Triangle Reordering for Efficient Rendering in Complex Scenes

Songfang Han
Hong Kong UST

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.

1. Introduction

Advanced real-time rendering applications often involve rendering large animated models using complex lighting and shading techniques. Depending on the relative complexity of 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 that 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 is 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 fragments will need to be shaded. In the worst case, when rendering in back-to-front order, *all* of the fragments 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 of the total number of fragments that passed the depth test and the number of visible fragments. 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 [Sander et al. 2007]. An efficient triangle order has both low ACMR and low OVR. In this paper, we propose a technique that finds a compromise between these two objectives. However, unlike 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 minimizes 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 bandwidth and generating compressed data structures, such as triangle strips [Akeley et al. 1990; Deering 1995; Chow 1997]. More recent methods simply utilize the transparent caching provided by modern GPUs and just reorder the triangles without further compressing the index buffer [Hoppe 1999; Lin and Yu 2006; Sander et al. 2007]. 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 Sander et al. [2007] 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 [Airey 1990; Teller and Séquin 1991; Greene et al. 1993]. Some techniques use hardware-based occlusion queries [Hillesland et al. 2002; Bittner et al. 2004; Govindaraju et al. 2005]. Most of these methods either operate at coarser levels or require fine-granular visibility sorting. Nehab et al. [2006] and Sander et al. [2007] 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. Chen et al. [2012] create a 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

Our 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 new 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 Sander et al. [2007] 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 λ 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 λ, it adds a patch break and starts a new patch on the following triangle. Refer to Sander et al. [2007] for additional details. Lower values of λ result in lower overall ACMR; however, due to the smaller number of patch breaks, this provides less flexibility for patch reordering to reduce overdraw.

3.2. Generating the Index Buffers

Next, we seek to reorder 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 Nehab et al. [2006]. We can increase the number

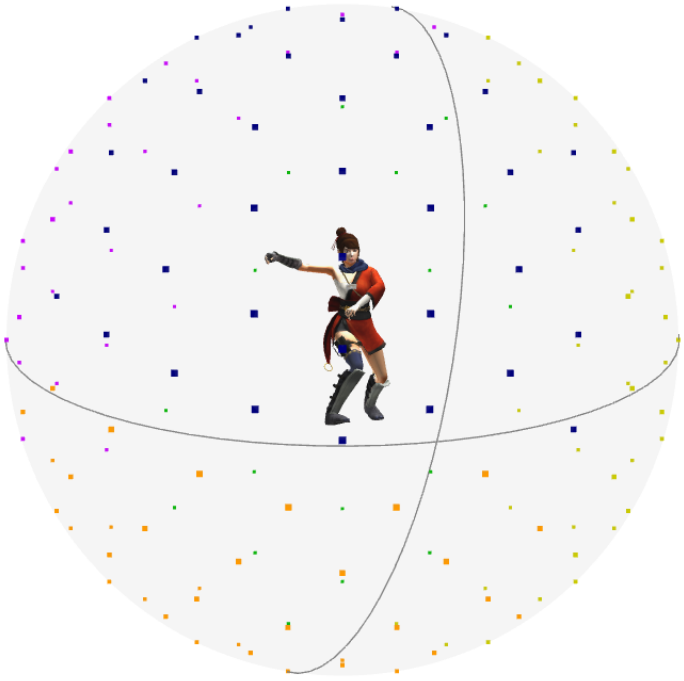

Figure 2. The points represent the vertices from the subdivided icosahedron that were used as viewpoints during clustering. The colors identify their clusters.

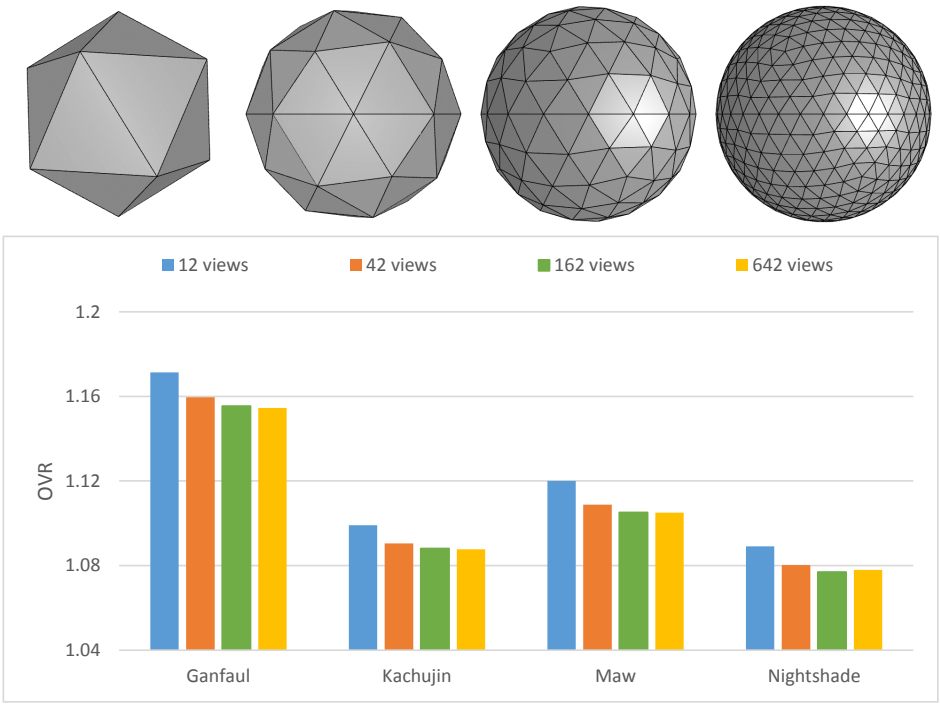

Figure 3. Average overdraw for all 12 animations of all four models using a progressively larger number of viewpoints (shown above the graph). The number of viewpoints trades off preprocessing time and the overdraw result. Our results show that using 162 viewpoints is sufficient and any additional gains in overdraw are marginal.

of viewpoints through further subdivision. More viewpoints will tend to give more accurate results at the expense of additional preprocessing time. From our experiments, 162 viewpoints suffice, and the added processing required by using more viewpoints does not improve the results significantly (less than 0.01 difference in overdraw; see Figure 3). If the viewpoint distribution of the target application differs significantly, a specialized set of viewpoints can be generated.

Framing. Some character animations involve significant global translations (e.g., a running character). Therefore, it is inefficient to use a single bounding sphere of viewpoints that encapsulates the model in all potential frames or poses. In order to maintain viewpoint consistency across frames while preserving a tight bounding sphere of viewpoints, we compute the model's bounding sphere in each frame and translate the model such that its center matches that of the subdivided icosahedron of viewpoints.

Clustering. A particular view configuration consists of a viewpoint and a frame in the animation which we seek to render. For each such configuration, we would like to have an available index buffer that has low OVR for use at runtime. In our algorithm,

we will refer to each such configuration as a $node$. Therefore, we have a node for each possible (keyframe, viewpoint) combination.

A single order that is suitable for all nodes (i.e., all animation keyframes when viewed from any viewpoint) cannot satisfactorily reduce overdraw (see single cluster results in Section 4). We instead create k node clusters 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.

As input, we are given a set of viewpoints V and a set of keyframes F. Since our approach must consider every possible keyframe viewed from every possible direction for our example animations, the total number of nodes is in the thousands ($|V| = 162$, $20 \leq |F| \leq 135$).

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 [MacQueen 1967]. The algorithm alternates between two steps, one which assigns nodes to clusters, and one which computes a new index buffer for each cluster.

Bootstrapping. The algorithm is initialized by choosing k initial viewpoints and creating an initial index buffer for each of them by sorting the patches in front-to-back order (i.e., by increasing distance between the patch centroid and the viewpoint). The patch positions used for sorting the patches are the average positions over all of the frames in the animation sequence. These k buffers represent our initial k clusters. The choice of k and the initial viewpoints are discussed in the results section.

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 compute a new 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. Sorting the patches from front-to-back is straightforward if we only consider one node (i.e., a single keyframe from a single viewpoint). However, in this case, the order must be suitable for all the nodes assigned to the cluster. Therefore, we must define a measure of distance for each patch that is applicable to all of these configurations and allows us to sort the patches accordingly. We accomplish this by computing an integrated distance $d(p)$ for each patch over all of the nodes n in the current cluster c:

$$d(p) = \sum_{n \in c} \mathrm{dist}(p, v_n, f_n)$$

where v_n and f_n are the viewpoint and frame associated with n, respectively, and $\mathrm{dist}(p, v_n, f_n)$ is the 3D Euclidean distance between v_n and the patch p centroid at frame f_n.

We then create a single mostly front-to-back order for the cluster by sorting the patches in increasing order based on $d(p)$. The complete pseudocode for the algorithm is given in Algorithm 1.

Algorithm 1

procedure CLUSTERNODES
Preprocessing:
    $F \leftarrow$ load animation frames
    $V \leftarrow$ generate representative viewpoints
    $N \leftarrow$ generate nodes from F and V
    $P \leftarrow$ generate vertex-cache optimized patches
    for $f \in F$ **do**
        translateToOrigin(f)
Bootstrapping:
    $C \leftarrow$ generate initial viewpoints for each cluster $c \in C$
    for $p \in P$ **do**
        averagePatch(p) compute average patch vertex positions over all frames
    for $c \in C$ **do**
        $IB_c \leftarrow$ computeBuffer(c) sort average patches to generate index buffer
Node assignment:
    for $n \in N$ **do**
        assignCluster(n) assign n to IB_c that gives smallest overdraw
    if no cluster assignment has changed **then**
        end
Index buffer computation:
    for $c \in C$ **do**
        for $p \in P$ **do**
            $d_p = \sum_{n \in c} dist(p, v_n, f_n)$
        $computeIB(c)$ update IB_c using the the patch distances computed above
    goto *Node assignment*

Note that prior to computing an integrated distance, we have also considered the simpler methods of either using the central viewpoint or the central frame of the cluster or both, to generate an order that is hopefully suitable for all nodes. This could reduce processing time significantly. However, it is not suitable for a large number of situations in which the model animates. Consider, for instance, even a simple animation

where a model induces a large rotation to one of its components. Different views can lead to considerably different orders among frames due to the varying orientations of parts of the model. In general, finding a representative viewpoint or frame from which to compute the order for an animated model is challenging, and even impossible in some cases. For these reasons, we adopted the brute-force integrated distance solution, which considers all frames and viewpoints equally.

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 table indexed by keyframe and viewpoint. Due to the framing described earlier, viewpoints used in preprocessing are relative to the center of the model's bounding sphere. During runtime selection, we therefore need to compute the relative view direction in order to calculate the corresponding viewpoint to be used. For simplicity, we index viewpoints based on polar and azimuth angles (θ, ϕ) (we use values from the closest original viewpoint when populating the table). While this distribution is less uniform than the subdivided icosahedron, it is only used to store the index buffer IDs for the purpose of simplifying the lookup at runtime. The time required for the lookup is negligible since it is only a single lookup for the entire model. The lookup parameters f, θ, and ϕ are rounded to the nearest valid parameter values.

4. Results

In this section, we present results of our approach. Our results use four models at zero, one, two and three levels of Loop subdivision and undergoing a set of twelve complex animations that have between 20 and 135 keyframes (see Figure 4). These are representative of animated characters often found in games and other real-time applications. Refer to the supplemental video for a demonstration of some of the

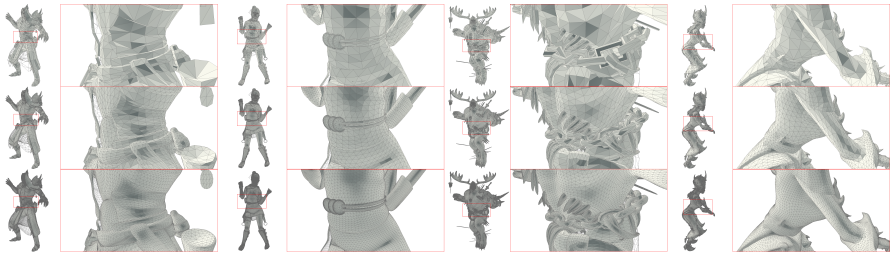

Figure 4. Representative animation frames with associated closeups of the four models that we used in our results. The top row shows the original model, while the bottom two rows shows the model after one and two levels of subdivision, respectively.

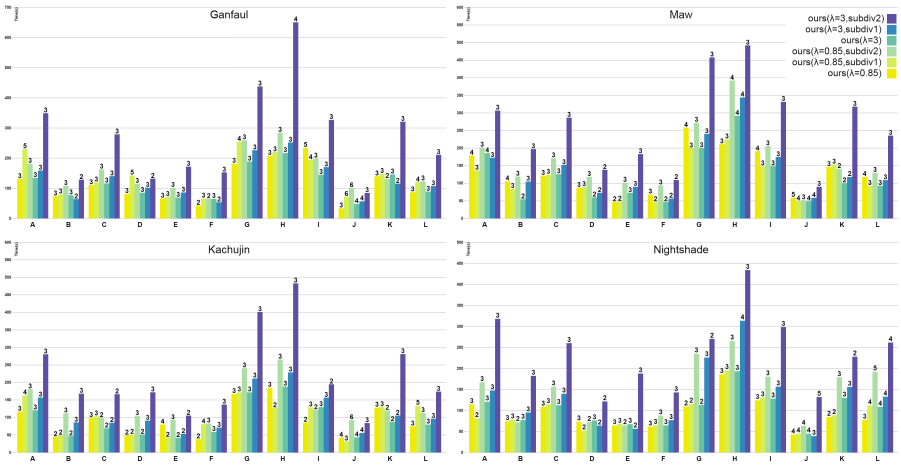

Figure 5. Processing times of our models. The number of iterations needed to reach convergence is shown above each bar.

animations. Figure 5 shows preprocessing times for creating index buffers for each animation (labeled *A*-*L*) as well as for a *joint* set of buffers that is optimized for all animations. Having a larger number of patches ($\lambda = 3$) and a larger number of triangles (subdivided models) both generally increase preprocessing time. A less predictable factor in the preprocessing time is the number of iterations before the process reaches convergence, which is anywhere between two and six in our results. To accelerate preprocessing, we use render-to-texture and instancing, which enable us to draw a frame from 162 viewpoints using a single draw call. An atomic counter is used to count the number of fragments that are processed. Note that we did not heavily optimize the preprocessing computation for speed, since this is done offline and only once after modeling. We focused on reducing overdraw for higher runtime performance on pixel bound scenes.

Choice of $\boldsymbol{\lambda}$. As mentioned earlier, rendering time has been shown to be directly proportional to ACMR for vertex-bound scenes, and OVR for pixel-bound scenes [Sander et al. 2007]. The algorithm trades off these objectives by controlling the desired ACMR through the λ parameter. Figure 6 shows how the overdraw ratio is affected by the choice of λ. We have found that setting λ 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. Note that even without considering vertex caching, it is not always possible to reduce the overdraw to 1 unless a huge number of clusters is used. As discussed below, we start getting diminishing returns in overdraw when using more than five clusters.

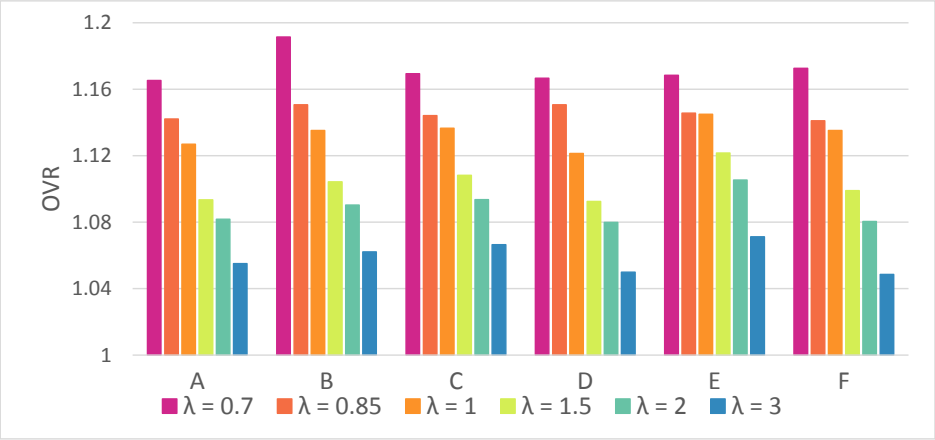

Figure 6. Adjusting λ provides a tradeoff between vertex caching and overdraw, as shown here for the six animations of the Ganfaul model.

Choice of initial viewpoints. As mentioned before, the algorithm starts by choosing a random viewpoint for each cluster in order to compute the initial index buffers. Due to the greedy nature of the algorithm, we noticed a discrepancy in the range of 0.02 in overdraw due to the choice of initial random viewpoints (e.g., 1.17 ± 0.02 for the Ganfaul model), which translate to marginal changes in rendering time. To accelerate the k-means convergence speed, during our random viewpoint selection, we ensure no two initial viewpoints are within a small distance threshold of each other. If this threshold is violated, we reject the viewpoint and randomly sample it anew.

Choice of number of clusters. Figure 7 shows results for different number of clusters for the Ganfaul model. Increasing the number of clusters reduces overdraw at the

Figure 7. 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.

expense of memory to store the additional index buffers. We have found that using more than five clusters only yields modest ovedraw 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 at different Loop subdivision levels (where ◬, ◬ denote one and two levels, respectively). 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 Sander et al. [2007] applied to *each individual frame*. It reduces overdraw significantly, but requires between 30-70 index buffers, depending on the animation, making 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 λ 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 8 further breaks down the results for each animation (labeled *motion A-L*). Note that the improvements of our algorithm are consistent over a variety of different character animations and

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	13795	0.654	1.377	0.869	1.302	2.939	1.211	0.861	1.153	2.715	1.065	0.860	1.16624	2.698	1.0882
Ganfaul ◬	55180	0.630	1.384	0.847	1.280	2.854	1.210	0.844	1.127	2.860	1.045	0.849	1.14309	2.847	1.06849
Ganfaul ◬	220720	0.618	1.376	0.849	1.248	2.875	1.209	0.848	1.091	2.926	1.034	0.858	1.11241	2.925	1.05853
Kachujin	12608	0.642	1.326	0.903	1.185	2.896	1.125	0.891	1.087	2.617	1.030	0.861	1.09377	2.585	1.03728
Kachujin ◬	50432	0.621	1.325	0.845	1.174	2.790	1.125	0.840	1.069	2.810	1.017	0.841	1.07874	2.799	1.02456
Kachujin ◬	201728	0.613	1.317	0.856	1.153	2.821	1.124	0.854	1.047	2.905	1.014	0.854	1.05456	2.846	1.01887
Maw	13908	0.620	1.423	0.879	1.262	2.918	1.167	0.860	1.103	2.696	1.044	0.859	1.11198	2.678	1.0597
Maw ◬	55632	0.631	1.416	0.847	1.231	2.850	1.166	0.843	1.081	2.851	1.031	0.843	1.09187	2.847	1.04768
Maw ◬	222528	0.619	1.417	0.855	1.193	2.874	1.166	0.853	1.058	2.927	1.025	0.853	1.07327	2.922	1.04204
Nightshade	12996	0.639	1.336	0.862	1.189	2.930	1.120	0.854	1.077	2.643	1.030	0.854	1.08915	2.614	1.0437
Nightshade ◬	51984	0.622	1.343	0.848	1.154	2.832	1.120	0.844	1.053	2.826	1.024	0.844	1.06377	2.820	1.03697
Nightshade ◬	207936	0.611	1.333	0.859	1.136	2.859	1.119	0.857	1.038	2.916	1.020	0.857	1.05144	2.905	1.03426

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 [Sander et al. 2007], 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).

Figure 8. Average overdraw ratio for different animations of the four models from Table 1.

brings the overdraw ratio very close to the optimal value of 1. Also note that as the subdivision level increases, the ACMR stays steady at approximately 0.85 while the overdraw ratio tends to decrease. Since we use $\lambda = 0.85$, clearly we should not see a significant change in ACMR, and the patches will have roughly the same number of triangles. However, since the subdivided meshes have higher resolution, the added granularity given to the ordering algorithm allows it to further decrease the overdraw ratio.

Rendering times. We conducted experiments to verify the dependency between OVR and rendering time [Sander et al. 2007]. We used the Kachujin model with $\lambda = 0.85$ in a heavily pixel-bound scene. We noticed the improvement in rendering time closely matched the improvement in OVR ($\sim$18-20%). With $\lambda = 3$, we achieved a slightly larger improvement, proportional to the change in OVR. With an inexpensive shader that simply outputs the color, the improvement was less significant ($\sim$10%).

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 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. For future work, we are exploring using a larger number of orderings that can *guarantee* front-to-back rendering, thus also making the approach more accurate for applications that require semi-transparent or translucent rendering.

Acknowledgments

This work was partly supported by Hong Kong GRF grants #619509 and #618513. The models used are from Mixamo.

References

AIREY, J. M. 1990. *Increasing update rates in the building walkthrough system with automatic model-space subdivision and potentially visible set calculations*. PhD thesis, UNC-CH. URL: http://www.cs.unc.edu/xcms/wpfiles/dissertations/airey.pdf. 40

AKELEY, K., HAEBERLI, P., AND BURNS, D., 1990. The tomesh.c program. Available on SGI computers and developers toolbox CD. 40

BITTNER, J., WIMMER, M., PIRINGER, H., AND PURGATHOFER, W. 2004. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum 23*, 3, 615–624. URL: https://www.cg.tuwien.ac.at/research/vr/chcull. 40

CHEN, G., SANDER, P. V., NEHAB, D., YANG, L., AND HU, L. 2012. Depth-presorted triangle lists. *ACM Transactions on Graphics 31*, 6, 160:1–160:9. URL: http://w3.impa.br/~diego/publications/ChenEtAl12.pdf. 40

CHOW, M. M. 1997. Optimized geometry compression for real-time rendering. In *Visualization'97*, IEEE Computer Society, Los Alamitos, CA, 347–354, 559. URL: https://doi.org/10.1109/VISUAL.1997.663902. 40

DEERING, M. 1995. Geometry compression. In *ACM SIGGRAPH '95*, ACM, New York, NY, 13–20. URL: https://doi.org/10.1145/218380.218391. 40

GOVINDARAJU, N. K., HENSON, M., LIN, M. C., AND MANOCHA, D. 2005. Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In *I3D*, ACM, New York, NY, 49–56. URL: https://doi.org/10.1145/1053427.1053435. 40

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical z-buffer visibility. In *ACM SIGGRAPH '93*, ACM, New York, NY, 231–238. URL: https://doi.org/10.1145/166117.166147. 40

HILLESLAND, K., SALOMON, B., LASTRA, A., AND MANOCHA, D. 2002. Fast and simple occlusion culling using hardware-based depth queries. Tech. Rep. TR02-039, Department of Computer Science, UNC-CH, Chapel Hil, NC. URL: http://www.cs.unc.edu/techreports/02-039.pdf. 40

HOPPE, H. 1999. Optimization of mesh locality for transparent vertex caching. In *ACM SIGGRAPH '99*, ACM, New York, NY, 269–276. URL: http://hhoppe.com/tvc.pdf. 40

LIN, G., AND YU, T. P.-Y. 2006. An improved vertex caching scheme for 3D mesh rendering. *TVCG 12*, 4, 640–648. URL: https://doi.org/10.1109/TVCG.2006.59. 40

MACQUEEN, J. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, University of California, Berkeley, CA, 281–297. URL: https://projecteuclid.org/euclid.bsmsp/1200512992. 43

NEHAB, D., BARCZAK, J., AND SANDER, P. V. 2006. Triangle order optimization for graphics hardware computation culling. In *I3D*, ACM, New York, NY, 207–211. 40, 41

SANDER, P. V., NEHAB, D., AND BARCZAK, J. 2007. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Transactions on Graphics 26*, 3, 89. URL: http://gfx.cs.princeton.edu/pubs/Sander_2007_>ETR. 39, 40, 41, 46, 48, 50

TELLER, S. J., AND SÉQUIN, C. H. 1991. Visibility preprocessing for interactive walkthroughs. In *ACM SIGGRAPH '91*, ACM, New York, 61–70. URL: https://doi.org/10.1145/127719.122725. 40

6. Author Contact Information

Songfang Han
HongKong UST
Clear Water Bay
Kowloon,HongKong
shanaf@connect.ust.hk

Pedro V. Sander
HongKong UST
Clear Water Bay
Kowloon,HongKong
psander@cse.ust.hk

Songfang Han, Pedro V. Sander, Triangle Reordering for Efficient Rendering in Complex Scenes, *Journal of Computer Graphics Techniques (JCGT)*, vol. 6, no. 3, 38–52, 2017
http://jcgt.org/published/0006/03/03/

Received:	2016-11-07		
Recommended:	2017-03-10	Corresponding Editor:	Eric Haines
Published:	2017-09-28	Editor-in-Chief:	Marc Olano