming any individual solution in nearly every case. Our new hybrid algorithm is easy to implement and suitable for usage in real-time applications such as games.

Keywords: lighting, culling, real-time

Concepts: • **Computing methodologies** → **Rasterization;**

*e-mail:yuriy.odonnell@ea.com
†e-mail:Matthaeus.Chajdas@amd.com

1 Introduction

Tiled shading, and more modern variants of it like tiled deferred shading, solve the problem of lighting a scene using many light sources by binning lights in screen-space. This drastically reduces the number of light sources that need to be processed for each tile, allowing games to use hundreds to thousands of light sources efficiently.

The general approach for tiled shading algorithms is to split the screen into rectangular tiles and intersect each light source with each tile's frustum. This uses the screen-space tile extents, as well as the min/max-depth of the tile to create a frustum against which all lights are tested [Andersson 2009]. Such algorithms share the problem that due to their 2D only nature, the per-tile frusta have generally a very large depth range and may intersect many light sources which do

I3D '17, February 25 - 27, 2017, San Francisco, CA, USA
ISBN: 978-1-4503-4886-7/17/03$15.00
DOI: http://dx.doi.org/10.1145/3023368.3023376

not affect any geometry within the tile. This led to the introduction of 2.5D algorithms which try to reduce the number of false positives. The key idea is to subdivide not only in screen-space, but also along the view direction. Typically a frustum-aligned grid is used and the lights are sorted into it, reducing the number of light/pixel tests due to the tighter bounds [Persson 2013]. The data structures used are highly uniform to allow for efficient assignment.

Our contribution is a novel approach which introduces "light trees" as a new spatial acceleration structure for light culling. Our approach is especially suited for handling many non-overlapping light sources, which are challenging for current algorithms. Similar to previous techniques, our light trees are built on the CPU and don't require CPU/GPU synchronization as they are geometry independent. We also present a hybrid approach which uses a 2.5D grid where each cell contains either a list or a tree of lights. It combines the strength of light tree structure with low-overhead list traversal for grid cells in which most light sources cover the entire cell. With this combined algorithm, we are able to improve the robustness of 2.5D lighting algorithms while maintaining all the properties which make them popular.

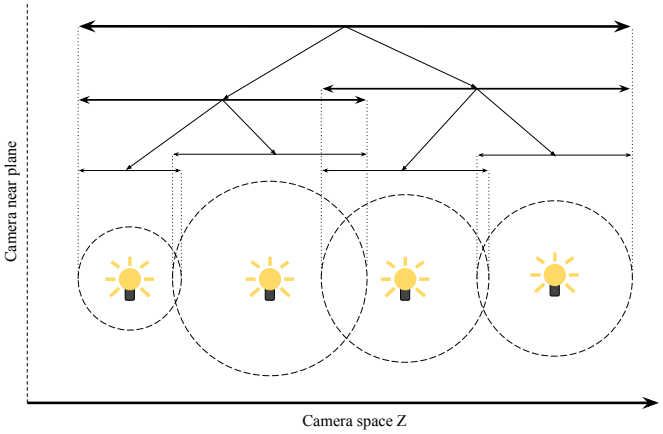


Figure 2: *Bounding interval hierarchy over light sources. The light sources are already sorted by their center along the z direction, so each tree node only needs to store the index of the first light and the number of light sources.*

2 Previous work

Binning lights in screen space has been popularized by the work of [Andersson 2009; Ferrier and Coffin 2011; Harada et al. 2012], which was the first to introduce *tiled lighting*. In tiled lighting, the screen space is subdivided into tiles. In the first step, lights are culled against each tile in parallel, and finally, each light intersecting the tile is evaluated. Since then, many game engines have used a variation of tile-based deferred lighting. [Harada 2012] introduced a 2.5D algorithm which splits the depth range and assigns lights to each cell. The main advantage of this algorithm is that lights which do not intersect the scene geometry can be quickly skipped, as each shaded pixel only processes lights intersecting its grid cell. This algorithm has been used in multiple game titles with various tweaks and improvements. In general, the main difference between the individual variants lies in the way the depth is subdivided as well as in the light/tile intersection test. In all cases, the goal is to reduce the amount of falsely-tested lights.

A recent advancement in this area is *clustered* shading [Olsson et al. 2012]. Clustered shading reorders the individual shading points to improve the light culling efficiency. By working on clusters of geometry samples, it can perform back-face culling against individual light sources. It uses a global BVH built over all light sources in the scene. For each cluster, the tree needs to be traversed. While similar to our work, we do not require a global data structure for the lights, and use a two-level hierarchy instead. This improves performance as our trees are much smaller, resulting in a more optimal traversal.

A production variant of clustered shading has been presented by [Persson 2013]. Compared to the original clustered shading, the new algorithm is simpler and only clusters the lights themselves, not the geometry. The lights are assigned to a 2.5D grid in camera space, which is subdivided along the z axis in exponentially larger steps. This keeps the size of each grid cell roughly constant in screen space and prevents far-away cells from becoming pixel-sized due to projection. The latest update to this technique includes more efficient light assignment on GPU using conservative rasterization [Örtegren 2015], which could be also used with our approach.

3 Our approach

We will first introduce "light trees", as this is the core idea behind our algorithm. Then we will describe a hybrid approach which combines it with clustered shading.

In all cases, our technique uses both the CPU and the GPU in each frame. Light sources are always pre-processed on the CPU, then uploaded to the GPU and finally used for lighting. On the CPU side, we perform the following steps: First, we frustum cull light sources, then we sort them by distance to the camera plane. Next, we bin the light sources into screen-space tiles and finally we build light-trees per screen-space tile.

On the GPU, we identify the light tree covering the current pixel and traverse it during the lighting pass. Notice that our technique only requires information about the camera and light sources, but not scene geometry. It can therefore be used in both forward and deferred shading contexts. Another advantage of not needing the geometry data is that light trees can be prepared in parallel with rendering the g-buffer or shadow maps.

The key idea of our algorithm is to use a *light tree* for each 2D screen tile. The light trees can perfectly adapt to the distribution of light sources in depth as they are not limited to a fixed grid along the z axis. Due to the local nature of the trees – each tile has its own unique tree – we can also generate very shallow trees with high coherency instead of a global tree for all light sources [Olsson et al. 2012]. Small, shallow trees are very suitable for GPUs which rely on very wide SIMD units for execution and require optimizations for both coherency as well as memory access to achieve good performance. In the following section, we'll describe the data structure used for the light trees.

3.1 Light tree structure

We build a simplified 1-D bounding interval tree [Cormen et al. 2009] over the light sources using their depth extents

in camera space as can be seen in figure 2. This assumes that every light source has a finite influence radius. The structure is a binary tree where every node contains a list of lights and their aggregate depth extents. We always generate full binary trees, rounding up the number of leaf nodes to the next power of two. Using full binary trees allows us to simplify the traversal and make it more coherent.

As mentioned before, execution coherency is not sufficient to ensure good GPU performance – we also need efficient memory access patterns. To this end, the trees are laid out in memory using depth-first order. This allows fast traversal on the GPU using a stack-less algorithm and improves cache efficiency.

The traversal is strictly in the same order as the tree is stored. That is, individual threads in a group only jump forward in the tree. In case all nodes have to be visited, memory will be traversed linearly, resulting in the optimal memory access pattern. If the reads are sparse – indicating very high coherency on the tree traversal – the execution will skip over consecutive nodes, but never jump back in the tree. Thus, the traversal is always efficient as tree nodes loaded as part of a cache line are either fully processed, or skipped.

3.2 Tree construction

Before the tree construction starts, we first sort the global list of light sources by their distance to the camera plane, using a parallel radix sort. This is important as it allows us to assign lights to tiles in the order required for efficient tree construction, avoiding a per-tile sorting step. We found global sorting to be cheaper than per-tile sorting when there are many lights that overlap multiple cells. Once we have the list of light sources for an individual tile, we process each tile independently and build binary trees.

We use a bottom-up tree construction algorithm, which builds a binary tree with breadth-first node memory layout. It takes advantage of the fact that all light sources are already sorted by their distance to the camera plane. Building each level of the tree is done by iterating over the previous level and combining every consecutive pair of nodes. The whole tree is constructed in $\mathcal{O}(n)$ time.

Listing 1: *The tree nodes as used on the GPU.*

```
struct Node
{
  float minDepth;
  float maxDepth;
  unsigned firstLightIndex;
  unsigned isLeaf : 1;
  unsigned lightCount : 16;
  unsigned skipCount : 15;
};
```

For the GPU, we want a depth-first tree as we want to have consecutive nodes stored next to each other in memory. We thus need to convert the tree from breadth-first order into depth-first order. As the number of leaf nodes is always a power-of-two, and generally small, we precomputed a few conversion tables so we can translate a breadth-first tree into a depth-first tree with a single pass over the data.

The alternative approach is to simply build the tree top-down. This requires a more expensive building algorithm which needs to compute the bounds of each node at every level. The total build requires $\mathcal{O}(n \log n)$ time. The upside is that it may allow an optimal choice of split location and will directly produce a depth-first tree.

Before uploading the tree to the GPU we pack it into the compact representation shown in listing 1. Note that we also store a skip count with each node which allows us to traverse any binary tree laid out in depth-first order (not only complete binary trees) [Smits 1998]. We store the skip count, the light count and a leaf flag packed into a 32-bit integer. The skip count is stored using 15 bits. We use one bit to indicate if a node is a leaf. The remaining 16 bits are used for the number of lights. This allows us to store up to 2^{16} lights per tile. We target several lights per leaf node, so the tree will contain generally much fewer nodes than we have light sources.

3.3 Hybrid algorithm

On their own, the light trees provide a highly adaptive data structure and often improve GPU lighting performance over the "practical clustered shading" approach. Unfortunately, there are still some combinations of view conditions in which the light trees are slightly slower than just clustered shading. In those cases, the light trees add a small, but noticeable amount of overhead over just storing a list of light sources and iterating over it. We thus combine our light trees with a 2.5D grid and choose the optimal algorithm per grid cell. The combined algorithm looks as follows: Just as before, we start with frustum culling of the light sources and the sorting by distance to the camera plane. The major change is in the next two steps. We first assign the light sources to frustum-aligned grid cells. Finally, we identify expensive grid cells and build a per-cell light tree for those.

Listing 2: *Heuristic for selecting between tree and list.*

```
float sum = 0.0;
for (auto& light : cell.lights) {
  sum += getClippedDepthExtents(cell, light);
}
if(sum / cell.lightCount < cell.radius) {
  buildTree(cell);
} else {
  buildList(cell);
}
```

We subdivide the depth similar to [Persson 2013] and decide per grid cell if we want to store a plain list of lights, or use the light trees. This requires a heuristic to choose between the two options. We use the average overlap in z axis between a cell and lights assigned to it to decide between lists and trees. When the average overlap is small compared to the extents of the cell, the cell is likely to contain a significant amount of empty space that can be efficiently skipped by using the light trees. As the average overlap approaches the cell extents, it indicates that there are many lights which cover most of the cell depth and a list should be used. Using the average overlap (light extent clipped to cell extent) instead of average light diameter makes the heuristic more robust in scenes with a mix of small and large lights. The net result is that in situations with many large overlapping lights our algorithm gracefully falls back to [Persson 2013] which is optimal for that case. For many small lights, only the light trees will be used, and for realistic distributions containing both large and small light sources our heuristic picks the best technique. The alternative approach for small lights, together with the ability to switch between techniques, is

Figure 3: *We classify each tile on the GPU into either list traversal (blue), tree traversal (green), or combined traversal (white). Notice the very high coherency in screen-space; very few tiles go down the mixed path. Scene shown is "CTF-TitanPass" ©Epic Games.*

one of the key reasons which allow our algorithm to provide the best performance for any light distribution.

3.4 GPU traversal

Listing 3: *GPU-friendly stack-less light tree traversal.*

```
unsigned nodeIndex = 0;
while(nodeIndex < nodeCount) {
  LightTreeNode node = lightTree[nodeIndex];
  if (nodeHit(node, distanceToCameraPlane)) {
    if (isLeaf(node)) evaluateLightSources(node);
    nodeIndex++;
  }
  else nodeIndex += getSkipCount(node);
}
```

Our hybrid algorithm has to combine both tree-traversal and list iteration for optimal performance. As our tree traversal code already allows for a variable amount of light sources per leaf node, we have chosen the following method to combine both: if the cell contains a list of items, we store a one-node tree where the root is also a leaf. In this case, the threads will just iterate over the individual light sources and the only overhead is a dependent fetch and a bit test.

Notice that this is not a uniform decision for a single tile – the threads in one tile may hit many different cells concurrently and will have to traverse both lists and trees at the same time. We have optimized the various cases using intrinsic instructions, separating the traversal into three different cases: a) all threads traverse a list, b) all threads traverse a tree, c) there is divergence and both lists and trees are traversed.

Which of the cases is executed is mostly dependent on the light distribution in the scene. Tiles with many overlapping light sources will go down the list path; and most of the remaining tiles will traverse trees. Traversing both lists and trees only happens for tiles with high depth variance (see also figure 3).

For the tree traversal case, we've implemented an optimization which loads the tree into local memory (LDS) to improve the main memory access pattern. We only use this optimization if all threads in a subgroup traverse the same tree, which is the common case.

4 Results

We have implemented our algorithm using Vulkan and evaluated the performance on an AMD RX480 as well as a NVIDIA GTX 980. We have compared our algorithm against "practical clustered shading" [Persson 2013]. For every scene, we have set the maximum depth extent of the frustum based on the scene extents; this is the only scene-specific setting we've done. While light trees are fully adaptive and do not require any scene-specific configuration, the performance of the "practical clustered shading" algorithm is highly sensitive to depth partitioning, light source sizes and distribution. [Persson 2013] describes a partitioning scheme with 15 exponentially distributed depth slices from 5 to 500 meters and a special "near" slice. Our benchmarks use a similar scheme (listing 4) with only minor modification to maximum depth. Identical grid configuration is used for hybrid and clustered modes. Even though our hybrid algorithm can degrade gracefully into either case, we have separate code paths for each implementation to have the minimal shader for each implementation.

We have chosen a light size distribution which is typical for many games; there's often a very large number of small lights such as street lamps, flashlights, car head and tail lights, but only few large light sources, especially with large on-screen contribution. We have tried to mimic this by using a light size distribution which results in low per-pixel overdraw; that is, only a few light sources affect each pixel.

Listing 4: *Depth slice index calculation.*

```
int computeSliceIndex(
  float logDepth,
  float logMinDepth, float logMaxDepth,
  int sliceCount)
{
  float scaleDenom = (logMaxDepth - logMinDepth)
                   * (sliceCount - 1.0f);
  float scale = 1.0f / scaleDenom;
  float bias = 1.0f - logMinDepth * scale;
  float slice = max(logDepth * scale + bias,
                    0.0f));
  return min(int(slice), sliceCount - 1);
};
```

4.1 Preprocessing

Our algorithm requires preprocessing per frame on the CPU – this includes the light frustum culling, the light sorting, the light binning, as well as the tree creation. Light binning is by far the most expensive step that accounts for 50% of total CPU time, followed by tree creation (25%), sorting (12%), data upload to the GPU (7%) and frustum culling (3%). Relative costs of different preprocessing steps are consistent for different light counts.

Our binning algorithm computes a frustum-aligned grid bounding box for each light using 2D bounds [Mara and McGuire 2013] and depth extents. Bounding boxes are used to scatter lights into grid cells, which takes most of the CPU time of this step. Binning is also required for clustered shading algorithm, where it accounts for 85% of the CPU cost.

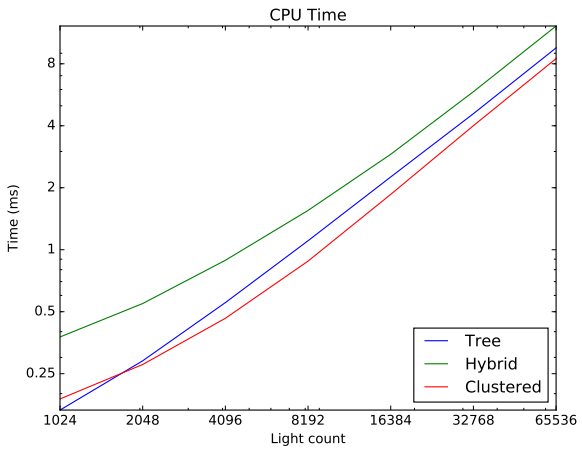


Figure 4: *Average CPU processing time comparison between the various approaches at different light counts. The CPU time scales linearly with the number of lights. For light counts below 8192, all algorithms require less than 2 ms. Performance data captured on "CTF-TitanPass" at* 1920×1080 *resolution and* 48^2 *tile size, 16 depth slices on Intel i7-4790K CPU (single threaded).*

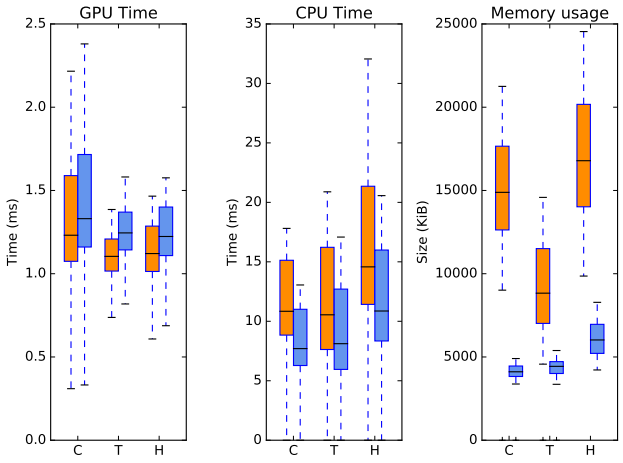


Figure 5: *Comparison between duplicated (orange) and indexed (blue) data for* 65536 *light sources on "CTF-TitanPass" at* 1920×1080 *resolution and* 48^2 *tile size, 16 depth slices on AMD Fury X GPU and Intel i7-4790K CPU (single threaded). Using indexed light data improves CPU execution performance and reduces memory usage, and is the method we use by default. C is Clustered, T is tree only, and H is the hybrid method.*

We have parallelized the tree creation using the Parallel Patterns Library [Microsoft 2016], which provides both parallel sorting as well as a generic parallel `for` loop and other constructs. All trees can be processed without locking or synchronization. This is possible because each tree is completely independent and all required memory is allocated upfront with information from the binning step. We have not performed any other low-level CPU optimizations, such as vectorization using SIMD intrinsics or ISPC [Intel 2011]. We expect that light binning, frustum culling and parts of the tree building could be significantly sped up by using SIMD instructions. Even without further optimization,the CPU cost is acceptable for real-time applications for all but the highest light count (see Figure 4).

We store the light data separately from our culling data structure. That is, the lists and the trees contain light indices and every fetch retrieves the light from a light buffer indirectly. We found this optimization to significantly reduce memory usage with minimal impact on the GPU cost. Besides cutting down memory, we also measured a significant improvement in CPU performance, roughly around 20% (see figure 5). If needed, light data expansion can be done on the GPU; which will keep the CPU benefits but requires an additional pass before the lighting can be performed and increases GPU memory footprint.

4.2 Rendering

We have tested our algorithm on two scenes – "CTF-TitanPass" and "EpicCitadel". The major difference between those scenes is that "EpicCitadel" features a much larger view distance. We have also used two different light distribution algorithms. In "CTF-TitanPass", we distribute the lights randomly throughout the scene bounding box, which produces light sources that do not intersect with the scene geometry. This is typical for situations where geometry is culled due to occlusion, but the light sources themselves remain visible. In "EpicCitadel", we distribute the light sources directly on the geometry by randomly selecting triangles and placing lights near them. Here, all light sources are guaranteed to intersect with geometry, which is the most taxing case in terms of light overdraw.

For all algorithms, the screen space tile size plays a critical performance role. We have measured the performance of various tile sizes (see figure 6.) For all other tests, we have used 48^2 sized tiles which provide the best trade-off between CPU and GPU performance.

In figure 7a, we show the performance on the "CTF-TitanPass" level from Unreal Tournament. We have distributed between 8 and 65 thousand light sources throughout the scene. The camera follows a fixed path through the scene, visiting places with short and long view ranges. We have logged the time spent on the GPU for lighting using time queries. At low light counts, we found that both clustered shading as well as our "hybrid" algorithm significantly outperform the "tree"-only approach. With increasing light numbers, "tree" becomes more competitive, finally surpassing the clustered shading at light counts on both tested GPUs. We also notice that on the RX 480, the "hybrid" algorithm consistently provides the best performance. On the GTX 980, "hybrid" is sometimes slightly outperformed by "tree", but in general they are quite close together.

At the beginning of the test, and around frame 1800, we have the longest view range in this scene. The number of pixels that traverse trees becomes very high in this case, as far away grid cells end up containing hundreds of small light sources. In this case, the tree traversal allows us to rapidly identify the light sources actually intersecting the pixel, while clustered shading has to traverse long lists for those cells.

Figure 7b presents the performance on "EpicCitadel". The light sources are evenly placed near the scene geometry. This level has a much longer view range than "CTF-TitanPass"

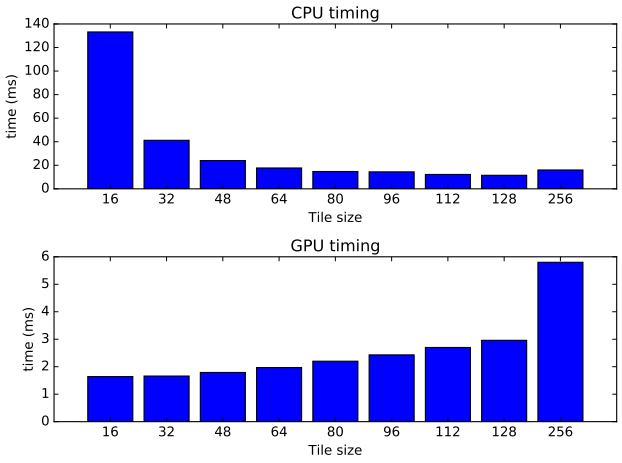


Figure 6: *CPU and GPU costs at different tile sizes for our hybrid algorithm, which is representative for all algorithms.*

which makes it particularly challenging for the clustered lighting approach. Our "tree"-only and "hybrid" algorithms consistently outperform clustered lighting in this scene. It highlights the improved robustness of our algorithm; while clustered lighting cost varies between 0.38ms and 7.0ms on a GTX 980, our hybrid approach has much smaller difference between the best (0.4ms) and worst (3.5ms) cases.

5 Conclusion

In this work, we have described a novel approach which improves upon the existing solutions, particularly in worst case scenarios. Our algorithm prepares an adaptive spatial data structure on the CPU without any knowledge of the scene geometry and works in tandem with a highly efficient traversal algorithm on the GPU. We have combined the light trees with existing 2.5D acceleration structures yielding a new hybrid approach which in many cases outperforms each of the algorithms individually.

As currently proposed, our algorithm prepares all data on the CPU, but going forward we envision moving more of the work to the GPU. The first step is likely the per-cell tree creation, which is already highly parallel on the CPU side. With other recent advancements in GPU binning [II and Pattanaik 2016], it also seems feasible to move the whole pre-process onto the GPU where it could be executed asynchronously with other scene preparation tasks like shadow map generation.

Acknowledgements

We'd like to thank Epic Games for allowing us to use the scenes from Unreal Tournament and the UDK. We also want to thank the anonymous reviewers and our colleagues at Electronic Arts and AMD for their valuable feedback.

References

Andersson, J., 2009. Parallel graphics in frostbite - current & future. SIGGRAPH Course: Beyond Programmable Shading.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2009. *Introduction to Algorithms*, 3rd ed. MIT Press and McGraw-Hill.

Ferrier, A., and Coffin, C. 2011. Deferred shading techniques using frostbite in "battlefield 3" and "need for speed the run". In *ACM SIGGRAPH 2011 Talks*, ACM, New York, NY, USA, SIGGRAPH '11, 33:1–33:1.

Garawany, R. E., 2016. Deferred lighting in uncharted 4. SIGGRAPH 2016, Advances in Real-Time Rendering.

Harada, T., McKee, J., and Yang, J. C. 2012. Forward+: Bringing Deferred Lighting to the Next Level. In *Eurographics 2012 - Short Papers*, The Eurographics Association, C. Andujar and E. Puppo, Eds.

Harada, T. 2012. A 2.5d culling for forward+. In *SIGGRAPH Asia 2012 Technical Briefs*, ACM, New York, NY, USA, SA '12, 18:1–18:4.

II, E. M. T., and Pattanaik, S. N. 2016. A memory efficient uniform grid build process for gpus. *Journal of Computer Graphics Techniques (JCGT) 5*, 3 (September), 50–67.

Intel, 2011. Intel spmd program compiler. https://ispc.github.io.

MacDonald, D. J., and Booth, K. S. 1990. Heuristics for ray tracing using space subdivision. *Vis. Comput. 6*, 3 (May), 153–166.

Mara, M., and McGuire, M. 2013. 2d polyhedral bounds of a clipped, perspective-projected 3d sphere. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (August), 70–83.

Microsoft, 2016. Parallel patterns library. https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl.

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*.

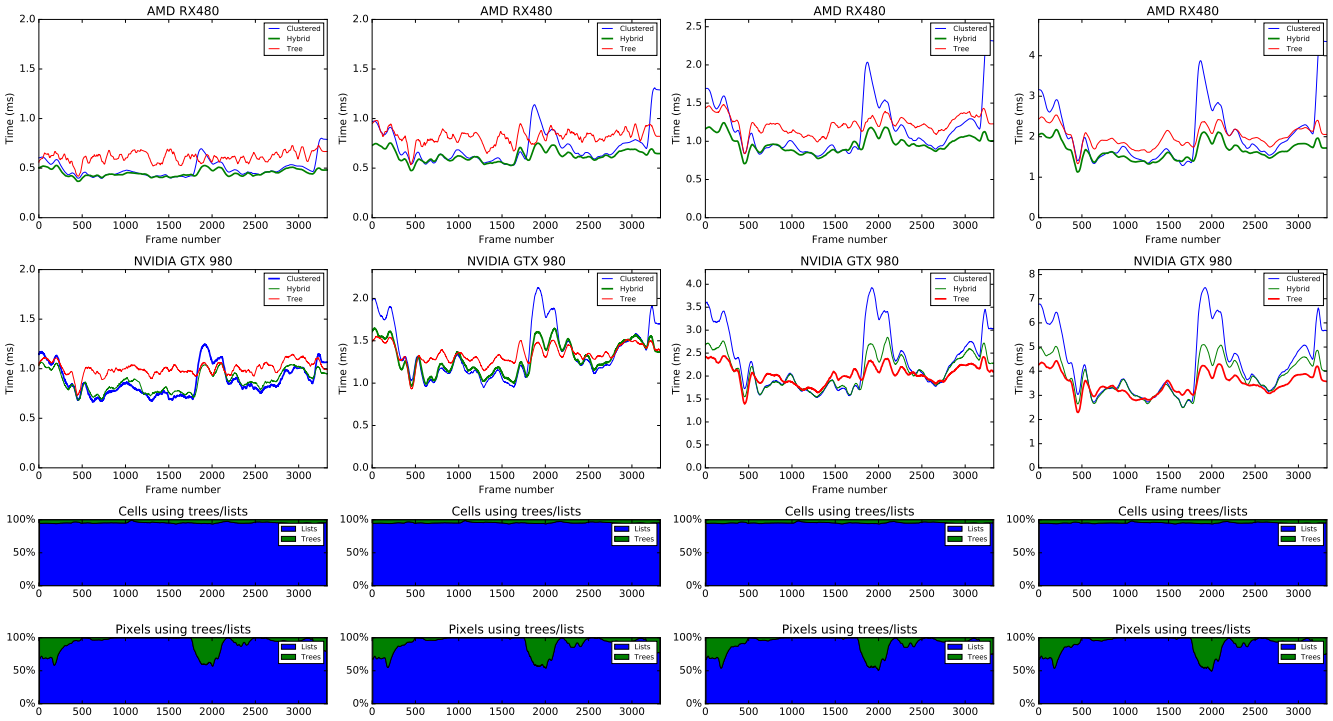
Olsson, O., Persson, E., and Billeter, M. 2015. Real-time many-light management and shadows with clustered shading. In *ACM SIGGRAPH 2015 Courses*, ACM, New York, NY, USA, SIGGRAPH '15, 12:1–12:398.

Örtegren, K. 2015. *Clustered Shading: Assigning arbitrarily shaped convex light volumes using conservative rasterization.* Master's thesis, Blekinge Institute of Technology.

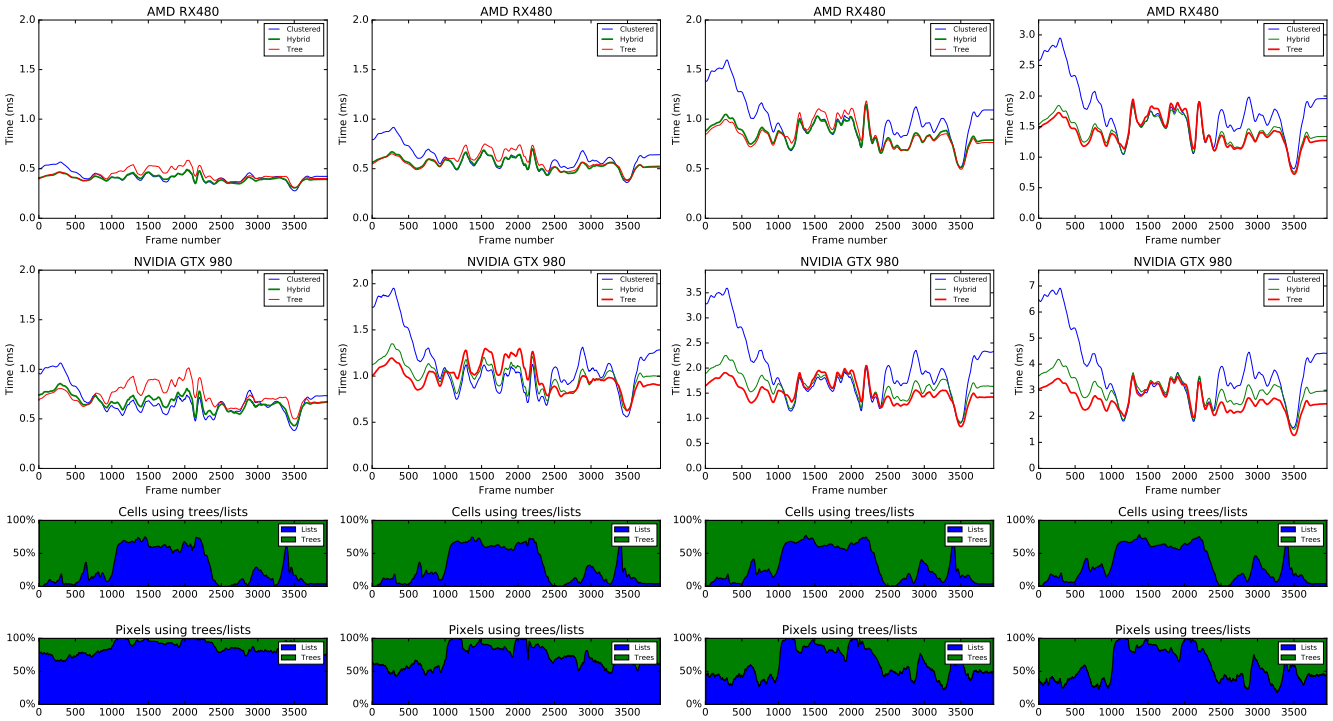
Persson, E., 2013. Practical clustered shading. SIGGRAPH Course: Advances in Real-Time Rendering in Games.

Smits, B. 1998. Efficiency issues for ray tracing. *J. Graph. Tools 3*, 2 (Feb.), 1–14.

Sousa, T., and Geffroy, J., 2016. The devil is in the details: idtech 666. SIGGRAPH 2016, Advances in Real-Time Rendering.



(a) *Results on the CTF-TitanPass level.*



(b) *Results on the EpicCitadel level.*

Figure 7: *Results on our test scenes for a fly-through at varying light counts (from left to right:* $8192, 16384, 32768, 65536$*). The fastest algorithm – with the lowest geometric average time over the test – is bold. Our tree and hybrid algorithms are generally very close, with a slight advantage for the hybrid algorithm on the AMD RX 480 card, and a slight advantage for the tree only algorithm on the GTX 980. The bottom two rows show how many grid cells have been assigned lists or trees in the hybrid case, and how many pixels have actually traversed a list or tree.*