k^+-buffer: An Efficient, Memory-Friendly and Dynamic k-buffer Framework

Andreas A. Vasilakis, Georgios Papaioannou, Ioannis Fudos, *Member, IEEE*

Abstract—Depth-sorted fragment determination is fundamental for a host of image-based techniques which simulates complex rendering effects. It is also a challenging task in terms of time and space required when rasterizing scenes with high depth complexity. When low graphics memory requirements are of utmost importance, k-buffer can objectively be considered as the most preferred framework which advantageously ensures the correct depth order on a subset of all generated fragments. Although various alternatives have been introduced to partially or completely alleviate the noticeable quality artifacts produced by the initial k-buffer algorithm in the expense of memory increase or performance downgrade, appropriate tools to automatically and dynamically compute the *most suitable* value of k are still missing. To this end, we introduce k^+-buffer, a fast framework that accurately simulates the behavior of k-buffer in a single rendering pass. Two memory-bounded data structures: (i) the *max-array* and (ii) the *max-heap* are developed on the GPU to concurrently maintain the k-foremost fragments per pixel by exploring *pixel synchronization* and *fragment culling*. Memory-friendly strategies are further introduced to dynamically (a) lessen the wasteful memory allocation of individual pixels with low depth complexity frequencies, (b) minimize the allocated size of k-buffer according to different application goals and hardware limitations via a straightforward depth histogram analysis and (c) manage local GPU cache with a fixed-memory depth-sorting mechanism. Finally, an extensive experimental evaluation is provided demonstrating the advantages of our work over all prior k-buffer variants in terms of memory usage, performance cost and image quality.

Index Terms—k-buffer, A-buffer, depth peeling, pixel synchronization, depth complexity histogram, dynamic geometry

1 Introduction

Depth-ordered fragment determination is a standard stage in developing numerous appealing and plausible visual effects for interactive 3D games and graphics applications. A variety of algorithms ranging from photorealistic rendering, such as global illumination [1], order-independent transparency for forward, deferred, volumetric shading [2], [3], [4] and shadowing [5] to volume visualization and processing of flow, molecular, hair and solid geometry [6], [7], [8], [9], [10] require accurate multi-fragment processing at interactive speeds.

In the last decade, significant research has been conducted on addressing the problem of visibility determination from different perspectives, usually classified in a broad level based on whether they perform depth ordering on the objects/primitives (geometric-space algorithms) or on the generated pixel fragments (image-space algorithms). Object- [11], and in higher granularity, primitive-sorting techniques [12], [13] exhibit increased popularity due to their unique combination of desirable properties (error-free, extremely efficient and easy integrated with the standard graphics pipeline). However, they require an expensive preprocessing step of building view-dependent data structures (e.g. BSP tree [14]), which unfortunately makes them unsuitable for scenes consisting of dynamic, or worse, self-intersecting geometry [15].

Avoiding these limitations, a family of GPU-accelerated buffers is traditionally responsible of treating the problem of *storing*, and subsequently *sorting*, the out-of-order surface intersections, namely *fragments*, generated when sampling the geometry at a pixel level. Figure 1 shows an illustrative example of the fragment generation process as a per-pixel ray-surface intersection process. While the GPU-accelerated A-buffer [5], and its subsequent variants that exploit fixed [16] or dynamic [17] paged memory management, are the dominant structures for maintaining multiple fragments via *one* or *more* [18] variable-length linked lists per-pixel, several alternatives have been proposed to alleviate the cost of excessive allocation and access of video-memory [19].

k-buffer [20] as well as its stencil-routed version [21] are widely-accepted A-buffer approximations, capable of capturing the *k-closest* to the viewer fragments by employing fixed-size vectors per-pixel on the GPU (see also an illustrative presentation of the idea in Fig 1). Despite their reduced memory and computation demands when compared to A-buffer solutions, they both 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 problem, by performing

- *A. A. Vasilakis and G. Papaioannou are with the Department of Informatics, Athens University of Economics & Business, Greece. E-mail: {abasilak,gepap}@aueb.gr*
- *I. Fudos is with the Department of Computer Science & Engineering, University of Ioannina, Greece. E-mail: fudos@cs.uoi.gr*

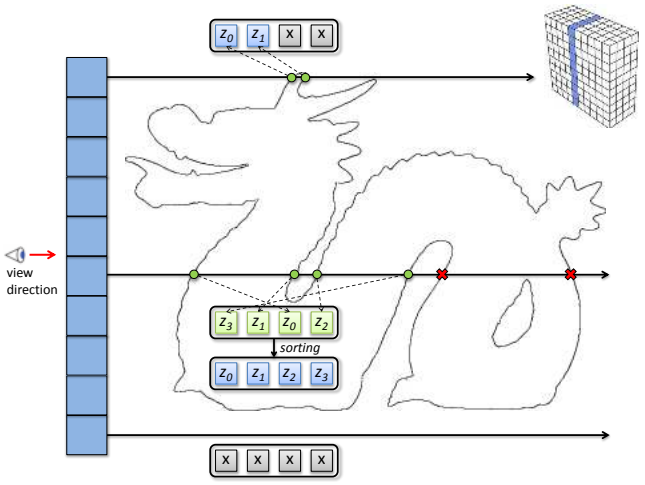

Fig. 1: Illustrating the construction process of a row of a 4-buffer (highlighted with blue at the top-right thumbnail), when ray casting the dragon model. A significant amount of memory space is wasted at pixels that consist of less than 4 fragments due to the pre-allocation of the same buffer length per pixel.

two [22] or multiple passes [23], constructing an auxiliary A-buffer [9], or exploring hardware-accelerated pixel synchronization mechanisms [24], [25] with the cost of additional performance and memory requirements as well as the necessity of specialized hardware.

From a development and production standpoint, complex and potentially animated environments, with high occlusion, multiple points of interest or depth layers can become very challenging in terms of finding the optimal per pixel fragment complexity limit (k) to correctly capture the designer's intent under constrained memory budget. Traditionally, this task is addressed by an iterative trial-and-error procedure, where the user manually configures the value of k, until an acceptable visual result is produced. However, this approach works with static geometry viewed from particular viewpoints. Thus, the local scope of the user intervention results in erroneously impacting other areas and viewing configurations in the scene, which finally distorts the output at a global scope.

In this paper, we extend our work on multi-fragment rendering, the k^+-buffer ($\mathbf{K^+B}$), first introduced in I3D'14 conference [26]. The k^+-buffer is an efficient k-buffer framework that overcomes the aforementioned performance bottlenecks, memory footprints, image artifacts and hardware restrictions.

Contrary to most of the other k-buffer alternatives, which store and sort the generated fragments on the fly, we follow a faster strategy similar to the one used by the A-buffer construction: The k-nearest fragments are captured in an unsorted sequence, followed by a post-sorting step that reorders them by their depth. We explore a GPU-accelerated *spin-lock* strategy via pixel semaphores to ensure real-time synchronized construction of the unsorted k-front fragments (Sec. 3.1). Two bounded array-based data structures for fragment data are introduced which enable a low cost culling test that concurrently discards outlier fragments (Sec. 3.2). A post-sorting process that correctly reorders fragments by their depth via a hybrid solution is finally performed (Sec. 3.3).

With respect to the original algorithm, we present several novelties and extensions in this article. The newly introduced contributions are the following:

- A novel fragment insertion strategy is introduced alleviating part of the fragment congestion when accessing the critical section (Sec. 3.2.1).
- Our framework may be easily extended to the general context of finding the "*best*", not necessarily the closest to viewer, k fragments (Sec. 3.2.2).
- A histogram-based depth complexity limit (k) estimator is employed to avoid delegating users with the non-intuitive task of manually setting the value of k under highly complex and potentially dynamic environments (Sec. 4.2). To our knowledge, this is the first k-buffer implementation with dynamic and precise allocation of the required storage space.
- The problem of over-sized local GPU caches when performing fragment sorting is resolved by exploiting fragment culling with a fixed max-array data structure (Sec. 4.4).
- A thorough experimental study is provided including comparisons with depth peeling methods as well as two recent competitive k-buffer implementations (Sec. 5).

The overall framework is described by offering shader-like pseudocode and the fragment processing pipeline. We highlight features and trade-offs of our framework, pointing out implementation details and light-weight modifications that can be used to guide the decision of which pipeline alternative to employ in a given setting. The structure of this paper is as follows: Section 2 offers a detailed overview of prior art. Section 3 introduces the algorithmic details of our main framework. Section 4 describes how the proposed pipeline is extended to support precise memory allocation as well as dynamic k determination. Section 5 provides extensive comparative results for several multi-fragment rendering alternatives. Finally, Section 6 offers conclusions and future research directions.

2 Related Work

The problem of visibility determination in screen-space has been thoroughly investigated in the computer graphics research community during the last two decades. Numerous multi-pass image-based techniques have been developed on the GPU to resolve fragment visibility ordering [19], aiming at minimizing more or less the computation cost and memory allocation needed to accurately extract a fragment

subset $s = \{1, \ldots, k\},\ k \leq n$ of all generated per pixel fragments n. We classify these techniques in two broad categories based on the number of fragment samples, denoted by f_s from now on, captured by each algorithm in a single iteration step.

Memory-unbounded algorithms aim at capturing *all* fragments per pixel in a single geometry pass ($f_s = n$). Fragments are stored into variable-length data structures per pixel during geometry rendering, followed by a post-sorting process that correctly reorders fragments by their depth.

On the other hand, *memory-bounded* algorithms succeed to process all fragment information through a multi-pass rendering pipeline, implemented with a constant video-memory budget. Depending on the algorithm, each iteration carries out one or two geometry passes to extract a fragment batch with guaranteed depth order ($f_s \leq k$). The main advantage of the latter class is the memory overflow-free behavior at the expense of increased computation requirements.

Memory-unbounded Methods. A-buffer [27] was the first method to capture all fragments via real-time concurrent construction of *variable-length* linked lists per pixel in a single rasterization pass. Subsequently, stored fragments are post-sorted according to their depth values. With the recent advent of atomic memory operations on graphics hardware, Yang et al. introduced an actual GPU-accelerated A-buffer implementation ($\mathbf{AB_{LL}}$) [5]. Although it initially suffered from performance bottlenecks due to the heavy contention and the random memory access when constructing and assembling the entire fragment list [16], it is currently the preeminent method for processing multiple fragments. The main reason is the L2 cache integration in the recent generation of GPU architectures, which has resulted in a significant increase of the atomic operation throughput. Uniform [28] and adaptive [1] tiling strategies have further been proposed to harness the potential fragment overflow risks.

On the other hand, FreePipe [29], a complete CUDA-based rasterization pipeline, maintains multiple fragments using *fixed-size* per pixel vectors. To ensure complete fragment extraction for all pixels, the buffer length must accurately be set prior rendering. FreePipe has been realized using modern OpenGL APIs, thus avoiding switching from the traditional graphics pipeline to a software rasterizer ($\mathbf{AB_{Array}}$) [30]. Despite its high performance, it requires the allocation of a large and in some cases, unnecessary quantity of graphics memory.

To overcome the limitations of both linked-list and fixed-array techniques, two-geometry-pass A-buffer variations were introduced by [17], [31]. While these methods self-adjust memory allocation to handle a variable number of fragments per pixel without wasting memory, only the S-buffer realization ($\mathbf{AB_{SB}}$) [17] is CUDA-free and exploits sparsity at the pixel-space.

In the context of the fragment post-sorting stage, several approaches have been proposed to alleviate more or less the performance bottlenecks, when handling high depth complexity scenes. To overcome the sequential nature of the sorting process on the number of depth layers, a CUDA-based technique was proposed by Patney et al. [32] extending the domain of parallelization to individual fragments. The idea of altering the sorting algorithm per pixel based on the number of stored fragments has shown significant benefits [33]. A novel register-based block sorting algorithm is recently introduced, better exploiting the memory hierarchy of the GPU [34]. Inspired by the works of [35], [36], Vasilakis and Fudos [18] proposed to concurrently store fragments into more than one per-pixel linked lists ($\mathbf{AB_{LL\text{-}BUN}}$), speeding up the subsequent sorting phase. Lindholm et al. [37] presented two novel components to improve the management of local GPU caches. The former minimizes the allocated size in the fast cache memory by adjusting the allocation to pixel depth complexity (also found here [38] as *backwards memory allocation*), while the latter partitions the depth sorting similarly to depth peeling [39] and recycles a smaller amount of allocated memory.

Memory-bounded Methods. Regardless of the data structure, the aforementioned class of methods suffers from (i) memory overflows as a result of the unbounded buffer needed to store all generated fragments, and (ii) performance bottlenecks that arise when the number of per-pixel fragments to be post-sorted increases significantly.

Probably the most well-known multi-pass peeling technique, due to its low and constant storage requirements, is front-to-back (**F2B**) depth peeling [39], which works by rendering the geometry multiple times, peeling off a single fragment layer per pass. Dual depth peeling (**DUAL**) [40] speeds up multi-fragment rendering by capturing both the nearest and the furthest fragments in each pass. DUAL was further extended by extracting two fragments per uniform clustered bucket (**BUN**) [35]. To alleviate *Z-fighting* issues in depth peeling, a number of solutions were recently introduced [18]. However, despite the advent of several optimizations (Z-Batches [41], coherent layer peeling [42], object culling [18]), multi-pass rendering is regularly required to carry out complex scenes, substantially failing to behave interactively.

k-buffer (**KB**) [20], [43] reduces computation cost by capturing the *k*-closest to the viewer fragments in a single geometry rasterization pass. However, it is susceptible to disturbing flickering artifacts caused by RMWH during fragment insertion updates. Liu et al. extended this work to a multi-pass approach (**KB-Multi**) [23] achieving robust rendering behavior with the trade-off of low frame rates. Moreover, Bavoil and Mayers eliminated most of the memory conflicts by performing *stencil routing* operations on a multi-

sample anti-aliasing buffer (**KB-SR**) [21]. Finally, a *memory-hazard-aware* solution (**KB-MHA**) [44] based on a depth-error correction coding scheme is explored, however do not guarantee, in practice, correct results in all cases. The image quality of the methods described above is highly dependent on a coarse CPU-based pre-sorting in primitive space, which eliminates the arrival of out-of-order fragments. Multiple rendering iterations, are further required to provide an A-buffer output, due to the limited number of multiple render targets on the GPU. As expected, this results in significant performance downgrade. Conversely, Wang and Xe proposed partitioning the input scene into components with a bounded number of layers and then rendering them individually to fit into the limited KB-SR buffer size [45]. However this scheme cannot support animated scenes and is not particularly suitable for order-dependent applications.

Multi-depth test scheme (**KB-MDT**$_{32}$), developed in both CUDA [29] and OpenGL [22] APIs, guarantees correct depth order results by capturing and sorting fragments on the fly via 32-bit atomic integer comparisons. However, its inability to simultaneously update the depth and color values necessitates an additional costly geometry pass. Fortunately, its 64-bit version (**KB-MDT**$_{64}$) [46] is currently feasible to run on modern NVIDIA graphics cards [47]. However, noisy images may be generated from both 32- and 64-bit versions due to the precision lost when converting floating depth values.

Similar to our method, Salvi extended the original k-buffer to avoid fragment racing by employing hardware-aware pixel synchronization (**KB-PS**) [24]. However, this method is compatible only with graphics cards based on the Haswell architecture.

Finally, Yu et al. proposed two linked-list-based solutions to accurately compute the k-foremost fragments [9]. The idea of the first one is to capture all fragments by initially constructing an A-buffer via linked lists [5], followed by a step that selects and sorts the k-nearest fragments (**KB-AB**$_{\mathbf{LL}}$). The same strategy was also followed by prior work [48] which adaptively compresses fragment data to closely approximates the ground-truth visibility solution. On the other hand, the second approach directly computes depth-ordered per-pixel linked lists avoiding the unnecessary A-buffer construction (**KB-LL**). Despite the fact that it theoretically requires less storage, fragments are sparsely stored in memory causing the additional allocation of contiguous blocks of memory.

Table 1 presents a comparative overview of all k-buffer alternatives with respect to memory requirements, rendering complexity, fragment extraction accuracy and sorting stage.

3 Core Framework Overview

We propose k^+-buffer, an efficient k-buffer implementation on the GPU, which is free from: (i) geometry sorting prior to rasterization, (ii) unbounded memory requirements, (iii) RMW memory-hazards, (iv) depth precision conversion artifacts and (iv) specialized hardware extensions (i.e. pixel synchronization, 64-bit atomic operations). Contrary to most k-buffer alternatives, which store and sort the generated fragments on the fly, we initially store the k-nearest fragments in an unsorted sequence, followed by a post-sorting step that reorders them by their depth. A semaphore-based *spin-lock* mechanism ensures atomicity of the per-pixel fragment operations in the shared memory. Implementation details are also provided to easily switch to the hardware-implemented pixel syncing solutions available on the modern architectures (Sec. 3.1). To alleviate contention (busy-waiting) of distant fragments, we concurrently perform *culling* checks that efficiently discard fragments that are farther from all currently maintained fragments.

Two array-based data structures are built on the GPU to accurately store the closest per-pixel fragments: (i) *max-array*, an array where the maximum element is always stored at the first entry and (ii) *max-heap*, a complete binary tree in which the value of each internal node is larger than or equal to the values of the children of that node. Despite its linear complexity, the former performs faster than the latter when the problem size is sufficiently small (Sec. 3.2). Finally, a post-sorting process correctly reorders fragments by their depth by exploring a hybrid solution (Sec. 3.3).

3.1 Spin-Lock (SL)

Per-pixel *binary semaphores* are utilized as a synchronization mechanism to ensure fragment-exclusive use of the critical storage section. Taking into account the possibility of simultaneous access to the lock, which could cause race conditions, an implementation of an atomic *test-and-set* operation is introduced. Typically, the calling thread obtains the lock if the old value was 0. It spins writing 1 to the variable until this occurs. One way to implement spin-lock strategy employing test-and-set into a pixel shader is shown in Algorithm 1 (ignore color-coded lines).

Algorithm 1 MutualExclusion (Texture t, Pixel p)

```
1: beginFragmentShaderOrderingINTEL();      ▷ acquire lock (PS)
2: beginInvocationInterlockNV();            ▷ acquire lock (FSI)
3: while true do                            ▷ spin until lock is free (SL)
4:     if !imageAtomicExchange(t, p, 1) then   ▷ acquire lock (SL)
5:         {enter critical section}             ▷ exclusive use
6:         imageStore(t, p, 0);                 ▷ release lock (SL)
7:         break;                               ▷ stop spinning (SL)
8:     end if
9: end while
10: endInvocationInterlockNV();             ▷ release lock (FSI)
▷ where 'text' and 'text' defines the PS and FSI implementations
to fragment-exclusive use of the critical section, respectively
```

A 32-bit unsigned integer texture with internal pixel format R_32UI is allocated to represent the per-pixel semaphores. First, a full-screen quad rendering

(**clear pass**) is executed to initialize the texture with zeros. Our method is enhanced by the OpenGL's *imageAtomicExchange*(texture lock, ivec2 P, uint V) function, which atomically replaces the value V of the atomic object with the argument into texel at coordinate P and returns its original value.

Pixel Synchronization (**PS**) is an extension that Intel has recently introduced for its IRIS Pro Graphics that provides an inexpensive mechanism to avoid fragment conflicts in the critical sections and ensures that RMW memory operations are performed in submission order [24]. Following this trend, a similar extension was developed, named Fragment Shader Interlock (**FSI**) [25], supported only by NVIDIA graphics cards with Maxwell architecture. Our framework can be enhanced by the use of PS/FSI without remodeling the proposed pipeline by minimal implementation-wise modifications (see colored code in Algorithm 1). Note that avoiding the usage of per-pixel semaphores also results in reduced memory demands.

3.2 Fragment Storing

A geometry rendering (**store pass**) is initially carried out to capture the closest fragment data per-pixel in a 64-bit floating point 3D array buffer with internal format of RG_32F, (R for color and G for depth) and k length. Figure 1 illustrates a k^+-buffer which can hold up to 400 fragments (screen size: 10×10, $k = 4$).

To alleviate the spinning of n generated fragments that do not belong to the closest k, a fast culling mechanism is performed. The idea is to efficiently discard each incoming fragment f_i, $\forall i \in \{0, \ldots, n-1\}$ that has equal or larger depth value (f_i.z) from all currently maintained fragments, before trying to acquire the semaphore. Note that i determines the submission order. Let $\mathrm{a}_i[:] = \{\mathrm{a}_i[j], j = 0 \ldots k-1\}$ denote the contents of the k^+-buffer when fragment f_i has been processed. Initially, we do not discard any incoming fragments until the fragment storage buffer is full ($\forall i < k$). Then, we discard all fragments f_i such that $f_i.\mathrm{z} \geq \max\{\mathrm{a}_{i-1}[:].\mathrm{z}\}$. On the other hand, a fragment with $f_i.\mathrm{z} < \max\{\mathrm{a}_{i-1}[:].\mathrm{z}\}$ replaces the fragment of the k^+-buffer with the largest depth value. This strategy guarantees that the k-nearest fragments will always survive since:

$$\begin{aligned} \max\{\mathrm{a}_{n-1}[:].\mathrm{z}\} &\leq \cdots \leq \max\{\mathrm{a}_{i-1}[:].\mathrm{z}\} \\ &\leq \cdots \leq \max\{\mathrm{a}_{k-1}[:].\mathrm{z}\} \end{aligned} \quad (1)$$

To achieve fragment culling without traversing the entire pixel row for every incoming fragment, we have developed two array-based data structures on the GPU that both store the maximum element at the first array position: (i) *max-array* (**K$^+$B-Array**) and (ii) *max-heap* (**K$^+$B-Heap**). Thus, this operation is performed in constant time. The implementation of this idea is shown in Algorithm 2 (ignore blue-colored code).

Max-array can be considered as an array where the fragment with the largest depth value is always stored at the first location and the rest are randomly positioned. When an incoming fragment obtains a semaphore, it stores its information in the first empty entry ($\mathrm{O}(1)$). In this case, a per-pixel *counter* (32-bit unsigned integer texture with internal pixel format R_32UI) is utilized as index and incremented after a successful insertion. Per-pixel counters are initialized to zero during the clear rendering pass. If the array is full ($counter == k$), it takes the place of the fragment with the largest depth value. Note that since the culling mechanism resides outside the critical section, an additional check is mandatory to guarantee correct results. To keep max-array consistent after an insertion on a completely filled array, we find the fragment with the largest depth value ($\mathrm{O}(k)$) and swap it with the newly added fragment (except when the latter is the largest one). This process is implemented without the use of any costly atomic operations since fragment atomicity is guaranteed.

However when the problem size increases ($k > 16$), fragment data information can be alternatively maintained in a max-heap data structure. Max-heap is a complete binary tree (*shape* property) in which all nodes are larger than or equal to each one of its children (*heap* property). Max-heap can be implemented using a simple k-sized array without allocating any space for pointers: If the tree root is at index 0, then each element at index $i \in [0, k)$ has children at indices $2i + 1$ and $2i + 2$ and its parent is located at index $\lfloor \frac{(i-1)}{2} \rfloor$. Since the first node contains the largest element, the core pipeline followed by max-array is not altered. Both inserting operations to an empty or a full heap modify the heap to conform to the shape property first, by adding nodes from the end of the heap or replacing the heap root ($\mathrm{O}(1)$). Then, the heap property is restored by traversing up-heap or down-heap ($\mathrm{O}(\log_2 k)$). Pseudocode for both insertion functions can be found in the original paper [26]. Figure 2 illustrates how both data structures with $k = 8$ are constructed and updated from a number of out-of-order fragment insertions. Notice that two incoming fragments are successfully discarded when the buffer is completely full. A comparison of internal data representations between max-array and max-heap node pointers is also shown.

3.2.1 Revising K$^+$B-Array

Based on the observation that the process initially requires the insertion of k fragments before it starts performing any culling test, we aim to lessen fragment racing by performing direct insertions of the first $k - 1$ incoming fragments. This is extremely efficient in cases where k^+-buffer is large enough to contain all ($n \leq k$) or most ($k \leq n \leq 2 \cdot k$) of the generated fragments ($n = \max_p\{f(p)\}$, where $f(p)$ is the number

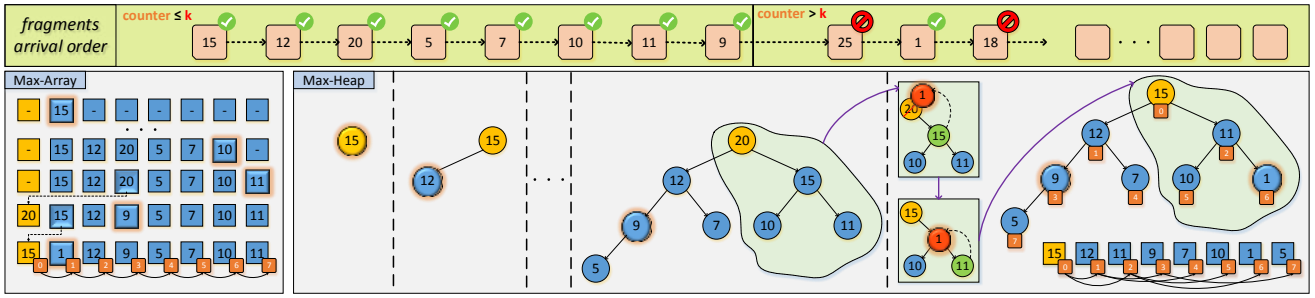

Fig. 2: Overview of the insertion process of an arbitrary sequence of out-of-order fragments when (left) max-array and (right) max-heap data structures with $k = 8$ are utilized. The incoming fragment in each step is highlighted with a glow effect. When the array is full, fragments with value larger than the maximum captured fragment (yellow-colored) are efficiently discarded (f_8.z= 25 and f_{10}.z= 18).

of generated fragments at pixel $p[x, y]$), behaving more or less as fast as AB_{Array}.

To ensure concurrency in all cases, a per-pixel atomic counter indicates the next available address position for the incoming fragment. Performance-wise, this idea is not suggested to be applied at the heap structure since a costly *heapify* operation must be performed (to ensure both heap properties) after the heap becomes completely full. The code upgrade needed to revise fragment culling shader is illustrated with blue in Algorithm 2.

Algorithm 2 K$^+$B-Array-R (Array a, Tex t, Pix p, Frag f, Int k)

```
1: p.counter ← p.counter+1;          ▷ atomic index increment
2: if p.counter < k then             ▷ first k fragments
3:     a[k − p.counter − 2] := f;    ▷ fast fragment store
4: else
5: if f.z < a[0].z then              ▷ fragment culling
6:     MutualExclusion(t, p);        ▷ slow fragment store (Alg. 1)
7: end if
8: end if
▷ where 'text' defines the revised fragment culling implementation
```

3.2.2 Generalized k^+-buffer

Without loss of generality, our framework may be modified to capture any unsorted sequence of k fragments (not being restricted to the k-closest ones). Instead of culling using as criteria the depth value of each incoming fragment f_i, we may remove the fragment node that generates the largest error to the integration over a general objective function $g(f_i)$ subject to a number of inequality constraints $y_j(x)$. Thus, fragment culling condition in Algorithm 2 can be generalized as the minimization of:

$$\min_{y_j(x)\leq 0, j=1\ldots J} g(f_i) = h(f_i) - h(a[0]) \qquad h(a[0]) = \max\{h(\mathsf{a}_{i-1}[:])\} \tag{2}$$

In the special case of K$^+$B-Heap, the formulation of culling optimization equals to:

$$h(f_i) = f_i.\text{z} \text{ and } y_1(p) = k - p.\text{counter} \tag{3}$$

since $a_i[0]$ is altered after each successful fragment insertion even from the first $i < k$ ones. However, due to our choice to store fragments without sorting them on the fly, applications that demand the depth-order property after every fragment insertion, like *Adaptive Transparency* [48], are hard to implement without altering the core data structure.

3.3 Fragment Sorting

Finally, a sorting process is employed to reorder the fragments for each pixel before generating the final image (**resolve pass**). Unsorted fragments are initially copied into a local array of size k per pixel before performing the depth sort, as it is relatively faster to perform read-write operations in the local space rather in the global graphics memory. Finally, based on the number of captured fragments, a mechanism decides which sorting algorithm is applied to each individual pixel [33]. Despite its quadratic complexity, *insertion sort* is faster for sorting small fragment sequences. When the number of generated fragments increases ($f(p) > 16$), *shell sort* is preferred. Note that the *backward memory allocation* [38] can easily be employed to harness local memory cache overflow and latency issues.

4 Memory-aware Extensions

Aiming at optimizing the GPU memory that a k-buffer-based rendering framework will allocate for frame buffers (with respect to local and global memory), we have extended our framework with four novel components:

- A pipeline extension is initially introduced to dynamically and precisely handle graphics memory allocation (Sec. 4.1).
- The value of k may be adaptively computed by analyzing the depth distribution histogram of the rasterized scene based on the specified application objectives and GPU global memory constraints (Sec. 4.2).
- Depending the adjusted/determined k, the proposed method can also be considered as a unified framework that successfully integrates

the functionalities of Z-buffer, k-buffer and A-buffer (Sec. 4.3).
- Finally, to avoid overflow and thread swap waiting that occurs at the local GPU memory, the K$^+$B-Array idea is also utilized to perform l^+-*depth-sorting*, reusing a constant amount (l) of allocated space (Sec. 4.4).

4.1 Precise Memory Allocation

Similar to all k-buffer alternatives where k is the same for all pixels, both K$^+$B-Array and K$^+$B-Heap require potentially large and unused pre-allocated storage requirements for pixels that contain less than k fragments (k-*fragmentless* pixels). For example, Figure 1 illustrates the wastefully allocated storage of a 4-buffer for (top) a pixel that consists of two fragments and (bottom) an empty-pixel. Note that the value of k is not automatically adjusted based on the rasterized scene and must be carefully set a priori by the user.

Inspired by AB$_{\text{SB}}$ [17], we introduce a memory-aware implementation using two geometry passes (**K$^+$B-AB$_{\text{SB}}$**). Memory is linearly organized into *variable contiguous regions* for each pixel, making it feasible to implement both proposed data structures. A precise allocation of the required memory space is achieved by performing an initial geometry rendering (**count pass**), which sums up the number of fragments covering each pixel via hardware occlusion queries. Contrary to AB$_{\text{SB}}$, where all fragments contribute to the per-pixel aggregation, we bound the number of fragments that affect a pixel by k when $f(p) > k$. For each incoming fragment, a per-pixel counter is atomically incremented (p.counterTotal). When the value of the counter reaches k, the subsequently arriving fragments are discarded. Then, the memory offset lookup table (**referencing pass**) is computed in a parallel fashion exploiting sparsity in pixel space. Finally, per-pixel counters (p.counter) are initialized to zero to guide the subsequent storing phase.

To increase memory caching, and therefore performance, we cluster non-empty pixels based on a uniform square tiling strategy replacing the pixel-column hashing function of the initial implementation. For additional information of the algorithmic details and shader implementations of this pass, readers are referred to the original AB$_{\text{SB}}$ paper [17].

A geometry rasterization is employed to store the most significant fragments to a hybrid buffer scheme starting from the memory offsets computed for each pixel. Knowing its fragment cardinality a priori (p.counterTotal), each pixel can efficiently choose the fastest way of storing its fragments in either a max-array or a max-heap storage. Since max-array structure inserts elements faster than max-heap, when the capacity is not full and k stays small, we apply the following strategy: if $f(p) > k$ and $k > 16$ then we pick max-heap, otherwise we use the max-array data structure as storage buffer.

In terms of performance, accessing random memory for concurrently storing all fragments becomes a significant bottleneck as opposed to the original single-pass versions, which benefits from the fast operations in sequential memory space. Last but not least, the need of an additional geometry rendering step also adds a tessellation-dependent computation cost.

The missing components for fulfilling K$^+$B-AB$_{\text{SB}}$ pipeline, including the original as well its dynamic version (discussed below), are shown in Algorithm 3. Note that insert_empty() and insert_full() are the abstract insertion functions. K$^+$B-Array and K$^+$B-Heap own versions of both functions are available in the source code provided as supplementary material.

4.2 Maximum Captured Layers (k) Prediction

Manually setting k for arbitrary geometry and viewing configuration easily leads to either a poor coverage of the depth complexity or an overestimation of its value, due to potential camera movement or animated/dynamically generated geometry. This inevitably results in visible and view-dependent artifacts or bad memory utilization.

We present here an intuitive and automatic method for the prediction of the maximum layers k captured by a k-buffer, based on the histogram of the per-pixel depth complexity. The idea is to perform an additional geometry pass (or easily embedded at the count pass of K$^+$B-AB$_{\text{SB}}$) to generate the depth complexity histogram on the GPU, followed by a host-side process that minimizes the allocated size of k according to different application goals and GPU memory limitations.

Generally, the essential design goal behind this process, and subsequently the principles and rules to adhere, depend on the pertinent application. Usually, we need to estimate a value for k that conforms with the total k-buffer memory bound while achieves the desired quality level. In this work, we adjust k aiming at matching a desired *fragment hit ratio* R_h as a quality goal, also called *robustness ratio*: the total number of extracted fragments over the total number of generated fragments. R_h is efficiently computed using hardware-accelerated occlusion queries at the count pass. The user-specified hard constraint of the maximum memory budget M_b is provided in MB.

Although, histogram generation is an inherently sequential operation, where thousands of pixels contribute votes to a reduced set of disjoint categories, known as *bins*, the analyzed variable (in our case *fragment samples*) accumulation can be carried out via atomic additions available on modern graphics hardware (see **histogram pass** in Algorithm 3). With the current hardware limitation on the number of available atomic counters, we can handle a per-fragment depth complexity up to 1024, which is more than adequate for most scenes.

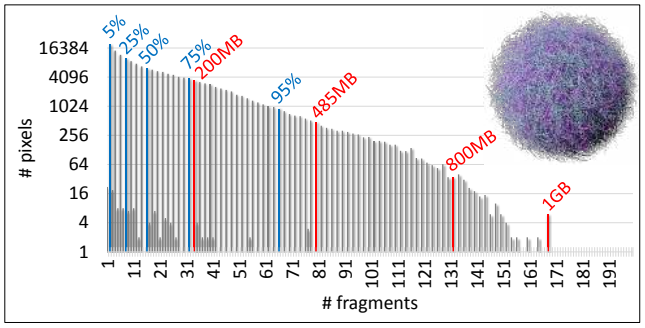

Fig. 3: Depth complexity histogram ($\log_2$ scale) of Hairball rasterization. Observe the computed k for different values of GPU memory and robustness ratio.

Given the width I_w and height I_h of the resolution of the rendering image I and the storage requirements of the selected k-buffer algorithm, which in general can be expressed as $x \cdot k + y$ bytes per pixel, we can compute an upper bound k_b for the final k based on the memory budget M_b as follows:

$$M_b = \frac{(x \cdot k_b + y) \cdot I_w \cdot I_h}{1024^2} \Leftrightarrow \quad (4)$$

$$k_b = \frac{1024^2 \cdot M_b / (I_w \cdot I_h) - y}{x} \quad (5)$$

Then, traversing the generated depth complexity histogram h from back-to-front, which is first transferred to the CPU from the GPU [49] and *normalized*, we can measure how many fragments we miss so far $(1023 \ldots k_b)$ and continue until the robustness ratio goal is met. This optimization can be formulated as:

$$\min_{k \le k_b} \left\{ \frac{\sum_{i=1023}^{k} (i-k) \cdot h[i]}{\sum_p f(p)} - R_h \right\} \quad (6)$$

Since it is not straightforward to compute k_b in the case of K$^+$B-AB$_{\text{SB}}$ due to

$$M_b = \frac{\sum_p x \cdot \min\{f(p), k_b\} + y \cdot I_w \cdot I_h}{1024^2} \quad (7)$$

we set $k_b = 1024$ and add an inequality constraint to Eq. 6: $\sum_p x \cdot \min\{f(p), k\} \le M_b - y \cdot I_w \cdot I_h$.

Figure 3 illustrates the depth complexity histogram when rasterizing the Hairball model from a fixed point of view. Observe the highlighted k for different values of GPU memory and robustness ratio. On the other hand, Figure 4 shows how k is dynamically adjusted to fit the user constraints (M_b = 200MB which results in $k_b = 32$ and $R_h = 95\%$) in scenes where (left) the virtual camera is allowed to freely roam inside the 3D scene and (right) many elementary objects are moving and interacting with each other.

4.3 Unified Fragment Buffer

k^+-buffer can also be considered as a unified framework that successfully integrates the functionality of

Algorithm 3 Dynamic K$^+$B-AB$_{\text{SB}}$ (Array a, Hist h, Pix p, Int k)

```
1: procedure CLEAR(h,p)                          ▷ full-screen quad pass
2:     h[:] := 0;                                ▷ init histogram to zero
3:     p.address := 0; p.counterTotal := 0;
4: end procedure

5: procedure COUNT(p, k)                         ▷ geometry pass
6:     if p.counterTotal < k then                ▷ bounded accumulation
7:         p.counterTotal ← (+1);
8:     else
9:         discard;
10:    end if
11: end procedure

12: procedure HISTOGRAM(h, p)                    ▷ full-screen quad pass
13:    if p.counterTotal > 0 then
14:        h[p.counterTotal−1] ↞ (+1);           ▷ increment bin's counter
15:    end if
16: end procedure

17: procedure REFERENCING(p, k)                  ▷ full-screen quad pass
18:    p.counterTotal := min{p.counterTotal,k}   ▷ bound counter
19:    p.address :=compute_pixel_offset(p.counterTotal);
20:    p.counter := 0;
21: end procedure

22: procedure STORE(a, t, f, p, k)               ▷ geometry pass
23:    p.method := (k < 16 or p.counterTotal < k) : Array ? Heap;
24:    if p.counter < k or f.z < a[0].z then     ▷ fragment culling
25:        acquire_lock();                       ▷ enter critical section (Alg. 1)
26:        if p.counter < k then                 ▷ array is not full
27:            insert_empty(p.counter++,p.method);
28:        else if f.z < a[0].z then             ▷ fragment culling
29:            insert_full(p.method);
30:        end if
31:        release_lock();                       ▷ exit critical section (Alg. 1)
32:    end if
33: end procedure

34: procedure RESOLVE(a, p)                      ▷ full-screen quad pass
35:    l⁺-depth-sorting (a, p, 32);              ▷ fragment sort (Alg. 4)
36: end procedure
```

▷ where 'text' and 'text' respectively must be inserted and be removed from overlapping functions when *dynamic k solution* is exploited. {←, ↞} denote atomic {store, increment} operations

Z-buffer, (multi-pass) k-buffer and A-buffer depending on the user-defined or dynamically computed k value. By allocating a single entry per pixel ($k = 1$), our method ensures displaying the closest fragment to the viewer. However, this comes with the additional expense of extra memory requirements and performance downgrade when compared to the hardware depth buffering.

On the other hand, depending on memory constraints and GPU processing capacity, k value can be set large enough to avoid any fragment-overflow ($k = \max_p\{f(p)\}$). More specifically, our framework can be considered as a hybrid scheme that correctly simulates the behavior of AB$_{\text{Array}}$ (when K$^+$B is used) or AB$_{\text{SB}}$ (when K$^+$B-AB$_{\text{SB}}$ is used). Performance-wise, the semaphore-free implementation of max-array structure (Sec. 3.2.1) should then be chosen due to its constant insertion complexity when the array is not full (since $\forall p[x, y] : f(p) \le k$).

Despite the fact that our framework is not restricted from multiple render targets and samples of the anti-

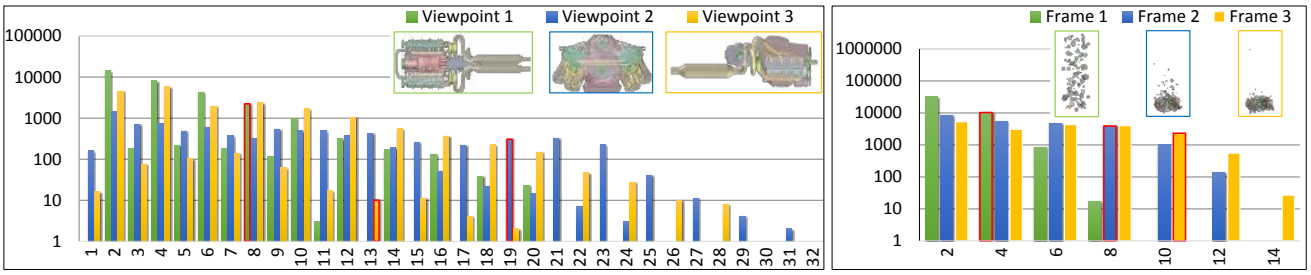

Fig. 4: Depth complexity histograms ($\log_{10}$ scale) for fixed $M_b = 200$MB and $R_h = 0.95$, when rendering (left) the motor engine model rendered from different viewing angles and (right) three frames of a multi-object animation. Observe how the k value adapts (left: $\{8, 19, 13\}$ and right: $\{4, 8, 10\}$) using our prediction process.

aliasing buffer, limited hardware resources may result at a low k ($< \max_p\{f(p)\}$). Thus, we have developed a k-depth peeling variation to achieve the functionality of an A-buffer under constrained memory demands by performing $\lceil \max_p\{f(p)\}/k \rceil$ rendering iterations. Following the formulation of Sec. 3.2.2, we insert an additional inequality constraint $y_2(f_i) = a_{i-1}[0] - f_i$ to offer multi-pass behavior using the furthest fragment of the previous iteration $i-1$ as the culling criterion for iteration i.

4.4 l^+-depth-sorting

While the fragment sorting strategy described in Sec. 3.3 does not suffer from local memory cache overflow and reduced hot swapping for small values of k ($\leq$ 64), this is not true when operating with dynamic scenarios (see Sec. 4.2). Extending the idea of fragment ordering using depth peeling at a local cache level [37], we propose a faster variation, called l^+-*depth-sorting* by extending K$^+$B-Array to increase the computational throughput as well as the maximum supported depth complexity in local memory.

In each iteration, the l-front fragments, where l is predetermined and in our tests fixed to 32, are efficiently chosen by traversing the global fragment memory and stored at an l-sized local array, following the strategy described in Sec. 3.2.1. Please note that semaphores are not needed in this case, since it is a problem treated sequentially. Subsequently, the captured fragments are sorted and resolved, finalizing the current iteration. The farthest fragment captured in this iteration is efficiently used in the following one for discarding all previously processed fragments.

The limitations of this approach are (i) its inability to handle z-fighting issues (we refer readers to a comprehensive review for eliminating this phenomenon [18]) and (ii) the requirement of reading the entire fragment list multiple times from global GPU memory ($\lceil f(p)/l \rceil$). On the other hand, the loop may terminate sooner, e.g. in the special case where opacity thresholding is performed [41] for transparent objects, avoiding lots of computations. The complete l^+-depth-peeling code is shown in Algorithm 4 (see Appendix provided as supplementary material).

Finally, Figure 5 illustrates the complete k^+-buffer framework, highlighting the flow among components (shaders) that should be followed to perform the corresponding functionality.

5 Results

We present an experimental analysis of our k^+-buffer approach versus a set of depth-peeling, k-buffer and A-buffer realizations focusing on performance, robustness, and memory requirements under different tested conditions. We have measured performance in terms of milliseconds (ms) and memory requirements in terms of megabytes (MB). For the purposes of comparison, we have developed two variations of KB-AB$_{\text{LL}}$, where instead of using per-pixel linked lists for the A-buffer construction, we have applied either fixed-length (**KB-AB$_{\text{Array}}$**) or variable-length (**KB-AB$_{\text{SB}}$**) arrays for each pixel. The shader source code from all tested methods is also provided as supplementary material. All methods are implemented using OpenGL 4.4 API and mainly performed on the NVIDIA GTX 780 Ti with 3 GB of memory. Finally, we have implemented our own semaphore mechanism instead of using INTEL's pixel synchronization to run KB-PS method on the above hardware.

Table 1 presents a comparative overview of all k-buffer alternatives with respect to memory requirements, rendering complexity, and other features.

5.1 Performance Analysis

We have performed an experimental performance evaluation of our methods against competing techniques using a collection of scenes under several different configurations. We aim to extend and update the performance analysis conclusions found in the original version of this paper [26], where experiments were carried out on an older NVIDIA GTX 480.

Instead of rendering scenes under different image resolutions, we have used a 1024×1024 viewport and performed zooming resulting in measurements at different image coverage values. For a fair comparison, all methods are tested in extreme conditions under artificially generated scenes that cover a percentage of

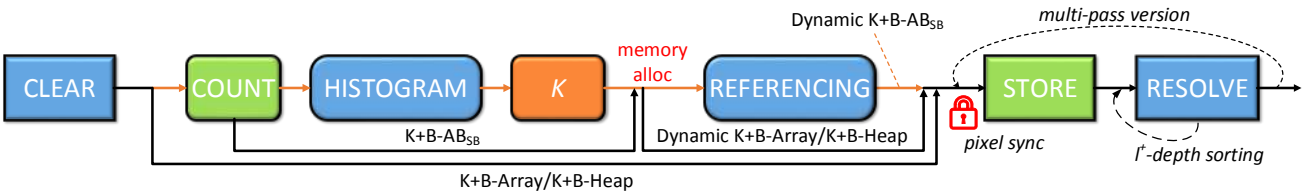

Fig. 5: Diagram of the k^+-buffer pipeline. Each box represents a shader program except the orange one which is implemented in CPU. The blue boxes are executed per-pixel using a full-screen quad rendering pass, while the green ones are executed for each geometry-rasterized fragment.

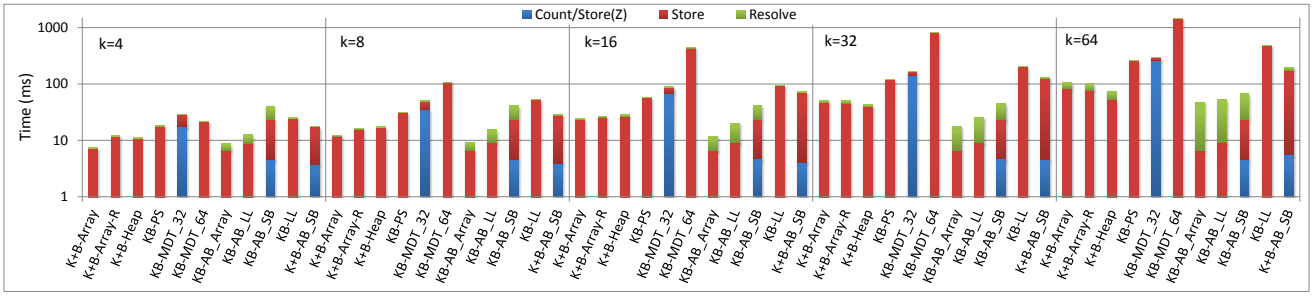

Fig. 6: Performance evaluation in ms ($\log_{10}$ scale) of k-buffer variants with varying k on a scene with $n = 128$.

screen size (or pixel density: p_d) and produce $n = r \cdot k$ randomly produced fragments per pixel, where $r \geq 1$.

5.1.1 k-buffer Comparison

Impact of k. Figure 6 shows how the computation time, for each rendering pass of a set of k-buffer methods, scales by increasing the value of $k = 4, 8, 16, 32, 64$ for a scene that consists of $n = 128$ fragments per-pixel. We observe that our K$^+$B variants perform better than the other memory-bounded techniques for all k values. As expected, K$^+$B-Heap performs better than K$^+$B-Array for larger values of k. Note that when $k = 64$, K$^+$B-Heap is up to $3.5\times$ and $4\times$ faster than the current implementations of KB-PS and KB-MDT$_{32}$, respectively. Although the one-pass KB-MDT$_{64}$ outperforms its 32-bit version for $k = 4$, major performance issues appear when fragment racing becomes more intense for larger values of k, probably due to the slower 64-bit atomic operations.

Our methods are slightly slower than the A-buffer-based ones due to the faster atomic operations on modern hardware (L2 cache). Note that the *resolve* step is more expensive for the former methods, since it has to locate the closest k fragments from all captured ones before sorting. Finally, we observe that the *count* and *resolve* passes of K$^+$B-AB$_{SB}$ cost less in terms of computations as compared to the ones of KB-AB$_{SB}$ due to the restricted operations carried out by the former. However, slow fragment storing in global memory results in a performance downgrade when the rasterized fragments are significantly increased.

Impact of Culling. Following the same tested configuration, Figure 7 illustrates how the performance significantly improves when the fragment culling mechanism is exploited to our K$^+$B-Array methods. KB-PS and KB-MDT$_{32}$ can efficiently be adjusted to support this functionality achieving a significant boost, however, not enough to beat the K$^+$B-Array performance. As expected, the mechanism soothes when k increases.

Unfortunately, the proposed fragment rejection process suffers from several **limitations**. First of all, the process initially requires the insertion of k fragments before it starts applying any culling test. Second, it depends on the fragment incoming depth order; having no impact at the worst case scenario of fragments arriving in descending order. Furthermore, the actual fragment elimination is unfortunately performed inside the pixel shader execution, avoiding the performance gain of exploiting the hardware-accelerated early-Z culling.

Impact of Memory Constraints. Figure 8 illustrates the performance evaluation in terms of ms per MB for a tested k-buffer method set when performance and memory are of utmost importance. To construct k-fragmentless pixels, we allow pixels to be influenced by up to $n = 10 \cdot k$ fragments. Thus, we define f_p as the probability of a generated fragment not to be discarded. We observe that K$^+$B-AB$_{SB}$ is preferred to be used for handling scenes with many empty pixels ($p_d = 25\%$) and small numbers of rasterized fragments ($f_p = 25\%$). When pixel and fragment densities increase ($p_d = 75\%, f_p = 75\%$), K$^+$B-AB$_{SB}$ performs better than the rest memory-aware methods. However, K$^+$B-AB$_{SB}$ behavior is normally worst than the bounded methods since it theoretically performs slower (e.g. one extra pass, storing data in global memory) in conjunction with the small unused mem-

ory of the bounded methods. Despite the fast speed of KB-AB_{SB} on sparse scenes, performance is significantly reduced when generated fragments blast off to high levels. Finally, KB-AB_{Array}, KB-AB_{SB} and KB-LL fail to work when fragment allocation results in memory overflow ($k = 64$).

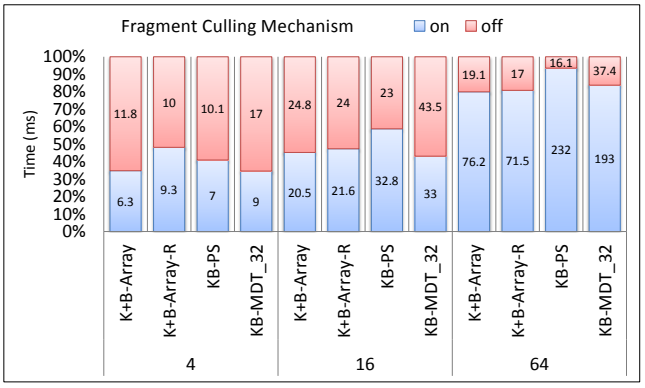

Fig. 7: Performance evaluation of k-buffer variants with and without enabling our fragment culling mechanism for $k = 4, 16, 64$.

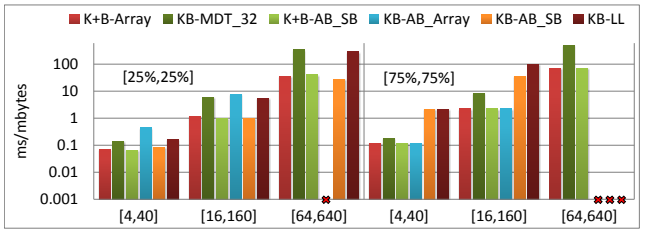

Fig. 8: Performance evaluation comparison in ms/MB ($\log_{10}$ scale) of k-buffer variants when moving from a scene with $r = 10$ from small number towards a large number of generated fragments.

Impact of GPU (Synchronization). Figure 9 (left) illustrates the performance evaluation for a tested k-buffer method set on different graphics hardware and synchronization implementations for fixed $k = 8$ and varying $n \in \{8, 16, 32\}$. Initially, we observe the superiority of hardware-accelerated PS/FSI extensions when compared with our SL implementation on both NVIDIA GTX 970 and Intel Iris Pro Graphics 5200 cards, respectively. Notice that all syncing techniques experience linear behavior when moving to higher fragment racing. An interesting note is that K$^+$B-Array has even better performance than KB-PS when syncing is hardware accelerated (Nvidia: $\approx 2\times$, Intel: $\approx 1.5\times$). When comparing the results on the older (GTX 480) versus the one on the modern (GTX $\{780$ Ti, $970\}$) graphics cards, we observe the enhanced behavior of KB-MDT_{32} and KB-AB_{Array} due to the advanced atomic operations at the latter cards.

5.1.2 Depth Peeling/A-buffer Comparison

Figure 9 (right) illustrates performance comparison of our methods against depth peeling and A-buffer alternatives for a scene with varying depth complexity. In this setting, k is set to the fragment cardinality, so that K$^+$B methods are able to capture all generated fragments. We initially observe that our multi-pass version of K$^+$B-Array with $k = \{8, 32\}$ performs better than the widely-used F2B and DUAL methods due to the additional rendering iterations of the latter ones. Note that it is preferred to use F2B for handling scenes with low geometric detail (small number of triangles) under high resolutions [18].

We observe that the revised version of K$^+$B-Array alleviates the burden of the unnecessary culling mechanism performing better from all memory-aware A-buffer variants and slightly worse than AB_{Array}, the fastest A-buffer implementation so far. Note that l^+-depth peeling improved sorting performance by $3.1\times, n = 64$ and $4.64\times, n = 128$. We omit to test K$^+$B-Heap since K$^+$B-Array performs slightly better than the former, enhanced by its constant-time insertion process on an unfilled array. On the other hand, K$^+$B-AB_{SB} is worse than AB_{SB} in all cases. Except from the unnecessary culling cost, the increased fragment syncing significantly affects performance.

5.2 Memory Allocation Analysis

Table 1 presents complexity in terms of memory consumption for all available methods that more or less simulate the behavior of k-buffer. We initially observe that our K$^+$B methods require slightly more storage (8-byte) per pixel than the rest of the memory bounded methods (KB, KB-SR, KB-PS, KB-$MDT_{32,64}$, KB-MHA) due to the additional allocation of the counter and semaphore textures. When moving to extreme screen resolutions this burden is noticeable. KB, KB-Multi and KB-MHA methods need more storage when data packing is explored ($\forall k > 1 : 4k > 2k + 2$). K$^+$B methods require less memory resources when compared to the KB-SR ($\forall k > 2 : 3k > 2k + 2$). Note that semaphore texture allocation is further avoided when the pixel synchronization extension is employed on Haswell or Maxwell hardware. On the other hand, video-memory consumption blasts off to high levels when A-buffer is constructed. Observe the increased memory requirements of KB-AB_{Array} due to its strategy to allocate the maximum memory per pixel p ($n = \max_p\{f(p)\} \gg k$). KB-AB_{LL}, KB-LL, KB-AB_{SB} require less storage resources by dynamically allocating storage only for non-empty pixels ($f(p) \in [1, n]$). Our memory-aware method K$^+$B-AB_{SB} requires equal (when $f(p) \leq k$) or less (when $f(p) > k$) storage than the unbounded A-buffer-based methods reducing the risk of a memory overflow. An interesting observation is that the K$^+$B-Array and K$^+$B-AB_{SB} when extended to capture all

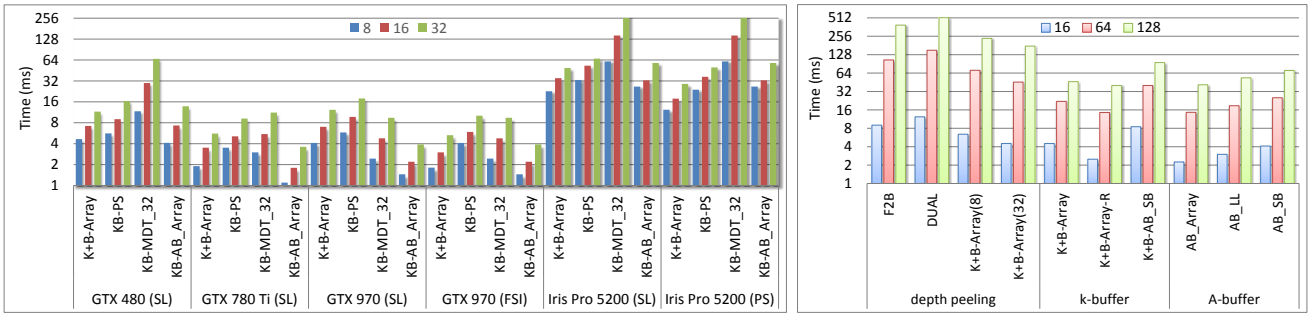

Fig. 9: Performance evaluation in ms ($\log_2$ scale) of (left) k-buffer alternatives on different graphics hardware and syncing implementations and (right) depth peeling and A-buffer alternatives on scenes with varying n.

Algorithm		Performance	Sorting need		Peeling Accuracy		Memory	
Acronym	Description	Geometry Passes	on primitives	on fragments	Max *k*	Artifacts	Per Pixel Allocation	Fixed
KB	Initial *k* -buffer implementation [20],[42]	1	✓	STORE	8; 16	MRT Hazards, Geom. Interpen.	2k; 4k ★ ; ★★	✓
KB-Multi	Multi-pass *k* -buffer [23]	1 to k	✓	RESOLVE		Geom. Interpen.	2k; 4k ★ ; ★★	
KB-SR	Stencil routed *k* -buffer [21]	1	✓	RESOLVE	32		3k ★☆	
KB-PS	*k* -buffer using pixel synchronization [24]	1	✗	STORE	-	✗	2k ★	
K⁺B-Array	k^+-buffer using max-array	1	✗	RESOLVE	-	✗	2k + 2 ★☆	
K⁺B-Heap	k^+-buffer using max-heap	1	✗	RESOLVE	-	✗	2k + 2 ★☆	
KB-MDT$_{32}$	Multi depth test scheme (32 bit) [22],[28]	2	✗	STORE	-	✗	2k ★	
KB-MDT$_{64}$	Multi depth test scheme (64 bit) [45]	1	✗	STORE	-	Can store only 32bit color	2k ★	
KB-MHA	Memory-hazard-aware *k* -buffer [43]	1	✓	STORE	8; 16	MRT Hazards, Geom. Interpen.	2k; 4k ★ ; ★★	
KB-AB$_{Array}$	*k* -buffer based on A-buffer (fixed-size arrays)	1	✗	RESOLVE	-	memory overflow risks	2n + 1 ★★ … ★	
KB-AB$_{LL}$	*k* -buffer based on A-buffer (dynamic linked lists) [9],[47]	1	✗	RESOLVE	-		3f + 1 ★★☆	✗
KB-LL	*k* -buffer based on Linked Lists [9]	1	✗	STORE	-		3f + 6 ★★☆	
KB-AB$_{SB}$	*k* -buffer based on S-buffer (variable-contigious regions)	2	✗	RESOLVE	-		2f + 2 ★★	
K⁺B-AB$_{SB}$	k^+-buffer based on S-buffer (variable-contigious regions)	2	✗	RESOLVE	-	✗	2f$_k$ + 3 ★☆	
f(p) = # *fragments at pixel* p[x,y]		n = max$_{x,y}${f(p)}			In A ; B, A denotes the layers/memory for the basic method and B for the variation using attribute packing			
f$_k$(p) = (f(p) < k) ? f(p) : k		f$_k$(p) ≤ k						

TABLE 1: Comprehensive comparison of the prior k-buffer approaches and the introduced k^+-buffer variants. Intuitively, more stars indicate larger memory requirements. Fragment sorting column indicates in which stage depth-ordering takes place.

fragments ($k = n$) require the same storage compared to the AB$_{\text{Array}}$ and AB$_{\text{SB}}$ methods, respectively. Finally, the tiny histogram allocation (4KBytes, resolution-independent) is the only storage demand when the dynamic k process is enabled.

5.3 Image Quality Analysis

Figure 10 shows the image differences of KB, KB-SR, KB-MDT$_{32}$, KB-LL and KB-AB$_{\text{Array}}$ methods when compared with the ground truth on rendering Hairball (180 max layers) with $k = 16$ (supported by all tested methods). Noticeable quality downgrade is observed in the bottom three images due to RMW hazards of (left) KB and (center) KB-SR methods as well as (right) depth conversion artifacts of KB-MDT$_{32}$. KB-LL is consistently constrained to guarantee an infinite loop-free behavior from repeated failed insertions of candidate fragments. (top, center) This results in a noticeable fragment loss. To avoid memory overflow of KB-AB$_{\text{Array}}$, we must allocate less storage (120 layers) than we actually need leading at (top, right) an information loss for a small pixel set.

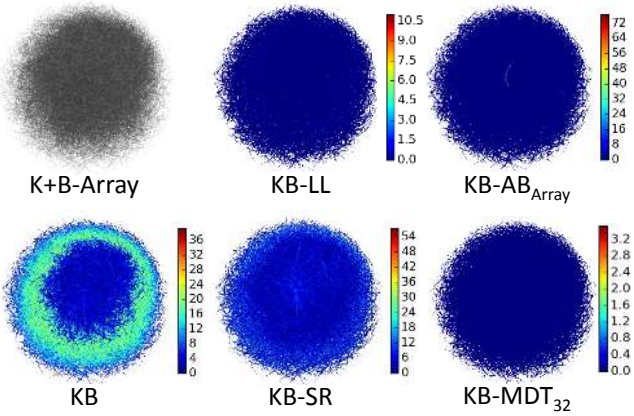

Fig. 10: Heatmap-coded differences between the image generated using K$^+$B-Array against the outputs of several k-buffer variants.

Considering completeness, KB, KB-SR, KB-MHA and KB-MDT$_{64}$ are limited to produce visual effects exploiting fragment features that can be compressed

to a 32-bit compact vector. Finally, k^+-buffer may easily be adapted to support multi-sample anti-aliasing (**MSAA**) by following the widely-accepted fragment coverage-based strategy [5].

6 Conclusions

We have introduced k^+-buffer, an improved pixel-synchronized and fragment culling-aware *k*-buffer framework built on two novel bounded array data structures maintained on the GPU. An additional geometry rendering may be also carried out to avoid the tedious task of manually tweaking the value of *k* by predicting a suitable value via on-the-fly depth complexity histogram analysis. A precise memory allocation strategy has also been incorporated into the proposed pipeline, tailoring storage utilization to the depth complexity of individual pixels. Implementation details and light-weight alternative implementations of various steps have been also provided to enable full support of our system on various GPU architectures. An extensive experimental comparison has demonstrated the superiority of our framework as compared to previous *k*-buffer solutions with regard to memory, performance and image quality.

Further directions may be explored for tackling the problem of visibility determination in multi-fragment rendering solutions. While our fragment outlier rejection mechanism is a key feature to our framework, additional research has to be conducted to alleviate its order-dependence performance nature. Additionally, fragment culling acceleration may be achieved by taking advantage of temporal coherence across adjacent frames [50]. Finally, our framework can be combined with an attention-based level-of-detail manager to downgrade fragment storage in areas that are expected to go unnoticed by an observer [51].

Acknowledgments

Thanks to Anthousis Andreadis and the rest of AUEB Computer Graphics Group for their contributions and support. Andreas Vasilakis was supported by hardware donations from Intel. *Stanford dragon* and *hairball* models were downloaded from Morgan McGuire's Computer Graphics Archive. *Marbles* animation was obtained from the University of Utah 3D Animation Repository. We would like to thank Louis Bavoil for providing us the *motor engine* mesh. 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). Andreas Vasilakis is the corresponding author.

References

[1] Y. Tokuyoshi, T. Sekine, T. da Silva, and T. Kanai, "Adaptive ray-bundle tracing with memory usage prediction: Efficient global illumination in large scenes," *Computer Graphics Forum*, vol. 32, no. 7, pp. 315–324, 2013.

[2] M. Salvi and K. Vaidyanathan, "Multi-layer alpha blending," in *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '14. New York, NY, USA: ACM, 2014, pp. 151–158.

[3] K. E. Hillesland, B. Bilodeau, and N. Thibieroz, "Deferred shading for order-independent transparency," in *Proceedings of Eurographics 2014 Short Papers*, ser. EG '14. The Eurographics Association, 2014, pp. 49–52.

[4] L. Szécsi, P. Barta, and B. Kovács, "Volumetric transparency with per-pixel fragment lists," *GPU PRO 3: Advanced Rendering Techniques*, p. 323, 2012.

[5] J. C. Yang, J. Hensley, H. Grun, and N. Thibieroz, "Real-time concurrent linked list construction on the GPU," *Computer Graphics Forum*, vol. 29, no. 4, pp. 1297–1304, 2010.

[6] R. Carnecky, R. Fuchs, S. Mehl, Y. Jang, and R. Peikert, "Smart transparency for illustrative visualization of complex flow surfaces," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 5, pp. 838–851, May 2013.

[7] D. Kauker, M. Krone, A. Panagiotidis, G. Reina, and T. Ertl, "Rendering molecular surfaces using order-independent transparency," in *Proceedings of the 13th Eurographics Symposium on Parallel Graphics and Visualization*, ser. EGPGV '13. Aire-la-Ville, Switzerland: Eurographics Association, 2013, pp. 33–40.

[8] J. Parulek and A. Brambilla, "Fast blending scheme for molecular surface representation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2653–2662, Dec. 2013.

[9] X. Yu, J. C. Yang, J. Hensley, T. Harada, and J. Yu, "A framework for rendering complex scattering effects on hair," in *Proceedings of the 2012 symposium on Interactive 3D Graphics and Games*, ser. I3D '12. NY, USA: ACM, 2012, pp. 111–118.

[10] Y.-S. Leung and C. C. L. Wang, "Conservative sampling of solids in image space," *IEEE Compututer Graphics and Applications*, vol. 33, no. 1, pp. 32–43, Jan. 2013.

[11] N. K. Govindaraju, M. Henson, M. C. Lin, and D. Manocha, "Interactive visibility ordering and transparency computations among geometric primitives in complex environments," in *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, ser. I3D '05. NY, USA: ACM, 2005, pp. 49–56.

[12] E. Sintorn and U. Assarsson, "Real-time approximate sorting for self shadowing and transparency in hair rendering," in *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, ser. I3D '08. NY, USA: ACM, 2008, pp. 157–162.

[13] G. Chen, P. V. Sander, D. Nehab, L. Yang, and L. Hu, "Depth-presorted triangle lists," *ACM Trans. Graph.*, vol. 31, no. 6, pp. 160:1–160:9, Nov. 2012.

[14] J. Huang and M. B. Carter, "Interactive transparency rendering for large cad models," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 5, pp. 584–595, Sep. 2005.

[15] J. Rossignac, I. Fudos, and A. A. Vasilakis, "Direct rendering of boolean combinations of self-trimmed surfaces," *Computer Aided Design*, vol. 45, no. 2, pp. 288–300, Feb. 2013.

[16] C. Crassin, "Linked lists of fragment pages," 2010.

[17] A. A. Vasilakis and I. Fudos, "S-buffer: Sparsity-aware multi-fragment rendering," in *Proceedings of Eurographics 2012 Short Papers*, ser. EG '12, Cagliari, Sardinia, Italy, 2012, pp. 101–104.

[18] A.-A. Vasilakis and I. Fudos, "Depth-fighting aware methods for multifragment rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 6, pp. 967–977, 2013.

[19] M. Maule, J. L. Comba, R. P. Torchelsen, and R. Bastos, "A survey of raster-based transparency techniques," *Computers & Graphics*, vol. 35, no. 6, pp. 1023 – 1034, 2011.

[20] L. Bavoil, S. P. Callahan, A. Lefohn, J. L. D. Comba, and C. T. Silva, "Multi-fragment effects on the GPU using the *k*-buffer," in *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, ser. I3D '07. NY, USA: ACM, 2007, pp. 97–104.

[21] L. Bavoil and K. Myers, "Deferred rendering using a stencil routed *k*-buffer," *ShaderX6: Advanced Rendering Techniques*, pp. 189–198, 2008.

[22] M. Maule, J. Comba, R. Torchelsen, and R. Bastos, "Hybrid transparency," in *Proceedings of the 2013 Symposium on Interac-*

tive 3D Graphics and Games, ser. I3D '13. New York, NY, USA: ACM, 2013, pp. 103–118.
[23] B. Liu, L.-Y. Wei, Y.-Q. Xu, and E. Wu, "Multi-layer depth peeling via fragment sort," in *Proceedings of 11th IEEE International Conference on Computer-Aided Design and Computer Graphics*, 2009, pp. 452–456.
[24] M. Salvi, "Advances in real-time rendering in games: Pixel synchronization: Solving old graphics problems with new data structures," in *ACM SIGGRAPH 2013 Courses*, ser. SIGGRAPH '13. New York, NY, USA: ACM, 2013.
[25] Z. Bolz and M. Heyer, "OpenGL extension: GL_NV_fragment_shader_interlock," 2014.
[26] A. A. Vasilakis and I. Fudos, "k^+-buffer: Fragment synchronized k-buffer," in *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '14. New York, NY, USA: ACM, 2014, pp. 143–150.
[27] L. Carpenter, "The A-buffer, an antialiased hidden surface method," in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, vol. 18, no. 3. ACM New York, NY, USA, 1984, pp. 103–108.
[28] N. Thibieroz, "Order-independent transparency using per-pixel linked lists," *GPU Pro 2*, pp. 409–431, 2011.
[29] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu, "FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects," in *Proceedings of the 2010 Symposium on Interactive 3D Graphics and Games*, ser. I3D '10. New York, NY, USA: ACM, 2010, pp. 75–82.
[30] C. Crassin, "Fast and accurate single-pass A-buffer," 2010.
[31] M. Maule, J. L. Comba, R. Torchelsen, and R. Bastos, "Memory-optimized order-independent transparency with dynamic fragment buffer," *Computers & Graphics*, vol. 38, pp. 1 – 9, 2014.
[32] A. Patney, S. Tzeng, and J. D. Owens, "Fragment-parallel composite and filter," *Computer Graphics Forum*, vol. 29, no. 4, pp. 1251–1258, 2010.
[33] P. Knowles, G. Leach, and F. Zambetta, "Efficient layered fragment buffer techniques," in *OpenGL Insights*, P. Cozzi and C. Riccio, Eds. CRC Press, 2012, pp. 279–292.
[34] P. Knowles, G. Leach, and F. Zambetta, "Fast sorting for exact oit of complex scenes," *The Visual Computer*, vol. 30, no. 6-8, pp. 603–613, 2014.
[35] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu, "Efficient depth peeling via bucket sort," in *Proceedings of the 2009 Conference on High Performance Graphics*, ser. HPG '09. New York, NY, USA: ACM, 2009, pp. 51–57.
[36] E. Sintorn and U. Assarsson, "Hair self shadowing and transparency depth ordering using occupancy maps," in *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, ser. I3D '09. New York, NY, USA: ACM, 2009, pp. 67–74.
[37] S. Lindholm, M. Falk, E. Sundn, A. Bock, A. Ynnerman, and T. Ropinski, "Hybrid data visualization based on depth complexity histogram analysis," *Computer Graphics Forum*, 2014.
[38] P. Knowles, G. Leach, and F. Zambetta, "Backwards Memory Allocation and Improved OIT," in *Proceedings of Pacific Graphics 2013 (Short Papers)*, ser. PG '13, October 2013, pp. 59–64.
[39] C. Everitt, "Interactive order-independent transparency," Nvidia Corporation, Tech. Rep., 2001.
[40] L. Bavoil and K. Myers, "Order independent transparency with dual depth peeling," Nvidia Corp., Tech. Rep., 2008.
[41] D. Wexler, L. Gritz, E. Enderton, and J. Rice, "GPU-accelerated high-quality hidden surface removal," in *Proceedings of the 2005 conference on Graphics Hardware*, ser. HWWS '05. New York, NY, USA: ACM, 2005, pp. 7–14.
[42] N. Carr, R. Měch, and G. Miller, "Coherent layer peeling for transparent high-depth-complexity scenes," in *Proceedings of the 2008 Symposium on Graphics Hardware*, ser. GH '08. Aire-la-Ville, Switzerland: Eurographics Association, 2008, pp. 33–40.
[43] S. P. Callahan, M. Ikits, J. L. D. Comba, and C. T. Silva, "Hardware-assisted visibility sorting for unstructured volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 3, pp. 285–295, May 2005.
[44] N. Zhang, "Memory-hazard-aware k-buffer algorithm for order-independent transparency rendering," *IEEE Trans. on Visualizat. and Comput. Graph.*, vol. 20, no. 2, pp. 238–248, 2014.
[45] W. Wang and G. Xie, "Memory-efficient single-pass GPU rendering of multifragment effects," *IEEE Transact. on Visualization and Computer Graphics*, vol. 19, no. 8, pp. 1307–1316, 2013.
[46] C. Kubisch, "Order independent transparency in OpenGL 4.x," in *GPU Technology Conference 2014*, ser. GTC '14, 2014.
[47] P. Brown, Z. Bolz, C. Crassin, and C. Kubisch, "OpenGL extension: GL_nv_shader_atomic_int64," 2014.
[48] M. Salvi, J. Montgomery, and A. Lefohn, "Adaptive transparency," in *Proceedings of the 2011 Symposium on High Performance Graphics*. NY, USA: ACM, 2011, pp. 119–126.
[49] L. Hrabcak and A. Masserann, "Asynchronous buffer transfers," in *OpenGL Insights*, P. Cozzi and C. Riccio, Eds. CRC Press, 2012, pp. 391–414.
[50] D. Scherzer, L. Yang, O. Mattausch, D. Nehab, P. V. Sander, M. Wimmer, and E. Eisemann, "Temporal coherence methods in real-time rendering," *Computer Graphics Forum*, vol. 31, no. 8, pp. 2378–2408, Dec. 2012.
[51] S. Lee, G. J. Kim, and S. Choi, "Real-time tracking of visually attended objects in virtual environments and its application to LOD," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 1, pp. 6–19, Jan. 2009.

Andreas-Alexandros Vasilakis received his PhD degree on the field of Computer Graphics from the Department of Computer Science & Engineering of the University of Ioannina in Greece, under the supervision of Prof. Ioannis Fudos. He has also received MSc and BSc degrees from the same institution in 2008 and 2006, respectively. Since March 2014, he has joined the Graphics Group at the Department of Informatics in Athens University of Economics and Business, where he is currently working on interactive framebuffer techniques as well as inverse lighting and global illumination problems.

Georgios Papaioannou received a BSc in Computer Science in 1996 and a PhD degree in Computer Graphics and Pattern Recognition in 2001, both from the University of Athens, Greece. He is currently an assistant professor at the Department of Informatics of the Athens University of Economics and Business and his research is focused on real-time computer graphics algorithms, photorealistic rendering, virtual reality, 3D pattern recognition and computational archaeology. Since 1997, he has worked as a research fellow and principal investigator in many national and EU-funded research and development projects. Prof. Papaioannou is also a member of ACM, SIGGRAPH, Eurographics Association and has been a member of the program committees of many conferences in the above fields.

Ioannis Fudos received the diploma in computer engineering and informatics from the University of Patras, Greece, in 1990 and the MSc and PhD degrees in computer science both from Purdue University in 1993 and 1995, respectively. He is an associate professor in the Department of Computer Science & Engineering at the University of Ioannina. His research interests include animation, rendering, morphing, CAD systems, reverse engineering, geometry compilers, solid modeling, and image retrieval. He has published in well established journals and conferences and has served as reviewer in various conferences and journals. He has received funding from EC, the General Secretariat of Research and Technology, Greece, and the Greek Ministry of National Education and Religious Affairs. He is a member of the IEEE.

APPENDIX

In this appendix, we provide an analytic pseudo-code description of the l^+-depth-sorting algorithm (more details at Section 4.4), essential component of the updated k^+-buffer pipeline (see also Figure 5 and Algorithm 3).

Algorithm 4 l^+-depth-sorting (Array a, Pixel p, Int s)

```
1: Array l[s];                                  ▷ allocate local array of length s
2: for i := 0 to ⌈p.counter/s⌉ do               ▷ total iterations
3:     num := 0; l[s − 1].z := 1.0; prev_max_z = 0.0;
4:     for j := 0 to p.counter do               ▷ traverse global memory
5:         if a[j].z < prev_max_z then
6:             continue;                        ▷ cull processed fragments
7:         end if
8:         if num < s − 1 then
9:             l[num++] := a[j];                ▷ insert to unfilled array
10:        else if a[j].z < l[s − 1].z then
11:            l[find_max()] := a[j];           ▷ insert to full array
12:        end if
13:    end for
14:    prev_max_z := l[s − 1].z;
15:    if num ≤ 16 then                         ▷ hybrid sorting
16:        insertion_sort(l, num);
17:    else
18:        shell_sort(l, num);
19:    end if
20:    compute_effect(l, num);                  ▷ fragment composition
21: end for
  ▷ where find_max() returns the index of the maximum value at l
```