Improving $k$-buffer methods via Occupancy Maps

Andreas A. Vasilakis † and Georgios Papaioannou

Dept. of Informatics, Athens University of Economics & Business, Greece

Abstract

In this work, we investigate an efficient approach to treat fragment racing when computing $k$-nearest fragments. Based on the observation that knowing the depth position of the $k$-th fragment we can optimally find the $k$-closest fragments, we introduce a novel fragment culling component by employing occupancy maps. Without any software-redesign, the proposed scheme can easily be attached at any $k$-buffer pipeline to efficiently perform early-$z$ culling. Finally, we report on the efficiency, memory space, and robustness of the upgraded $k$-buffer alternatives providing comprehensive comparison results.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Hidden line/surface removal

1. Introduction

Multi-fragment visibility determination is a standard stage in developing numerous effects for interactive 3D games and graphics applications. Order independent transparency for deferred [BM08, HBT14] and forward rendering [MCTB11] can be considered typical examples of such applications that also comprise benchmark cases for testing multi-fragment processing frameworks for scenes with high depth complexity (e.g. flow data [GSM 14] or hair geometry [YYH 12]). Traditionally, a family of GPU-accelerated buffers is responsible on treating the problem of storing the out-of-order in-

DOI: 10.2312/egsh.20151017
coming fragments generated by the rasterization pipeline (see Fig. 2). While the GPU-accelerated A-buffer [YHGT10, VF12] is the dominant structures for holding multiple fragments via variable-length per-pixel structures, several alternatives have been proposed to alleviate the cost of excessive allocation and access of video-memory [MCTB11].

$k$-buffer [BCL’07] as well as its stencil-routed version [BM08] are widely-accepted A-buffer approximations, capable of capturing the $k$-closest fragments to the viewer by employing fixed-size per-pixel vectors on the GPU. Despite their reduced memory demands when compared to A-buffer solutions, both techniques suffer more or less from read-modify-write hazards (RMWH) caused when the generated fragments are inserted in arbitrary depth order. To this end, an abundance of $k$-buffer variants have been recently introduced aiming at eliminating the disturbing dotted or heavily speckled surface areas that result from the aforementioned problems [Sal13, VF14]. However, these modifications come at the cost of additional computation and memory requirements or the requirement of specialized hardware.

While exploiting hardware-accelerated pixel synchronization [Sal13, BH14] ensures accurate $k$-buffer construction, its performance degrades rapidly due to the heavy fragment contention of accessing the critical section when rendering highly-complex scenes. To this end, $k^*$-buffer [VF14] significantly alleviates fragment racing by concurrently performing efficient culling checks to discard fragments that are farther from all currently maintained fragments. Unfortunately, the proposed fragment rejection process has several limitations. First of all, the process initially requires the insertion of $k$ fragments before it starts performing any culling tests. Secondly, it depends on the depth order of the incoming fragments, with no impact at the worst case scenario of fragments arriving in descending order. Last but not least, the actual fragment elimination is unfortunately performed by performing bit counting operations and subsequently utilizing fragments, with no impact at the worst case scenario of fragments arriving in descending order. As a result, fragments whose depth is larger than the computed boundary, do not belong to the potential final result and eventually fail the depth test, avoiding their pixel shading execution cost. In essence, this process maximizes the number of fragments to be rejected and significantly reduces the total workload. An experimental evaluation is provided demonstrating the advantages of our work over the original $k$-buffer variants in terms of memory usage, performance cost and image quality.

2. Occupancy-based Fragment Culling

Ideally, when constructing a $k$-buffer, knowing the exact depth of the $k$-th fragment a priori allows us to insert all fragments with smaller or equal depth in constant time, discarding the rest (farther ones). In this work, we present an efficient approach for approximating the depth of the $k$-th fragment that is the nearest largest to the $k$-th one: $k_a \geq k$. Inspired by [LHLW09], a 32-bit unsigned integer 3D fragment occupancy array buffer is utilized to define a per-pixel bitmap, whose entries indicate the presence of fragments in each sub-interval. The depth range of each pixel $p$ is divided into $S = 32 \cdot d$ uniform consecutive sub-intervals $[p.s_j, p.s_{j+1})$, where $p.s_j = p.z_n + \frac{d}{k} \cdot (p.z_f - p.z_n)$, $j = 0, 1, \ldots, S - 1$ and $d > k/32$ define the depth range subdivision. A depth-range map is initially computed, containing for each pixel $p$ the nearest and farthest fragment depth values from the camera, $p.z_n$ and $p.z_f$, respectively [SA09]. However, this additional geometry pass is highly expensive especially for highly-tessellated scenes, so a bounding box [LHLW09] or a smaller representation as a convex hull may be instead chosen to approximate the geometry (see Fig. 2).

Initially, a clear pass initializes the occupancy buffer to zero. During the first geometry rasterization (fill pass), the
j-th bit of the occupancy map is set to 1 \((p.b_j = 1)\) for each arriving fragment \(j\) falling within the corresponding sub-interval \((p.s_j \leq j < p.s_{j+1})\) via blending or atomic operations, depending on the hardware. A full-screen rendering stage follows (count pass) to concurrently count the number of 1s in the bit-array \(p.b\) by adjusting the Brian Kernighan’s algorithm [Ker88]; count the number of times \(p.b \& (p.b - 1)\) is performed in a loop. When the counter reaches \(k\), the algorithm halts and the current bucket’s depth is returned to the Z-buffer (\(O(k)\) time). Then, the \(k\)-buffer is efficiently constructed (peel pass) by taking advantage of the early-z culling capabilities of the GPU. Finally, a sorting process [LFS’14] is employed to reorder the captured fragments before generating the final image (resolve pass).

Figure 3 illustrates how to modify any \(k\)-buffer pipeline in order to introduce our occupancy-based fragment culling.

**Lemma 2.1** Given that more than one fragments are routed to the same subinterval, the foremost \(k\) fragments always succeeded to pass the culling mechanism.

**Proof** If \(n_j \geq 0\) is the number of fragments falling into packet \(j, b_j = n_j > 0 \) ? 1 : 0 and \(N = \sum n_j > k\) is the total generated fragments at an arbitrary pixel, then exists an index \(i\) where \(\sum_{j=0}^{N} b_j = k\). Let’s assume that the statement is false, thus \(\sum_{j=0}^{N} b_j < k \iff \sum_{j=0}^{N} n_j < \sum_{j=0}^{N} b_j\), which does not hold, since \(n_j \geq b_j, \forall j \leq i\).  

![Figure 3: The modified \(k\)-buffer pipeline.](image)

### 3. Results

We present an experimental analysis of our fragment culling mechanism extending a set of \(k\)-buffer realizations focusing on performance, memory requirements and image quality under different testing conditions (see also a comparative overview at Table 1). All experiments were conducted using a 1024\(^2\) viewport with varying \(d, k\) on an NVIDIA GTX780 Ti with 3 GB of memory. We have also provided as supplementary material the shader code, including implementations for older and modern hardware, of the proposed components in the order shown in Figure 3.

<table>
<thead>
<tr>
<th>Method</th>
<th>Perform.</th>
<th>Memory</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>[BCL’07]</td>
<td>↓</td>
<td>↑</td>
<td>×</td>
</tr>
<tr>
<td>[BM08]</td>
<td>↓</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>[SML11,YHY’12]</td>
<td>↑</td>
<td>↓</td>
<td>✓</td>
</tr>
<tr>
<td>[MCTB13,Sal13,VF14]</td>
<td>↑</td>
<td>↑</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1: Comparative analysis of augmented \(k\)-buffer approaches with fragment culling tests.

**Performance Analysis.** Figure 4 shows how the performance exponentially improves when our fragment culling mechanism with \(d = 32\) is exploited at \(k^d\)-buffer [VF14] when rendering three complex multilayer scenes with \(k = \{4, 8, 16, 32\}\). Similar boost is observed for the rest variations [SML11,YHY’12,MCTB13,Sal13] (omitted for space reasons). Conversely, stencil-routed \(k\)-buffer [BM08] is significantly lessened from the additional geometry pass.

![Figure 4: Performance evaluation in ms of two representative \(k\)-buffer methods [BM08,VF14] without and with enabling our fragment culling on three scenes with high depth complexity (shown in brackets) and varying \(k\).](image)

Figure 5 illustrates how the performance of Hairball rendering (see Fig. 1) scales when our fragment culling is exploited in \(k^d\)-buffer [VF14], under a set of increasing \(d = \{1,\ldots,64\}\) and fixed \(k = 8\). We initially observe the small performance gain when the \(k^d\)-buffer’s fragment culling is exploited. On the other hand, even with large depth sub-intervals the modified method is highly accelerated by the use of our technique. Note that when moving to higher values of \(d\), computation of the culling layer is slightly increased. Finally, for \(d = 32\), we performed culling testing inside the shader to show that moving from software- to hardware-implemented depth testing we achieved a 2x computation improvement.

![Figure 5: Our mechanism speeds up the \(k^d\)-buffer [VF14] even when \(d\) remains at low levels. [VF14]* denotes a modified fragment culling-free \(k^d\)-buffer and \(\dagger\) defines a shader-based fragment culling execution.](image)
Memory Allocation Analysis. Concerning memory demands, our mechanism requires specifically \((d - k + 3): 32\text{-bit additional per-pixel storage (for } d > k\), when compared with the rest of memory-bounded \(k\)-buffer variants. Specifically, this extra memory overhead comes from the allocation of the near, far and \(k\)-th fragment depth textures and fragment occupancy buffer. A memory-friendly representation can also be implemented that reuses the fragment occupancy buffer for storing the color information of the actual \(k\)-buffer. On the other hand, the risk of a potential memory-overflow can also be implemented that reuses the fragment occupancy buffer for storing the color information of the actual \(k\)-buffer.

On the other hand, the risk of a potential memory-overflow can also be implemented that reuses the fragment occupancy buffer for storing the color information of the actual \(k\)-buffer. In this work, we have introduced a simple fragment culling introduced extension is restricted to work for applications that aim to capture only the \(k\)-closest to the viewer fragments and not the “best” \(k\) ones [SML11]. Finally, RMWH of the original \(k\)-buffer [BCL’07] cannot be completely alleviated.

4. Conclusions

In this work, we have introduced a simple fragment culling process easily integrated to any \(k\)-buffer pipeline. The \(k\)-th fragment, well-approximated via fragment occupancy maps, defines the depth testing criterion. We expect that the included comparative experiments with respect to performance, memory usage and robustness will provide a useful guide for effectively addressing multi-fragment rendering on high-depth-complexity scenes.

Acknowledgments. This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program “Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF) - Research Funding Program: ARISTEIA II-GLIDE (grant no.3712).

References


[BM08] Bavoil L., Myers K.: Deferred rendering using a stencil routed \(k\)-buffer. ShaderX6: Advanced Rendering Techniques (2008), 189–198. 1, 2, 3, 4


© The Eurographics Association 2015.