# Using Annotated KD-Trees to Accelerate Shadow Ray Queries

Peter Djeu* Stan Volchenok†

The University of Texas at Austin

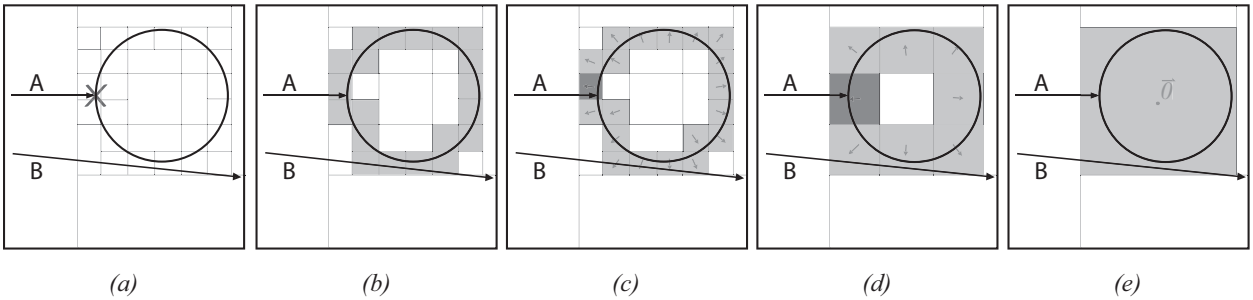


Figure 1: A 2D kd-tree built over a circle. A grid-like kd-tree is shown for clarity, although in practice the tree is less uniform. Nodes that are used as voxel proxies appear darker. (a) Standard kd-tree intersection. (b) The kd-tree is annotated with a boundary bit that indicates the presence of voxel proxies. (c) The kd-tree is further annotated with average normals. (d) The annotations exist within higher levels of the kd-tree. (e) The average normal can indicate that the voxel proxy is not appropriate for intersection.

**ABSTRACT**

We present preliminary work on a new algorithm to accelerate shadow ray visibility queries by using a volumetric proxy representation of the original shadow caster, which we assume is specified as a polygonal mesh. The algorithm creates this proxy representation by building an annotated kd-tree over the mesh. Certain nodes in the kd-tree are marked as *voxel proxies* for the original mesh based on a classification heuristic. These voxel proxies are used in tandem with the original mesh to accelerate shadow ray queries while maintaining high shadow quality. In addition to proposing the algorithm, we present results on the importance of using an appropriate surface area metric during build and extend the approach to use average normals in order to reduce blockiness artifacts along the silhouette. Finally, we present some static measurements of the required computation which indicate that this work is a promising, although not-yet-fully-realized approach to improving real-time shadow ray evaluation.

**Index Terms:** I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Boundary representations; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

## 1 INTRODUCTION

A ray tracer must evaluate shadow ray queries in order to produce a visually plausible image. Performance is often bottlenecked, however, by these shadow ray queries due to their volume and incoherent nature. To alleviate these problems, ray tracers employ acceleration structures. KD-trees are one such acceleration structure, and they accelerate the process of ray-triangle intersection by sorting the scene geometry into an adaptive partitioning of space. In this paper we present a new use for kd-trees: they can represent a coarsified version of a polygonal mesh in addition to serving their original purpose. We present preliminary work on an algorithm designed to accelerate the evaluation of shadow ray queries by approximating the mesh as opaque voxels within an annotated kd-tree. We refer to these opaque voxels as *voxel proxies* of the original mesh. The voxel proxies have the same spatial extent as nodes within the kd-tree, meaning that unlike the faces of the original mesh, the proxies are volumetric in nature.

An annotated kd-tree is created from a polygonal mesh representing a single closed object in the scene (such as a teapot). The voxel proxies are then used in tandem with the original polygonal mesh to accelerate shadow ray visibility queries. Contrast this technique with the traditional way of evaluating shadow ray queries, which involves traversing shadow rays through the kd-tree and then intersecting them with faces from the polygonal mesh. We propose intersecting the shadow rays whenever appropriate with the voxel proxies (which are just nodes from the kd-tree) in lieu of intersecting against the actual geometry. When this substitution is not appropriate, the algorithm falls back on ray-triangle intersection. When this substitution is appropriate, the annotated kd-tree provides an opportunity for even faster, coarser approximations. The tree represents a hierarchy of voxel proxies, meaning that interior nodes in the tree can also be used as proxies, rather than just the leaves.

Like most methods dealing with coarsification, this algorithm involves approximation which can produce shadowing artifacts (e.g. blockiness), thereby reducing the *quality* of the resulting shadows. We will discuss two important considerations that affect the quality of the voxel proxy shadows.

The first consideration is whether the kd-tree can be used effectively as a representation of voxel proxies. Certain trees are more suited than others based on the heuristics used in their build. We will examine the effect of two common kd-tree build heuristics as they relate to blockiness in the resulting annotated kd-tree.

The second consideration is deciding when an annotated kd-tree node can be used as a voxel proxy. We will discuss two simple heuristics for making this decision. The first is the naïve heuristic which assumes that all proxies are usable. The second heuristic determines the usability of a proxy based on the average normal associated with the annotated node. While more expensive, this heuristic leads to a substantial increase in the quality of the shadows.

Many of the ideas in this paper can be applied in a very straight-

*e-mail: djeu@cs.utexas.edu

†e-mail: stanv@cs.utexas.edu

forward manner to other ray tracing acceleration structures, for example annotated bounding volume hierarchies (BVHs) and annotated uniform grids. One of our key contributions is to suggest that an acceleration structure, which a ray tracer has readily available, can be given new uses, such as representing a coarsified version of a mesh. The additional use of the acceleration structure leverages much of the work that was already invested in its creation, such as the determination of voxels that tightly bound subsections of the mesh in the case of kd-trees and BVHs.

Finally, we present initial results on the appropriateness of this algorithm for use in real-time ray tracing. Although this paper does not provide a final validation via real-time performance, it does study the savings in computation, measured in operation count. In particular, it compares the number of operations needed when using voxel proxies versus the number of operations needed when using normal ray-triangle intersection. It should be emphasized that this paper only explores the potential for real-time performance gains. Future work is needed to fully validate this technique.

## 2 Related Work

The primary inspiration for this work is Dammertz and Keller's work in improving the numerical robustness of ray-surface intersections [1]. Their technique computes a 3D interval on which the ray-surface intersection must lie. They recursively refine this intersection interval until the low and high bounds of the interval no longer change or are degenerate, at which point they return an intersection value. The primary focus of their work is to guarantee that free-form surfaces are sufficiently refined and remove self-intersection artifacts by narrowing their intersection intervals until they approach machine precision. Our goal, on the other hand, is to accelerate ray-surface intersection while maintaining reasonable shadow quality. Unlike Dammertz and Keller, who refine their input geometry towards machine epsilon granularity, we preserve the input mesh as the finest level of detail. Were we to perform refinement on the input mesh, we would only increase the number of geometric components that would need to be considered during intersection.

Other work has used volumetric proxies to produce expensive rendering effects at real-time rates on the GPU. Shanmugam and Arikan use spherical proxies to compute the low-frequency component of ambient occlusion induced by distant occluders [14]. Sloan et al. use similar spherical proxies to produce soft shadows and indirect illumination in addition to ambient occlusion [15]. Our work differs from this category of previous work in that we embed our proxies directly into the ray tracing acceleration structure and determine the size and placement of the proxies by leveraging the work needed to construct the acceleration structure in the first place. Our technique has an added benefit of being able to transition between ray-proxy intersection and ray-triangle intersection when necessary.

Other work has used proxies within the context of a ray tracing acceleration structure. The R-LOD technique of Yoon et al. replaces the geometry in each node with a single plane created using principle component analysis on the original geometry [16]. Light bleeding can occur because the planes in each node are derived independently and may not line up with one another. Lacewell et al. replace the geometry in each acceleration structure node with a prefiltered representation of opacity in order to render soft shadows and ambient occlusion for meshes with many pieces of partially transparent, uncorrelated geometry (i.e. foliage and hair) [9]. Their opacity proxies are based on the assumption that geometry between nodes is uncorrelated, which could also lead to light leakage. In contrast, our work is primarily concerned with producing leakage-free hard shadows while studying and minimizing the artifacts that arise when using proxies for hard shadows, particularly along the silhouette.

Another approach is to use 2D rather than 3D proxies to accelerate intersection. Popescu et al. use billboards and depth maps to render reflections at interactive rates [12]. Objects that cast a reflection are converted into 2D proxies which are quite cheap to intersect. It would be interesting to compare in future work this type of proxy with our approach.

This work shares many similarities with level of detail techniques. The voxel proxies in the annotated kd-tree can be viewed as a coarsification of the original mesh. However, unlike a traditional level of detail algorithm, which converts a polygonal mesh into a coarser polygonal mesh (discussed in detail in Luebke et al [10]), this work converts a polygonal mesh into a volumetric representation.

## 3 Creating the Voxel Proxies

The first step in accelerating shadow queries is to create the voxel proxies, which are actually nodes within an annotated kd-tree. The kd-tree is built over a mesh of a single closed object in the scene (such as a teapot), (Figure 1.a). This kd-tree includes only polygonal faces from the object; it ignores geometry from all other objects in the scene. A scene graph can easily provide the semantically meaningful grouping of faces into objects [2].

The kd-tree is built in the traditional way using the surface area heuristic down to the leaves [3, 11, 4]. We would like to reduce shadow artifacts by using tight-fitting kd-trees (see the next section) so we use an exhaustive kd-tree builder which considers all split candidates along all three axes.

After the kd-tree has been built, another pass is made which marks the nodes that can be used as voxel proxies for the original mesh. This extra annotation is in the form of a *boundary bit* that is set when a node belongs to the boundary represented by the polygonal mesh (recall that a polygonal mesh is a boundary representation that partitions space into three sets: interior, exterior, and boundary). The node's boundary bit is set based on the following simple recursive rules:

1. If a node is a leaf, set the boundary bit if the node contains any triangles.

2. Otherwise, set the boundary bit if one or both children have their boundary bit set.

A kd-tree created in this way can be seen in Figure 1.b, where the lightly shaded kd-nodes have their boundary bit set. These are the nodes that represent potential voxel proxies.

## 4 The Importance of KD-Tree Tightness

Creating an annotated kd-tree produces a serviceable voxelization of the original polygonal mesh. However, this voxelization will not always produce *high quality* shadows, where quality is inversely proportional to the number of shadow artifacts. The reason why shadow quality suffers is because the nodes that serve as the voxel proxies do not tightly enclose their contents. Figure 2 shows an example of what happens when the kd-tree is built using the basic, unmodified Surface Area Metric [3, 11, 4]. Notice that certain kd-nodes deviate significantly from the silhouette of the bunny formed by the remaining nodes.

The problem is that the voxel proxies in Figure 2 do not tightly bound their contents. Specifically, there is empty space in these proxies that has not been isolated by a kd-split. This problem can be mitigated by introducing the empty space bonus to the surface area metric [7]. We use a multiplier of 0.85x to reduce the cost of a split that produces an empty node. When this multiplier is used, the results improve dramatically, as seen in Figure 3.

In another attempt to improve the tightness of the kd-tree's voxel proxies, we also build trees that use a correction term to the Surface

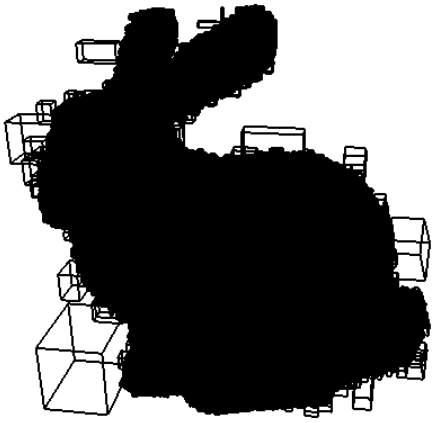

Figure 2: KD-tree built for the 69k Bunny using the Basic SAM. Only the kd-nodes that contain geometry are shown.

Figure 3: KD-tree built for the 69k Bunny using the SAM with the Empty Space Bonus. Only the kd-nodes that contain geometry are shown.

Area Heuristic to account for mailboxing [5] due to similarities between using mailboxing and using voxel proxies. We found, however, that this did not provide as much improvement as the empty space bonus.

In the results section, we will provide a measure of the tightness of annotated kd-trees by counting the number of leaves in the tree that do not tightly bound their contents. We introduce the term *loose leaf* to refer to such a node. A kd-tree with many loose leaves will produce a shadow that appears much blockier.

## 5 Modifying Ray Traversal to Use Voxel Proxies

In order to use the voxel proxies within the annotated kd-tree, it is necessary to modify kd-tree traversal. We start with standard kd-tree traversal for shadow rays:

```
bool traverseP(Ray r, KdNode n, Interval i) {
    // MODIFICATION POINT 1
    if (n.isLeaf()) {
        // MODIFICATION POINT 2
        return isectP(r, n.geometry, i);
    }

    // find the ray-plane intersection
    float tSplit = isect(r, n.splitPlane);

    // subsequent traversal of child nodes
    // is unchanged
}
```

We have implemented two modifications to traversal to incorporate voxel proxies. In both methods, a decision criterion (i.e. heuristic) is introduced to determine which voxel proxies, if any, should be used to occlude the shadow ray. The first such method involves a simple and fast heuristic that chooses any leaf of the annotated kd-tree that has its boundary bit set. The second method involves using the average normal within a node to determine whether that node is an appropriate voxel proxy. This second technique is more expensive, but produces images with fewer artifacts.

### 5.1 Heuristic 1: Boundary Bit Set at the Leaves

Perhaps the simplest way to incorporate voxel proxies into traversal is to use all of them when the ray is at a leaf in the kd-tree. To do so, a check is performed on whether the boundary bit is set when a ray enters a leaf. The line right after "Modification Point 2" in the original kd-traversal algorithm is replaced with:

```
return n.boundaryBitSet();
```

Not too surprisingly, this technique produces the blockiness artifacts seen in Figure 4. The geometry has been completely replaced with voxel proxies, which is a gross approximation even at the leaves of the kd-tree. We observe that the shadows have high quality in areas where the geometry is mostly front facing to the shadow ray (mostly back facing is also fine). However, if the ray approaches the geometry at a glancing angle, then the voxel proxy is unable to provide the resolution necessary to distinguish between when the ray hits and when it does not. In this case, intersecting against the regular geometry would be better.

### 5.2 Heuristic 2: Dot Product with the Average Normal

To improve the quality of the shadows, we introduce a new annotation that summarizes the orientation of the geometry within a node. This annotation is then used in a new heuristic based on the dot product to decide which proxies are appropriate for the current ray.

#### 5.2.1 Annotation using average normals

We add a new step to the creation of the voxel proxies. In addition to having a boundary bit, annotated kd-nodes now include a normal vector. This normal is called the *average normal* and is assigned to each node in the kd-tree with the boundary bit set according to the following rule: average together the normals of all faces that are within the bounds of the current kd-node.

These normals can be computed in an efficient manner bottom-up starting from the leaves of the tree. To do so, each node needs to keep track of its average normal and the total number of vectors that are included in the average normal (although this latter value can be discarded after the preprocess phase).

Figure 4: Shadows produced when using Heuristic 1 on the 69k Bunny. The shadow silhouette has a ragged edge because this heuristic emphasizes the spatial quantization of the voxel proxies along the silhouette.

Unlike most computations involving normal averaging, the result of the averaging is not normalized. This means that the average normal may have unit length but is often shorter.

The meshes used in this paper have a fairly uniform tessellation rate across the mesh so we found that using a simple average is good enough for the normal that annotates each kd-node. For meshes with a non-uniform tessellation rate, the face normals can be weighted by the area of the face to account for different faces contributing more or less to occlusion. Using a weighted average will also require clipping or a remeshing of the larger faces so that they do not overcontribute to nodes with which they have only partial overlap.

#### 5.2.2 Dot product test

We now propose a new heuristic for selecting when a voxel proxy can occlude the current ray. We take the dot product of the current unit ray direction with the voxel proxy's average normal. If the magnitude of this dot product is greater than a threshold value, then the current voxel proxy passes the *dot product test* and should be used to occlude the current ray.

Intuitively, the dot product test is used to find places where the mesh is almost entirely front facing or almost entirely back facing to the incoming ray. In such cases, the voxel proxy is a good proxy for the original mesh, such as the case for Ray A in Figure 1.c. Contrast this situation with Ray B, which passes outside of the object near its silhouette. Incorrectly using a voxel proxy for rays such as B leads to blockiness along the silhouette. However, the dot product test will indicate that the voxel proxy should not be used for Ray B; rather, Ray B should be tested against the actual geometry.

An additional benefit to using the dot product test is that interior nodes in the kd-tree can now be used as voxel proxies. Previously, the use of these interior nodes led to severe blockiness artifacts in the shadow. However, the dot product test indicates when it is appropriate to use these nodes. For example, the large voxel proxy can be used for Ray A in Figure 1.d because it passes the dot product test. The voxel proxy, an interior kd-node, represents a region of the mesh that is mostly front-facing to the ray. Even though there are smaller voxel proxies that could be explored, the larger voxel proxy suffices. Contrast this with Ray B in the same figure, which is still approaching at a glancing angle to the voxel proxies at this level of the kd-tree. Interestingly, annotating kd-tree nodes with the average normal of its geometry can lead to nodes that are assigned the zero vector. This case is illustrated in Figure 1.e. The root node of the kd-tree contains the average normal over the entire sphere, which amounts to the zero vector. No ray is able to use this node as a voxel proxy due to its degenerate average normal.

Setting the threshold value for the dot product test is crucial to determining the quality of the resulting shadows. Although it is difficult to find a value that works well with all scenes, we found that a value of 0.9 produces high quality shadows in the scenes that we tested, in contrast to a threshold value of 0.7, which contains noticeable artifacts (Figures 5 and 6). The dot product threshold also controls how much error appears in the shadow. When it is set to a low value, greater approximations are allowed when determining where the geometry lies (effectively larger nodes in the kd-tree become usable voxel proxies). At the same time, fewer traversal steps and fewer ray-triangle intersections are needed when larger nodes are used as proxies. The dot product threshold thus controls the balance between rendering speed and image quality, and leaving it as a free parameter may be useful. For example, the renderer may choose to adapt the dot product threshold value based on *a priori* knowledge of the scene and the requirements of the rendering.

#### 5.2.3 Modifying traversal for the dot product test

We will add the dot product test to the original, unmodified traversal algorithm in the following way. The following lines of code should be inserted at "Modification Point 1."

```
if (n.boundaryBitSet()) {
    float dotProd = dot(r.dir, n.avgNormal);

    if (abs(dotProd) > DOT_PROD_THRESHOLD) {
        return true;
    }
}
```

When the ray first enters a node, a check is performed to determine whether the kd-node has its boundary bit set (a prerequisite for being a voxel proxy). If so, the dot product test is performed, and if it passes, true is returned to indicate that the proxy occludes the current ray. Notice that the original check for intersecting the actual geometry in leaf nodes is preserved. When using the dot product test, if a usable voxel proxy is not found, the algorithm falls back to intersecting actual geometry. This characteristic allows the algorithm to preserve quality in the silhouettes of shadows.

## 6 Experimental Validation

We will now discuss our implementation of this algorithm and the experimental results gathered. There are two goals to this validation: 1) the algorithm should produce images that are mostly free from artifacts (here we use our subjective judgment, although a more formal method can be used in the future) and 2) the amount of computation performed and saved when using voxel proxies should be promising for future improvements. We would like to emphasize that this work is still preliminary and does not present final run-time speed-up. However, by carefully performing a static analysis of the algorithm, we will have a better understanding of run-time speedup or slowdown in later work.

### 6.1 Methodology

The following results include only measures of computation during traversal. The cost of constructing the annotated kd-tree is not reported in this section because build is considered a preprocess in this work.

| Bunny 69k | nodes | d | leaves | loose leaves | % loose |
|---|---|---|---|---|---|
| Basic SAM | 303531 | 28 | 151766 | 65882 | 43.4% |
| SAM w/ ESB | 363879 | 33 | 181940 | 38607 | 21.2% |
| Dragon 203k | nodes | d | leaves | loose leaves | % loose |
| Basic SAM | 699319 | 31 | 349660 | 147449 | 42.2% |
| SAM w/ ESB | 838463 | 33 | 419232 | 79655 | 19.0% |

Table 1: KD-tree statistics. ESB is empty space bonus. d is depth.

We vary the following experimental parameters. The Surface Area Heuristic used to build the initial kd-tree is first set to the Basic Surface Area Metric (Basic SAM), and then to the Surface Area Metric with an Empty Space Bonus (SAM w/ ESB). We set our empty space bonus multiplier in the SAM w/ ESB to 0.85x whenever a split produces an empty node. Also, during shadow rendering we exclusively use Heuristic 2 (dot product test) because Heuristic 1 produces too many blockiness artifacts. We study the tradeoffs in Heuristic 2 by using dot product threshold values of 0.9 and 0.7.

All images are rendered at 512x512 using the viewpoint shown. Although 512x512 is not a realistic rendering resolution, the results presented in this paper are primarily concerned with percentage savings over the baseline technique of direct geometry intersection. These percentage savings are to a large degree independent of resolution.

We use the Stanford Bunny (Figure 5) and the Stanford Dragon (Figure 6) as the test models.

### 6.2 Analysis of Results

Table 1 shows that the number of loose leaves decreases by roughly a factor of two as the Basic SAM is switched to the SAM with Empty Space Bonus. The kd-tree, in turn, contains voxel proxies that better approximate the original mesh.

Our measure of the number of operations required by this algorithm is included in Table 2. Notice that by using the voxel proxies, the shadow queries can be evaluated using fewer ray-triangle intersections and kd-node traversals. At the 0.9 threshold, the savings are only within the 10% to 15% range for ray-triangle intersection, but using the 0.7 threshold, these savings go up to as high as 30% to 35%. The number of kd-tree nodes traversed decreases negligibly, by only around 1% to 5%. It seems that in general, using the Basic SAM produces more traversal and intersection savings than using the SAM with Empty Space Bonus. The extra savings is most likely attributable to more loose leaves in the tree and hence larger voxel proxies throughout the tree.

Interestingly, models with higher tessellation are more tolerant to a low dot product threshold. Compare Figures 5 and 6 to see that a threshold value of 0.7 causes noticeable artifacts in the Bunny (69k triangles) but not the Dragon (203k triangles). We suspect that highly tessellated models respond better to this approximation technique because they produce much tighter kd-trees.

### 6.3 Negligible Temporal Artifacts

A very real concern when using voxel proxies is the presence of temporal artifacts, such as popping along the shadow silhouette when the position of the light moves relative to the shadow caster. We demonstrate in the associated video that a dot product threshold of 0.9 creates a voxel proxy shadow that is nearly indistinguishable from the ground truth generated via ray-triangle intersection.

## 7 Summary

It is important to realize that this work is not the final word on this topic, but rather an initial foray into a new avenue of research. Here is a summary of the things that have been learned in this work. The SAM with the empty space bonus seems to produce a noticeable increase in shadow quality at low additional cost and minimal algorithmic change. Using Heuristic 2 (dot product test) is also a promising approach. It leads to significantly fewer artifacts in the resulting image and provides a decision criterion for when to use voxel proxies that are interior to the kd-tree. Empirically, more highly tessellated models still produce high quality images even when the dot product threshold is decreased. For instance, the Bunny shows quality loss at a threshold of 0.7 while the Dragon does not. The algorithm's robustness when dealing with highly tessellated models such as the Dragon is encouraging because these models stand to benefit the most from a technique such as voxel proxies.

## 8 Future Work

It remains to be seen whether this technique can be adapted for use in real-time ray tracing. Combining the static counts that are presented in this work with measured run-time latencies should provide a good idea of whether this technique can be made fast enough. From here, it is a matter of engineering the code to achieve real-time performance.

Another avenue of future work is to build or update the annotated kd-tree every frame to support dynamic meshes. The recent advances in fast kd-tree building suggest that such a task is possible [6, 13]. Additionally, since the algorithm operates only on one mesh at a time, it is very amenable to lazy build of the kd-tree [2]. New annotated kd-trees can be built or updated for only the meshes that interact with the current frame. One glaring problem which needs to be addressed, however, is whether computing the average normal within a kd-tree node can be done quickly enough for real-time use.

A big component that is missing from this work is a mechanism to account for acceptable error once the shadow becomes small in screen space. This paper explores the cost of shadows that take up a majority of the screen, in which case image quality must be kept high. However, the real benefit to approximating the mesh's shape with voxel proxies comes when some amount of error is acceptable. By combining lazy build with the contribution that the shadow query will have on the final image (for example, via ray differentials [8]), it might be possible to build a much shallower annotated kd-tree with voxel proxies that are coarser but still sufficient for the current frame. In these cases a simpler heuristic for choosing proxies (such as Heuristic 1) may produce acceptable results.

This paper also introduces a framework to explore new ideas in using volumetric proxies for ray traced shadows. We present only a small set of techniques and heuristics in this paper; many others are possible.

## Acknowledgements

The authors wish to thank Warren Hunt and Sean Keely for insightful discussion and help with implementing the rendering platform. The models were provided courtesy of the Stanford 3D Scanning Repository. This work was supported in part by NSF CAREER award #0546236 and by a research grant from Intel Corporation.

## References

[1] H. Dammertz and A. Keller. Improving ray tracing precision by object space intersection computation. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, September.

[2] P. Djeu, W. Hunt, R. Wang, I. Elhassan, G. Stoll, and W. R. Mark. Razor: An architecture for dynamic multiresolution ray tracing. Technical Report TR-07-52, The University of Texas at Austin, Department of Computer Sciences, January 24 2007.

[3] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, 1987.

[4] V. Havran and J. Bittner. On improving kd-trees for ray shooting. *Journal of WSCG*, 10(1):209–216, February 2002.

| Bunny 69k, Basic SAM | Normal shadows | 0.9 Voxel Proxies | % Improvement | 0.7 Voxel Proxies | % Improvement |
|---|---|---|---|---|---|
| ray-tri isects | 649125 | 588664 | 9.3% | 425885 | 34.4% |
| kd-nodes traversed | 5793631 | 5721574 | 1.2% | 5441030 | 6.1% |
| boundary bit checks | 0 | 5721574 | N/A | 5441030 | N/A |
| dot product compares | 0 | 4871535 | N/A | 4615918 | N/A |
| Bunny 69k, SAM w/ ESB | | | | | |
| ray-tri isects | 551089 | 508665 | 7.7% | 397811 | 27.8% |
| kd-nodes traversed | 5597824 | 5584896 | 0.2% | 5527477 | 1.3% |
| boundary bit checks | 0 | 5584896 | N/A | 5527477 | N/A |
| dot product compares | 0 | 4634215 | N/A | 4579346 | N/A |
| | | | | | |
| Dragon 203k, Basic SAM | Normal shadows | 0.9 Voxel Proxies | % Improvement | 0.7 Voxel Proxies | % Improvement |
| ray-tri isects | 511291 | 442111 | 13.5% | 331441 | 35.2% |
| kd-nodes traversed | 5392612 | 5332964 | 1.1% | 5124576 | 5.0% |
| boundary bit checks | 0 | 5332964 | N/A | 5124576 | N/A |
| dot product compares | 0 | 4528236 | N/A | 4340596 | N/A |
| Dragon 203k, SAM w/ ESB | | | | | |
| ray-tri isects | 443938 | 387612 | 12.7% | 302779 | 31.8% |
| kd-nodes traversed | 4688839 | 4672679 | 0.3% | 4618153 | 1.5% |
| boundary bit checks | 0 | 4672679 | N/A | 4618153 | N/A |
| dot product compares | 0 | 3926972 | N/A | 3875946 | N/A |

Table 2: Static counts. ESB stands for empty space bonus.

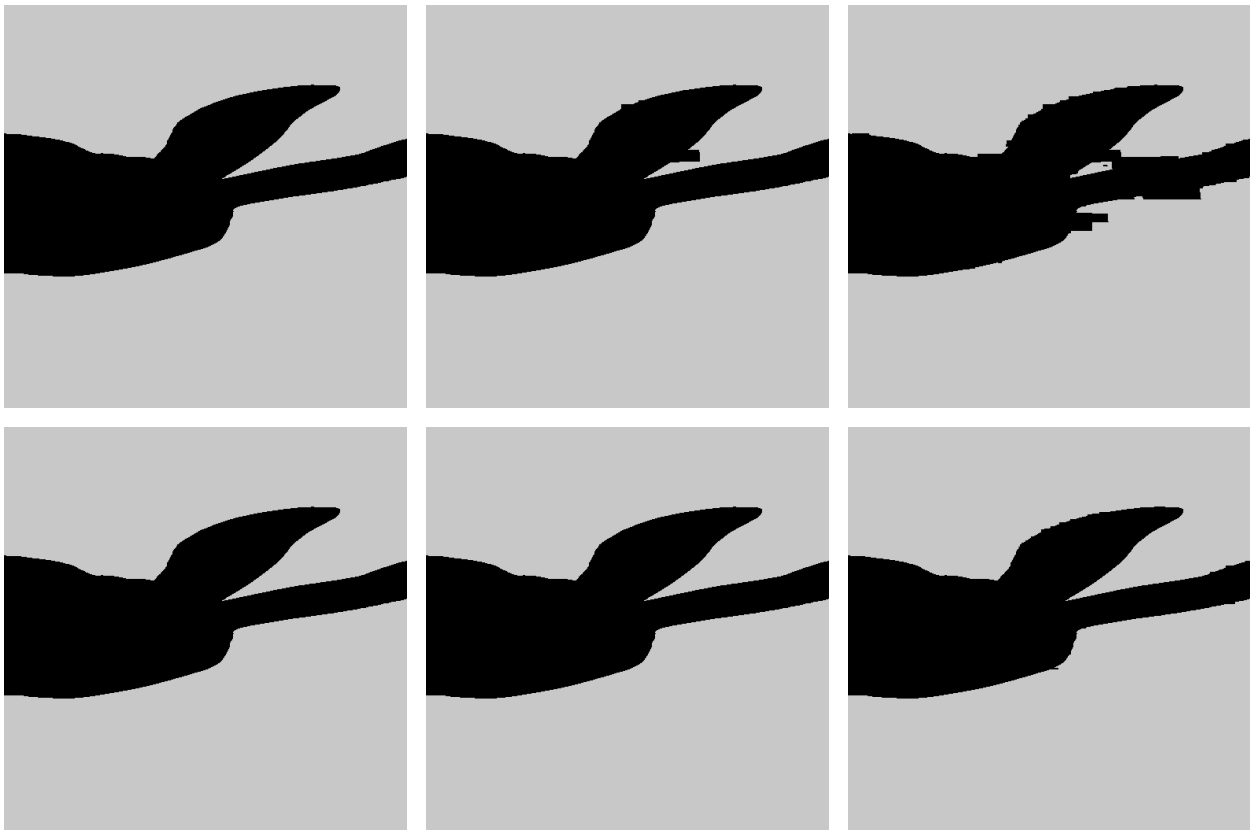

Figure 5: 69k Bunny shadow quality. Top Row: KD-tree built using the Basic SAM. Bottom Row: KD-tree built using the SAM w/ ESB. Left Column: Geometry only (reference). Middle Column: Voxel proxies with a dot product threshold of 0.9. Right Column: Voxel proxies with a dot product threshold of 0.7. Note that the two images in the left column are identical.

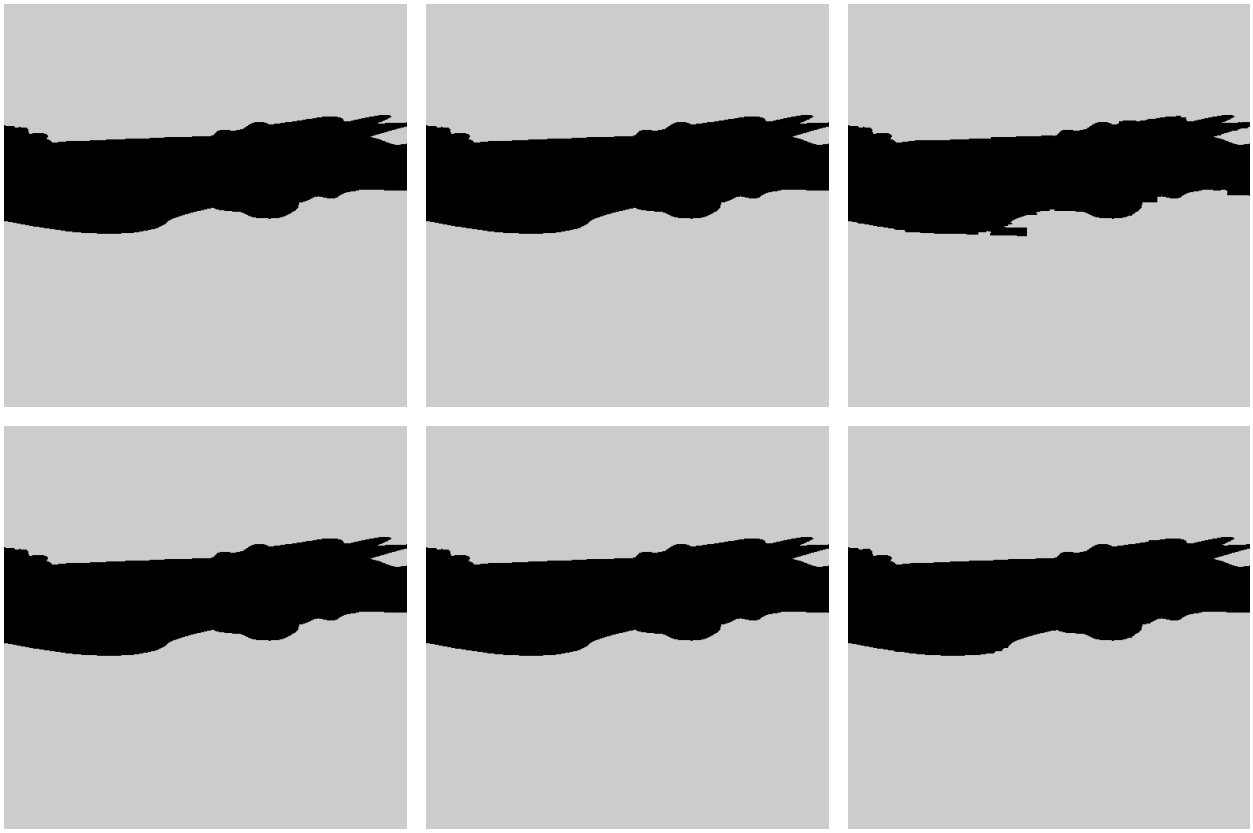

Figure 6: 203k Dragon shadow quality. The layout matches Figure 5.


[5] W. Hunt. Corrections to the surface area metric with respect to mailboxing. In *IEEE Symposium on Interactive Raytracing*, 2008.

[6] W. Hunt, W. R. Mark, and G. Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. In *IEEE Symposium on Interactive Ray Tracing*, pages 81–88, 2006.

[7] J. Hurley, A. Kapustin, A. Reshetov, and A. Soupikov. Fast ray tracing for modern general purpose CPU. In *Graphicon 2002*, 2002.

[8] H. Igehy. Tracing ray differentials. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 179–186, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[9] D. Lacewell, B. Burley, S. Boulos, and P. Shirley. Raytracing prefiltered occlusion for aggregate geometry. In *IEEE Symposium on Interactive Raytracing*, 2008.

[10] D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2003.

[11] D. J. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer: International Journal of Computer Graphics*, 6(3):153–166, 1990.

[12] V. Popescu, C. Mei, J. Dauble, and E. Sacks. Reflected-scene impostors for realistic reflections at interactive rates. In *Computer Graphics Forum, volume 25, issue 3 (EG 2006)*, Sep 2006.

[13] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Experiences with streaming construction of SAH KD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 89–94, 2006.

[14] P. Shanmugam and O. Arikan. Hardware accelerated ambient occlusion techniques on GPUs. In *I3D '07: Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, pages 73–80, New York, NY, USA, 2007. ACM.

[15] P.-P. Sloan, N. K. Govindaraju, D. Nowrouzezahrai, and J. Snyder. Image-based proxy accumulation for real-time soft global illumination. In *PG '07: Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, pages 97–105, Washington, DC, USA, 2007. IEEE Computer Society.

[16] S.-E. Yoon, C. Lauterbach, and D. Manocha. R-lods: Fast lod-based ray tracing of massive models. *The Visual Computer*, 22(9–11):772–784, September 2006.