EUROGRAPHICS 2020
R. Mantiuk and V. Sundstedt
(Guest Editors)

Volume 39 (2020), Number 2
STAR – State of The Art Report

A Survey of Multifragment Rendering

A. A. Vasilakis[†1] and K. Vardis[†1] and G. Papaioannou[1]

[1]Department of Informatics, Athens University of Economics and Business, Greece

Abstract

In the past few years, advances in graphics hardware have fuelled an explosion of research and development in the field of interactive and real-time rendering in screen space. Following this trend, a rapidly increasing number of applications rely on multifragment rendering solutions to develop visually convincing graphics applications with dynamic content. The main advantage of these approaches is that they encompass additional rasterised geometry, by retaining more information from the fragment sampling domain, thus augmenting the visibility determination stage. With this survey, we provide an overview of and insight into the extensive, yet active research and respective literature on multifragment rendering. We formally present the multifragment rendering pipeline, clearly identifying the construction strategies, the core image operation categories and their mapping to the respective applications. We describe features and trade-offs for each class of techniques, pointing out GPU optimisations and limitations and provide practical recommendations for choosing an appropriate method for each application. Finally, we offer fruitful context for discussion by outlining some existing problems and challenges as well as by presenting opportunities for impactful future research directions.

CCS Concepts
• ***Computing methodologies*** → ***Rasterization; Visibility;***

1. Introduction

Multifragment rendering (MFR) is a genre of image synthesis techniques and associated data structures tightly coupled with the rasterisation pipeline, which has helped deliver important improvements to the visual quality of primitive-order rendering and has enabled the real-time display of complex phenomena and structures. A multifragment method encompasses the algorithms and image-space data structures that are necessary to produce, maintain, process and exploit a set of geometry fragments that are associated with a single image, in the sense that multiple samples correspond to the *same location* in image space. Currently, and in the years to come, MFR has an important role to play with the increasing demand for ray-tracing-enabled applications, since the rich image-space information that is provided can facilitate the creation of hybrid rendering techniques, providing freedom to the creative mind.

Scope. In this survey, we examine the underlying mechanisms, complexity, strengths and weaknesses of multifragment-based solutions and their applications, ranging from order-independent transparency to global illumination. We present the most important techniques for the creation and population of a multifragment buffer and discuss in detail the various operations that can be performed on the stored data to implement fundamental methods for image synthesis.

Multifragment rendering is not to be confused with multiview approaches, such as cube map rendering, which are orthogonal in both their generation procedure and use in applications. In fact, later in this survey, some techniques that simultaneously use both strategies are reported. Similarly, methods based on layered depth images (LDIs) [SGHS98] also encompass data from multiple views and, as such, are not included here. However, some MFR approaches in the literature are referred as LDIs, therefore they are considered part of this survey. Finally, we exclude from the discussion voxelisation techniques that rely on the rasterisation pipeline, which generate data structures addressable in object space.

Throughout the years, many different terms have been used in the literature to identify the *data structure* responsible for storing multiple per-pixel fragments and often reflect the intended application, including *Deep Images* [VAN*19], *Deep G-buffers* [MMNL16], *Layered Fragment Buffer* [KLZ12], and *Layer Depth Images* [SGHS98]. In this work, we avoid complicating matters further and use the term multifragment rendering for the rest of the manuscript, since it neither makes any assumption on the data type that is captured and stored nor contains the notion of multiple viewpoints.

† These authors contributed equally to this work.

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd. Published by John Wiley & Sons Ltd.

Related Surveys. In a recent tutorial course [HB14], Havran and Bittner showed that many rendering algorithms are tightly connected to sorting and searching methods. Despite the extensive analysis on the underlying data structures and their enhancements in the context of specific rendering algorithms, the scope of this work was, contrary to our case, object-space data structures.

On the other hand, several surveys have attempted to classify MFR techniques with regard to the specific topic of interactive order-independent transparency [MCTB11, Wym16] as well as in the broader field of light transport [RDGK12]. Nevertheless, no resources exist to inform readers on the large volume of research on the general image-space rendering domain, leaving several aspects of valuable knowledge obscured and ambiguous.

Report Organisation. The paper is organised as follows. Section 2 deals with the fundamental concepts of the rasterisation pipeline and visibility determination as well as how these can be generalised, beyond hidden surface elimination. Section 3 provides a comprehensive and comparative analysis of the current methods used to route and store the samples of the displayed geometry, by classifying them based on their memory allocation strategy. The different stages of each algorithm are further discussed and explained, focusing on function rather than implementation. Section 4 outlines how fundamental image operations have been adjusted to encompass the generic multifragment nature of image-space rendering in order to give a significant effect on the speed and quality at which images are produced. Section 5 presents how novel MFR solutions have been exploited in a large variety of diverse problems and applications and summarises several key strategies that have been found to perform well on specific scenarios. Section 6 discusses open problems and undiscovered, or with a rare attention, research areas. Finally, Section 7 offers conclusions and final thoughts in the MFR topic.

2. Fundamentals

This section presents what is commonly considered to be the heart of real-time rendering, namely the *rasterisation pipeline*. For both the expert and inexperienced reader in the domain of real-time rendering, it can help establish a common ground for concepts and conventions, used in the rest of the paper. While the different stages of the rendering pipeline will be shortly mentioned (Sec. 2.1), an in-depth discussion on the *visibility determination* problem will be provided (Sec. 2.2), since it involves some of the most complex operations that are relevant to MFR techniques and differentiate many of the algorithms presented. More importantly, we also discuss how the visibility determination process can be generalised to support indirect visibility (Sec. 2.3). Finally, Table 1 summarises most of the symbol notation used throughout this document, where some of them will be described at some length in the following subsections.

2.1. Rasterisation

In its simplest form, a *rasterisation* process receives as input a mathematical representation of geometric shapes and converts the latter into a set of coloured dots, or *pixels*, to fill a rectangular grid, called a *raster*, which represents a synthesised image. At a more abstract level, and the one that reflects a modern architecture, it constitutes a geometry sampling process that leads to a set of records of information associated with each sample, or *fragments*, that are forwarded down a graphics pipeline for shading, routing and compositing to an image buffer.

Conceptually, this process can be pipelined into four basic steps, which are directly mapped to the graphics processing unit (GPU). First, each geometric shape is decomposed into a set of basic primitives, such as points, lines, and triangles. Each primitive is represented as an ordered sequence of *vertices*, where each vertex is further associated with a set of vertex attributes, such as position, normal, etc. Then, a series of vertex processing stages is executed, responsible for both vertex and primitive manipulation. Each primitive can be subdivided further and/or spawn new primitives, while each vertex is subjected to a series of transformations and clipping that eventually projects it onto the image plane. Third, the scan conversion step samples the projected objects, usually at a fixed spatial rate, to form a record of the interpolated attributes of the primitive at each sampling location. The relationship between fragments and pixels is many to one; many surfaces may overlap the same pixel area, thus generating *multiple* per-pixel fragments, or *layers*, $F(p) = \{f_1, \ldots, f_{n(p)}\}$, where $n(p) \geq 0$ corresponds to the fragment count at pixel p. The fragments are potentially tested for visibility and the surviving ones, assuming a fixed shading rate, are forwarded for fragment processing. Finally, the results of this procedure are routed to one or more output buffers after a merging stage performs a series of post-processing steps to compute the final values to the output buffers based on the sampled information.

In graphics hardware architectures, the fragment generation stage is executed in parallel and fragments corresponding to the same pixel location are produced in a primitive order, forming a generally *unsorted sequence* with respect to the projection axis. Therefore, to produce a correct image, a *visibility determination* algorithm is employed, to determine how surfaces occlude each other

Table 1: *Summary of the symbol notation used in this survey.*

Symbol	*Description*
Rasterisation	
p	*Pixel*
f_j	*Fragment* j
z_j	*Depth of fragment* f_j
$n(p)$	*Depth complexity of* p
n	*Maximum depth complexity for all pixels* p: $\max(n(p))$
$F(p)$	*Fragment set* $\{f_1, \ldots, f_{n(p)}\}$ *of* p
Multifragment Rendering	
i	*Algorithm iteration*
r_g	*Geometry rendering passes per iteration* i
$s(p)$	*Maximum number of stored fragments of* p *in a single* i
$F_s(p)$	*Stored fragment set* $\{f_1, \ldots, f_{s(p)}\}$ *of* p, where $F_s \subseteq F$
$k(p)$	*Capacity of stored fragments of* p *in a single* i
k	*Capacity of stored fragments for all* p *in a single* i
b	*Number of depth bins*
n_b	*Depth complexity of depth-divided scene*
$m(p)$	*Memory allocation for each pixel* p

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

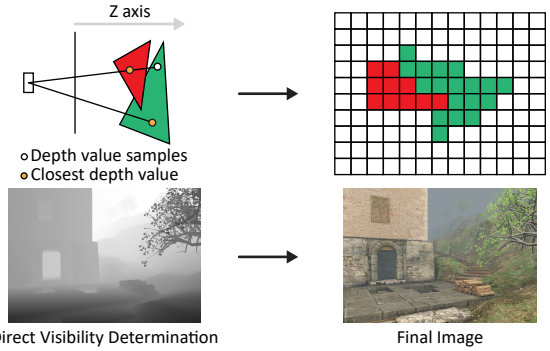

Figure 1: *The correct depth order, with respect to the camera viewpoint, of the projected primitives is determined via the use of a depth buffer (top). The depth buffer and the corresponding resolved image for an example 3D scene (bottom).*

and write the appropriate values to the image buffer. In particular, the rasterization pipeline is very efficient at resolving *primary* visibility, i.e. identifying the fragments associated with the first visible surface locations with respect to the virtual sensor (Fig. 1).

2.2. Primary Visibility Determination and the Z-buffer

The most widely used algorithm for primary visibility determination is the Z-buffer [Cat74]. An image buffer, called the *depth buffer* is created to hold the closest depth values with respect to the viewer. Each fragment's depth is tested against the currently stored value in the same image-plane location and prevailing fragments are marked as visible, updating the depth buffer with their own depth values. The Z-buffer algorithm is a trivial yet effective technique for resolving direct surface visibility. It works with arbitrarily ordered fragments and any geometric entity whose depth can be sampled in image-space, and is thus efficiently implemented in commodity hardware. As a result, it has been exploited to produce a vast number of real-time techniques [TPK01, RDGK12, AMHH*18].

After processing all fragments, the Z-buffer algorithm effectively produces a *single layer* of discretised geometric information about the surfaces in view, which in the case of opaque surfaces corresponds to the nearest visible geometry to the virtual sensor. However, this is not generally the case if fragment blending is involved at the merge stage, e.g. to compute transparency effects. It is actually the reason that spurred the research of many methods on *order-independent transparency*; transparent surfaces require multiple per-pixel fragments to be captured in sorted back-to-front order and then evaluated using a compositing operator [MCTB11]. In general, with a single layer of view-dependent geometric information, computations that require samples from hidden surfaces or outside the current field of view are difficult to implement. Phenomena such as indirect illumination [DS05], ambient occlusion [Mit07], etc., where a visible point interacts with geometry and lighting from potentially any part of the scene, cannot be properly implemented solely by relying on the depth buffer samples.

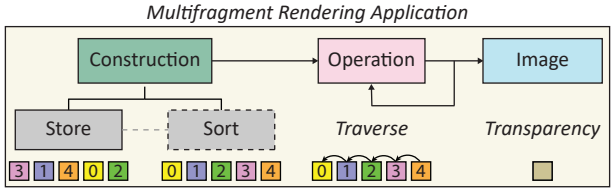

Figure 2: *Diagram of building an application using the MFR pipeline (top). Order-independent transparency requires the sorting of an arbitrary sequence of out-of-order fragments before alpha compositing them in a linear traversal fashion (bottom).*

2.3. Generalisation

Historically, the most prominent example of a method that went beyond the single layer of rasterised geometry was Carpenter's A-buffer [Car84]; a software implementation of multiple per-pixel linked lists for transparency and antialiasing purposes (Sec 3.2). This was later followed by hardware architecture proposals, such as the F-buffer [MP01] and the R-buffer [Wit01], based on FIFO and recirculating fragment buffers, respectively.

At a conceptual level, the complex primary visibility determination is part of a more general *multifragment* pipeline responsible for generating, storing, processing and evaluating information from incoming fragments. This pipeline is comprised of two main steps: *construction* and *operation*, both of which are affected by the context of the particular application in mind (Fig. 2).

The construction step is responsible for generating and storing per-pixel fragments through a common rasterisation procedure (Sec. 3), which is repeated according to a fixed number of *iterations* i. Each iteration is executed in one or more *geometry rendering passes* r_g. In every geometry pass, a fragment subset $F_s(p) \subseteq F(p) = \{f_1, \ldots, f_{s(p)}\}$, $s(p) \leq n(p)$, associated with pixel p is selected, sorted and eventually stored. The outcome of each iteration is the starting point of the next iteration.

Even though storing and sorting are two fundamental building blocks of the construction step (Sec. 3.2 and 3.3), the latter stage is implicitly performed for the depth-peeling techniques (Sec. 3.1). Concerning implementation, it can be applied either as post-processing step or as an online algorithm, i.e. reordering fragments as they arrive.

After construction has taken place, the next step performs one or more operations on the ordered fragment data structure. The most critical operation here is fragment data traversal, followed by other optional, but more advanced ones, such as mipmapping, compression, etc. (Sec. 4). The application accesses and exploits the fragment information stored in the MFR structure to compose the final output (Sec. 5).

3. Construction

We classify the multifragment buffer construction techniques according to the maximum number of fragment samples per-pixel $s(p)$ captured by each method in a single iteration step, in three broad categories: depth-peeling, A-buffer and k-buffer methods

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

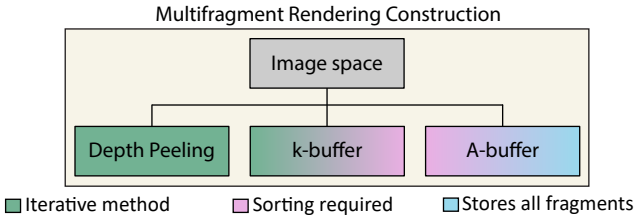

Figure 3: *Basic classification of multifragment buffer construction methods based on the number of fragment samples per-pixel captured by each one in a single algorithm iteration.*

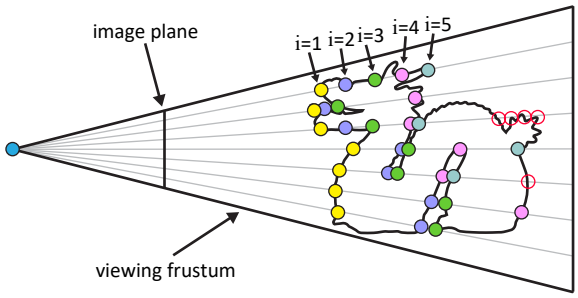

Figure 4: *The fragment samples of each extracted layer when front-to-back depth peeling [Eve01] is performed for multiple iterations $i = 1 \ldots 5$. Note that distant from the camera fragments (outlined with red colour) are not captured.*

(Sec. 3.1-3.3). Unlike depth peeling approaches, which directly yield an ordered set of fragments per pixel, fragments captured by A-buffer or *k*-buffer solutions are depth-ordered either during the construction process via *insertion sort* or by a consecutive post-sorting process (Sec. 3.2.2). Figure 3 presents a generic taxonomy of the MFR domain based on this categorisation.

Depth-peeling methods process all fragment information through an *iterative* rendering pipeline with $i > 1$. Depending on the algorithm, each iteration carries out 1 or 2 passes r_g, to extract a limited fragment batch $s(p) = \{1,2\}$, with *guaranteed depth order* (Sec. 3.1). The main advantages of methods in this class are that they allocate constant memory, they do not need post-sorting and they support older hardware, increasing portability and ease of implementation in diverse architectures. Their downside is that they can induce a significant, and often prohibitive, overhead on the geometry processing stage and rasterisation, cumulative over all iteration steps. Nevertheless, depth-peeling algorithms are still used nowadays despite their outmoded multipass mechanism [LKE18, TDDD18]. In practice, they are usually employed when sufficient visual quality can be achieved by processing even a small number of fragments per pixel, for example $s(p) = 4$.

A-buffer methods aim at capturing all fragments per pixel, $s(p) = n(p)$, in a *single* iteration step ($i = 1$). Typically, fragments are stored into GPU-accelerated data structures of fixed- or variable-length per pixel during a single geometry rendering pass ($r_g = 1$), followed by a sorting process that reorders them according to their depth. A-buffer variants are currently the dominant method for maintaining and processing multiple fragments even in commodity graphics hardware. However, this class of methods suffers from memory overflows as a result of the unpredictable memory space needed to store all generated fragments as well as performance bottlenecks that arise when the number of per-pixel fragments to be sorted increases significantly. To this end, a multitude of novel A-buffer alternatives were proposed aiming at overcoming these limitations (Sec. 3.2).

k-buffer approaches offer a middle ground between the depth peeling and A-buffer classes of methods by efficiently combining the best features of both pipelines. While, in some cases, the *k*-buffer has been referred as *multilayer depth peeling* in the literature [LWXW09, LCD10], we think that it defines a unique class of methods with distinct characteristics and essential capabilities and, thus, we dedicate a separate section for it. In general, *k*-buffer approaches are capable of capturing the *k-best* fragments, $s(p) = k$, in each iteration by employing a fixed-size per-pixel array buffer (Sec. 3.3). Without loss of generality, the *k-best* fragments in a pixel p are defined as the k-subset of $F(p)$ that minimise the value of a specific cost function $C(p)$. For example, finding the *k-nearest* fragments can be expressed as the minimisation of function $C(p) = \sum_k z_j$, where z_j corresponds to the depth of f_j. As a result, using a *k*-buffer requires fewer iterations, $i = \lceil n/k \rceil$, when compared to the traditional depth-peeling in order to capture all generated fragments n. In practical scenarios, where only a specific portion of the fragment pool is enough for achieving plausible simulation effects, such as transparency (Sec. 5.1), only a single iteration is usually performed. The *k*-buffer can objectively be considered as the most preferred framework for optimal fragment subset selection, especially when low graphics memory requirements are of the utmost importance. Unfortunately, the standard practice of employing a fixed number of k for all pixels can lead to various issues [VVPM17]. On one hand, setting k to a small number, can result in view-dependent artefacts as more than k fragments might be required for some pixels at a particular viewing configuration to correctly simulate the desired effect. On the other hand, employing a large value of k can result in wasteful memory allocation, as a large and potentially unused storage is preallocated for pixels that contain less than k fragments.

Table 2 presents a comparative overview of the most representative MFR methods in each class with respect to the rendering complexity, fragment acquisition, memory requirements, and sorting stage.

3.1. Depth Peeling

Probably the most well-known iterative MFR technique is the *front-to-back depth peeling*, which works by rendering the geometry multiple times, extracting a single fragment layer per iteration, in ascending depth order. The technique was first described in [Mam89] and later implemented by Everitt [Eve01] gaining a lot of popularity due to its low and constant storage requirements. Specifically, the algorithm starts by rendering the scene normally with a depth test (Sec. 2.2), which returns the closest per-pixel fragment to the camera. The prepared depth buffer is then used as the minimum acceptable depth in the subsequent pass. Together with

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

the normal depth test, this effectively discards previously extracted fragments, producing the next nearest layer underneath. This process is performed iteratively and halts either when the maximum number of iterations, set by the user, is reached or if no more fragments are generated. Figure 4 shows how the consecutive fragment samples are stored, illustrated with a different colour per iteration, when depth peeling in a front-to-back fashion. While traditional depth peeling extracts layers in a front-to-back order, *Reverse front-to-back peeling* [Thi08] technique peels the layers in back-to-front order, allowing to blend them with the output frame buffer immediately, in the special case of order-independent transparency. Unfortunately, both methods require $\mathcal{O}(n)$ complexity. Furthermore, the n passes of rasterisation makes them unsuitable for real-time applications with complex geometries.

Dual depth peeling [BM08b] speeds up the rendering process, by bidirectionally capturing both the nearest and furthest fragments in each iteration, thus dropping the complexity to $\mathcal{O}(n/2)$. The idea is to apply the original depth peeling method for the front-to-back and the back-to-front directions simultaneously. Due to the unavailable support of multiple depth buffers on the GPU, a custom min-max depth buffer is necessary. In every iteration, the algorithm extracts the fragment information which matches the min-max depth values of the previous iteration and performs depth peeling on the fragments inside this depth range. An additional rendering pass is needed to initialise the depth buffer to the closest and the further layers.

The dual depth peeling method was extended to extract two fragments per *uniform depth bucket* in each iteration [LHLW09], thus reducing further the necessary iterations to $\lceil n_b/2 \rceil$, where n_b corresponds to the depth complexity $n(b_j)$ of the bucket b_j with the most fragment layers in all pixels: $n_b = \max\{n(b_j)\}$, $n_b < n$. Each bucket corresponds to a uniformly divided interval of the pixel depth range. To quickly approximate the depth extent of each pixel, a rendering pass of the scene's bounding box has to be initially performed. To reduce collisions in scenes with highly non-uniform distributions of fragments, the authors further proposed to *adaptively subdivide depth range* according to the fragment occupancy in each pixel: $n_b = \lceil n/b \rceil$, where b is the number of buckets. However, this comes with the expense of additional rendering passes and a larger memory overhead.

Similar to the Z-buffer, depth peeling methods are susceptible to flickering artefacts (*z-fighting*), when two or more generated fragments in the same pixel have identical depth values. This phenomenon introduces various undesirable and unintuitive results, when rendering complex multilayer scenes. Vasilakis and Fudos introduced a number of solutions [VF13] to, fully or partially, alleviate the depth coplanarity issues in depth peeling methods with the expense of either performance downgrade or excessive memory allocation, respectively, and potential fragment overflow.

Recently, Mara and McGuire [MMNL16] proposed a minimum separation selection criterion during depth peeling, where they select only the most relevant, with respect to shading, geometry fragments; the ones that are immediately accessible after a certain distance past the visible surfaces. The authors also applied reverse reprojection to closely estimate the second depth layer in a single geometry rendering pass ($r_g = 1$). A generalisation of the traditional single-layer *G-buffer* [ST90] was also presented where additional geometry properties can be stored per depth layer.

Discussion. Despite the development of several speed optimisations via acceleration schemes, such as *z-clipping planes* [WGER05], *tiling* [KSN08], *partial geometry ordering* [CMM08] and *object-based occlusion culling* [VF13], depth peeling methods fail to behave interactively on complex environments since the geometry is repeatedly rasterised multiple times. On the other hand, they require minimal graphics memory and can be the optimal choice when the exhaustive capture of all fragments is not imperative [MMNL16].

Table 2: *Overview of selected representative MFR construction methods. Symbols notation can be found in Table 1. For clarity, the $m(p)$ column presents relative per-pixel storage requirements, assuming equally-sized fragment payload and depth.*

Method	i	r_g	$s(p)$	$m(p)$	Post-Sort
Depth Peeling					
Front-to-back [Eve01]	n	1	1	3	x
Dual [BM08b]	$\lceil \frac{n}{2} \rceil + 1$	1	2	6	x
Depth Bins [LHLW09]	$\lceil n_b/2 \rceil$	1	$2b$	$4b+2$	x
A-buffer					
Linked Lists [YHGT10]	1	1	$n(p)$	$3n(p)+1$	√
Arrays [LHLW10]	1	1	$n(p)$	$2n+1$	√
Variable-size Arrays [VF12, MCTB14]	1	2	$n(p)$	$2n(p)+2$	√
k-buffer					
Arrays [BCL*07, Sal13]	$\lceil n/k \rceil$	1	k	$2k+1$	x
Decoupled Arrays [MCTB13]	$\lceil n/k \rceil$	2	k	$2k+1$	x
Linked Lists [YYH*12]	$\lceil n/k \rceil$	1	k	$3k+6$	x
Max-Heaps/Arrays [VF14, TH14]	$\lceil n/k \rceil$	1	k	$2k+2$	√
Variable-size Arrays [VVPM17]	$\lceil n/k \rceil$	2	$k(p)$	$2k(p)+2$	x

3.2. A-Buffer

Conceptually, the construction of an A-buffer data structure requires two stages: (i) a *store* operation, where all fragments are captured in the form of an unsorted sequence, and (ii) a *sorting* one, where fragments are reordered correctly according to their depth. Since each operation deals with different issues and research questions, the remaining of the discussion is organised as follows: Section 3.2.1 reviews methods related to the generation, processing and storing of all incoming fragments, in parallel. Section 3.2.2 discusses the problem of efficient parallel sorting in the graphics hardware.

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

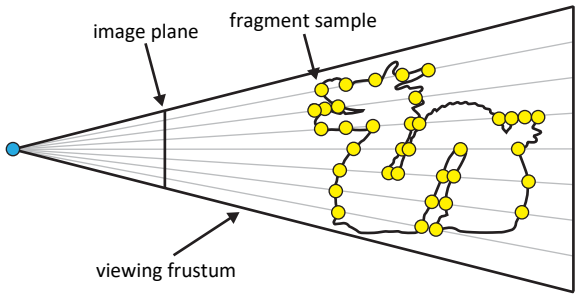

Figure 5: *The A-buffer is able to capture all fragment samples in a single iteration step.*

3.2.1. Store

Historically, the first MFR approach was based on the A-buffer method for resolving visibility of hidden surfaces [Car84]. The main idea is to store all generated fragments (Fig. 5) in variable-length lists instead of keeping only the closest visible fragment, as in the Z-buffer. Each fragment in the list contains additional information such as opacity, area and coverage, in order to handle antialiasing and transparent surfaces. A depth-sorting operation arranges the fragment lists in a back-to-front sequence. The final image is obtained by traversing the list and resolving the final colour of each pixel.

Yang et al. [YHGT10] proposed a GPU-variant of the original method using per-pixel linked lists, by exploiting the capability of graphics hardware to perform atomic operations. The implementation requires two buffers: a *head* buffer, containing head pointers to each list and a *node* buffer, storing node connectivity and data information of each node for all lists. The algorithm runs in every frame and operates in three steps. In the first step, both buffers are cleared to zero, representing null head pointers and no data, respectively. In the second step, the lists are filled in a subsequent geometry rendering pass. For each incoming fragment, the address of a new node is obtained by atomically incrementing a global atomic counter. Then, an atomic exchange operation sets the new node's next pointer to the pointer that is currently stored in the head buffer and replaces the head pointer with new node's address (Fig. 6, left). This operation creates the linked list in reverse, where all per-pixel fragments are stored in an unsorted manner. Finally, a post-process operation reorders the fragments based on their depth.

The algorithm exploits efficiently the rasterisation pipeline and is able to capture all fragments in a single iteration. However, it suffers from two limitations. First, it requires a memory preallocation operation to reserve the memory space needed, since dynamic memory allocation for new nodes is not possible within a shader. This is decided manually and can result either in wasted space or memory overflow; the latter manifests as flickering. This can only be avoided by performing an initial step, responsible for counting the total number of fragments required. Second, the non-contiguous allocation patterns of the linked list result in memory indirections and cache misses and can, as a result, cause performance degradation in applications that typically operate on sequential per-pixel elements, such as order-independent transparency (Sec. 5.1).

The research community has since focused on ways to alleviate the limitations of GPU-based linked lists, improve upon their efficiency as a data structure, or even propose alternative representations. As an example, Crassin [Cra10b] used a *memory paging* mechanism, where fragments are allocated in small fixed-size blocks (4-6 fragments each) in each pixel instead of individually. This way, cache coherency is improved when accessing neighbouring per-pixel fragments, at the cost of slight fragment overallocation.

Classic linked lists perform well as long as the mode of operation is unidirectional. For efficient *bidirectional* traversal, one [FMS13] or multiple [VVP16a] double-linked lists can be used. This can be achieved by augmenting the single list with a *prev* pointer in each node and a *tail* buffer. Double linked-lists can support more complex effects at the non-negligible cost of extra memory space.

Decoupling visibility from shading data allows for compact A-buffer representations. This is accomplished by maintaining only depth information in the node buffer [LHL15, VVP16a] and deferring the rest of the data to a separate buffer, in order to reduce memory bandwidth during the sorting stage. One step further, *primitive-based* A-buffers [KWBG13, VVP16b] aim at decoupling visibility with per-fragment shading. Using such a layer of indirection can impose a small performance decrease but can also reduce the memory storage of the entire data structure significantly. To mitigate memory overflow issues, tiling approaches split the image domain, uniformly [Thi11] or adaptively [TSdSK13], into smaller regions and render the scene in multiple geometry passes.

Instead of a linked list, Liu et al. [LHLW10] proposed *FreePipe*, a CUDA-based implementation of the rasterisation pipeline. Their approach allows the storage of all fragments in *fixed-size per-pixel arrays*. Compared to a linked list, the need to maintain node pointers is removed and fragments are stored in contiguous memory locations. A similar approach was also implemented in the traditional rasterisation pipeline [Cra10a]. In general, fixed-size approaches can offer much higher efficiency in complex environments, since cache coherence is improved significantly. On the downside, scenes with uneven fragment distribution can cause large blocks of memory space to be wasted, as the size of each array is fixed for all pixels (Fig. 6, right).

Lefebvre et al. [LHL13] proposed the *HA*-buffer, where fragments are stored in a *hash-table* rather than a traditional array. To preserve spatial coherence on the keys, the authors employed a parallel implementation of Robin Hood hashing. One of the key properties of this technique is that construction and depth-ordering can be combined in a single geometry pass, since fragments are sorted as they are inserted in the hash. The HA-buffer exhibits good performance characteristics, but the overall construction speed depends on the occupancy of the hash-table, where high load factors require more hash-table lookups to find the appropriate storage location.

Variable-size arrays attempt to combine the advantages of fixed-size arrays and linked-list structures [VF12, KLZ12, MCTB14]. Typically, they require two additional passes before the storing stage: an initial geometry pass, responsible for counting the number of fragments in each pixel and a screen-space pass, for the generation of per-pixel memory offsets (heads) through a prefix sum

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

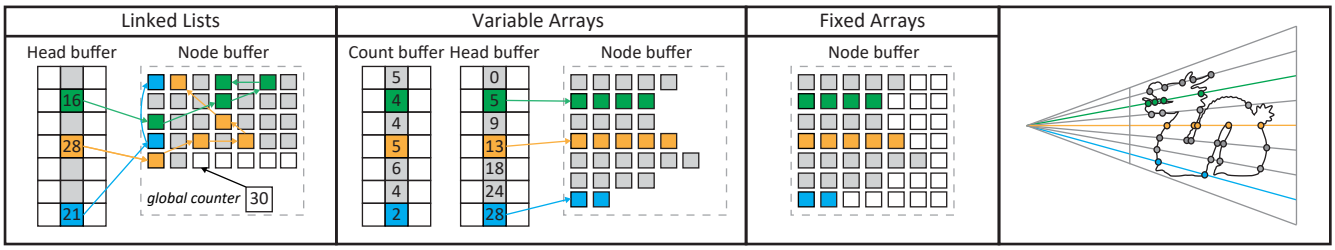

Figure 6: *Basic A-buffer variants: Variable and fixed array structures pack pixel fragments in contiguous memory regions (centre, right), avoiding the memory indirections when accessing the nodes in the linked list (left). Variable arrays achieve this by performing additional passes and maintaining extra buffers (centre) while fixed arrays allocate the same number of entries per pixel resulting at significant waste of memory space (right).*

operation. These methods are able to allocate per-pixel information in contiguous regions and without overestimating memory space (Fig. 6, centre), thus, offering a memory-friendly alternative to the other approaches with improved cache coherence. On the downside, they are more complex to implement and can be less efficient on scenes with high geometric complexity.

Discussion. The various A-buffer alternatives that exist in the literature cover a broad range of characteristics and, therefore, there is no optimal solution but rather a more suitable match for each application's requirements. Fixed-size arrays are the simplest and most efficient implementation but can also waste a significant amount of memory. So they should be employed on scenes with uniform fragment distribution, or applications with relaxed memory constraints. Traditional linked lists can accommodate more complex environments at the same memory budget, by trading tighter memory allocation with less efficient cache utilisation. Finally, variable-size arrays are preferable for applications with tight memory budget and environments with medium geometric complexity.

3.2.2. Sort

After all fragments have been generated and saved to the A-buffer variant (except the HA-buffer [LHL13] which is an exception), they are depth-ordered in a subsequent *sorting* stage. Conventionally, this task is parallelised across pixels, but serialised along the individual depth fragments, by being implemented in a single full-screen rendering pass. The process can be divided into four main consecutive steps: *read* the depth values of the captured fragments from the global memory, *store* the data in a local temporal structure, *sort* the temporal data based on the depth information, and finally *save* the sorted data back to the global memory. The reason for copying the unsorted fragment list to a local cache before sorting is that the high latency when operating on global memory is minimised. While altering the global structure of the A-buffer may improve more or less the performance (Sec. 3.2.1), several approaches have been proposed for alleviating the bottlenecks that arise in managing local caches in graphics hardware. High fragment scattering can impact the actual sorting algorithm due to thread divergence, while poor coalescing and bad utilisation of the local memory can affect all the other steps.

The graphics architecture is significantly different from that of a CPU, thus directly applying typical parallel sorting strategies is neither optimal nor practical. Even though GPU-based sorting algorithms have been explored in the literature [SJC18], reordering millions of fragment lists in parallel remains a non-trivial task. Knowles et al. [KLZ12] showed that altering the sorting algorithm per pixel based on $n(p)$ has resulted significant speed benefits. Specifically, they demonstrated that the *insertion sort* algorithm behaves faster for low depth complexity, e.g. $n(p) \leq 16$, while *merge* or *shell sort* is better suited for higher fragment counts. To reduce divergence and improve memory caches, Vasilakis and Fudos [VF13] performed *bin sort* by distributing the fragment elements into a number of buckets, and each bucket is then sorted individually. However, the computational complexity depends on the algorithm used to sort each bucket, the number of buckets to use, and most importantly on whether the input is uniformly distributed. To reduce the amount of local memory access, a novel *register-based block* sorting algorithm was further introduced by using an insertion sort network of fast registers, thus better exploiting the memory hierarchy of the GPU [KLZ14]. Recently, Archer and Leach [AL18] showed that the latter method can be improved by modularising parts of the network and by tuning the loop unrolling, thus reducing the total sort code size for better cache behaviour.

Proper memory cache utilisation can have a large impact on the overall performance. The prevailing approach described in the literature [YHGT10, Cra10a] for managing local GPU caches is to allocate a fixed-sized array of length n per pixel. However, larger array sizes can significantly reduce the number of active threads due to low occupancy. Lindholm et al. [LFS*15] presented two novel approaches to improve the management of local GPU caches. The first minimises the allocated size in the fast cache memory by adjusting the allocation to pixel depth complexity (also proposed in [KLZ13] as *backwards memory allocation*), while the second partitions the depth sorting similarly to the iterative k-buffer (Sec. 3.3), thus recycling a smaller amount of local memory. A faster variation was further introduced in [VPF15] that alters the traditional array with a max-array structure [VF14, TH14] at a local cache level.

To overcome the sequential nature of the sorting process on the number of depth layers, a CUDA-based technique extended the domain of parallelisation to individual fragments [PTO10]. A fragment-parallel procedure is responsible for sorting the list of

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

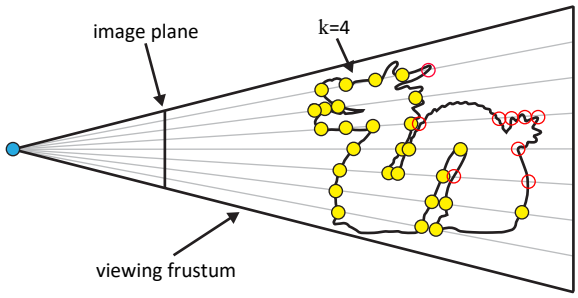

Figure 7: *The fragment samples captured from a k-buffer, with $k = 4$, when a single iteration is performed. Note that distant from the camera fragments (outlined with red colour) are not captured.*

generated fragments using a two-key sorting scheme, with pixel identification as the first key and the depth value as the second one. However, the performance advantages of this method are obvious only for scenes with very high depth complexity ($n > 50$).

Discussion. The behaviour of the graphics memory caching, locality and occupancy is highly-sensitive to the irregular fragment sampling. Choosing the appropriate sorting mechanism is application-dependent and most specifically, related to the generated fragment distribution. In practice, efficient sorting implementations generally use *hybrid* solutions combining an asymptotically efficient algorithm for the overall ordering with insertion sort for small lists.

3.3. k-buffer

Z^3 [JC99] is considered the first hardware architecture to use a fixed number of fragments per pixel in order to allocate less memory than the A-buffer variants. When the maximum number of fragments per pixel is reached, it selects the two closest fragments and merges them together using a set of heuristics based on pixel coverage. The *k*-buffer, initially defined in [CICS05] for volume rendering and later extended in [BCL*07] to support a larger variety of applications, can be seen as a generalisation of the Z^3 algorithm, where the storage and insertion of the fragments has been made programmable.

k-buffer [CICS05, BCL*07] reduces the computation cost by capturing the best *k*-subset of all generated fragments, usually the *closest* to the camera (Fig. 7), in a single geometry rasterisation pass. In general, the algorithmic structure of any *k*-buffer variant is as follows. For each incoming fragment $f_j \in F(p)$, we initially *read* the captured elements of *k*-buffer for pixel p from the graphics memory and copy them to a temporary buffer. If this fragment is *important*, compared to the preexisting fragments in $F(p)$, we *modify* the current fragment subset accordingly (e.g. with insertion sort) and finally *write* the updated elements back to the *k*-buffer memory. If the fragment is not significant, e.g. it is positioned after all k already captured fragments, then it is discarded and the *k*-buffer is not modified. Note that the importance function for fragment inclusion and order in $F(p)$ varies between applications.

Despite the increased computational demands and reduced memory when compared to depth peeling and A-buffer solutions respectively, the original algorithm suffers more or less from disturbing flickering artefacts caused by read-modify-write hazards raised when the generated fragments are inserted in arbitrary depth order. The image quality may be significantly improved based on a coarse CPU-based presorting in primitive space [HCNS18] which can reduce the arrival of out-of-order fragments significantly. Bavoil et al. [BCL*07] further proposed high-level hardware modifications that would avoid these hazards in future hardware, but these were never adopted by graphics hardware vendors.

Numerous *k*-buffer variants have been introduced that aim to eliminate any race conditions that can occur with fragments belonging to the same pixel. Liu et al. [LWXW09] extended the original method to an *iterative* approach, achieving robust rendering behaviour with the trade-off of low frame rates. Bavoil and Mayers [BM08a] eliminated most of the memory conflicts by performing *stencil routing* operations on a multisample anti-aliasing buffer. Wang and Xie [WX13] proposed to *partition* the input scene into components with a bounded number of layers and then render them individually to fit into the limited data size of the stencil-routed *k*-buffer. However this scheme cannot support dynamic scenes and is not particularly suitable for order-dependent rendering. A memory-hazard-aware solution based on a depth error *correction coding* scheme was explored by Zhang [Zha14], however in practice, correct results are not guaranteed for all cases. Leung and Wang [LW13] proposed to convert a solid voxelisation into a *k*-buffer representation that ensures conservative sampling. However, the proposed solution limited the applicability of the particular structure to solid shape representation applications.

Multidepth testing, developed in both CUDA [LHLW10] and OpenGL [MCTB13], 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 value and the rest of the shading parameters necessitates an additional costly geometry rendering pass ($r_g = 2$). While its 64-bit version [Kub14] is feasible to run on modern graphics cards, it is limited to maintain only 32-bit additional geometry information per sample. Furthermore, noisy images may be generated from both 32- and 64-bit versions of this algorithm due to the precision lost when converting floating-point depth values.

Yu et al. [YYH*12] proposed two linked-list-based solutions to accurately compute the *k*-closest fragments. The idea of the first one is to capture all fragments by initially constructing an A-buffer via linked lists [YHGT10], followed by a step that selects and sorts the k-nearest fragments. The same strategy was also followed by prior work [SML11], which adaptively compresses fragment data to closely approximate the ground-truth visibility solution. The second approach directly computes depth-ordered per-pixel linked lists, thus avoiding the memory-consuming A-buffer construction. The idea is to perform fragment insertion sort in parallel without mutual exclusion. A verification step is responsible to guarantee valid insertions during fragment race condition. Despite the fact that this approach theoretically requires less storage, failed insertion attempts resulted in fragments being sparsely stored in the data buffer, causing a notable memory overhead.

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

The original k-buffer algorithm [BCL*07] has been revised to perform without quality artefacts, caused by read-write-modify hazards, by utilising hardware-accelerated *pixel synchronisation* [Sal13]. However, 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 alleviated fragment race conditions by concurrently performing efficient culling checks to discard fragments that are farther from all currently maintained fragments based on *max-array* and *max-heap* data structures. Using max-array buffer storage can perform faster compared to the max-heap data structure, when the selected fragment subset remains at low levels ($k \leq 16$). A pixel-synchronised max-array k-buffer implementation [TH14] is also used in *AMD TressFX* [MET*14] - a software library for advanced simulation and rendering of hair, fur, and grass. A drawback of the pixel-synced methods is that they require an additional post-sorting of the captured fragments. *Insertion sort* is usually employed here since, despite its quadratic complexity, it behaves quite efficiently when sorting small fragment sequences ($k \leq 8$) [KLZ12].

Nevertheless, fragment culling of pixel-synced methods depends on the depth order of the incoming fragments, where no culling is being performed when fragments arrive in descending depth order. Vasilakis and Papaioannou [VP15] introduced a fast fragment culling mechanism via *occupancy maps* to maximise the number of fragments to be rejected. By computing a well-approximated depth of the k-th fragment a priori, the insertion of all fragments with smaller or equal depth can be performed in constant time.

With regard to memory, the k-buffer assumes a preassigned, and global, value of k fragment layers across the entire image. From a development and production standpoint, the process of finding the optimal k, which correctly captures the user's intent, while keeping memory budget low, can become very challenging. Traditionally, this task is addressed in an iterative trial-and-error basis, to obtain an acceptable visual result, albeit at very specific viewing conditions. The intended fidelity can easily degrade for arbitrary views, while at the same time under-utilising fixed preallocated memory for image parts of shallow depth complexity. To this end, a *dynamic* k-buffer [VPF15] was proposed, where k is automatically determined under constrained memory budget via on-the-fly *depth complexity histogram* analysis. In contrast to previous approaches, where the manually or automatically chosen k is considered to be the same for all pixels, Vasilakis et al. [VVPM17] introduced the *variable* k-buffer, the first *selective* MFR solution, which dynamically assigns k on a per-pixel level, $k(p)$, according to an importance-based distribution function, thus, allowing higher depth complexity in regions that are deemed important.

Discussion. Without doubt, practical implementations for supporting complex effects must severely constrain both the memory budget and the computation time for the depth-sorted fragment determination, thus leading to the adoption of bounded memory MFR configurations such as the k-buffer strategy. Similar to the A-buffer implementations, the fixed-size k-buffer approach outperforms the variable solution, thus being more suitable for real-time rendering applications. However, relying on manual configuration of the value of k, can inevitably result in bad memory utilisation and view-dependent artefacts. While synchronisation and selective rendering solutions move towards the right direction, further research has to be conducted to address fragment race conditions and selection more efficiently in terms of speed and perceptual importance.

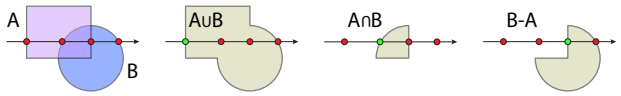

Figure 8: *Depth traversal is commonly applied for efficient computing CSG operations. Fragments illustrated with green colour define the surface boundary for each operation as observed from the user's point of view: union, intersection and difference.*

4. Operations

In this section, we identify and discuss some of the most widely used operations in the MFR genre. In general, applications (discussed in detail in Sec. 5), need to access multiple fragments in an ordered manner (Sec. 4.1 and 4.2) to evaluate pixel quantities or query visibility and spatial intervals (Sec. 4.3). A number of other, infrequent operations can also be performed in order to process the fragments and alter the MFR data structure (Sec. 4.4). Certain operations are performed locally, per pixel, while others transcend the single-pixel cell boundaries and access laterally adjacent information, often in an iterative manner. Note that each operation is usually implemented as a screen-space post-processing step.

4.1. Depth-only Traversal

The most common multifragment operation is the unidirectional traversal along the depth direction, where the fragment elements for each pixel in the data structure are accessed sequentially starting from the first node until reaching the last one. Order-independent transparency (Sec. 5.1) and Constructive Solid Geometry (Sec. 5.5) are the most typical applications of this operation. The sorted fragment node structure is visited either in back-to-front order, to blend transparent fragments (Fig. 2, bottom), or in front-to-back order according to a Boolean operation, until its surface boundary is reached (Fig. 8). For such an operation, array-based data structures are typically more efficient, compared to linked-lists (Tab. 2), as they support binary search and have better cache locality that can make a significant difference in the access and traversal performance [MCTB11].

4.2. Image-space Traversal

Treating the sampled fragment information as a viable geometric representation has been exploited in screen-space and environment map ray tracing, especially with a single depth layer (Z-buffer). The most widely used method for image-space traversal has been the linear ray marching using regular object-space increments [SKS11]. The same principle can be directly applied in the multilayer case. Each ray is sampled incrementally at fixed object-space intervals, projected in screen-space, and finally tested for termination only at these pixel locations. The termination condition for a single-layer buffer is simply the crossing of the depth boundary. However, in MFR data structures, it involves more elaborate

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

computations, which are detailed in Sec. 4.3. This method is simple and efficient, but its object-space nature suffers from under and over-sampling issues. A small number of steps will eventually skip pixels representing correct intersection locations, while a larger one might result in accessing the same pixel multiple times (Fig. 9, left). McGuire and Mara [MM14] proposed an efficient 2D Digital Differential Analyser (DDA) for densely marching rays, capable of capturing both contact details and distant geometry with no severe sampling issues (Fig. 9, middle). Instead of following the Bresenham-style traversal scheme, Hoffman et al. [HBSS17] relied on the exact 3D DDA scheme proposed by Amanatides and Woo [AW87] to ensure conservative pixel traversal.

Iterating in single-pixel increments can be prohibitive for real-time applications at high resolution configurations, unless only near-field effects are considered. To capture the entire image-space domain effectively, ray traversal can be performed in a *hierarchical* manner (Fig. 9, right). Essentially, this reduces the image-domain traversal from linear to logarithmic complexity by efficiently skipping regions that are completely below or above the ray. This can be done in two ways. Build a boundary-based hierarchy on the nearest and/or farthest layer(s), inspired by the work of Cohen and Shaked on height field rendering [CS93]. This approach enables effective empty space skipping only on the shell of the represented fragments, e.g. [Ulu14]. Conversely, one can consider fragment clustering in both image and depth domains, resulting in more effective data structures. Widmer et al. [WPS*15] proposed a quad-tree ray traversal acceleration structure, where each set of adjacent and non-overlapping tiles is represented in the hierarchy as either a planar or axis-aligned bounding-box, depending on fragment coplanarity criteria. Hofmann et al. [HBSS17] also followed a spatial hierarchical approach, generating a pyramidal representation by merging neighbouring fragments with similar depth intervals.

For near-field tracing, e.g. ambient occlusion shading, it is usually preferable to use a fixed number of uniformly spaced samples, since the probability of missing a geometric feature and the respective error is small. For other ray tracing tasks, we have found that hierarchical traversal is generally advantageous. However, in dense environments and especially when rays emanate from surfaces, this performance gain is lost.

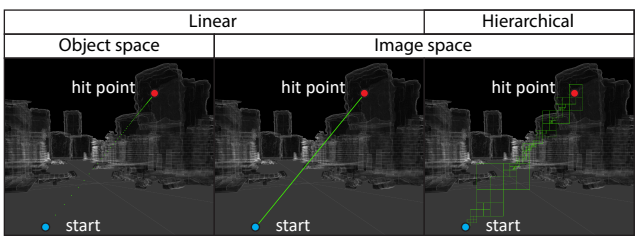

Figure 9: *Image domain traversal is performed by moving in linear object space (left), linear (middle) or hierarchical image space steps (right).*

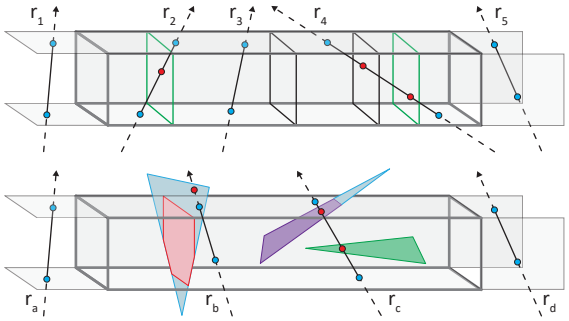

Figure 10: *The different cases for testing a ray against fragment samples (top) [VVP16a] and pixel-clipped primitives (bottom) [VVP16b]. Empty space skipping is performed when rays pass outside depth boundaries (r_1, r_5, r_a, r_d). In both cases, the closest valid hit is determined by the ray's direction (r_4, r_c). Ray-pixel and ray-fragment/primitive intersections are illustrated with cyan and red colours, respectively.*

4.3. Ray-Pixel Hit

Identifying a potential intersection between a ray and a pixel can be performed by testing the image-projected ray against all fragment samples assigned to a pixel [MM14]. To mitigate intersection errors due to the fragment discretisation process of the rasterisation pipeline, each fragment is considered as a frustum-shaped voxel of predetermined thickness. This approach is relatively efficient, but adds a small view dependency. Ray-fragment intersection is commonly performed in view space, since screen space depth is nonlinear [BHKW07, XTP07]. For every fragment, the ray's depth extents are compared with the fragments depth extents in order to identify a hit. In the multifragment case, four different ray-fragment hit cases may arise (Fig. 10, top); a ray may pass outside pixel depth boundaries (r_1, r_5), it may cross zero (r_3), one (r_2) or more (r_4) fragments. For the latter case, rays crossing multiple fragments use their direction to determine the hit. Considering image-space ray tracing efficiency, significant performance boost can be achieved by subdividing the pixel-sized frusta along the depth axis into multiple bins and enabling bidirectional traversal, thus, allowing for empty space skipping [VVP16a]. While replacing fragments with frustum-shaped voxels can improve hit-ratio, it cannot guarantee accurate intersection behaviour. Therefore, analytic intersection tests were introduced, operating on primitive indices captured in per-pixel linked lists [WHL15, VVP16b] (Fig. 10, bottom).

For rendering operations that use a wide distribution of rays, such as ambient occlusion, diffuse inter-reflection and soft shadows, approximate, fragment-based intersections suffice to achieve an acceptable quality level. Otherwise, an analytical solution is preferable.

4.4. Other Operations

Neighbourhood Discovery. Computing the connectivity between fragments of neighbour pixels allows for more complex sampling and filtering operations at the fragment level, such as spatially-aware transparency. Carnecky et al. [CFM*13] proposed to com-

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

pute the multilayer neighbourhood graph of adjacent pixels by linking geodesically neighbouring fragments. A comparison is performed between a fragment and all fragments contained in an adjacent pixel and the best neighbour candidate corresponds to the fragment which maximises the connectivity function. Murray et al. [MBG16] lower the complexity of this operation by grouping together per-pixel fragments that belong to the same object. Thus, the number of fragment pairs to test is restricted to a subset of candidates, i.e. the ones with the same object identifier. Regarding collision detection, Radwan et al. [ROW14] computed the connectivity between dense fragment samples in order to estimate a thickened boundary representation of the surface geometry in screen-space.

Compositing. Multilayer compositing has recently become a commonly used technique for generating complex production images from multiple sources [HHHF12, EDL15, VAN*19]. Elements rendered into separate multilayer images can be combined accurately to produce a single composited image of the entire scene, even if the images are interleaved in depth [Duf17]. This allows for great flexibility, since it enables additional capabilities such as accurately inserting volumetric details [HHHF12] or other opaque surfaces [EDL15]. Object modification does not require the entire frame to be re-rendered, and the scene can be divided into elements without needing to consider how they will be combined. Archer et al. [ALvS18] proposed several implementations for compositing multifragment buffer pairs on the GPU.

Compression. In order to handle and visualise large fragment sets, provision of effective data reduction mechanisms is necessary for addressing problems related to sampling overflow. Offering visual scalability is crucial, when processing and rendering multiple fragment layers. A number of approximation techniques has been introduced, in which compressed sets of data are computed based on both *sampling* and *aggregation* strategies, in a way that do not cause a loss of information with a significant visual impact on the target application. To reduce the memory storage, Duan and Li [DL03] proposed to separately compress colour and depth information for each layer using lossless image coding schemes. To keep multilayer shadow images to manageable sizes, Lokovic and Veach [LV00] performed sample compression when a subsequent fragment falls within an error tolerance of the extrapolated transmittance from the previous samples. In order to avoid memory overflow, Salvi et al. [SVLL10, SML11] followed a similar idea by performing on-the-fly rejection of incoming fragments. The algorithm chooses to eliminate the fragment that if removed, would minimise the error with respect to the uncompressed visibility function (Fig. 11). Based on the observation that the far fragments, tend to have less contribution to per-pixel visibility, two works split the fragment layers of a pixel in two classes; a core, that is preserved as it is and a tail, which is approximated via weighted average blending [MCTB13] or online alpha compositing [SV14]. Kerzner et al. [KS14] presented a novel G-buffer antialiasing method based on a lossy streaming compression algorithm suitable for deferred shading. During the compression (accumulation) pass, they merge fragments with coplanar geometric features that overlap in depth, and cover mutually exclusive multisample antialiasing samples.

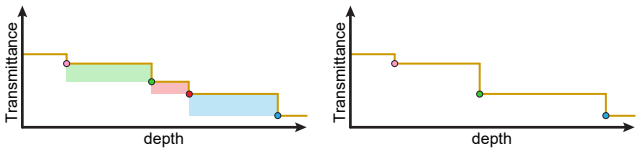

Figure 11: *The visibility function models how light is absorbed as it travels through space (left). To compress this, the fragment that generates the smallest area variation is removed [SML11] (right).*

5. Applications

With the advent of modern graphics architectures along with the corresponding software APIs, MFR research plays key role in a wide range of application areas covering several fields in the industry (Fig. 12), by enabling developers to perform robustly a series of tasks that are not always possible with common single-layer techniques (Tab. 3). In this section, we identify key application domains of MFR and provide recommendations for choosing an appropriate method for each problem.

5.1. Order-Independent Transparency (OIT)

The majority of research ideas in the MFR domain has spurred by the demand for efficient and correct simulation of transparent (see-through) surfaces. Transparency is an order-dependent operation, i.e. primitives must be submitted in sorted back-to-front order before alpha compositing them [HCNS18]. However, this procedure requires significant rendering time per frame since the ordering is different under dynamic geometry or animated camera view. Moreover, it will not always produce a correct solution in cases of intersecting or circularly overlapping geometry.

Order-independent transparency, on the other hand, lifts this restriction and performs sorting on a per-pixel level by exploiting the capabilities of modern graphics hardware. The traditional approach is as follows: First, the opaque surfaces are rendered normally and stored in a separate buffer. Then, for each pixel, a list of transparent fragments is captured, stored and sorted (Sec. 3). The list is traversed in a back-to-front manner (Sec. 4.1) and a composition operator, such as the *over* operator [PD84], is applied on the fragments to compute the final colour of the transparent surfaces. The final result is then merged with the colours from the opaque buffer.

Since this application encompasses a large body of research, this report concentrates mainly on research publications with an emphasis on notable recent advances and refers the interested reader to the comprehensive survey by Maule et al. [MCTB11] for more information on this specific topic. While A-buffer solutions produce accurate results, storing the entire fragment list of transparent objects in memory can quickly exhaust GPU resources in real-time applications, since the number of primitives in hair [YYH*12, JCLR19] or computer-aided design (CAD) models [SBF15, MBG16] (Fig. 12) can be quite large. As such, researchers focused on capturing only the most *important* information with a k-buffer scheme (Sec. 3.3) and compactly representing the resulting transmittance as a function of depth.

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

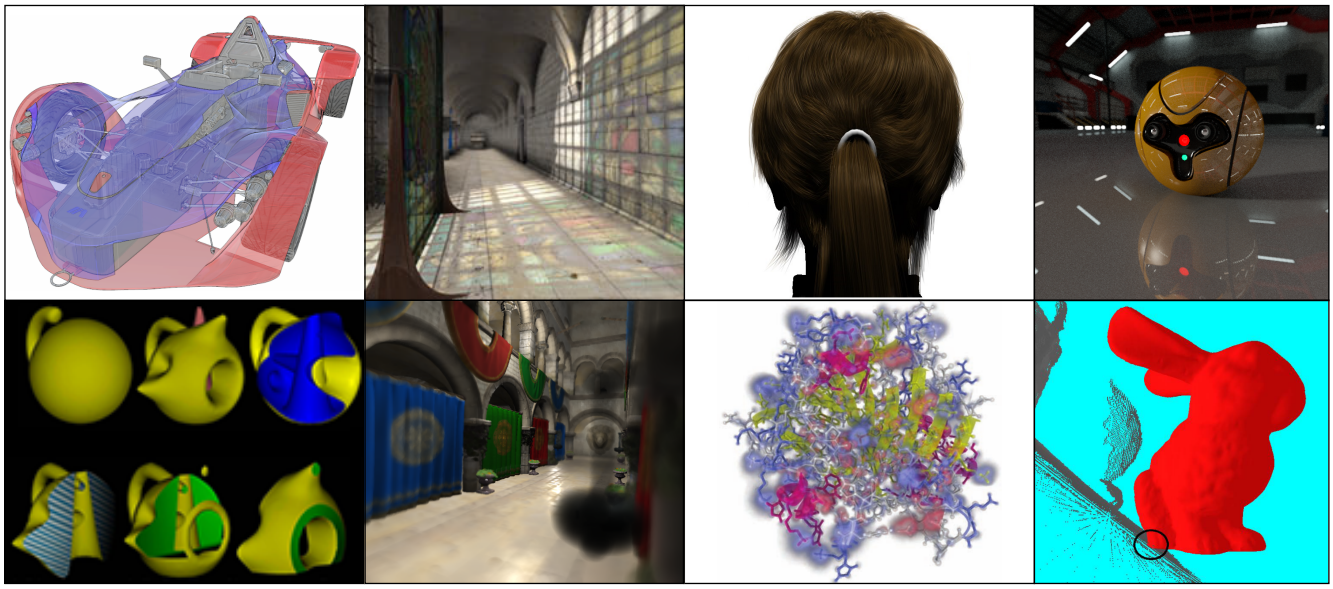

Figure 12: *MFR has been deployed in a wide spectrum of rendering applications in order to generate compelling graphics effects at interactive frame rates. Top left to bottom right: order-independent-transparency [MBG16], shadow mapping [SDMS15], hair rendering [JCLR19] (https://github.com/CaffeineViking/vkhr), global illumination [VVP16b], trimming and CSG operations [RFV13], depth of field [FHSS18], hybrid visualisation [LFS*15], and collision detection [ROW14].*

Specifically, Salvi et al. [SML11] captured and compressed the k-best fragment data on the fly, by *adaptively* removing and merging fragments with smaller contribution, in order to minimise the error in the visibility function and more closely approximate the ground-truth solution. The original method required hardware modifications in pixel synchronisation that were made available later on [Sal13]. Further on, Maule et al. [MCTB13] tried to approximate better the transmittance function by suggesting that fragments farther away from the viewer are not as important as the closer ones, and their contribution can be approximated with a smaller impact to the pixel colour. Thus, they proposed a hybrid approach, where the k closest fragments are captured, sorted and iteratively blended, while the remaining ones contribute to the final result via a fast weighted average blending. Going one step further, Salvi and Vaidyanathan [SV14] introduced *multi-layer alpha blending*; an online merging strategy that alpha-composites the fragments with the lowest transmittance in a streaming fashion. Later on, Wyman [Wym16] offered a general analysis on the continuum of OIT algorithms and further proposed a k-buffer solution with stochastic transparency characteristics, i.e. fragments are stored in a probabilistic manner.

Regardless of the way fragments are captured and merged, embedding the material properties of transparent surfaces in the fragment buffer can significantly affect the rendering cost. Hillesland et al. [HBT14] traded memory for efficiency by storing G-buffer information in a k-buffer instead of the final shading colour. Shading is deferred to a later stage, where lower quality shading computations are applied at the furthest fragments. Finally, Schollmeyer et al. [SBF15] went a step further and proposed a complete deferred shading pipeline that accommodates MFR and OIT.

To summarise, many specific applications related to OIT, including hair rendering, can benefit from using relatively shallow k-buffers instead of an A-buffer, due to the rapid attenuation of transmittance along the depth axis. CAD model visualisation or similar object inspection applications are ideal candidates for adaptive and variable allocation of a fixed memory budget across the image buffer, since depth complexity can take dramatically different values in each pixel, also leaving many pixels unoccupied.

5.2. Shadows (SH)

Traditional shadow mapping, i.e. based on the Z-buffer, has been the most popular approach for shadow generation in real-time rendering. Despite its simplicity, it suffers from various issues, due to either the discretisation nature of the rasterisation pipeline or the limited occluded information stored in the depth map [ESAW11, SWP11]. As such, the research community has also explored techniques to improve the quality of the produced shadows by maintaining visibility information from multiple per-pixel occluders.

Rather than storing a single depth at each pixel, Lokovic and Veach [LV00] proposed *deep shadow maps* that store a transmittance function, representing the fractional visibility through a pixel at all possible depths. To suppress self-shadowing aliasing issues, Woo et al. [Woo92] proposed to generate shadow maps based on the midpoint of the two closest depth values to the light source, an approach that can be implemented today via depth peeling or a k-buffer. Bavoil et al. [BCS08] used multilayer shadow maps to improve soft shadowing issues of the original backprojection algorithm [GBP06]. They exploited depth peeling in order to reduce light bleeding in thin objects by computing occlusion from

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

each per-pixel sample and to minimise self-shadowing by augmenting midpoint shadow maps [Woo92] with multilayer information. Further on, Salvi et al. [SVLL10] proposed *adaptive volumetric shadow maps* that generate an adaptively-sampled representation of the volumetric transmittance in a linked-list data structure, where each texel stores a compressed approximation (Sec. 4.4) to the transmittance curve along the corresponding light ray. Later on, Selgrad et al. [SDMS15] performed prefiltering and merging of multiple occluders stored in variable-length lists, in order to select the appropriate filter width for the generation of soft shadows (Fig. 12).

To generate alias-free shadows for point lights, the shadow mapping process can be reversed using the concept of irregular Z-buffers [JLBM05]. Instead of storing occluders and testing receivers against them, receivers are projected to the light and stored in per-pixel fragment lists. Then, occluders are rendered using conservative rasterization and tested against the data stored in the linked lists [WHL15].

Recently, Lee et al [LKE18] used iterative depth warping to synthesise multiple shadow maps, each one corresponding to an area light sample, for high-quality soft shadows. In this work, depth peeling is used to generate multilayer depth buffers in order to find good candidates for warping and, consequently, handle potential disocclusions that may appear when hidden surfaces become visible in the new frame.

In general, shadow generation via MFR can benefit from compact buffers, since the stored information may correspond to either an approximate transmittance function or partial occlusion estimate. A key enabling factor here is the fact that small discrepancies in indirect visibility are less visually objectionable, compared to other MFR applications, such as OIT.

5.3. Global Illumination (GI)

Image-based global illumination techniques use either an image-space ray or cone tracing mechanism to sample paths due to one or more indirect lighting scattering events, or a more approximate gathering operation on the MFR buffer to reconstruct far-field ambient irradiance attenuation (ambient occlusion), irradiance (sky lighting) or near-field colour bleeding. Traditional single-layer techniques are able to efficiently approximate indirect illumination phenomena [RDGK12]. Despite their fast performance though, they suffer from view-dependent artefacts, due to the missing geometric information, both inside and outside the view frustum, causing - often severe - visual instabilities and erroneous results. Multifragment techniques can augment the information stored in the data structure, thus, reduce view dependencies significantly.

Ambient Occlusion. Bavoil and Sainz [BS09] demonstrated multilayer ambient occlusion (AO) using several depth-peeled screen-space layers. The layers are generated with an enlarged field of view for accessing visibility information outside the screen bounds, thus increasing stability. The approach computes screen-space AO for each layer and maintains the highest contribution. This way, the AO estimator is improved, at the cost of slightly overestimating occlusion. Moreover, Bauer et al. [BKKB13] stored additional properties for translucent objects, such as opacity, on an linked-list A-buffer and computed AO using a modified estimator, considering both the interior and exterior occlusion of objects. For each sample covered by translucent objects, AO is computed for each fragment in the list separately. The final AO is the highest AO contribution from all visited fragments.

Indirect Illumination. Hermes et al. [HHGM10] exploited a *k*-buffer to capture the intersection layers for bundled parallel rays from multiple directions [Hac05], in order to progressively compute irradiance and approximate glossy reflections, via an additional transfer atlas, for one or more light bounces. Widmer et al. [WPS*15] used a two-layer *k*-buffer as input to the generation of a screen-space quad-tree for screen-space reflections. To support multiple bounces, they performed hierarchical screen-space ray traversal on a cubemap representation of their data structure. Mara and McGuire [MMNL16] stored a two-layer depth peeling representation in a single geometry rendering pass, where the second layer was selected based on a minimum fixed separation metric, so that potentially similar geometry can be skipped. By storing only the two-best layers, global illumination computations can be performed in real time. As such, the authors presented various illumination-based applications related to AO, reflections and approximate single-bounce diffuse irradiance gathering. McGuire and Mara [MM14] captured near and far-field illumination effects using per-pixel image space line traversal and per-fragment thickness-based intersection tests. The authors used depth peeling to provide results in several illumination methods, such as AO, reflection and screen-space radiosity. Vardis et al. [VVP16a] used an A-buffer to accommodate multiple layers and views as well as employed various optimisation strategies to accommodate AO and unidirectional path tracing applications on fully-dynamic scenes of arbitrary geometric complexity. The same authors further improved the accuracy of the approach via analytic ray-triangle intersection tests, indexed by the stored fragments [VVP16b] (Fig. 12).

The use of multifragment buffers in global illumination calculations can be clearly grouped in two categories. For low-frequency scattering events and high ambience, shallow (even two-layer) buffers suffice to produce visually convincing results, especially for real-time applications. These methods, however, can easily break down in highly-directional scattering events and complex or expansive environments, where missing light interactions and severe light leaking can occur. For accurate results, either an A-buffer is recommended, ideally coupled with analytic primitive intersections to eliminate holes caused by the sampling of oblique geometry, or a hybrid rendering solution, combining a limited depth *k*-buffer with object-space acceleration data structures for the geometry not captured by the first.

5.4. Distribution Effects and Filtering (DF)

Defocus Blur. The simulation of focal imaging from real world lens systems, such as depth-of-field, is of high importance in real-time rendering as it can increase visual realism significantly. Since out-of-focus regions require information from occluded geometry not available in a single-layer buffer, multifragment techniques have been exploited to augment the missing information and produce more visually convincing results.

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

Specifically, Lee et al. [LES10] used a custom depth peeling approach by observing that certain layers cannot be reached by lens rays and, thus, can be skipped. To simulate high quality depth-of-field, image-space ray tracing is further employed (Sec. 4.2) to traverse fragment layers efficiently. Using ray tracing enables the incorporation of effects from more general lens models, such as anisotropic blur, spherical and chromatic aberration. Selgrad et al. [SRP*15] applied a similar multilayer filtering data structure that is used in the generation of soft shadows [SDMS15] to simulate depth-of-field effects at real-time framerates, without any variance artefacts that may appear in ray tracing approaches. Recently, Franke et al. [FHSS18] combined the observations of Lee at al. [LES10] with the minimum separation metric of Mara and McGuire [MMNL16] to generate a partial multilayer data structure of surfaces affected only by the circle of confusion. This representation is then splatted to screen-space tiles, enabling fast sorting and blending while properly handling occlusion of out-of-focus near-field objects objects at depth discontinuities (Fig. 12).

Antialiasing. Aliasing riddles real-time image synthesis in many ways and one of the typical manifestations is the jagged appearance of primitive edges or the erratic capture of thin structures, due to insufficient sampling in image space.

The original A-buffer algorithm [Car84] was the first method to perform antialiasing at the pixel level using *sampled coverage* in the form of a 32-bit mask. The clipped polygon *fragments* are maintained in a depth-sorted list, where they are also merged for efficiency based on surface congruity and coverage mask overlap and finally resolved to produce the pixel colour and alpha value. Inspired by A-buffer, Jouppi et al. [JC99] proposed a hardware implementation of a polygon antialiasing mechanism that also maintained polygon fragments in a small, fixed memory, by utilising a more aggressive merging operation. Lee and Kim [LK00] also proposed a modified A-buffer algorithm and a respective hardware architecture for antialiased polygon rendering, using dynamic storage allocation. More recently, Kerzner and Salvi [KS14] reduced the memory usage and shading costs associated with multisample anti-aliased G-buffer construction. This is achieved by intercepting fragments on an MSAA-enabled buffer and storing them in a custom, fixed-length array of surface attributes and G-buffer data per pixel. Fragments with similar geometric features are merged, resulting in at most three records in the antialiased G-buffer for eight visibility samples per pixel.

Denoising. Among the many strategies explored to reduce image noise, image-space *denoising* has emerged as a particularly attractive solution due to its effectiveness and ease of integration into rendering pipelines [ZJL*15]. However, the vast majority of image-space methods operate on a single layer of information, where all per-pixel samples are first aggregated in a single value per data channel. To this end, Vicini et al. [VAN*19] presented a novel framework that preserves the full expressiveness of the deep structure in order to denoise multilayer physically-based rendering images. They combine an image-space non-local means filtering of pixel colours with a deep cross-bilateral filter operating on auxiliary features. Despite its relatively straightforward design and pipeline, this work represents the first step towards integrating denoising into modern multilayer-compositing workflows.

From the diverse specialised applications in this category, the common denominator is the flexibility and programmable pipeline offered by the availability of multiple layers of information. For antialiasing purposes, retaining a low-count, fixed-size fragment list can result in large memory savings in the cost of some small quality loss. Denoising operations applied on a fixed number of layers can more easily enable the use of future deep learning approaches for filtering. Finally, defocus blurring can be conveniently mapped to shallow multifragment buffers through the clustering of fragments into focal zones.

5.5. Shape Representation (SR)

Constructive Solid Geometry. Computing the boundary of solids defined in constructive solid geometry (CSG) with geometry primitives is a computationally expensive and numerically delicate procedure. As such, the visualisation of CSG operations has also been performed via multilayer data structures in the literature, for efficient and interactive GPU rendering. The traditional approach involves traversing the sorted list of fragments (Sec. 4.1) in order to classify surfaces as interior/exterior and apply common CSG operations, such as union, intersection and difference, during image composition (Fig. 8).

Specifically, Lefebvre et al [LHL13] used an A-buffer to store shape information in per-pixel linked lists. The combined CSG model is determined by updating bitfields through Boolean expressions, where each bit represents whether the ray is inside or outside a particular primitive. Kauker et al. [KKP*13] stored fragments either in per-pixel linked lists or variable-length arrays to render molecular surfaces with OIT, where atoms are represented as spheres. The final colour of a molecule is produced through a compositing stage that removes the interior parts of atoms through a CSG union operation. Surface boundaries are determined simply by examining the orientation of fragments with respect to the viewer. Furthermore, Rossignac et al. [RFV13] performed GPU-based trimming and CSG on self-intersected surfaces with support for interactive free-form editing of deformable objects (Fig. 12). The approach was evaluated on various depth-peeling and A-buffer variants.

Scientific Visualisation. MFR has been widely exploited to perform interactive visual exploration of large, complex, and multimodal scientific data. In many applications, visualisation of hybrid data representations is required, i.e. the fusion of multiple volumetric and potentially transparent surface data sources. Several screen-space techniques have been developed to compute and process the fragments in a correct composition order.

As such, Busking et al. [BBF*11] proposed an image-space pipeline, based on depth-peeling and deferred shading, to enhance comparative visualisation of medical images at interactive rates. Differences between surfaces are extracted using CSG operations with in/out classification masking. Kanamori et al. [KSN08] used depth-peeling to render particle-based density fields, where ray-isosurface intersection tests are performed by iteratively peeling layers of rasterised spheres. The iterative approach of depth peeling was later replaced by A-buffer variants to further accelerate the process of ray casting [SI12, PB13, KKP*13].

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

Liu et al. [LCD10] used a hybrid representation, based on quadtrees and multilayer point rendering, to generate compact ray segments for empty space skipping during the ray casting stage. A similar methodology was applied by Hadwiger et al. [HAB*18] for interactive visualisation of both sparse and dense volumes. In this work, volumes are converted to a hierarchical geometric representation in advance in order to generate bounding boxes of different volume occupancy classes. During rendering, ray-segments of these classes are constructed and stored in per-pixel lists to enable efficient skipping of unoccupied regions of volume space.

Lindholm et al [LFS*15] employed an A-buffer for hybrid visualisation of both geometric and volumetric datasets. The authors employed a depth histogram analysis with memory management optimisations, during the post-sorting stage, in order to minimise cache allocation and maximise throughput. As such, they enabled interactive rendering in various applications, such as computational fluid dynamics, space weather simulation, and biomedical data visualisation (Fig. 12).

Gunther et al. [GRT13] used per-pixel linked lists for opacity-based visualisation of dense line fields, a method adapted later for surface flow visualisation and other data types [GTG17]. Apart from storing colour and depth, for blending transparent layers, they also capture occlusion measurements between lines. This information is used on a subsequent step as part of a minimisation function, to adaptively fade out line parts based on their importance.

The potentially large number of layers in scientific and CAD visualisation tasks, could mandate the use of an A-buffer. Massive dataset visualisation, with multiple overlaid surfaces or information layers can benefit from techniques that compress the farthest fragments, since their contribution in the composition of the final image and therefore, any resulting error, is negligible.

5.6. Collision Detection (CD)

Screen-space intersection techniques rely on the rasterisation of the objects in two collision groups into fragment-based representations and the determination of intersection via fragment traversal (Sec. 4.1). By avoiding construction of spatial hierarchies, they can support all types of rasterisable primitives and can deal well with highly-deformable geometry. However, they rely on an approximate fragment-to-fragment intersection test that heavily depends on the buffer resolution.

For instance, Jang and Han [JH08] proposed a hybrid CPU-GPU screen-space collision detection method (Fig. 12). Regions of interest, i.e. overlapping bounding boxes, are sent to a stencil routed k-buffer to identify potentially colliding sets of polygons. These sets are then fed back to the CPU for ray-triangle intersection tests. Furthermore, Morvan et al. [MRS12] used a slightly different approach for efficient proximity queries. They identify regions of interest on the GPU and store the vertices of the contained objects in an orthographic per-pixel linked list. A subsequent pass then renders the primitives and computes potential intersections and minimum distances. Radwan et al. [ROW14] computed intersections between densely sampled point clouds by rasterising the points to multifragment buffers and merging multiple depth values to form sampled surface boundaries.

6. Open Problems/Challenges

Despite the fact that MFR techniques have become essential in many rendering tasks, efficient construction and processing of multiple fragments is a non-trivial process due to the large amount of data generated even on medium-resolution displays. The optimisation of current algorithms and the development of novel solutions for MFR is still an ongoing and active field of research. In the following part of this section, we summarise a number of open and challenging issues that remain partially covered, or even unresolved, along with a fruitful discussion on potential directions, which, we believe could shape the future research investigation on this field.

Data Reuse. While taking advantage of *spatio-temporal coherence* can vastly increase performance in a very large number of rendering scenarios [SYM*11], we have noticed that not many approaches have leveraged the redundancy of information over time and space, e.g. [MMNL16]. Therefore, a very promising direction is to exploit *reprojection* of fragment data from previous frames to other MFR variants in order to achieve deep buffer construction at a lower cost. A larger amount of redundancy can be alleviated by adaptively scheduling the A-buffer construction over several frames, based on the observation that small spatio-temporal changes occur in a limited time interval. Moreover, the k-buffer construction process could gain a significant performance boost by predicting the exact depth of the k-th fragment a priori, thus allowing the capture of fragments with smaller or equal depth in constant time, discarding the remaining ones [VF14]. Regarding virtual reality, MFR strategies have not been applied for efficient stereoscopic rendering. Specifically, data stored in a multilayer G-buffer structure could be reprojected to render the view for each eye with greater reprojection success ratio and smaller error, compared to similar methods applied to single-layer depth information.

Data Compression. Although graphics hardware has evolved remarkably in the past decade, graphics applications have become aggressively more demanding in terms of content complexity and physically-based realism, striving at the same time to meet the demand for interactive framerates in high-resolution displays. As geometry and display resolution continue to grow rapidly in order to reach the bar of expected user experience, they are simultaneously stretching and testing the boundaries of MFR solutions. Following the conventional approach of rendering via rasterisation is marred with limitations, directly leading to the natural explosion of the amount of dynamic and heterogeneous fragment datasets that have to eventually be processed in each frame. Although this issue has been sufficiently covered only for improving OIT and shadow generation (Sec. 5.1 and 5.2), new promising directions have to be revealed towards a breakthrough in the general multilayer image/buffer compression problem guided by modest single-layer solutions [HAM06].

Data Sampling. The process of rasterisation, which is the heart of MFR, imposes some very severe limitations in the graphics pipeline. The scan conversion procedure approximates the underlying geometry; a continuous primitive is represented by discretised samples at specific (usually regular) locations on the image plane. This results in three main sources of spatial aliasing: (i) polygon edges and oblique geometry are sparsely sampled, while geometry

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

Table 3: *Categorising the MFR papers, discussed in this report, based on the construction type and the operation(s) used to realise one or more GPU-accelerated application use cases. Symbols correspond to OIT: Order-independent Transparency, SH: Shadows, GI: Global Illumination, DF: Distribution Effects and Filtering, SR: Shape Representation and CD: Collision Detection.*

	Operations			Applications					
Method	**Z-Traversal** 4.1	**Ray Tracing** 4.2 & 4.3	**Other** 4.4	**OIT** 5.1	**SH** 5.2	**GI** 5.3	**DF** 5.4	**SR** 5.5	**CD** 5.6
Depth Peeling (3.1)									
[Eve01, BM08b, Thi08]	-	-	-	√	-	-	-	-	-
[BS09]	√	-	-	-	-	√	-	-	-
[KSN08, BBF*11]	√	-	-	-	-	-	-	√	-
[BCS08, LKE18]	√	-	-	-	√	-	-	-	-
[LHLW09, VF13]	√	-	-	√	-	-	-	-	-
[LES10]	√	√	-	-	-	-	√	-	-
[ROW14]	√	-	√	-	-	-	-	-	√
[MM14, MMNL16]	√	√	-	-	-	√	-	-	-
[FHSS18]	√	√	√	-	-	-	√	-	-
A-buffer (3.2)									
[JH08, MRS12]	√	-	-	-	-	-	-	-	√
[SVLL10, SDMS15, WHL15]	√	-	√	-	√	-	-	-	-
[YHGT10, LHLW10, KLZ12, VF12, MCTB14, SBF15, JCLR19]	√	-	-	√	-	-	-	-	-
[LCD10, SI12, PB13, GRT13, RFV13, HAB*18]	√	-	-	-	-	-	-	√	-
[GRT13, KKP*13, LHL13, LFS*15]	√	-	-	√	-	-	-	√	-
[CFM*13]	√	-	√	√	-	-	-	-	-
[BKKB13, VVP16a, VVP16b]	√	√	-	-	-	√	-	-	-
[SRP*15, VAN*19]	√	-	√	-	-	-	√	-	-
[HBSS17]	√	√	√	-	-	√	-	-	-
k-buffer (3.3)									
[BCL*07]	√	-	-	√	-	-	√	√	-
[LWXW09, YYH*12, Sal13, TH14, HBT14, Wym16, VVPM17]	√	-	-	√	-	-	-	-	-
[HHGM10]	√	-	-	-	-	√	-	-	-
[SML11, MCTB13, SV14, MBG16]	√	-	√	√	-	-	-	-	-
[KS14]	√	-	√	-	-	-	√	-	-
[VF14]	√	-	-	√	-	-	-	√	-
[WPS*15]	√	√	-	-	-	√	√	-	-

(ii) either parallel to the view direction or (iii) residing outside the view frustum are skipped entirely. Although *conservative rasterisation* [HAMO05], which guarantees that a fragment will always be generated as long as there exists some partial overlap between a pixel and a primitive, can resolve the first issue, zero area projected primitives will not generate any fragments making the second sampling issue inevitable. In order to limit view dependencies to a minimum, voxelisation [HHZ*14, KFR*16] and GPGPU-ray tracing [GD15, MBJ*15, WMB19] have been cooperatively operating alongside traditional rasterisation in order to accurately capture and accelerate certain effects that demand accurate partial visibility such as global illumination. Further on, the launch of NVIDIA's RTX platform for hardware-accelerated ray tracing computations got an enthusiastic welcome from the games industry, showing the significance of switching to more physically accurate solutions for real-time graphics [HAM19]. Although this architecture modification can provide a significant performance and quality boost for certain types of content and visual effects, it is not enough to support full, photorealistic global-illumination in generalised and dynamic geometry streams [KVBB*19]. So, we strongly believe that harnessing this trend and hardware support to devise smart, hybrid MFR/ray-tracing methods can be the key direction on which novel research may endeavour.

Presentation Device. The possible graphics applications are rapidly growing to include casual users on mobile devices [AGM*17] or immersive devices such as virtual and augmented reality devices [KAS*19]. On the one hand, mobile graphics hardware has huge constraints in terms of energy, power and silicon area they use, resulting in lower bandwidth, performance and more conservative use of texturing, compared to their desktop counterparts. On the other hand, regarding the visual aspect of mixed reality, making the transition from 2D display graphics to an immersive display poses several significant challenges, including dramatically increased resolutions under lower latency constraints, to preserve the continuous illusion of reality. Thus, thorough research investigation and exploitation of MFR is necessary to expand the boundaries of realism and immersion, under these limited budget constraints and strict rendering requirements, for the rapidly developing application domain of virtual and augmented reality in current and emerging platforms.

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

Graphics Hardware. Over the past decade, graphics hardware has undergone an incredible transformation mainly driven by the video game industry and its economic momentum. Although the rapid evolution of GPUs has greatly impacted the multifragment processing capabilities in more that one ways, including atomic operations, dynamic memory location writes [YHGT10] and pixel synchronisation [Sal13], novel specialised modifications to the current hardware pipeline are ultimately required to enable full hardware support of k-buffer or A-buffer solutions on future GPUs. Predicting fragment storage overflow at the global graphics hardware (Sec. 3.2.1) as well as improving GPU local memory caching and occupancy (Sec. 3.2.2) when highly-irregular per-pixel fragment sampling is manifested, should be of major importance. Moreover, augmenting *variable rate shading* in the depth domain via selective MFR strategies [VVPM17] can be a promising direction for investigation.

Application Domain. In the most part, improving the quality and performance of OIT (Sec. 5.1) was the driving force for advancing the MFR domain in both software and hardware throughout the past few years (Tab. 3). The requirement of correct and convincing display of transparent geometry in real-time framerates in games had excessively revealed several bottlenecks and challenges with respect to fragment sampling and processing in graphics hardware. However, despite the rapid increase of number of graphics applications that progressively rely on MFR solutions to develop visually convincing images, we observe that the MFR is inadequately covered to a full extent, allowing immediate opportunities for future research. For example, screen-space reflections, which is currently a trend in the video games industry [Ulu14], can be highly accelerated by optimising the pixel-space ray traversal operation (Sec. 4.2). Last but not least, we have identified that a mapping of successful rendering methods, such as photon mapping and motion blur, to the screen-space paradigm is still missing.

7. Conclusion

In this survey, we have described the principles and fundamentals of multifragment rendering. This paper aimed to include the majority of the most relevant and state-of-the-art MFR solutions categorised by the approach taken for storing and processing the generated fragment samples. We have showed that a large body of applications have successfully explored MFR for many aspects of rendering and visualisation domains. We have also identified aspects, inadequately covered by existing literature, that may provide immediate opportunities for future research.

Historically, the divergent design considerations between offline and real-time applications have resulted in two key approaches to resolve direct and indirect visibility: rasterization and ray tracing. Despite the tremendous progress on the landscape of both fields, we believe that the MFR, alone or in hybrid schemes, will continue to play an important role of handling partial visibility determination. Working closely with the field of ray tracing and the application domains we can uncover, and find solutions, in an area with renewed research interest, where high potential for software and hardware improvements is feasible in the near future.

Acknowledgements

We are grateful to all the authors for granting permission to use their images and to the anonymous reviewers for their constructive comments. This research is co-financed by Greece and the European Union (European Social Fund-ESF) through the Operational Programme "Human Resources Development, Education and Lifelong Learning 2014-2020" in the context of the project "Modular Light Transport for Photorealistic Rendering on Low-power Graphics Processors" (5049904).

References

[AGM*17] AGUS M., GOBBETTI E., MARTON F., PINTORE G., VÁZQUEZ ALCOCER P. P.: Mobile Graphics. In *EG 2017: Tutorials* (2017), The Eurographics Association, pp. 1–5. 16

[AL18] ARCHER J., LEACH G.: Further Improvements to OIT Sort Performance. In *Proceedings of Computer Graphics International 2018* (New York, NY, USA, 2018), CGI 2018, ACM, pp. 147–152. 7

[ALvS18] ARCHER J., LEACH G., VAN SCHYNDEL R.: GPU-based Techniques for Deep Image Merging. *Computational Visual Media 4*, 3 (Sep 2018), 277–285. 11

[AMHH*18] AKENINE-MÖLLER T., HAINES E., HOFFMAN N., PESCE A., IWANICKI M., HILLAIRE S.: *Real-Time Rendering 4th Edition*. A K Peters/CRC Press, Boca Raton, FL, USA, 2018. 3

[AW87] AMANATIDES J., WOO A.: A Fast Voxel Traversal Algorithm for Ray Tracing. In *EG 1987-Technical Papers* (1987), Eurographics Association. 10

[BBF*11] BUSKING S., BOTHA C. P., FERRARINI L., MILLES J., POST F. H.: Image-based Rendering of Intersecting Surfaces for Dynamic Comparative Visualization. *Vis. Comput. 27*, 5 (May 2011), 347–363. 14, 16

[BCL*07] BAVOIL L., CALLAHAN S. P., LEFOHN A., COMBA J. A. L. D., SILVA C. T.: Multi-fragment Effects on the GPU Using the k-buffer. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (NY, USA, 2007), I3D '07, ACM, pp. 97–104. 5, 8, 9, 16

[BCS08] BAVOIL L., CALLAHAN S. P., SILVA C. T.: Robust Soft Shadow Mapping with Backprojection and Depth Peeling. *Journal of Graphics Tools 13*, 1 (2008), 19–30. 12, 16

[BHKW07] BÜRGER K., HERTEL S., KRÜGER J., WESTERMANN R.: GPU Rendering of Secondary Effects. In *Vision, Modeling and Visualization 2007* (2007). 10

[BKKB13] BAUER F., KNUTH M., KUIJPER A., BENDER J.: Screen-Space Ambient Occlusion Using A-Buffer Techniques. In *2013 International Conference on Computer-Aided Design and Computer Graphics* (Nov 2013), pp. 140–147. 13, 16

[BM08a] BAVOIL L., MYERS K.: Deferred Rendering using a Stencil Routed k-Buffer. *ShaderX6: Advanced Rendering Techniques* (2008), 189–198. 8

[BM08b] BAVOIL L., MYERS K.: *Order Independent Transparency with Dual Depth Peeling*. Tech. rep., Nvidia Corporation, 2008. 5, 16

[BS09] BAVOIL L., SAINZ M.: Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH 2009: Talks* (New York, NY, USA, 2009), SIGGRAPH '09, ACM, pp. 45:1–45:1. 13, 16

[Car84] CARPENTER L.: The A-buffer, an Antialiased Hidden Surface Method. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 103–108. 3, 6, 14

[Cat74] CATMULL E. E.: *A subdivision algorithm for computer display of curved surfaces*. PhD thesis, The University of Utah, 1974. 3

[CFM*13] CARNECKY R., FUCHS R., MEHL S., JANG Y., PEIKERT R.: Smart Transparency for Illustrative Visualization of Complex Flow Surfaces. *IEEE Transactions on Visualization and Computer Graphics 19*, 5 (May 2013), 838–851. 10, 16

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

[CICS05] CALLAHAN S. P., IKITS M., COMBA J. L. D., SILVA C. T.: Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3 (May 2005), 285–295. 8

[CMM08] CARR N., MĚCH R., MILLER G.: Coherent Layer Peeling for Transparent High-depth-complexity Scenes. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (Switzerland, 2008), GH '08, Eurographics Association, pp. 33–40. 5

[Cra10a] CRASSIN C.: Fast and accurate single-pass A-buffer, 2010. 6, 7

[Cra10b] CRASSIN C.: Linked lists of fragment pages, 2010. 6

[CS93] COHEN D., SHAKED A.: Photo-Realistic Imaging of Digital Terrains. *Computer Graphics Forum* (1993). 10

[DL03] DUAN J., LI J.: Compression of the layered depth image. *IEEE Transactions on Image Processing 12*, 3 (March 2003), 365–372. 11

[DS05] DACHSBACHER C., STAMMINGER M.: Reflective Shadow Maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games* (NY, USA, 2005), I3D '05, ACM, pp. 203–231. 3

[Duf17] DUFF T.: Deep Compositing Using Lie Algebras. *ACM Trans. Graph. 36*, 3 (June 2017). 11

[EDL15] EGSTAD J., DAVIS M., LACEWELL D.: Improved Deep Image Compositing Using Subpixel Masks. In *Proceedings of the 2015 Symposium on Digital Production* (New York, NY, USA, 2015), DigiPro '15, ACM, pp. 21–27. 11

[ESAW11] EISEMANN E., SCHWARZ M., ASSARSSON U., WIMMER M.: *Real-Time Shadows*. A. K. Peters, Ltd., MA, USA, 2011. 12

[Eve01] EVERITT C.: *Interactive Order-Independent Transparency*. Tech. rep., Nvidia Corporation, 2001. 4, 5, 16

[FHSS18] FRANKE L., HOFMANN N., STAMMINGER M., SELGRAD K.: Multi-Layer Depth of Field Rendering with Tiled Splatting. *Proc. ACM Comput. Graph. Interact. Tech. 1*, 1 (2018), 6:1–6:17. 12, 14, 16

[FMS13] FIIRST R., MATTAUSCH O., SCHERZER D.: Real-Time Deep Shadow Maps. *GPU Pro 4: Advanced Rendering Techniques 4* (2013), 253. 6

[GBP06] GUENNEBAUD G., BARTHE L., PAULIN M.: Real-time Soft Shadow Mapping by Backprojection. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques* (Switzerland, 2006), EGSR '06, Eurographics Association, pp. 227–234. 12

[GD15] GANESTAM P., DOGGETT M.: Real-time multiply recursive reflections and refractions using hybrid rendering. *The Visual Computer 31*, 10 (2015), 1395–1403. 16

[GRT13] GÜNTHER T., RÖSSL C., THEISEL H.: Opacity Optimization for 3D Line Fields. *ACM Trans. Graph. 32*, 4 (2013), 120:1–120:8. 15, 16

[GTG17] GÜNTHER T., THEISEL H., GROSS M.: Decoupled Opacity Optimization for Points, Lines and Surfaces. *Computer Graphics Forum 36*, 2 (2017), 153–162. 15

[HAB*18] HADWIGER M., AL-AWAMI A. K., BEYER J., AGUS M., PFISTER H.: SparseLeap: Efficient Empty Space Skipping for Large-Scale Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 24*, 1 (Jan 2018), 974–983. 15, 16

[Hac05] HACHISUKA T.: High-Quality Global Illumination Rendering Using Rasterization. In *GPU Gems 2* (2005), Addison-Wesley Professional. 13

[HAM06] HASSELGREN J., AKENINE-MÖLLER T.: Efficient Depth Buffer Compression. In *Graphics Hardware* (2006), The Eurographics Association. 15

[HAM19] HAINES E., AKENINE-MÖLLER T. (Eds.): *Ray Tracing Gems*. Apress, 2019. http://raytracinggems.com. 16

[HAMO05] HASSELGREN J., AKENINE-MÖLLER T., OHLSSON L.: Conservative rasterization. *GPU Gems 2* (2005), 677–690. 16

[HB14] HAVRAN V., BITTNER J.: Efficient Sorting and Searching in Rendering Algorithms. In *Eurographics 2014: Tutorials* (2014), The Eurographics Association. 2

[HBSS17] HOFMANN N., BOGENDÖRFER P., STAMMINGER M., SELGRAD K.: Hierarchical Multi-layer Screen-space Ray Tracing. In *Proceedings of High Performance Graphics* (New York, NY, USA, 2017), HPG '17, ACM, pp. 18:1–18:10. 10, 16

[HBT14] HILLESLAND K. E., BILODEAU B., THIBIEROZ N.: Deferred Shading for Order-Independent Transparency. In *Proceedings of Eurographics 2014 Short Papers* (2014), EG '14, The Eurographics Association, pp. 49–52. 12, 16

[HCNS18] HAN S., CHEN G., NEHAB D., SANDER P. V.: In-Depth Buffers. *Proc. ACM Comput. Graph. Interact. Tech. 1*, 1 (July 2018), 2:1–2:14. 8, 11

[HHGM10] HERMES J., HENRICH N., GROSCH T., MUELLER S.: Global Illumination using Parallel Global Ray-Bundles. In *Vision, Modeling, and Visualization (2010)* (2010), The Eurographics Association. 13, 16

[HHHF12] HANIKA J., HILLMAN P., HILL M., FASCIONE L.: Camera Space Volumetric Shadows. In *Proceedings of the Digital Production Symposium* (NY, USA, 2012), DigiPro '12, ACM, pp. 7–14. 11

[HHZ*14] HU W., HUANG Y., ZHANG F., YUAN G., LI W.: Ray tracing via GPU rasterization. *The Visual Computer 30*, 6 (2014), 697–706. 16

[JC99] JOUPPI N. P., CHANG C.-F.: Z3: An Economical Hardware Technique for High-quality Antialiasing and Transparency. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (NY, USA, 1999), HWWS '99, ACM, pp. 85–93. 8, 14

[JCLR19] JANSSON E. S., CHAJDAS M. G., LACROIX J., RAGNEMALM I.: Real-Time Hybrid Hair Rendering. In *Eurographics Symposium on Rendering - DL-only and Industry Track* (2019), The Eurographics Association. 11, 12, 16

[JH08] JANG H., HAN J.: Fast collision detection using the A-buffer. *The Visual Computer 24*, 7 (Jul 2008), 659–667. 15, 16

[JLBM05] JOHNSON G. S., LEE J., BURNS C. A., MARK W. R.: The Irregular Z-buffer: Hardware Acceleration for Irregular Data Structures. *ACM Trans. Graph. 24*, 4 (Oct. 2005), 1462–1482. 13

[KAS*19] KOULIERIS G. A., AKŞIT K., STENGEL M., MANTIUK R., MANIA K., RICHARDT C.: Near-Eye Display and Tracking Technologies for Virtual and Augmented Reality. *Computer Graphics Forum 38*, 2 (2019), 493–519. 16

[KFR*16] KAUKER D., FALK M., REINA G., YNNERMAN A., ERTL T.: VoxLink-Combining sparse volumetric data and geometry for efficient rendering. *Computational Visual Media 2*, 1 (2016), 45–56. 16

[KKP*13] KAUKER D., KRONE M., PANAGIOTIDIS A., REINA G., ERTL T.: Rendering Molecular Surfaces Using Order-independent Transparency. In *Proceedings of the 13th Eurographics Symposium on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, 2013), EGPGV '13, Eurographics Association, pp. 33–40. 14, 16

[KLZ12] KNOWLES P., LEACH G., ZAMBETTA F.: Efficient Layered Fragment Buffer Techniques. In *OpenGL Insights*. CRC Press, 2012, pp. 279–292. 1, 6, 7, 9, 16

[KLZ13] KNOWLES P., LEACH G., ZAMBETTA F.: Backwards Memory Allocation and Improved OIT. In *Proceedings of Pacific Graphics 2013 (Short Papers)* (October 2013), PG '13, pp. 59–64. 7

[KLZ14] KNOWLES P., LEACH G., ZAMBETTA F.: Fast sorting for exact OIT of complex scenes. *The Visual Computer 30*, 6 (2014), 603–613. 7

[KS14] KERZNER E., SALVI M.: Streaming G-buffer Compression for Multi-sample Anti-aliasing. In *Proceedings of High Performance Graphics* (Goslar Germany, Germany, 2014), HPG '14, Eurographics Association, pp. 1–7. 11, 14, 16

[KSN08] KANAMORI Y., SZEGO Z., NISHITA T.: GPU-based Fast Ray Casting for a Large Number of Metaballs. *Computer Graphics Forum 27*, 2 (2008), 351–360. 5, 14, 16

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

[Kub14] KUBISCH C.: Order Independent Transparency In OpenGL 4.x. In *GPU Technology Conference 2014* (2014), GTC '14. 8

[KVBB*19] KELLER A., VIITANEN T., BARRÉ-BRISEBOIS C., SCHIED C., MCGUIRE M.: Are We Done with Ray Tracing? In *ACM SIGGRAPH 2019 Courses* (New York, NY, USA, 2019), SIGGRAPH '19, ACM, pp. 3:1–3:381. 16

[KWBG13] KERZNER E., WYMAN C., BUTLER L., GRIBBLE C.: Toward Efficient and Accurate Order-independent Transparency. In *ACM SIGGRAPH 2013 Posters* (New York, NY, USA, 2013), SIGGRAPH '13, ACM, pp. 109:1–109:1. 6

[LCD10] LIU B., CLAPWORTHY G. J., DONG F.: Multi-layer Depth Peeling by Single-Pass Rasterisation for Faster Isosurface Raytracing on GPUs. *Computer Graphics Forum 29*, 3 (2010), 1231–1240. 4, 15, 16

[LES10] LEE S., EISEMANN E., SEIDEL H.-P.: Real-time Lens Blur Effects and Focus Control. *ACM Trans. Graph. 29*, 4 (July 2010), 65:1–65:7. 14, 16

[LFS*15] LINDHOLM S., FALK M., SUNDÉN E., BOCK A., YNNERMAN A., ROPINSKI T.: Hybrid Data Visualization Based on Depth Complexity Histogram Analysis. *Computer Graphics Forum 34*, 1 (2015), 74–85. 7, 12, 15, 16

[LHL13] LEFEBVRE S., HORNUS S., LASRAM A.: *HA-Buffer: Coherent Hashing for single-pass A-buffer*. Research Report RR-8282, INRIA, Apr. 2013. 6, 7, 14, 16

[LHL15] LEFEBVRE S., HORNUS S., LASRAM A.: Per-Pixel Lists for Single Pass A-Buffer. In *GPU Pro 5: Advanced Rendering Techniques*. A K Peter / CRC Press, 2015. 6

[LHLW09] LIU F., HUANG M.-C., LIU X.-H., WU E.-H.: Efficient Depth Peeling via Bucket Sort. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 51–57. 5, 16

[LHLW10] LIU F., HUANG M.-C., LIU X.-H., WU E.-H.: FreePipe: A Programmable Parallel Rendering Architecture for Efficient Multi-fragment Effects. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), I3D '10, ACM, pp. 75–82. 5, 6, 8, 16

[LK00] LEE J.-A., KIM L.-S.: Single-pass full-screen hardware accelerated antialiasing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (New York, NY, USA, 2000), HWWS '00, ACM, pp. 67–75. 14

[LKE18] LEE S., KIM Y., EISEMANN E.: Iterative Depth Warping. *ACM Trans. Graph. 37*, 5 (Oct. 2018), 177:1–177:13. 4, 13, 16

[LV00] LOKOVIC T., VEACH E.: Deep Shadow Maps. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., pp. 385–392. 11, 12

[LW13] LEUNG Y.-S., WANG C. C. L.: Conservative Sampling of Solids in Image Space. *IEEE Comput. Graph. Appl. 33*, 1 (Jan. 2013), 32–43. 8

[LWXW09] LIU B., WEI L., XU Y., WU E.: Multi-layer depth peeling via fragment sort. In *11th IEEE International Conference on Computer-Aided Design and Computer Graphics* (2009), pp. 452–456. 4, 8, 16

[Mam89] MAMMEN A.: Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications 9*, 4 (July 1989), 43–55. 4

[MBG16] MURRAY D., BARIL J., GRANIER X.: Shape Depiction for Transparent Objects with Bucketed k-Buffer. In *Eurographics Symposium on Rendering - Experimental Ideas & Implementations* (2016), The Eurographics Association. 11, 12, 16

[MBJ*15] MATTAUSCH O., BITTNER J., JASPE A., GOBBETTI E., WIMMER M., PAJAROLA R.: CHC+RT: Coherent Hierarchical Culling for Ray Tracing. *Computer Graphics Forum 34*, 2 (2015), 537–548. 16

[MCTB11] MAULE M., COMBA J. L., TORCHELSEN R. P., BASTOS R.: A survey of raster-based transparency techniques. *Computers & Graphics 35*, 6 (2011), 1023 – 1034. 2, 3, 9, 11

[MCTB13] MAULE M., COMBA J., TORCHELSEN R., BASTOS R.: Hybrid Transparency. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2013), I3D '13, ACM, pp. 103–118. 5, 8, 11, 12, 16

[MCTB14] MAULE M., COMBA J. L., TORCHELSEN R., BASTOS R.: Memory-optimized order-independent transparency with Dynamic Fragment Buffer. *Computers & Graphics 38* (2014), 1 – 9. 5, 6, 16

[MET*14] MARTIN T., ENGEL W., THIBIEROZ N., YANG J., LACROIX J.: TressFX: Advanced real-time hair rendering. *GPU Pro 5* (2014), 193–209. 9

[Mit07] MITTRING M.: Finding next gen: CryEngine 2. In *SIGGRAPH 2007 courses* (NY, USA, 2007), SIGGRAPH '07, ACM, pp. 97–121. 3

[MM14] MCGUIRE M., MARA M.: Efficient GPU Screen-Space Ray Tracing. *Journal of Computer Graphics Techniques (JCGT) 3*, 4 (December 2014), 73–85. 10, 13, 16

[MMNL16] MARA M., MCGUIRE M., NOWROUZEZAHRAI D., LUEBKE D.: Deep G-buffers for Stable Global Illumination Approximation. In *Proceedings of High Performance Graphics* (Goslar, Germany, 2016), HPG '16, Eurographics Association, pp. 87–98. 1, 5, 13, 14, 15, 16

[MP01] MARK W. R., PROUDFOOT K.: The F-buffer: A Rasterization-order FIFO Buffer for Multi-pass Rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (New York, NY, USA, 2001), HWWS '01, ACM, pp. 57–64. 3

[MRS12] MORVAN T., REIMERS M., SAMSET E.: Efficient Image-Based Proximity Queries with Object-Space Precision. *Comput. Graph. Forum 31*, 1 (Feb. 2012), 62–74. 15, 16

[PB13] PARULEK J., BRAMBILLA A.: Fast Blending Scheme for Molecular Surface Representation. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (Dec 2013), 2653–2662. 14, 16

[PD84] PORTER T., DUFF T.: Compositing Digital Images. *SIGGRAPH Comput. Graph. 18*, 3 (Jan. 1984), 253–259. 11

[PTO10] PATNEY A., TZENG S., OWENS J. D.: Fragment-Parallel Composite and Filter. *Comput. Graph. Forum 29*, 4 (2010), 1251–1258. 7

[RDGK12] RITSCHEL T., DACHSBACHER C., GROSCH T., KAUTZ J.: The State of the Art in Interactive Global Illumination. *Computer Graphics Forum 31*, 1 (2012), 160–188. 2, 3, 13

[RFV13] ROSSIGNAC J., FUDOS I., VASILAKIS A. A.: Direct rendering of Boolean combinations of self-trimmed surfaces. *Computer-Aided Design 45*, 2 (2013), 288 – 300. 12, 14, 16

[ROW14] RADWAN M., OHRHALLINGER S., WIMMER M.: Efficient Collision Detection While Rendering Dynamic Point Clouds. In *Proceedings of Graphics Interface 2014* (Toronto, Canada, 2014), GI '14, Canadian Information Processing Society, pp. 25–33. 11, 12, 15, 16

[Sal13] SALVI M.: Advances in Real-Time Rendering in Games: Pixel Synchronization: Solving old graphics problems with new data structures. In *ACM SIGGRAPH 2013 Courses* (New York, NY, USA, 2013), SIGGRAPH '13, ACM. 5, 9, 12, 16, 17

[SBF15] SCHOLLMEYER A., BABANIN A., FROEHLICH B.: Order-Independent Transparency for Programmable Deferred Shading Pipelines. *Computer Graphics Forum 34*, 7 (2015), 67–76. 11, 12, 16

[SDMS15] SELGRAD K., DACHSBACHER C., MEYER Q., STAMMINGER M.: Filtering Multi-Layer Shadow Maps for Accurate Soft Shadows. *Comput. Graph. Forum 34*, 1 (2015), 205–215. 12, 13, 14, 16

[SGHS98] SHADE J., GORTLER S., HE L.-W., SZELISKI R.: Layered Depth Images. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 231–242. 1

[SI12] SZÉCSI L., ILLÉS D.: Real-Time Metaball Ray Casting with Fragment Lists. In *Eurographics 2012 - Short Papers* (2012), The Eurographics Association. 14, 16

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

[SJC18] SINGH D. P., JOSHI I., CHOUDHARY J.: Survey of GPU Based Sorting Algorithms. *Int. J. Parallel Program. 46*, 6 (Dec. 2018), 1017–1034. 7

[SKS11] SOUSA T., KASYAN N., SCHULZ N.: Secrets of cryengine 3 graphics technology. In *ACM SIGGRAPH Talks* (2011). 9

[SML11] SALVI M., MONTGOMERY J., LEFOHN A.: Adaptive Transparency. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics* (2011), ACM. 8, 11, 12, 16

[SRP*15] SELGRAD K., REINTGES C., PENK D., WAGNER P., STAMMINGER M.: Real-time Depth of Field Using Multi-layer Filtering. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games* (NY, USA, 2015), i3D '15, ACM, pp. 121–127. 14, 16

[ST90] SAITO T., TAKAHASHI T.: Comprehensible Rendering of 3-D Shapes. *SIGGRAPH Comput. Graph. 24*, 4 (Sept. 1990), 197–206. 5

[SV14] SALVI M., VAIDYANATHAN K.: Multi-layer Alpha Blending. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2014), I3D '14, ACM, pp. 151–158. 11, 12, 16

[SVLL10] SALVI M., VIDIMČE K., LAURITZEN A., LEFOHN A.: Adaptive Volumetric Shadow Maps. In *Proceedings of the 21st Eurographics Conference on Rendering* (Aire-la-Ville, Switzerland, 2010), EGSR'10, Eurographics Association, pp. 1289–1296. 11, 13, 16

[SWP11] SCHERZER D., WIMMER M., PURGATHOFER W.: A Survey of Real-Time Hard Shadow Mapping Methods. *Computer Graphics Forum 30*, 1 (2011), 169–186. 12

[SYM*11] SCHERZER D., YANG L., MATTAUSCH O., NEHAB D., SANDER P. V., WIMMER M., EISEMANN E.: A Survey on Temporal Coherence Methods in Real-Time Rendering. In *Eurographics 2011 - State of the Art Reports* (2011), The Eurographics Association. 15

[TDDD18] THONAT T., DJELOUAH A., DURAND F., DRETTAKIS G.: Thin Structures in Image Based Rendering. *Computer Graphics Forum 37*, 4 (2018), 107–118. 4

[TH14] THIBIEROZ N., HILLESLAND K.: Grass Fur and All Things Hairy. In *Game Developer Conference 2014* (2014), GDC '14. 5, 7, 9, 16

[Thi08] THIBIEROZ N.: Robust order-independent transparency via reverse depth peeling in Direct3D 10. *ShaderX 6: Advanced Rendering Techniques* (2008), 211–226. 5, 16

[Thi11] THIBIEROZ N.: Order-independent transparency using per-pixel linked lists. *GPU Pro 2* (2011), 409–431. 6

[TPK01] THEOHARIS T., PAPAIOANNOU G., KARABASSI E.-A.: The Magic of the Z-Buffer: A Survey. In *Proc. of 9th International Conference on Computer Graphics, Visualization and Computer Vision, WSCG* (2001). 3

[TSdSK13] TOKUYOSHI Y., SEKINE T., DA SILVA T., KANAI T.: Adaptive Ray-bundle Tracing with Memory Usage Prediction: Efficient Global Illumination in Large Scenes. *Computer Graphics Forum 32*, 7 (2013), 315–324. 6

[Ulu14] ULUDAG Y.: Hi-Z Screen-Space Cone-Traced Reflections. In *GPU Pro 5*. CRC Press, 2014, pp. 149–192. 10, 17

[VAN*19] VICINI D., ADLER D., NOVÁK J., ROUSSELLE F., BURLEY B.: Denoising Deep Monte Carlo Renderings. *Computer Graphics Forum 38*, 1 (2019), 316–327. 1, 11, 14, 16

[VF12] VASILAKIS A. A., FUDOS I.: S-buffer: Sparsity-aware multifragment rendering. In *Proceedings of Eurographics 2012 Short Papers* (Cagliari, Sardinia, Italy, 2012), EG '12, pp. 101–104. 5, 6, 16

[VF13] VASILAKIS A. A., FUDOS I.: Depth-Fighting Aware Methods for Multifragment Rendering. *IEEE Transactions on Visualization and Computer Graphics 19*, 6 (June 2013), 967–977. 5, 7, 16

[VF14] VASILAKIS A. A., FUDOS I.: k^+-buffer: Fragment Synchronized k-buffer. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2014), I3D '14, ACM, pp. 143–150. 5, 7, 9, 15, 16

[VP15] VASILAKIS A. A., PAPAIOANNOU G.: Improving k-buffer Methods via Occupancy Maps. In *Eurographics 2015 - Short Papers, Zurich, Switzerland, May 4th - 8th, 2015.* (2015), pp. 69–72. 9

[VPF15] VASILAKIS A. A., PAPAIOANNOU G., FUDOS I.: k^+-buffer: An Efficient, Memory-Friendly and Dynamic k-buffer Framework. *IEEE Transactions on Visualization and Computer Graphics 21*, 6 (June 2015), 688–700. 7, 9

[VVP16a] VARDIS K., VASILAKIS A. A., PAPAIOANNOU G.: A Multi-view and Multilayer Approach for Interactive Ray Tracing. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2016), I3D '16, ACM, pp. 171–178. 6, 10, 13, 16

[VVP16b] VARDIS K., VASILAKIS A. A., PAPAIOANNOU G.: DIRT: Deferred Image-based Ray Tracing. In *Proceedings of High Performance Graphics* (Goslar Germany, Germany, 2016), HPG '16, Eurographics Association, pp. 63–73. 6, 10, 12, 13, 16

[VVPM17] VASILAKIS A. A., VARDIS K., PAPAIOANNOU G., MOUSTAKAS K.: Variable k-buffer using Importance Maps. In *Eurographics 2017 - Short Papers* (2017), The Eurographics Association, pp. 21–24. 4, 5, 9, 16, 17

[WGER05] WEXLER D., GRITZ L., ENDERTON E., RICE J.: GPU-Accelerated High-Quality Hidden Surface Removal. In *Graphics Hardware* (2005), The Eurographics Association. 5

[WHL15] WYMAN C., HOETZLEIN R., LEFOHN A.: Frustum-traced Raster Shadows: Revisiting Irregular Z-buffers. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2015), i3D '15, ACM, pp. 15–23. 10, 13, 16

[Wit01] WITTENBRINK C. M.: R-Buffer: A Pointerless A-Buffer Hardware Architecture. In *Eurographics/SIGGRAPH Graphics Hardware Workshop 2001* (2001), The Eurographics Association. 3

[WMB19] WILLBERGER T., MUSTERLE C., BERGMANN S.: Deferred Hybrid Path Tracing. In *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs* (Berkeley, CA, 2019), Apress, pp. 475–492. 16

[Woo92] WOO A.: The shadow depth map revisited. In *Graphics Gems III*. Academic Press Professional, Inc., 1992, pp. 338–342. 12, 13

[WPS*15] WIDMER S., PAJĄK D., SCHULZ A., PULLI K., KAUTZ J., GOESELE M., LUEBKE D.: An Adaptive Acceleration Structure for Screen-space Ray Tracing. In *Proceedings of the 7th Conference on High-Performance Graphics* (New York, NY, USA, 2015), HPG '15, ACM, pp. 67–76. 10, 13, 16

[WX13] WANG W., XIE G.: Memory-Efficient Single-Pass GPU Rendering of Multifragment Effects. *IEEE Transactions on Visualization and Computer Graphics 19*, 8 (Aug 2013), 1307–1316. 8

[Wym16] WYMAN C.: Exploring and Expanding the Continuum of OIT Algorithms. In *Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics* (2016), The Eurographics Association. 2, 12, 16

[XTP07] XIE F., TABELLION E., PEARCE A.: Soft Shadows by Ray Tracing Multilayer Transparent Shadow Maps. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques* (Aire-la-Ville, Switzerland, 2007), EGSR'07, Eurograph. Association, pp. 265–276. 10

[YHGT10] YANG J. C., HENSLEY J., GRÜN H., THIBIEROZ N.: Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum 29*, 4 (2010), 1297–1304. 5, 6, 7, 8, 16, 17

[YYH*12] YU X., YANG J. C., HENSLEY J., HARADA T., YU J.: A Framework for Rendering Complex Scattering Effects on Hair. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (NY, USA, 2012), I3D '12, ACM, pp. 111–118. 5, 8, 11, 16

[Zha14] ZHANG N.: Memory-Hazard-Aware K-Buffer Algorithm for Order-Independent Transparency Rendering. *IEEE Transactions on Visualization and Computer Graphics 20*, 2 (Feb 2014), 238–248. 8

[ZJL*15] ZWICKER M., JAROSZ W., LEHTINEN J., MOON B., RAMAMOORTHI R., ROUSSELLE F., SEN P., SOLER C., YOON S.-E.: Recent Advances in Adaptive Sampling and Reconstruction for Monte Carlo Rendering. *Comput. Graph. Forum 34*, 2 (2015), 667–681. 14

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.