Per-Triangle Shadow Volumes Using a View-Sample Cluster Hierarchy

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

Chalmers University of Technology

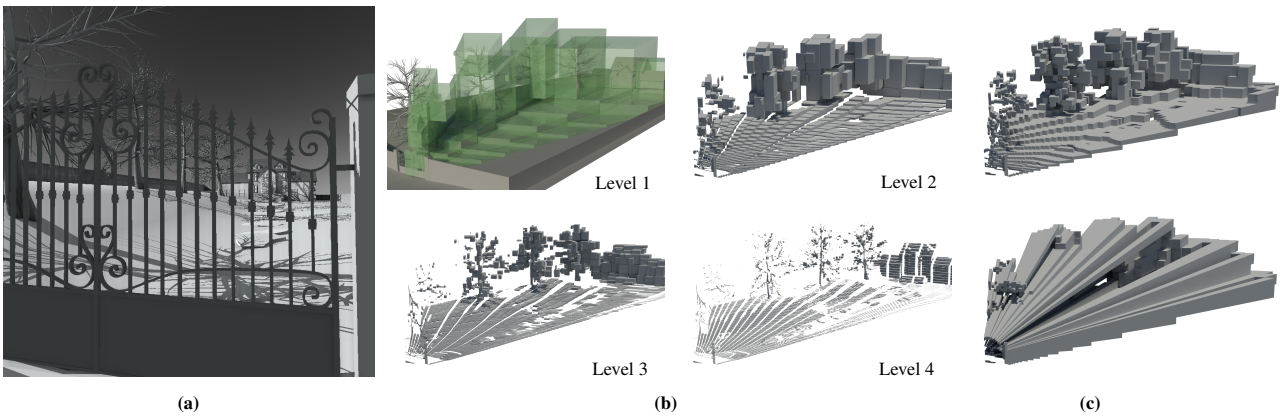


Figure 1: *a) A challenging scene for most real-time shadow algorithms, rendered in 4.2ms with our cluster hierarchy, 7.46ms with the original PTSV, and 12.16ms with ZPASS. b) The final four levels of our view-sample acceleration structure visualized. c) Top: Level 3 without explicit bounds. Bottom: The corresponding level when using the original PTSV algorithm.*

Abstract

Rendering pixel-accurate shadows in scenes lit by a point light-source in real time is still a challenging problem. For scenes of moderate complexity, algorithms based on Shadow Volumes are by far the most efficient in most cases, but traditionally, these algorithms struggle with views where the volumes generate a very high depth complexity. Recently, a method was suggested that alleviates this problem by testing each individual triangle shadow volume against a hierarchical depth map, allowing volumes that are in front of, or behind, the rendered view samples to be efficiently culled. In this paper, we show that this algorithm can be greatly improved by building a full 3D acceleration structure over the view samples and testing per-triangle shadow volumes against that. We show that our algorithm can elegantly maintain high frame-rates even for views with very high-frequency depth-buffers where previous algorithms perform poorly. Our algorithm also performs better than previous work in general, making it, to the best of our knowledge, the fastest pixel-accurate shadow algorithm to date. It can be used with any arbitrary polygon soup as input, with no restrictions on geometry or required pre-processing, and trivially supports transparent and textured shadow-casters.

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

Keywords: shadows, alias-free, real-time

*e-mail:erik.sintorn|kampe|olaolss|uffe@chalmers.se

1 Introduction and Previous Work

In this paper, we suggest a novel method for rendering pixel-accurate shadows from point-light sources in real time. While an abundance of very fast shadow algorithms are available (see, e.g., Real Time Shadows [Eisemann et al. 2011] for a recent overview of shadow algorithms in general), the vast majority are image-based approximative approaches, based on Shadow Mapping [Williams 1978]. In these algorithms, a discrete image (a shadow map) that contains point samples of the closest distance to the shadow casters from the light is generated. This shadow map is then queried while shading to determine whether an arbitrary point is in shadow or not. As the shadow map is generated without taking the actual points to be shaded into account, the result of the query can only be approximate and has to be filtered to reduce aliasing artifacts. To avoid having to use too large shadow maps and too large filter kernels, it is common practice to partition the view frustrum and use different shadow-maps for each partition (e.g. [Zhang et al. 2006]). With proper filtering and a good partitioning scheme, it is possible to obtain very high quality shadows that may be sufficient for many real-time applications, such as video games, but pixel-perfect shadows cannot be guaranteed. With virtual texturing, extremely high resolutions are possible, however [Lefohn et al. 2007], at the cost of view-dependent performance and memory requirements.

A second class of algorithms are those where the actual view samples to be considered are first collected and organized in some form of acceleration structure [Aila and Laine 2004; Johnson et al. 2005]. The shadow-casting polygons are then rendered over these irregular samples to determine light visibility for each view sample. While GPU-based implementations of these algorithms exist (e.g. [Sintorn et al. 2008]), the irregular rasterization process often leads to very unbalanced workloads, which results in uneven and potentially very poor performance.

A third class of algorithms are those based on Shadow Volumes [Crow 1977]. Here, for each shadow-casting object, a polygonal mesh (the shadow volume) that encloses the space that is blocked from the light by that object is generated. The shadow volumes are

then tested against all view samples, and a view sample is considered in shadow if it lies within any of the volumes. Shadow volumes were for a while frequently used in practice after a version of the original shadow-volume algorithm was introduced in which these inclusion tests could be performed efficiently on graphics hardware [Heidmann 1991]. We will refer to this algorithm as ZPASS in the remainder of this text. The basic idea is to extract all *sillhouette edges* from the mesh, each frame, and extrude these infinitely far away from the light source to form a *shadow quad*. The scene is first rendered from the camera's point of view into a depth buffer. Then, the shadow quads are rasterized as polygons onto a cleared stencil buffer while testing against this depth buffer. The stencil buffer is incremented for every front-facing polygon and decremented for every back-facing polygon. The resulting stencil value will be zero only when the view sample does not lie within any shadow volume. This algorithm only works well as long as the camera itself is not inside a shadow volume. To avoid that problem, the ZFAIL algorithm was introduced [Carmack 2000; Bilodeau and Songy 1999]. The only difference here is that the standard depth test is reversed so that all shadow quads that lie *behind* the z-buffer are rendered instead. This algorithm is more robust, but typically slower due to a higher fill rate.

The main problem with the traditional shadow volume algorithms lie in that they cull view samples only on their two-dimensional position in view space. Thus, a shadow quad must be tested against the potentially very large number of samples that lie within the volume formed by the quad and the camera position. Sintorn et al. [2011] alleviate this significantly by building a min-max hierarchy over the depth buffer and testing individual triangle shadow volumes against this hierarchy. A shadow volume can then be culled as long as it does not intersect the frustum formed by a node (or if the entire node is within the shadow volume). The authors show that the number of actual test-and-set operations required are dramatically reduced and that performance of their implementation is on par with, or better than, previous algorithms. Additionally, since each view sample can be tested against each triangle shadow volume, this algorithm trivially supports textured and semi-transparent shadow casters, and it can robustly handle any arbitrary set of shadow-casting polygons, without connectivity information. We will refer to this algorithm as *Per Triangle Shadow Volumes* (PTSV) in the remainder of this paper.

There are several other papers that attempt to reduce the fill-rate problems inherent in the traditional shadow-volume algorithm. In the work by LLoyd et al. [2004], the shadow volumes are culled and clamped per object in the scene graph, to reduce unnecessary overdraw. These methods are orthogonal to our algorithm. Aila and Akenine Möller [2004] identify tiles that lie on the shadow boundary and need to perform full per pixel tests only for these tiles. Chan and Durand [2004] attempt to find umbra regions and identify shadow boundaries using a shadow map before reverting to standard shadow volumes for only the pixels that lie on these boundaries. Finally, in Split Plane Shadow Volumes [Laine 2005], the number of stencil updates are reduced by locally (per tile) choosing whether to use the ZPASS or ZFAIL algorithm.

In this paper, we improve on PTSV by building a complete three-dimensional acceleration structure over the view samples, allowing *clusters* of samples that have the same two-dimensional bounds to be considered separately when they lie at different depths. Figure 1 illustrates a case where all previous work will perform very badly. A house in the far distance is viewed through a nearby gate, and a number of trees cast complex shadows that intersect the volume in between. The ZPASS algorithm [Heidmann 1991] will need to consider all view samples that have a depth which is further away than the shadow quad, i.e., essentially all view samples that do not lie on the gate. The ZFAIL algorithm [Bilodeau and Songy 1999; Carmack 2000], in contrast, must consider all samples that lie in *front* of the shadow quad. Unfortunately, the PTSV algorithm cannot do much better in this case, since most 8×4 tiles in the image will contain view samples from both the foreground and the background (see Figure 1c).

A very similar problem exists in the realm of real-time shading with many lights, where tiled shading algorithms have recently become popular [Olsson and Assarsson 2011; Harada 2012]. Similarly to PTSV, tiled-shading algorithms are sensitive to depth discontinuities. Our proposed algorithm is inspired by a recent solution to that problem, called *Clustered Shading* [Olsson et al. 2012]. The problem in tiled shading is analogous to that of PTSV, with the difference that the volumes considered are not shadow volumes but the bounding volumes of lights with a finite range (as is common in real-time applications). Olsson et al. observe that depth discontinuities can lead to many false positives where light volumes intersect tile volumes but none of the samples within, and that this problem is highly view dependent, leading to high variability in rendering times. They show that by clustering samples into three-dimensional subdivisions, as opposed to two dimensions for tiled, light-culling efficiency becomes much higher and view dependence lower, especially when considering many small light sources.

The main contributions in this paper are:

- A better view-sample acceleration structure for PTSV, which has a much smaller total volume and improves efficiency and performance significantly, making our algorithm the fastest real-time alias-free shadow method to date.
- A two-pass algorithm which removes performance spikes that are due to poor load balancing, at no extra cost.
- An improved set of culling planes over PTSV, which significantly reduces the number of false positives during traversal.

Meanwhile, our algorithm still maintains all the good properties of the PTSV algorithm. Shadow casters can be any arbitrary triangle soup with no additional connectivity information and we also inherit the ability to trivially support textured or semi-transparent shadow casters.

2 Algorithm

The goal of our algorithm is to establish, for every view sample in the G-buffer, whether that view sample is directly visible by a point light source or not. Samples that are blocked from the light source will be in shadow and the rest will have direct lighting applied in a final shading pass. We accomplish this by generating an acceleration structure over the view samples and then testing the shadow volume of each triangle against this structure. This approach has been attempted several times before (e.g. [Aila and Laine 2004; Sintorn et al. 2008; Sintorn et al. 2011]) but in this paper we will suggest that the quality of the acceleration structure is critical to obtaining good and reliable performance, and so we will generate a tightly fitting, fully three-dimensional hierarchy.

To this end, the view frustum will be divided into a coarse three-dimensional grid and each view sample will be processed to mark those grid-cells that are occupied. From this grid we then build a hierarchy against which we can traverse triangle shadow-volumes. A shadow volume can be tested against any node in the hierarchy as the bounds of the corresponding AABB are implicitly defined. When a leaf node is found to be intersecting with a shadow volume, all view samples that reside in the same two-dimensional tile as that node are tested. We show that performance can be further improved by calculating the explicit bounds of each node, and that these can be efficiently calculated while building the hierarchy. The

main improvement over previous work comes from this much tighter acceleration structure, with which a shadow volume will only need to traverse down to the leaf nodes if it actually lies very close to the samples contained therein.

We will begin by describing the basic steps of our algorithm, starting with an overview and then discussing the different parts in detail. We will then discuss some shortcomings of our initial implementation and how they can be overcome. Our algorithm and implementation closely follow the steps of the PTSV algorithm detailed by Sintorn et al. [2011]. We have implemented the algorithm in CUDA, and it runs entirely on the GPU, without reading any data back to the CPU. The steps of the algorithm are:

- **Building a Hierarchy** Using the current G-Buffer, build an acceleration structure that groups view samples that are close to each other.
- **Triangle Setup** For each shadow casting triangle, create its shadow volume, i.e., a set of planes that enclose the volume of space that is in shadow due to that triangle.
- **Traversal** Traverse each triangle shadow volume through the hierarchy, culling nodes that lie completely outside and marking those that are completely inside the volume as *in shadow*. Only when a node might intersect the shadow-volume planes do we traverse into its children.

Building the acceleration structure As in the PTSV algorithm, we choose to build a tree that has a branching factor of 32. This allows the SIMD lanes of one multiprocessor of the GPU to work in parallel with the intersection tests that make up the bulk of our traversal algorithm, and so we can utilize the hardware efficiently. Using another fanout is a trivial change to the algorithm, however, so it could easily be varied for different hardware.

We chose to group view samples into clusters that are 8×8 pixels. Thus, a single cluster can contain a maximum of 64 view samples. Given a view sample whose pixel coordinates are (x, y) and which has a view space depth z, we calculate the *cluster coordinate*, $\mathbf{x}'$, as:

$$\mathbf{x}' = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \lfloor x/8 \rfloor \\ \lfloor y/8 \rfloor \\ \left\lfloor \frac{log(-z/near)}{log(1+\frac{2tan\Theta}{S_y})} \right\rfloor \end{pmatrix}, \quad (1)$$

where $near$ is the distance to the near plane, Θ is the field-of-view, and S_y is the number of cluster divisions in height. The cluster coordinate's z' component is chosen as in the work by Olsson et al. [2012], i.e., we subdivide the frustum exponentially in depth to obtain leaf nodes whose implicit bounds are frustums with a depth that is roughly equal to their width and height in world space.

We interleave these integer coordinates to produce a key in morton order [Morton 1966], which we call the *cluster key*. This key uniquely defines a leaf node in our hierarchy, and several view samples may have the same key. For now, we will consider a screen resolution of 1024×1024, and so we only need seven bits for the x' and y' coordinates. The number of bits needed for the z' component depends on the ratio between the near and far distances used in the projection. At this resolution, we found nine bits to be more than sufficient for all of our scenes. We therefore rearrange the cluster key somewhat, as shown in Figure 2, to allow for a more shallow tree as discussed below.

We now need to build an acceleration structure over these keys. In our initial attempts, we followed the approach of Olsson et al. [2012], building a list of cluster keys and compacting this list so that we had a

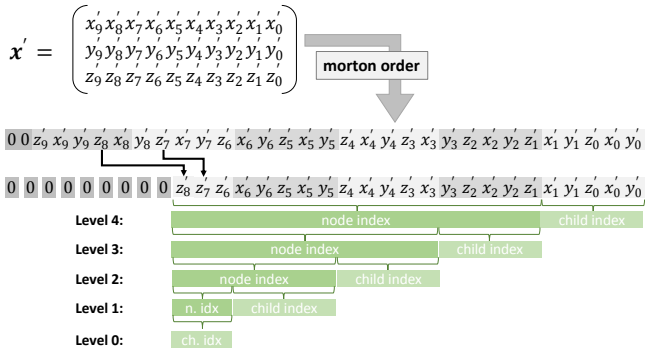


Figure 2: *The cluster coordinate is packed into a 23-bit integer using a slightly rearranged morton order. This key can then efficiently be used to populate a full tree hierarchy.*

minimal set of clusters from which to build the tree. However, when building a hierarchy on top of these clusters, we need to maintain a pointer from each node to where its children are stored, along with a bitmask that tells us which of the children exist. While we were able to build this compact list and hierarchy very quickly ($<$ 1ms for a resolution of 1024×1024), both building the tree and traversing it is significantly faster if we choose to sacrifice some memory and store a full tree instead.

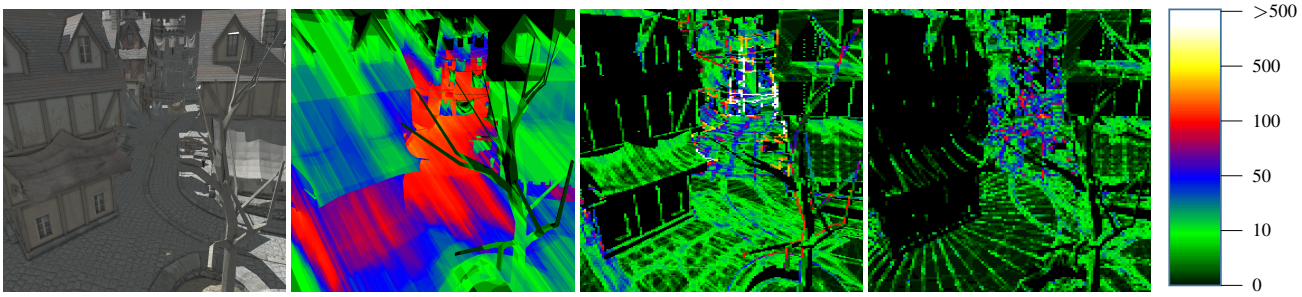
The five levels of our hierarchy are represented by five arrays of 32-bit words. Each word is a child mask, where a set bit indicates that the corresponding child exists in the tree. To build the final level of the tree (where a set bit indicates the existence of a leaf node, or cluster), we can simply process all view samples and calculate their cluster keys. The 18 most significant bits of the key give us the index to the node in the level-four array in which the view sample resides. The five least significant bits tell us which of the bits of this node shall be set to indicate that this cluster exists. Having populated the lowest level of our tree, we shift the cluster key 5 bits to the right and repeat the process to populate the next level above. This goes on until we have a single root node at level 0 represented by a single 32 bit word (see Figure 2).

Our initial implementation of this step is to simply start a thread per view sample and let that thread atomically update all the nodes in which it resides (in Section 2.1, we will add optimizations).

Triangle Setup For each triangle, we calculate the four planes that enclose a triangle shadow volume. Care must be taken to ensure that two triangles that share an edge will calculate exactly the same plane for that edge, and when not using light front-face culling, a small bias must be added to the triangle plane to avoid self shadowing. We transform these planes into clip space, using the camera's model-view-projection matrix, and store them in a list. This basic triangle setup is exactly the same as is done in PTSV.

These four planes are sufficient to produce correct results, but will, during traversal, produce a lot of false positives. In PTSV, tiles are also culled against the two 2D silhouette edges of the shadow volume, which alleviates the problem. We have found that this method still produces a significant amount of false positives (see Figure 6) and suggest another set of culling planes.

The problem with false positives occur when an Axis Aligned Bounding Box (AABB) does not lie completely outside all shadow volume planes. This can happen in the wedges formed by any two of the planes (see Figure 4). For each vertex (and so, each wedge), we add a plane that contains both the light source and the vertex. Any rotation of that plane is legal as long as it does not intersect the



(a) CITADEL *scene.* **(b)** *ZPASS* **(c)** *PTSV* **(d)** *Our algorithm*

Figure 3: *Heat maps showing the number of times each pixel is tested against a shadow volume. The PTSV algorithm has a very unbalanced workload around steep depth discontinuities, where the view-sample cluster hierarchy performs much better.*

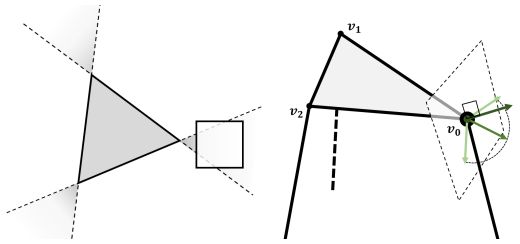


Figure 4: *Left: False positives occur when a node that does not intersect the triangle still does not lie completely outside any plane. Right: We add new planes, attempting to cull nodes that fall within the false-positive wedge.*

shadow volume. We choose to create the half vector between the two shadow-volume plane normals, make that orthonormal to the vector pointing from the vertex to the light source, and use that as the plane normal for our culling plane.

A node that lies completely outside any of these new planes is guaranteed to lie outside the shadow volume, and testing against all three planes improves culling (and performance) significantly. Note that for degenerated triangles or triangles whose normals are close to orthogonal to the light direction, these calculations can be unstable, and so we simply skip these culling planes in such cases.

Traversal The traversal kernel is written in a *persistent-threads* fashion, where we start enough warps to keep the machine fully utilized and let each warp pick jobs from a list by atomically updating a counter to get an index. Each job, in this case, is one triangle shadow volume, which all threads in the warp will cooperatively traverse against the hierarchy, starting with the root node. Each thread will consider one child of the current node and test that child against the four planes that make up the shadow volume. The thread will first consult the child mask for the current node to make sure that the child exists in the tree. If it does, but the child is completely outside any of the shadow volume planes, it is rejected. If the child lies completely inside all four planes, it can be trivially accepted, which is noted by simply clearing the corresponding bit in the current nodes child mask. If neither is true, the child is tested against the extra culling planes described above and if it does not lie completely outside any of those, it is considered intersecting.

After the intersection test, the results are broadcast (using CUDAs ballot instruction) so that all threads know which children are intersected. These will now be tested in turn in the same way. In order to know where in the tree we are, we maintain a current *key* throughout the process (as detailed in Algorithm 1).

When the leaf-node level is reached, the threads will instead cooperate in testing the individual view samples within the tile. At this level, we have no information about which view samples actually lie in the cluster and which lie in another cluster with the same 2D bounds, so we simply test all samples and update the shadow buffer if the sample lies within the shadow volume. We could of course store this extra information while building the hierarchy, but as 32 intersection tests will always be done in parallel, it seems unlikely that this would improve overall performance.

So far, our traversal algorithm does not differ much from that of Sintorn et al. [2011]. One difference is that their acceleration structure *is* a full tree with samples in every node, whereas, while we store our structure as a full tree, it is actually very sparse, and so we need to fetch the child mask for each node in order to know which children to test. Another difference lies in how the clip-space coordinates for the AABBs of the nodes are calculated. Unlike their algorithm, in which the z-components are fetched from a texture, we get all coordinates implicitly from the cluster keys. The traversal algorithm is outlined in Algorithm 1. We have implemented this algorithm in CUDA, both in an iterative fashion, using a small stack, and as written in Algorithm 1, using template metaprogramming for the recursion. The latter performs slightly better, probably due to better optimization opportunities.

After traversal, we will know for each pixel whether it is in shadow or not, *except* for pixels that have only been trivially accepted as part of some node. We must therefore run one final pass where we start one thread per view sample. Each thread will perform almost exactly the same job as when building the hierarchy (as explained above), except that this time, instead of updating the hierarchy, it will just make sure that all the nodes it resides in are still marked as existing. Otherwise, the node has been trivially accepted and the view sample is set as shadowed.

Remaining problems The algorithm, as described so far, works well and we can see from our measurements that we have mostly eliminated the problem where the amount of work increases significantly at steep depth discontinuities (see Figure 6). Unfortunately, we can also see that the total number of test and set operations that we need to do each frame is overall significantly higher than in PTSV. This is partly due to us having a deeper tree but also because the size of our nodes is completely unaffected by the actual view samples within. In PTSV, tiles that do not contain depth-discontinuities will have a fairly well fitting bounding box, while ours will be fixed size and potentially very conservative.

Another problem, which we share with PTSV but which will be more exaggerated in our case, is the possibility of poor load balancing. Even with our persistent threads model, we see that some

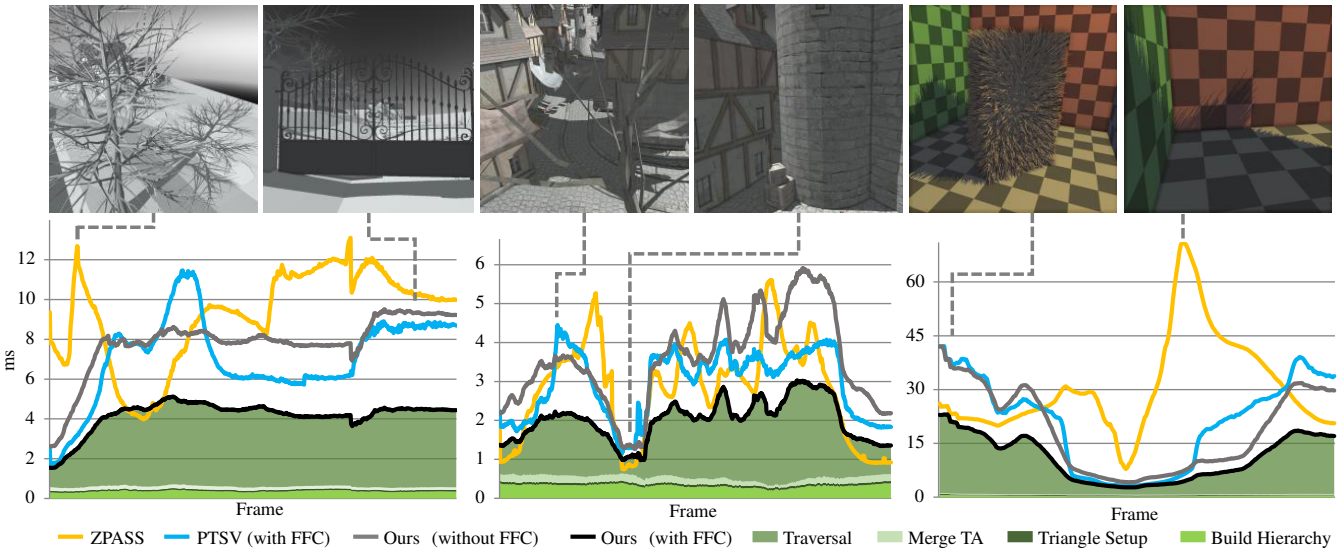


Figure 5: *Time taken by each part of our algorithm, along with total time taken for ZPASS and PTSV, for three different animations. Total performance of our algorithm is presented with and without Front Face Culling (FFC). From left to right,* VILLA *(88k triangles),* CITADEL *(60k triangles) and* FUZZY *(400k triangles).*

multiprocessors have to wait for a long time, while a few finish processing triangles that generate much more work than the average. This problem is illustrated in Figure 7.

Finally, where PTSV only has to build a small mip-map hierarchy over the depth buffer, we have to build a much larger hierarchy. Initially, we did this as explained above, with one thread per view sample, which proved to be a less than optimal solution. We will discuss these three problems in the upcoming sections.

2.1 Explicit bounds

In our hierarchy, the existence of a node is indicated by a single bit in its parent's child mask. The size of each node is implicitly defined by its position in the tree. This makes for a very memory-efficient representation, but, as explained above, does not give us much opportunity to cull shadow volumes. We will now explore the possibility of storing a complete AABB with each node to improve culling performance. Building the hierarchy with AABB information is not much more complicated than without (although doing so efficiently takes some extra thought, as detailed below). We can simply let each view sample atomically update all of its parents bounding boxes. As we currently store a full tree, the memory requirements are however significantly increased (see Section 3 for details).

With explicit bounds, the performance of the traversal is dramatically improved (see Figure 8). This is partly due to the traversal kernel becoming much simpler, as we no longer have to calculate the implicit trivial-reject and trivial-accept points for each node, but much more due to the significant decrease in volume of our acceleration structure (see Figure 1c). Previously, a node which only contained a few samples could still be very large, and a shadow-volume that touched that node would have to traverse all of its children. Now, each node will have a bounding box which tightly fits the contained view samples. The number of test-and-set operations required are now almost always fewer than in PTSV, despite our deeper tree. In frames with steep depth discontinuities, we often perform better than a factor of $2\times$ fewer operations (see Figure 6).

2.2 Load Balancing

As illustrated in Figure 7, a small fraction of the triangles may require much more work than the average, which leads to very poor load balancing with our method. To combat this, we simply split the traversal step into two parts. In a first pass, each warp will pick one triangle as before and will then traverse down to a predefined level, L. Every time the traversal reaches that level, it will atomically push a new job onto a list and then return to traverse the rest of the upper part of the tree. This job is simply a tuple (t, k) where t is the triangle ID and k is the key of the node in which traversal was aborted.

In a second pass, each warp will instead pick a job from the newly created list and start traversal where the previous pass left off. Note that we do not have to retrace the path taken to reach the node, but can immediately continue traversal. Therefore, the only additional work that is required by this two-pass method is writing and reading the intermediate job list which fortunately turns out to be quite modest in size even if we let the first pass go all the way down to the leaf-node level. The performance gain from improved load balancing is sometimes very large (as can be seen in Figure 8), and the two-pass method has a much more stable performance.

2.3 Efficiently building the hierarchy

Building the hierarchy in the way described so far (by launching one thread per view sample) leads to very high contention, as many threads will attempt to update the same node. We also perform a lot of redundant work, as several threads will consider view samples that lie in the same cluster and will all mark that cluster as existing. In fact, after having implemented the optimizations described above, building the hierarchy is actually often the most time-consuming part of the algorithm. We reduce the amount of redundant work by launching one warp per 8×8 pixel tile. While all view samples in this tile *could* lie in different clusters, they will usually occupy only a few. The threads within the warp will cooperate in finding the bounding box of each occupied cluster, and then, a single thread can atomically update the nodes in global memory. To reduce the

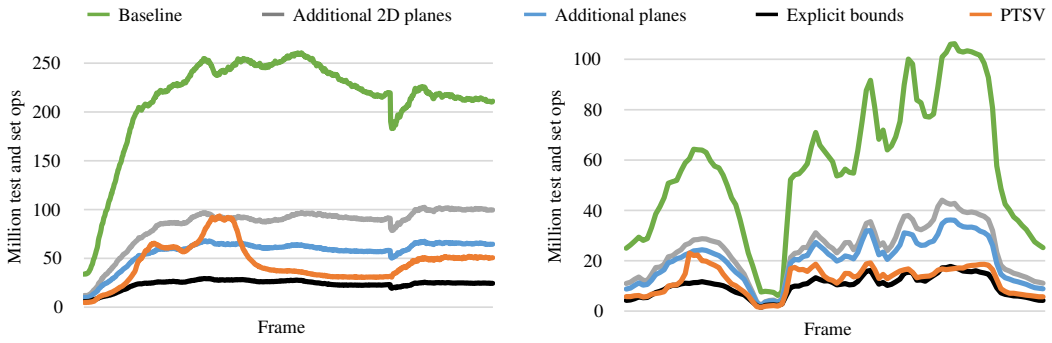


Figure 6: *Number of test-and-set operations required by various versions of our algorithm and by PTSV, for animations in two different scenes (left is* VILLA *and right is* CITADEL*). Culling of shadow-casting triangles that face the light source is enabled in these experiments.*

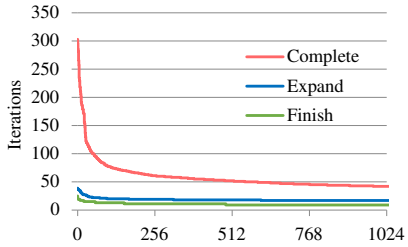


Figure 7: *Number of traversal iterations performed in the 1024 largest jobs. The red line shows the original algorithm where each triangle is processed by a single warp. When we turn to a two-pass algorithm, no single warp will have to do many more iterations than the average.*

amount of contention in the top of the tree, we perform the hierarchy construction in two passes. A first pass will build the tree up to an intermediate level (as described above), and in a second pass, we launch a warp per node at this intermediate level to perform the atomic insertion of bits, and updating of AABBs, upwards to the root. As can be seen in Figure 5, building the hierarchy is now extremely fast. In fact, it takes about the same time as building the hierarchical depth buffer in PTSV.

3 Discussion and Results

We have measured the performance of our algorithm using animated fly-throughs of three different scenes. CITADEL is the scene used to measure performance in the PTSV paper, with 60k triangles. VILLA is a scene designed to stress shadow volume algorithms, with 88k triangles. Finally, FUZZY is a more complex scene with 400k very dense shadow-casting triangles. All experiments were done on an NVIDIA Titan GPU with a screen resolution of 1024×1024 unless otherwise stated. In Figure 5, we show the total time taken by each part of our algorithm, and we can see that we pay a fairly low and constant price for building the hierarchy (Build Hierarchy), calculating the shadow volume planes (Triangle Setup) and finding view samples that have been trivially accepted as part of some node (Merge TA). The cost for traversing the shadow volumes against the hierarchy is naturally view dependent, but the worst-case performance seems to be much more stable than in previous work. In the same plot, we see the performance of the PTSV algorithm and our carefully tuned ZPASS implementation. We can see that our algorithm outperforms both, except in views where there are very few shadow volumes on screen, where ZPASS performs better. It is interesting to note that, unlike the measurements provided in the PTSV paper (which were done on a GTX480 GPU), PTSV seems to perform as well as ZPASS throughout the CITADEL sequence. This is probably due to ZPASS being entirely limited by the pixel throughput of the GPU, which has not changed much between the GTX480 and the Titan GPUs. We measured the performance of our ZPASS implementation on a GTX480 as well and found that performance was indeed very similar.

As in PTSV, we can use *Front Face Culling (FFC)* of shadow-casting triangles when objects are closed, which improves performance significantly. In Figure 5, we also present the total time taken by our algorithm when this optimization is turned off. All timings reported for PTSV are as obtained with the optimization applied.

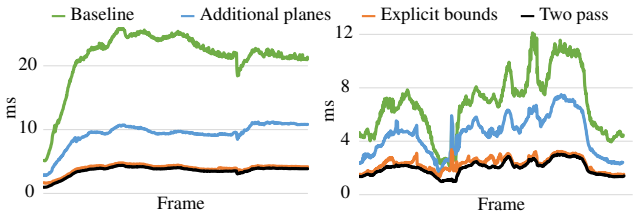


Figure 8: *Time taken by the traversal part of our algorithm when applying different optimizations (left is* VILLA *and right is* CITADEL*).*

In Figure 8, we show the time taken by the traversal stage of our algorithm with different optimizations applied. The Baseline is where we only test implicit bounding boxes against the basic shadow-volume planes. We then add the additional culling planes, as described in Section 2. We improve performance significantly again by calculating the explicit bounding boxes for each node, and finally remove spikes that are due to poor load balancing by performing the traversal in two passes.

The problem with load balancing is illustrated in Figure 7. The lines here show the number of iterations performed in the traversal algorithm for the 1024 most demanding jobs when rendering the view shown in Figure 1. For the original algorithm (Complete), we can see that there are a few triangles that have much more work to do ($\sim 10\times$ average), and therefore, most multiprocessors will have to idle until one multiprocessor has finished the last of these. When we instead divide the work into two passes (Expand and Finish), the number of iterations are more evenly distributed and we avoid sudden unexpected spikes in rendering time.

We have measured the number of test-and-set operations required by our algorithm, as it provides an implementation-independent metric of the efficiency of the algorithm. In Figure 6, we have plotted this metric for our algorithm (with different optimizations applied) and for the PTSV algorithm. As expected, we see that our algorithm is much less sensitive to high-frequency depth buffers and that it performs an equal amount of work as, or less work than, the PTSV algorithm throughout the animations. While the improvement from our additional culling planes is modest in the CITADEL scene, it helps significantly in the VILLA scene, which contains many more small shadow casting triangles.

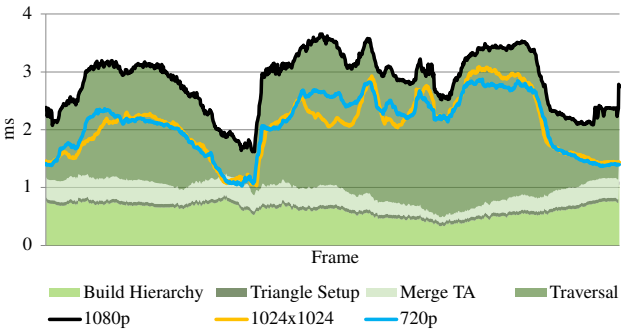


Figure 9: *Time taken by various parts of our algorithm when rendering the* CITADEL *animation to resolution 1080p and the total time when rendering to resolutions 720p and 1024×1024.*

When rendering to higher resolutions, we must make a few small changes to the algorithm. First, our implementation as described so far has fairly high memory requirements due to storing bounding boxes for the hierarchy as a full tree. With a maximum resolution of 1024×1024, a cluster size of 8×8, and with the maximum number of z' bits in the cluster key set to 9, the total number of nodes in a full tree is 8.65 million. The hierarchy information, with a single bit per node, costs only about 1 MB of memory. But if we store six floats per bounding-box, the cost is 198MB, which might be too much in some cases. We have implemented an alternative version, where we only store a pointer per node (33MB) and a compact list of only the existing bounding boxes. The compact list never exceeds 52000 elements in our tests (but the worst case size is 1 million elements), and this version runs almost as fast as the original (at most 5% slower). With this alternative version, then, the expected working memory requirements are around $(33 + 1)$MB and worst case is $(33 + 25)$MB. We can now support a full HD resolution with expected $(66 + 2)$MB and worst case $(66 + 50)$MB.

Secondly, to support resolutions larger than 1024 pixels wide or high, we must store a longer morton key than the one described in Figure 2. When rendering to 1080p, we simply add one bit each for the x, y and z components, which increases the largest supported resolution to 2048×2048, with the same depth range. With a 32-bit morton code we could support resolutions up to 8192×8192.

In Figure 9 we show the same timings as in Figure 5, but when rendering to a resolution of 1920×1080. For reference, we have also plotted the total time taken when rendering to resolutions 1280×720 and 1024×1024. All timings use the more compact memory layout for bounding boxes described above. As expected, building the hierarchy takes approximately twice as long as for the lower resolutions, since it will contain roughly twice the number of nodes. Traversal scales much better however, since many more nodes can be trivially accepted. The slight differences in running time when comparing the two lower resolutions are mostly due to the images having different horizontal field-of-views.

Acknowledgements

This work was supported by the Swedish Foundation for Strategic Research under Grant RIT10-0033. The TITAN GPU used for this research was donated by the NVIDIA Corporation. The CITADEL scene is a part of the Epic Citadel level distributed with the Unreal Development Kit by Epic Games.

References

AILA, T., AND AKENINE-MÖLLER, T. 2004. A hierarchical shadow volume algorithm. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conf. on Graphics hardware*, HWWS '04, 15–23.

AILA, T., AND LAINE, S. 2004. Alias-free shadow maps. In *Proc. of EGSR 2004*, 161–166.

BILODEAU, W., AND SONGY, M., 1999. Real time shadows. Creativity 1999, Creative Labs Inc. Sponsored game developer conferences, Los Angeles, California, and Surrey, England.

CARMACK, J., 2000. Z-fail shadow volumes. Internet Forum.

CHAN, E., AND DURAND, F. 2004. An efficient hybrid shadow rendering algorithm. In *Proc. of the EGSR*, 185–195.

CROW, F. C. 1977. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph. 11* (July), 242–248.

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

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.

HEIDMANN, T. 1991. Real shadows, real time. *Iris Universe 18*, 28–31. Silicon Graphics, Inc.

JOHNSON, G. S., LEE, J., BURNS, C. A., AND MARK, W. R. 2005. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. on Graphics 24*, 4, 1462–1482.

LAINE, S. 2005. Split-plane shadow volumes. In *Proceedings of Graphics Hardware 2005*, Eurographics Association, 23–32.

LEFOHN, A. E., SENGUPTA, S., AND OWENS, J. D. 2007. Resolution matched shadow maps. *ACM TOG 26*, 4, 20:1–20:17.

LLOYD, B., WEND, J., GOVINDARAJU, N. K., AND MANOCHA, D. 2004. Cc shadow volumes. In *EGSR/Eurographics Workshop on Rendering Techniques*, 197–206.

MORTON. 1966. A computer oriented geodetic data base and a new technique in file sequencing. Tech. Rep. Ottawa, Ontario, Canada.

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 *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGGH-HPG'12, 87–96.

SINTORN, E., EISEMANN, E., AND ASSARSSON, U. 2008. Sample-based visibility for soft shadows using alias-free shadow maps. *CG Forum (EGSR 2008) 27*, 4 (June), 1285–1292.

SINTORN, E., OLSSON, O., AND ASSARSSON, U. 2011. An efficient alias-free shadow algorithm for opaque and transparent

objects using per-triangle shadow volumes. *ACM Trans. Graph. 30*, 6 (Dec.), 153:1–153:10.

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

ZHANG, F., SUN, H., XU, L., AND LUN, L. K. 2006. Parallel-split shadow maps for large-scale virtual environments. In *Proc. of the 2006 ACM international conf. on Virtual reality continuum and its applications*, VRCIA '06, 311–318.

A Appendix

Here, we provide pseudocode that describes the traversal algorithm suggested in this paper. This code is run cooperatively by all threads in a warp. The TRAVERSENODE procedure is started for each triangle shadow volume (SV), with *level* and *key* initially set to 0, and *childMask* set to the root node childMask. MAXLEVELS is the total number of levels of the tree (6 with our layout for 1024×1024).

Algorithm 1 Pseudocode describing how the view-sample cluster hierarchy is traversed for a triangle's shadow volume. The symbols $\gg$ and $\ll$ denote right and left shift. & denotes bitwise AND. ! denotes bitwise invert.

```
1:  procedure TRAVERSENODE(SV, level, key, childMask)
2:      if level = MAXLEVELS-1 then
3:          TESTVIEWSAMPLES(SV, key)
4:          return
5:      nodeBitPos ← (MAXLEVELS − level + 1) ∗ 5
6:      childBitPos ← (MAXLEVELS − level) ∗ 5
7:      childKey ← key|(laneId ≪ childBitPos)
8:      intersect ← true
9:      trivialAccept ← true
10:     if TESTBIT(laneId, childMask) then
11:         for each plane in SV do
12:             p ← CLIPSPACETRPOINT(childKey, plane)
13:             if p dot plane > 0 then
14:                 intersect ← false
15:                 trivialAccept ← false
16:                 break
17:             p ← CLIPSPACETAPOINT(childKey, plane)
18:             if p dot plane < 0 then
19:                 trivialAccept ← false
20:     intersectionResult ← BALLOT(intersect)
21:     TAResult ← BALLOT(trivialAccept)
22:     if laneID = 0 and TAResult ≠ 0 then
23:         nodeIdx ← key ≫ nodeBitPos
24:         ATOMICAND(hierarchy [level] [offset], !TAResult)
25:     childMask ← intersectionResult&!TAResult&childMask
26:     while childMask ≠ 0 do
27:         nextChild ←31-CLZ(childMask)
28:         nextKey ← key|(nextChild ≪ childBitPos)
29:         nextNodeIdx ← nextKey ≫ childBitPos
30:         nextChildMask ← hierarchy [level+1] [nextNodeIdx]
31:         TRAVERSENODE(SV, level+1, nextKey, nextChildMask)
32:         UNSETBIT(nextChild, childMask)
```

TESTVIEWSAMPLES is run when the final level is reached and simply tests the individual view samples of a tile in parallel. CLIPSPACETRPOINT and CLIPSPACETAPOINT find the trivial-reject and trivial-accept points respectively of a node's AABB. When using implicit bounds, this is done by calculating the AABB from the provided key, and with explicit bounds, the key is used to look up a bounding box in memory. TESTBIT(a, b) returns true if bit a is set in word b. UNSETBIT(a, b) clears bit a in word b. CLZ(a) is the CUDA intrinsic that counts leading zeroes in word a. BALLOT(a) is the CUDA warp vote function, which returns a word with bit b is set if a is *true* for thread b.