

# Triangle Reordering for Reduced Overdraw in Animated Scenes

Songfang Han    Pedro V. Sander  
Hong Kong UST



**Figure 1:** Renderings of the four animated characters used in our results. The small images visualize overdraw for several key frames in the animation (dark regions indicate overdraw). Refer to the supplemental video for a full demonstration.

## Abstract

We introduce an automatic approach for optimizing the triangle rendering order of animated meshes with the objective of reducing overdraw while maintaining good post-transform vertex cache efficiency. Our approach is based on prior methods designed for static meshes. We propose an algorithm that clusters the space of viewpoints and key frames. For each cluster, we generate a triangle order that exhibits satisfactory vertex cache efficiency and low overdraw. Results show that our approach significantly improves overdraw throughout the entire animation sequence while only requiring a few index buffers. We expect that this approach will be useful for games and other real-time rendering applications that involve complex shading of articulated characters.

**Keywords:** real-time rendering, depth sort, order-independent transparency

**Concepts:** •Computing methodologies → Image manipulation; Computational photography;

## 1 Introduction

Advanced real-time rendering applications often involve rendering large animated models using complex lighting and shading techniques. Depending of the relative complexity between the rendered geometry and the fragment shading algorithm, the rendering process is often bottlenecked at either the *vertex shader* or *fragment shader* stage. Scenes that fall into one of these categories are referred to as *vertex-bound* and *fill-bound* scenes, respectively. Approaches have been proposed to reorder the triangles of a mesh so as to alleviate these bottlenecks.

In order to reduce vertex computation, the application can leverage the GPU’s post-transform vertex caching mechanism, which stores

the vertex shading output of a small set of recently processed vertices. When processing a particular vertex, recomputation can be avoided if the vertex has recently been processed by an adjacent triangle within the same hardware unit and thus is still cached. This encourages a triangle order with vertex reference locality (i.e., mesh triangles that share vertices should be close to each other in the index buffer). The average cache miss ratio (ACMR) of a particular triangle order measures the ratio between processed vertices and rendered triangles for a given caching scheme (usually a FIFO scheme is assumed). Generating triangle orders that reduce ACMR results in a significant improvement in rendering time for heavily vertex-bound scenes.

Scenes may also have very complex lighting and shading techniques, resulting in computationally intensive fragment shaders. In this case, reducing the number of fragments that need to be shaded can significantly reduce rendering time. When rasterizing triangles, GPUs apply *early-Z culling*, which performs depth testing prior to fragment shading. Thus, if the triangles happen to be processed in perfect front-to-back order, *none* of the occluded pixels will need to be shaded. In the worst case, when rendering in back-to-front order, *all* of the pixels need to be shaded, even those that are completely occluded by subsequent triangles. The overdraw ratio (OVR) of a triangle order refers to the ratio between the total number of pixels that passed the depth test and the number of visible pixels. An overdraw ratio of 1 is optimal and means no overdraw.

It has been shown that for heavily vertex-bound scenes, ACMR is directly proportional to rendering time, while for heavily fill-bound scenes, OVR is directly proportional to rendering time [?]. An efficient triangle order has both low ACMR and low OVR. In this paper we propose a technique that, as in ? and ?, compromises between these two objectives. However, unlike these previous techniques, our approach handles animated meshes. Since we are addressing keyframe animations where mesh connectivity does not change, ACMR is invariant to the animation. On the other hand, since vertices change their relative positions over the course of the animation, OVR can be significantly affected. Our algorithm generates a set of triangle orders that minimize OVR over the entire animation sequence, while still maintaining a low ACMR.

## 2 Related work

**Vertex caching** Vertex cache optimization has been extensively researched. Early techniques made advances by reducing badwidth

and generating compressed data structures, such as triangle strips (e.g., ???). More recent methods simply utilize the transparent caching provided by modern GPUs and just reorder the triangles without further compressing the index buffer (e.g., ???). These approaches directly target the post-transform cache, where most of the vertex processing gain can be achieved. In this paper, we do not propose new methods for improving cache efficiency, but rather directly employ the method of ? to generate mesh patches with low ACMR. We later use these patches in our algorithm to create orders that reduce OVR over entire animation sequences.

**Overdraw** A popular strategy to reduce overdraw of fill-bound scenes is to prime the Z-buffer by rendering the geometry without writing to the framebuffer. On a subsequent pass, the geometry is rendered again, but this time writing to the framebuffer and using a *less than or equal* depth test. This approach ensures that only the visible fragments are shaded. Note, however, that it doubles the amount of vertex processing, which could be unacceptable in many scenarios. Alternative ways to reduce overdraw include visibility sorting and occlusion culling (e.g., ???). Some techniques use hardware-based occlusion queries (e.g., ???). Most of these methods either operate at coarser levels, or require fine-granular visibility sorting. ? and ? take an alternative approach of creating a single index buffer with a view-independent order that is optimized to reduce overdraw. The approach is completely transparent to the application, which simply directly renders this pre-sorted buffer. ? creates set of buffers that guarantee front-to-back order by duplicating triangles in the index buffer and selectively drawing these triangles based on a shader test so as to guarantee that the order of the rendered triangles is correct. While these techniques provide good results for static meshes, they do not address animated scenes. Our proposed technique addresses this problem by jointly clustering sets of animation key frames and viewpoints that can share the same index buffer.

### 3 Our approach

Next, we describe our approach. The algorithm first partitions the mesh into patches that are locally optimized for reduced ACMR. It then generates a set of index buffers that contain different orderings of these patches. These different orders are optimized for reducing overdraw for different keyframes and viewpoints.

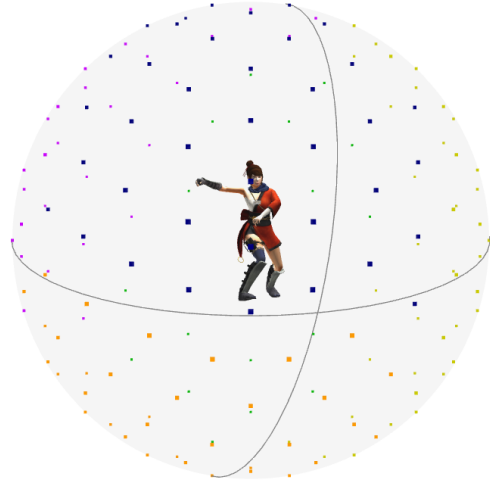
#### 3.1 Generating cache-efficient patches

We follow the *fast linear clustering* approach of ? to quickly generate cache-optimized surface patches of triangles. The basic idea is to first optimize the entire mesh to reduce ACMR, and then break the output index buffer into contiguous triangle sequences, or patches. The approach uses a parameter  $\lambda$  to regulate the resulting ACMR. Essentially, the method traverses the order one triangle at a time, and when the ACMR of the current patch drops below  $\lambda$ , it adds a patch break and starts a new patch on the following triangle. Refer to ? for additional details. Lower values of  $\lambda$  result in lower overall ACMR, however due to the smaller number of patch breaks, it provides less flexibility for patch reordering to reduce overdraw.

#### 3.2 Generating the index buffers

Next, we seek to reorder of these cache-optimized patches for overdraw reduction.

**Viewpoints** We assume that the animated model may be viewed from all directions. We first generate a set  $V$  of 162 viewpoints that lie on a sphere enclosing the model to represent the potential viewing directions (figure 2). The viewpoints are computed by subdividing an icosahedron as in ?. If the viewpoint distribution of the target



**Figure 2:** The points represent the vertices from the subdivided icosahedron that were used as viewpoints during clustering. The colors identify their clusters.

application differs significantly, a specialized set of viewpoints can be generated.

**Clustering** A single order that is suitable for all animation keyframes when viewed from any viewpoint cannot satisfactorily reduce overdraw (see single cluster results in section 4). We instead create  $k$  clusters of (keyframe, viewpoint) pairs that can share index buffers. This results in a total of  $k$  index buffers. At runtime, the rendering algorithm picks the appropriate one based on the current viewpoint and keyframe.

We are given a set of viewpoints  $V$  and a set of keyframes  $F$ . Our approach must consider every possible keyframe viewed from every possible direction. Since  $|V| = 162$  and  $30 \leq |F| \leq 70$  for our example animations, the total number of nodes (i.e., keyframe and viewpoint combinations) is in the thousands.

We seek to find a partitioning of all nodes that yields low overdraw with a small number of index buffers. Our approach is based on k-means clustering [?]. The algorithm alternates between two steps, one which assigns nodes to clusters, and one which computes a new index buffer for each cluster.

**Step 1: Node assignment** Each node is assigned to the cluster whose index buffer results in the lowest overdraw when used to render that keyframe from that particular viewpoint. This is accomplished by rendering the scene using each of the  $k$  candidate index buffers and using hardware occlusion queries to read back the overdraw results.

**Step 2: Index buffer computation** For each cluster, we seek a triangle order with reduced average overdraw for all of its currently assigned nodes. We accomplish this by creating an order that roughly sorts the patches from front-to-back. The problem of sorting from front-to-back is well defined when rendering a single keyframe from a single viewpoint. We can simply sort the patches by increasing distance from the viewpoint. We directly extend this approach to multiple nodes by integrating the computed distance over all nodes in the cluster. We then create a single global order that sorts the patches in front-to-back order based on this distance result. In computing the order, we use the patch’s centroid as its position. Additionally, patches whose triangles are all backfacing are ignored in the distance

computation, and the integrated distance is normalized based on the number of nodes in which it is visible.

**Bootstrapping** The above algorithm is initialized by choosing  $k$  random nodes and creating an initial index buffer ordered front-to-back solely based on those nodes. The choice of  $k$  is discussed in the results section.

### 3.3 Runtime selection

When rendering the model, the target application has to choose between one of the  $k$  index buffers. This is accomplished by using a lookup  $\text{IBid}[f, \theta_v, \phi_v]$  indexed by keyframe  $f$  and viewpoint  $v$ . The vertices from the subdivided icosahedron (i.e., viewpoints) are grouped by their polar angles, resulting in  $n$  lists  $L_i$ , of vertices that share the same polar angle  $\theta_i$  ( $n = 13$  in our case). Within each list, the vertices are sorted by azimuth angle  $\phi$ . Polar angles of the lists are uniformly separated by  $\pi/(n - 1)$ , and the azimuth angles within each list  $L_i$  are uniformly separated by  $2\pi/(|L_i| - 1)$ . Thus, given  $\theta_v, \phi_v$ , we can determine the closest vertex to the queried viewpoint in constant time. The overall time required for this lookup is negligible since it is only a single lookup for each rendered model.

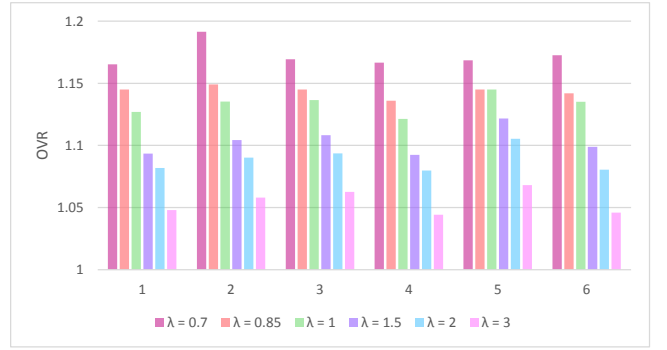
## 4 Results

In this section, we present results of our approach. Our results use four models undergoing a set of complex animations that have between 30 and 70 keyframes. These are representative of animated characters often found in games and other real-time applications. Refer to the supplemental video for a demonstration of these animations. Preprocessing each animation takes approximately half an hour. Note that we did not heavily optimize the preprocessing computation for speed. We focused on reducing overdraw for higher runtime performance on pixel bound scenes.

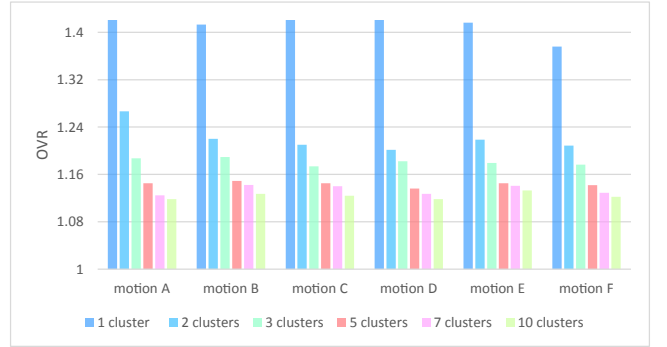
**Choice of  $\lambda$**  As mentioned earlier, rendering time has been shown to be directly proportional to ACMR for vertex-bound scenes, and OVR for pixel-bound scenes [?]. The algorithm trades off these objectives by controlling the desired ACMR through the  $\lambda$  parameter. Figure 3 shows how the overdraw ratio is affected by the choice of  $\lambda$ . We have found that setting  $\lambda$  in the range of 0.75 – 0.95 provides a good balance between ACMR and OVR objectives for our animated scenes. For the remaining results in this paper, we use  $\lambda = 0.85$ . In addition, we also provide results for  $\lambda = 3$ , which is the extreme case where vertex cache performance is not taken into account and results are solely optimized for reduced overdraw.

**Choice of number of clusters** Figure 4 shows results for different number of clusters for the Ganfaul model. Increasing the number of clusters reduces overdraw at the expense of memory to store the additional index buffers. We have found that using more than five clusters only yields modest overdraw reduction for the models and animations that we tested. Thus, for the remaining results in the paper, we use five clusters.

**Overall results** Table 1 shows the statistics of each of our four models. Results are averaged over all keyframes and over a set of 300 random viewpoints within a radius three times that of the object’s bounding sphere. The *original* results use a single index buffer that is optimized solely for reduced ACMR. It has a low memory footprint since it only requires one buffer, however it results in an order with high overdraw. The *tipsify* results use the state-of-the-art algorithm for static scenes described in [?] applied to *each individual frame*. It reduces overdraw significantly, but requires between 30-70 index buffers, depending on the animation, making



**Figure 3:** Adjusting  $\lambda$  provides a tradeoff between vertex caching and overdraw, as shown here for the six animations of the Ganfaul model.



**Figure 4:** The number of clusters trades off memory (one index buffer per cluster) and overdraw, as shown here for the six animations of the Ganfaul model.

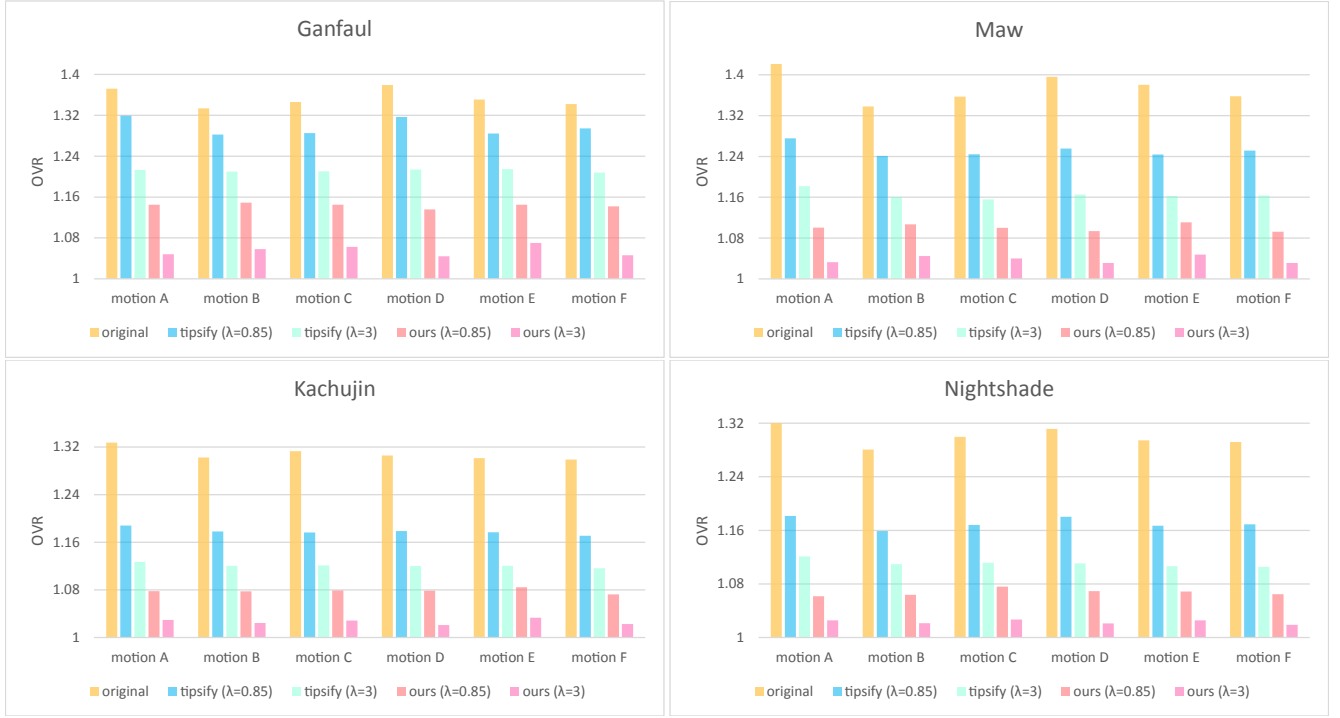
its direct use impractical. Our *single* set of results uses five clusters for each single animation, thus requiring only five index buffers per animation and our cluster-based approach further reduces overdraw significantly for both  $\lambda = 0.85$  and  $\lambda = 3$ . Our *joint* set of results optimize for the same five clusters for all frames in *all* animations. It therefore does slightly worse than the single animation results. As described earlier, the choice of  $\lambda$  affects the resulting ACMR. With  $\lambda = 0.85$ , ACMR is kept under 0.9, while for  $\lambda = 3$ , the order is solely optimized for overdraw, thus ACMR is significantly sacrificed to achieve this additional reduction in overdraw. Figure 5 further breaks down the results for each animation (labeled *motion A-F*). Note that the improvements of our algorithm are consistent over a variety of different character animations and brings the overdraw ratio very close to the optimal value of 1.

## 5 Conclusion

We introduced a new algorithm to efficiently reorder triangles of animated models in order to reduce overdraw. To our knowledge, this is the first technique that generates such optimized triangle orders for animations. By using a small number of index buffers, the proposed approach produces triangle orders that have significantly lower overdraw even when compared to techniques that are optimized for static meshes. We presented results that balance between vertex cache and overdraw performance as well as results that are solely optimized for reduced overdraw. The approach is very general and widely applicable to arbitrary animations in a variety of real-time rendering applications.

**Table 1:** Average cache miss ratio (ACMR) and overdraw ratio (OVR) results for several character animations. We contrast our method with a triangle order that only considers vertex caching (original), and with the approach for static meshes of [1], which applies the algorithm to each frame independently, resulting in 30-70 index buffers per animation (tipsify). We present our results using a separate set of five index buffers per motion (single) and with a joint set of five index buffers for all motions combined (joint).

Model	# tris	Original		Tipsify ( $\lambda = 0.85$ )		Tipsify ( $\lambda = 3$ )		Single ( $\lambda = 0.85$ )		Single ( $\lambda = 3$ )		Joint ( $\lambda = 0.85$ )		Joint ( $\lambda = 3$ )	
		ACMR	OVR	ACMR	OVR	ACMR	OVR	ACMR	OVR	ACMR	OVR	ACMR	OVR	ACMR	OVR
Ganfaul	13,801	0.676	1.355	0.869	1.298	2.939	1.213	0.862	1.144	2.715	1.055	0.860	1.144	2.684	1.064
Kachujin	12,610	0.645	1.310	0.903	1.172	2.896	1.122	0.891	1.068	2.617	1.027	0.887	1.102	2.570	1.030
Maw	13,908	0.619	1.380	0.879	1.253	2.918	1.166	0.860	1.101	2.696	1.038	0.859	1.109	2.675	1.044
Nightshade	12,996	0.656	1.302	0.862	1.172	2.930	1.112	0.854	1.068	2.643	1.024	0.854	1.071	2.612	1.027



**Figure 5:** Average overdraw ratio for different animations of the four models from Table 1.

## Acknowledgments