Computers & Graphics 35 (2011) 1023–1034

Contents lists available at SciVerse ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Technical Section

A survey of raster-based transparency techniques ☆

Marilena Maule[a], João L.D. Comba[a,*], Rafael P. Torchelsen[b], Rui Bastos[c]

[a] UFRGS, Brazil
[b] UFFS, Brazil
[c] NVIDIA Corporation, USA

ARTICLE INFO

Article history:
Received 31 December 2010
Received in revised form
1 July 2011
Accepted 25 July 2011
Available online 3 August 2011

Keywords:
Transparency
Fragment sorting
Order-independent transparency

ABSTRACT

Transparency is an important effect for several graphics applications. Correct transparency rendering requires fragment-sorting, which can be more expensive than sorting geometry primitives, and can handle situations that might not be solved in geometry space, such as object interpenetrations. In this paper we survey different transparency techniques and analyze them in terms of processing time, memory consumption, and accuracy. Ideally, the perfect method computes correct transparency in real-time with low memory usage. However, achieving these goals simultaneously is still a challenging task. We describe features and trade-offs adopted by each technique, pointing out pros and cons that can be used to help with the decision of which method to use in a given situation.

1. Introduction

Transparency is the physical property of materials that allows light to pass through objects. This property is important to estimate the appearance of real objects, and it is largely used to denote the relationship among structures in visual interaction. The focus of this survey is to summarize and describe specific techniques for rendering transparent objects using raster-based pipelines.

In raster-based pipelines, the computation of transparency is simplified based on the assumption that there is no refraction when light passes from one medium to another. In this formulation, computing the correct transparency requires properly blending fragment colors, considering their surface opacities and their distances to the viewpoint, as described by Porter and Duff [1]. For transparency to be correctly computed, fragments must be composed in the same order of their distance to the camera, either in front-to-back (FTB) or back-to-front (BTF)[1] ordering.

Fragments combined in unsorted depth order might not have their contributions properly evaluated. For example, consider the computation of a pixel color involving three fragments. Assume that in correct depth order the closest and farthest fragments are transparent, while the middle fragment is opaque. If the opaque fragment is combined last (after the transparent ones), the incorrect contribution of the farthest fragment cannot be discarded anymore.

Fig. 1(a) illustrates the expected result after rasterization of three triangles facing the camera. In this example, the green triangle is the closest to the camera, while the red one is the farthest. The BTF blending equation expects triangles in the following order: red, blue, and green (as in Painter's algorithm [2]). This way, the green contribution is correctly identified as the nearest to the camera. Fig. 1(b) illustrates color blending in incorrect depth order. Note that the green triangle is not drawn in front of the others, due to the out-of-order blending. Tables 1 and 2 give numerical examples for both situations.

As we can conclude that sorting is the main topic of the papers we survey, and we use it as the criteria to classify methods into the following categories:

- *Geometry-sorting*: sort geometry (meshes or primitives) before rasterization;
- *Fragment-sorting*: sort rasterization fragments before blending, using buffer-based or depth peeling;
- *Hybrid-sorting*: combine geometry-sorting with fragment-sorting;
- *Depth-sorting-independent*: blend fragments without considering their depth order;
- *Probabilistic*: estimate visibility without sorting.

We organize our presentation using the above classification in the sections that follow. To clarify the discussion, parameters used by the methods are given in Table 3.

2. Geometry-sorting methods

The geometry-sorting methods render transparent objects (usually composed of triangle meshes) by either sorting the objects

☆ This article was recommended for publication by E. Reinhard.

* Corresponding author. Tel.: +55 51 3308 6930; fax: +55 51 3308 7308.
E-mail addresses: joao.comba@gmail.com (J.L.D. Comba), rafael.torchelsen@gmail.com (R.P. Torchelsen), rbastos@nvidia.com (R. Bastos).

[1] Back-to-front blending equation [1]: $C'_{dst} = (1-\alpha)C_{dst} + \alpha C_{frag}$.

0097-8493/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2011.07.006

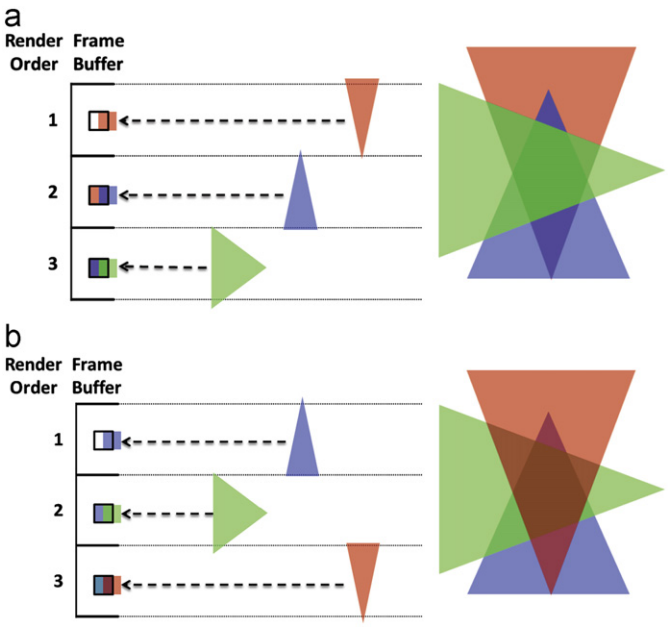

Fig. 1. Blending evaluation for green, blue, and red triangles (in viewing order). On the left, we illustrate the blending of fragments generated during the rasterization of each triangle. The arrow length represents the fragment depth. On the right, we show the resulting image after blending. Primitives processed in-depth order (a) produce different results compared to objects processed in out-of-depth order (b), whereas blending in-depth order produces the expected result. (a) Depth ordering: correct blending and (b) Out-of-order: incorrect blending. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 1
Numerical example of a pixel evaluated with the BTF order. Each line represents a fragment blending order (top to bottom).

Fragment	RGBα	Depth	$BTF_{blended\ result}$
Background	(1, 1, 1, 1)	∞	(1, 1, 1, 1)
1	(1, 0, 0, 0.4)	3	(1, 0.6, 0.6, 0.76)
2	(0, 0, 1, 0.4)	2	(0.6, 0.36, 0.76, 0.616)
3	(0, 1, 0, 0.4)	1	(0.36, 0.616, 0.456, 0.53)

Table 2
Numerical example of a pixel evaluated out-of-depth order with the BTF blending equation [1]. Each line represents a fragment and the blending order (top to bottom).

Fragment	RGBα	Depth	$BTF_{blended\ result}$
Background	(1, 1, 1, 1)	∞	(1, 1, 1, 1)
1	(0, 0, 1, 0.4)	2	(0.6, 0.6, 1, 0.76)
2	(0, 1, 0, 0.4)	1	(0.36, 0.76, 0.6, 0.616)
3	(1, 0, 0, 0.4)	3	(0.616, 0.456, 0.36, 0.53)

Table 3
Parameters used by different methods.

W	Screen width
H	Screen height
S	Number of samples for super sampling
m	Number of objects
n	Number of geometry primitives ($n \gg m$)
p	Pixel size: 12B (8B: RGB 8b per channel, 8b alpha, 4B depth value)
l	Number of transparent layers
d	Average number of transparent layers ($d < l$)
k	Buffer entries per pixel (in fragments)
f	Min(k, d)
s	Samples per pixel

or at a finer granularity, by sorting their primitives (i.e. triangles). Both approaches are simple to implement and can directly leverage the alpha-blending support on GPUs (graphics processing units). These methods sort objects or primitives before rasterization and are arguably the most widely used transparency technique in games.

Two situations can lead to image artifacts when using geometry-sorting methods: interpenetrating geometry and out-of-order arrivals. Interpenetrating geometry makes it difficult to properly sort objects and primitives, which leads to blending artifacts. Out-of-order arrival can arise when the technique sorts entire meshes instead of triangles, which can also lead to blending artifacts. Both situations can be properly handled by sorting at the fragment level.

2.1. Object sorting

This approach sorts objects as single entities, usually by the centroid of their meshes. It is prone to artifacts due to the out-of-order arrival of fragments, since object ordering is approximate and does not guarantee correct ordering at the primitive level.

Object sorting methods first sort all the m objects in $O(m \log m)$, followed by rasterization of object primitives using $O(n+WH)$ operations, and blending using $O(WHd)$ operations. The scene is rendered in one geometry pass, with no need for extra memory, apart from the color and depth buffers, which require $O(WHp)$ bytes.

2.2. Primitive sorting

Sorting geometry at the primitive level is the finest granularity that can be obtained in object space. It allows solving the out-of-order problem when no interpenetration occurs. One way to solve interpenetration cases is to split primitives, which might have a high computational cost and still generate z-fighting problems. Another option is to solve interpenetration analytically at even higher costs.

Primitive sorting requires sorting n primitives (triangles) in $O(n \log n)$ operations. The remaining steps and analysis are analogous to the object-sorting given above.

3. Fragment-sorting methods

Fragment-sorting methods compute transparency by z-sorting at fragment level before blending. Fig. 2 shows an example of out-of-order rendering of triangles, where fragments are sorted before blending. We further subdivide the fragment-sorting methods into two categories:

- *Buffer-based* methods store and sort the fragments before blending;
- *Depth peeling* methods extract depth order implicitly through a multi-pass rendering approach.

The main advantage of fragment-sorting methods is the quality of the image, often superior to all other methods. On the other hand, the computational cost and/or memory footprint, due to sorting, is considerably higher.

3.1. Buffer-based methods

Buffer-based methods use a buffer to store fragments while they are generated. After rasterization, a sorting step computes the correct blending ordering. The advantage of these methods is image quality, when compared to sorting-based methods like geometry-sorting, and performance, when compared to *depth peeling* methods,

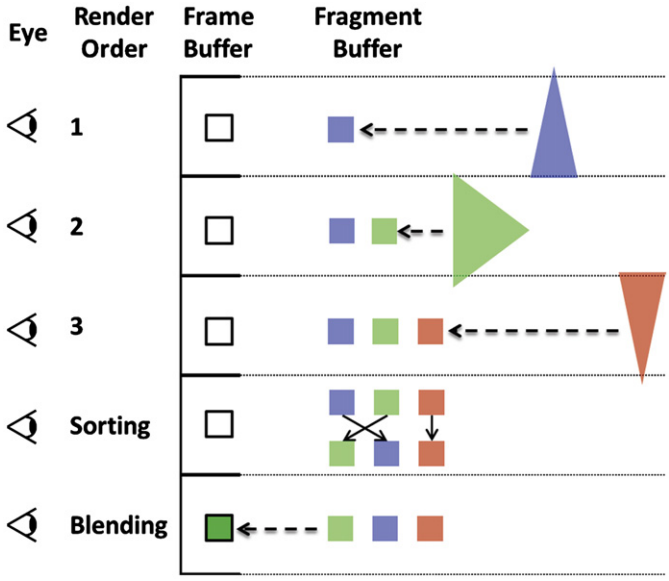

Fig. 2. Fragment re-ordering example. A buffer holds all out-of-order fragments. A post-rasterization phase sorts these fragments by depth. Finally, the fragments are correctly blended in-depth order.

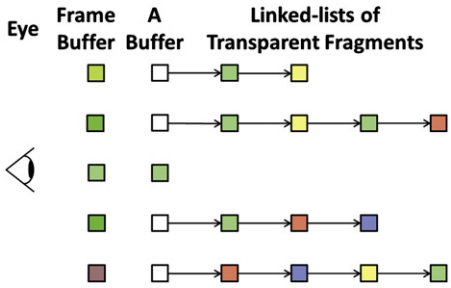

Fig. 3. A-buffer stores for each pixel with transparency either a color (for opaque pixels) or a linked list of fragments. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

at the expense of high memory consumption. Ideas for handling the storage issue in several methods are given below.

3.1.1. A-buffer

The A-buffer [3] (accumulation, area-averaged, and anti-aliased buffer) was the first method to address the order-independent transparency problem, but its original focus was a hidden surface algorithm with support for anti-aliasing. The algorithm first stores *all* fragments per pixel during rendering, followed by a sorting step to reorder fragments by depth. In a final step, fragments associated to each pixel are blended in depth-sorted order.

The A-buffer stores for each pixel of the final image either a color or a list of all fragments associated with that pixel, as shown in Fig. 3. Fragments are appended to the appropriate pixel list as they are generated during rasterization. After rasterization, the list of fragments of each pixel is sorted and blended in FTB (or BTF) order.

The high image quality, at the cost of performance, makes the A-buffer suitable for offline applications, such as animation movies and special effects. The method represents a robust solution for order-independent transparency, visibility, and anti-aliasing. However, it uses unbounded memory and requires random memory access imposed by the linked lists of fragments. Such limitations place constraints on the achievable performance of the method and are addressed by the techniques that follow.

The A-buffer first rasterizes all geometry primitives in $O(n+WH)$ operations, generating and storing *all* the $O(d)$ fragments per pixel. After rasterization, the fragment list of one pixel is sorted in $O(d \log d)$ operations, and traversed to blend fragments in $O(d)$. For the entire screen, it takes $O(WHd \log d)$ operations for sorting, and $O(WHd)$ operations for blending. The scene is rendered with one single geometry pass, with *all* fragments stored on a per pixel basis, taking $O(WHdp)$ bytes.

3.1.2. Z^3

Similar to the A-buffer, the Z^3 technique [4] uses a buffer for holding multiple fragments per pixel; those are also sorted before blending. However, instead of relying on unbounded memory, the Z^3 constrains the number of fragments stored per pixel. This constraint alleviates memory requirements, but it introduces image artifacts.

Instead of keeping all transparent fragments generated by rasterization for each pixel, the Z^3 algorithm holds only k fragments per pixel. When rasterization generates more than k fragments for a given pixel, an overflow occurs. Upon overflow, once the pixel fragments in the buffer are sorted, the nearest fragments are blended and stored back in the buffer, allowing the algorithm to continue (without requiring more memory). The choice of which fragments to blend is guided by the number of samples covered by each fragment. Fragments with fewer samples have less contribution in the final color, and therefore their blending when overflow occurs is bound to generate less error. Once rasterization finishes and overflows are properly handled, fragments left in the buffer are blended in FTB order.

Besides carefully choosing the fragments to blend (upon overflow), the Z^3 algorithm tracks information about the depth of each fragment, which includes three depth values and a set of coverage samples. The depth values—i.e., a central z and two slopes (in x and y directions)—enable a better treatment of interpenetrating fragments by reconstructing good approximations of the depth in each sample. Fig. 4 illustrates the Z^3 data structure.

The Z^3 technique has lower memory cost than the A-buffer, since it uses a fixed amount of memory to evaluate pixel colors, compared to the unbounded memory of the A-buffer. In addition, Z^3 improves the precision of the blending by storing more depth information about the fragment samples. The main drawback is on image quality, due to the incorrect blending when overflows occur, and on performance, because it needs atomic operations to test and write into the buffer.

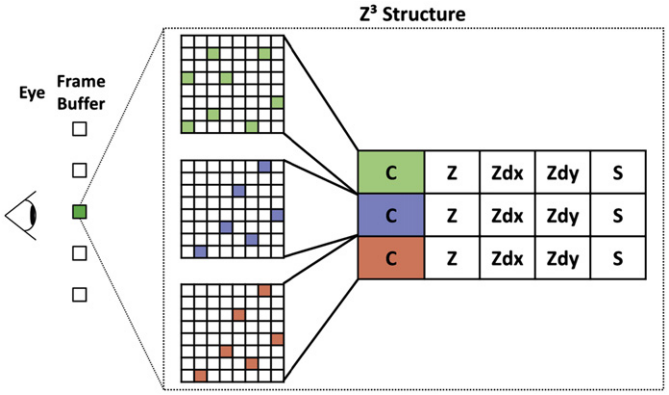

Fig. 4. Z^3 structure stores information about fragments: color samples (C), central depth (z), slopes in x and y directions (Z_{dx} and Z_{dy}), and a stencil bit (S). Figure adapted from [4]. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Z^3 computes transparency in one geometry pass with $O(n+WH)$ raster operations, storing $O(WHd)$ fragments into k slots per pixel, which takes $O(WHkp)$ bytes. While fragments are stored, their depth order is updated at a cost of $O(WHd \log f)$ operations, since during overflow one fragment may be compared against f previously stored values to find the two nearest to be blended. Z^3 processes all fragments, and thus blending takes $O(WHd)$ operations.

3.1.3. FIFO buffers

In this section we review two FIFO buffer methods (F-buffer and R-buffer) that use intermediate buffers to store fragment information produced by rasterization. Both methods are inspired on the A-buffer, but use different approaches to store the incoming fragments. A similar architectural proposal is presented in Amor et al. [5].

The F-buffer (fragment-stream-buffer) [6] stores *all* incoming fragments in a FIFO buffer during rasterization. Once rasterization ends, the fragments in the F-buffer are processed in a multi-pass fashion, which peels one transparent layer of the scene at each pass by testing against a depth map. Rejected fragments are sent to another F-buffer, which serves as input in the next pass. Selected fragments compose the transparent layer and are blended to an accumulation buffer. When the F-buffer is empty, the accumulated color buffer is written into the framebuffer.

The R-buffer [7] is a modified A-buffer that stores all fragments of a scene sequentially in a FIFO buffer, instead of using a linked list approach. Order-independent transparency is computed in multiple passes over the stored fragments, similar to the F-buffer, but using two R-Buffers and an extra Z-buffer. In the first pass, the closest opaque fragment is captured in the framebuffer, and the un-occluded transparent fragments are stored in the R-buffer.

Multiple passes process the fragments captured in the R-buffer. In each pass, the input R-buffer is traversed to identify the farthest transparent fragment of each pixel to blend into the framebuffer, while writing the remaining fragments into an output R-buffer. The input and output R-buffers switch roles at each pass, and computation proceeds until the output R-buffer is empty (Fig. 5).

The methods described in this section are similar. Both methods use a single geometry pass to capture all fragments of a scene, followed by multiple sorting and blending passes over the fragments stored in FIFO buffers. The main difference between them is that the F-buffer does not allow writing into the framebuffer until the processing is done, whereas the R-buffer operates directly in the framebuffer.

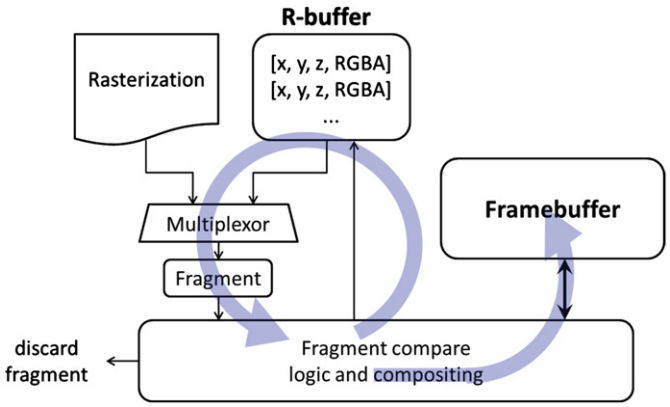

Fig. 5. R-buffer interface, with recirculating pipeline, can perform OIT in one single geometry pass, with many passes over the stored fragments. The multiplexor selects the input from the rasterizer or the R-buffer. Fragments are tested and blended, or sent back (recirculated) to the R-buffer. Figure adapted from [7].

Like the A-buffer [3], the main drawback of both methods is the unbounded memory usage, since these methods need to store all incoming fragments, which can exhaust memory. Differently from the original A-buffer, these approaches do not use linked lists, which is emulated by storing additional information per fragment and performing multiple passes over the stored data. In a multi-thread system, they have the advantage of not suffering from read-after-write hazards. This is caused by many threads trying to store fragments into the same pixel list, as fragment storage is not directly associated to each individual pixel.

FIFO techniques handle transparency in a single geometry pass with $O(n+WH)$ raster operations. The generated fragments are stored in the FIFO buffer in $O(WHd)$ operations taking $O(WHdp)$ bytes. In a post-rasterization phase, sorting is performed in $O(WHd^2)$ operations due to the disassociation of fragments to their pixel location. During sorting, fragments are blended in $O(WHd)$ operations.

3.1.4. Stencil-routed A-buffer

The A-buffer variation given by Myers and Bavoil [8] uses the stencil buffer to route fragments into a multi-sample buffer. It leverages multi-sample anti-aliasing (MSAA) to capture multiple fragments per pixel, which are later processed by a pixel shader for sorting and blending.

The scene is first rasterized to capture up to k fragments per pixel, where k is the maximum number of samples available in the MSAA implementation. Each incoming fragment is routed to a sample of the MSAA buffer. If there are more than k fragments per pixel, overflow occurs and an additional geometry pass is needed. Fragment routing uses a stencil mask that stores an incoming fragment in the next free MSAA sample, building a vector of fragments per pixel. Once all fragments are captured, a full-screen quadrilateral is rendered with a pixel shader that reads all fragments of a given pixel; it sorts and blends them accordingly. Fig. 6 illustrates the stencil-routed algorithm for a single pixel.

Since the stencil-routed A-buffer uses hardware resources, the use of MSAA and stencil buffer is not supported by this method. In terms of performance, the stencil-routed A-buffer is up to k times faster than the basic *depth peeling* method to be described in Section 3.2. It stores up to k fragments per pixel on each geometry pass, using $O(n+WH)$ raster operations per pass. When the number of transparent fragments exceeds the k value, the algorithm needs $O(l/k)$ geometry passes to properly evaluate

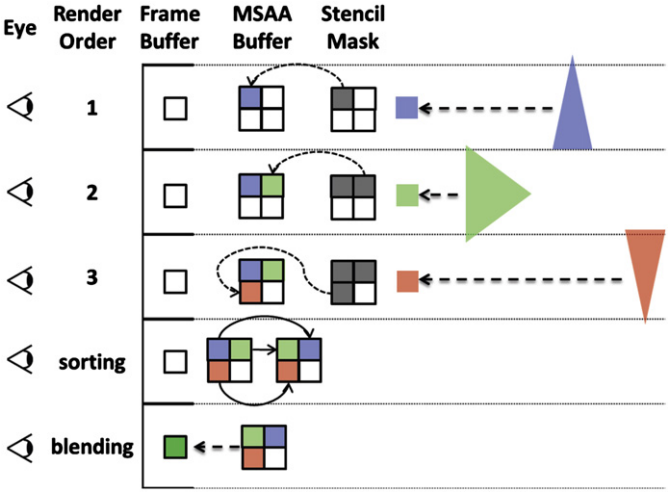

Fig. 6. Stencil-routed evaluation for one pixel. The stencil mask routes fragments into the MSAA buffer. The fragments are then sorted and blended accordingly in a post-rasterization stage.

transparency, with l limited by the 8-bit word of the stencil buffer. Writing of all fragments takes O($WHkp$) bytes. A subsequent sorting of all lists (in chunks of k fragments per pass) takes O($d \log k$) operations per pixel, followed by O(WHd) blending operations.

3.1.5. FreePipe

Liu et al. [9] proposed *FreePipe*, an implementation of the entire graphics pipeline in CUDA. FreePipe allowed two variations of the A-buffer to be explored, based on a fixed-size vector of k fragments per pixel (Fig. 7).

The first approach tests each incoming fragment against the corresponding buffer entry (using atomic operations available in graphics hardware) to find its corresponding z-sorted location in the list. If keeping the correct sorting order is needed, fragments already stored in the buffer are moved to other positions. This data movement increases memory traffic, but results in the sorted list of fragments per pixel—i.e., no sorting is needed, after the rasterization pass is finished. It can also lead to blocking threads that compete to store fragments into the same pixel list, because the store operation must be atomic. At the end of this process, the fragments are ready for blending.

The second approach uses a counter index per pixel, which indicates the next available position in the pixel list to store the incoming fragment. Only this counter is updated atomically. The fragments are stored in order of arrival rather than in sorted order, avoiding more atomic operations (and their hazards). In a post-processing phase, the fragments are sorted and then blended.

The limitation of both approaches is the large memory requirements because of the fixed-size buffer, as well as loss of fragments when overflow occurs. The first approach can detect pixel saturation and early terminate the processing of additional fragments. The second approach, similar to the A-buffer, captures all fragments per pixel, before they are sorted. Both approaches allocate the same number of entries per pixel, regardless of whether or not the pixel has transparent fragments, leading to waste of memory. Atomic operations degrade performance, since they serialize threads associated to the same pixel locations.

Both approaches compute transparency in a single geometry pass with O($n+WH$) raster operations, using O($WHkp$) bytes and performing blending in O(WHk) operations. The first one sorts fragments at generation time, comparing against previously stored values. This requires O(df) operations, and does not need sorting before blending. The second one stores fragments upon generation and sorts them in O($f \log f$) in a separate step after rasterization.

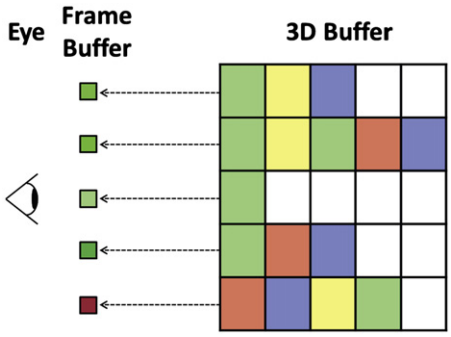

Fig. 7. Side view of the buffer used in the FreePipe approach, which has a fixed number of slots per pixel to store fragments.

3.1.6. Linked lists

The creation and the update of linked lists entirely on GPUs, using atomic operations in shader programs, is explored in the proposal of Yang et al. [10]. Since current GPUs do not allow dynamic memory allocation, two buffers are used to emulate the A-buffer linked lists. The first buffer holds, for each pixel, an index to the last rasterized fragment. This index points to a location in the second buffer, which keeps all fragments generated for all pixels. Upon the arrival of a new fragment, the index is updated to point to the new entry, which receives the previous index. Thus, each entry points to the previously rasterized fragment, forming a linked list. Fig. 8 shows the first buffer, called *head pointer*, the second buffer, *node buffer*, and their states after the rasterization of three triangles, where two transparent fragments are generated for a pixel.

Since the node buffer is shared, a single global counter is used to index the next available position where a thread can store an incoming fragment. This shared counter needs to be updated atomically, which represents a bottleneck, since all threads trying to store a new fragment will be blocked here. The random memory accesses of the linked list also degrades performance. The main advantages of this method include the fact that no memory is allocated for pixels or layers that are not covered (the node buffer size may be controlled to fit the number of fragments generated), and the way it leverages the graphics pipeline already optimized in hardware.

Linked lists is implemented similar to the A-buffer approach. In a single geometry pass, using O($n+WH$) raster operations, fragments are stored in O($WHdp$) bytes. After rasterization, fragment lists are sorted in O($WHd \log d$) and blended in O(WHd) operations.

3.2. Depth peeling-based methods

Depth peeling methods extract layers of visibility from a graphical model using multiple rendering passes. Layers are captured in-depth order, which eliminates the need to sort the resulting fragments. Transparency is computed by blending transparent layers in FTB

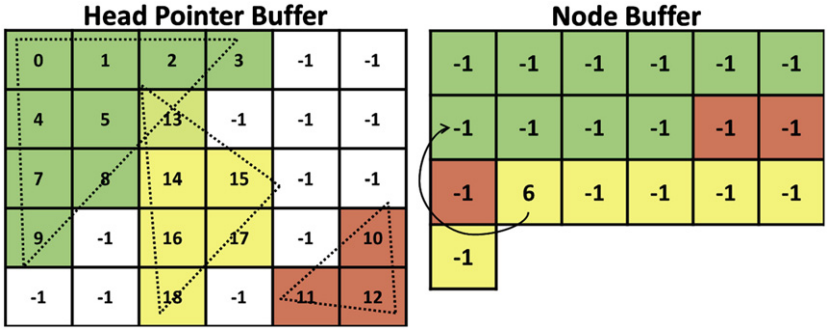

Fig. 8. Head pointer buffer and node buffer states after inserting the green, red, and yellow triangles, respectively. The head pointer buffer points to the last fragment inserted for each pixel in the node buffer. The last fragment inserted in the node buffer points to the previous fragment. Figure adapted from [10]. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

or BTF order. In this section, we summarize three depth peeling variations.

3.2.1. Virtual pixel maps

The virtual pixel maps (VPM) introduced by Mammen [11] proposed the first depth peeling technique. Transparency is computed in multiple geometry passes that extract one depth layer in each pass. The first geometry pass renders all opaque objects into an opaque pixel map, which stores a depth and a color value per pixel. At each subsequent geometry pass, the un-occluded fragments (with respect to the currently captured layer) are tested to find the nearest to the previously captured layer. After each pass, selected fragments are blended with the color already stored, and corresponding depth values updated. The process ends when no more fragments are blended, or when the number of passes exceeds a user-defined threshold. Fig. 9 shows the evaluation for a pixel using VPM.

VPM does not have large memory requirements, since it needs only one extra z-buffer and one extra framebuffer to peel the occluded fragments and accumulate the nearest fragments colors. Memory requirements double the color and depth buffer O(WHp) bytes. The geometry passes can be costly for scenes with high depth complexity. For each transparent layer, VPM needs a geometry pass, totalizing O($l(n+WH)$) raster operations for all layers, at the same time that layers are blended in O(WHd).

3.2.2. Interactive order-independent transparency

The iOIT technique presented by Everitt [12] uses a depth peeling approach that leverages the implementation in graphics hardware of the shadow-mapping algorithm [13]. It starts by using the z-buffer to extract the transparent layer nearest to the eye. The following passes use the previous z-buffer values to find the subsequent nearest layers by eliminating all fragments that are in front or match the current layer. Remaining fragments are depth sorted to generate the frontmost fragment behind the current layer—i.e. the next depth layer. After each geometry pass, the nth layer is ready to be blended in the color buffer.

An approach to peel two depth layers in each geometry pass is proposed by Bavoil and Myers [14]. The first and last visible layers are computed in each geometry pass (an example is given in Fig. 10). Since it processes depth peeling simultaneously in FTB and BTF fashion, it can be twice faster than the iOIT depth peeling.

The advantage of iOIT techniques over virtual pixel maps is the FTB approach, which enables early termination when pixel opacity saturates. This means that iOIT needs up to l geometry passes to cover all layers. Similar to VPM, iOIT does not have large memory requirements, but it also incurs in repeated geometry passes. The remaining steps and analysis are analogous to VPM described above.

3.2.3. Bucket sort

Two GPU-based bucket sort (BS) implementations were proposed by Liu et al. [15] to peel multiple depth layers per geometry pass. It allows peeling up to 32 layers in each pass using multiple render targets to store buckets.

In the first implementation, the depth range is uniformly subdivided and each subdivision is mapped to one bucket. Incoming fragments are mapped to the bucket corresponding to its depth value. A collision occurs when more than one fragment is mapped to the same bucket, which may be alleviated using a

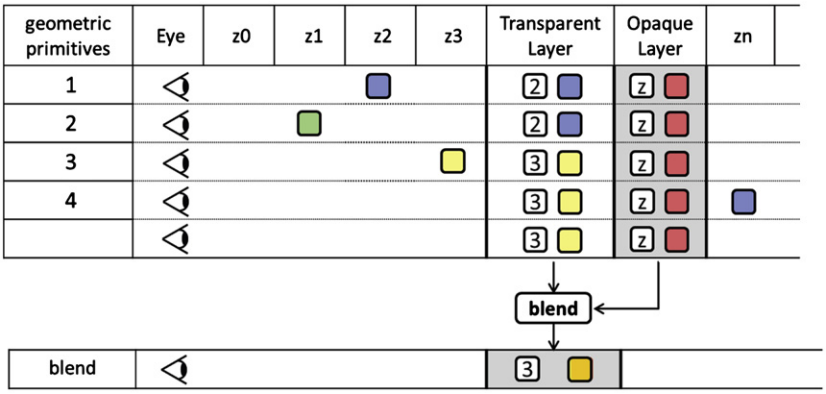

Fig. 9. VPM transparency evaluation for one pixel. In each pass, the visible fragment nearest to the opaque layer is blended to the opaque fragment and its depth is updated. Figure adapted from [11].

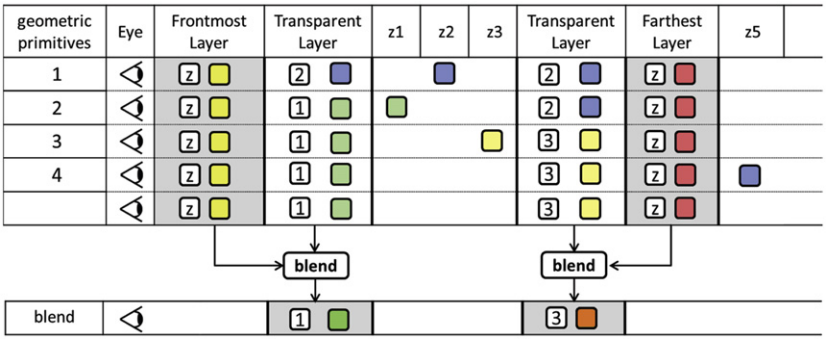

Fig. 10. Dual depth peeling transparency evaluation for a pixel. In each geometry pass, the visible fragment farthest from the eye is composed into the farthest layer and its depth is updated. In the same pass, the visible fragment nearest to the eye is blended into the frontmost layer and its depth is updated. This example has the same configuration used in Fig. 9. Figure adapted from [14].

two-pass approach over the geometry. The first geometry pass renders the scene bounding box into bucket intervals to obtain a depth estimate for each pixel. The depth range is subdivided according to fragment occupation into fixed intervals. In the second pass, each incoming fragment is mapped into the bucket corresponding to its depth interval. See Liu et al. [15] for more details and the mapping equation.

In the fixed approach, some buckets might remain empty while others are overloaded, according to the depth complexity distribution of the scene. Instead of finding the ideal subdivision that maps a fragment into each bucket, the second implementation uses an adaptive approach. To create intervals that better fit the depth density distribution, an equalized-depth histogram of the geometry distribution is computed in a first geometry pass (Fig. 11). This histogram is used in the second pass to map incoming fragments into the correct depth buckets (Fig. 12).

The bucket sort is suitable for scenes with uniform depth distributions. For scenes with a high concentration of fragments in a small depth interval, or for scenes with high depth ranges, it may generate artifacts due to the irregular mapping of fragments into buckets. The advantage is the low cost in mapping fragments ready to be blended.

The bucket sort technique needs two geometry passes. One to calculate the equalized histogram, and another to render the scene; both with $O(n+WH)$ raster operations. The second geometry pass takes $O(WHd)$ operations to route fragments into $O(WHkp)$ bytes. Since fragment lists are generated in order, there is no need for sorting, and lists are blended in $O(WHf)$ operations.

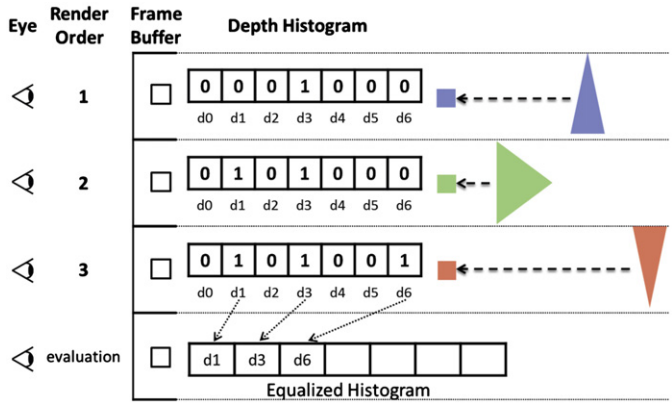

Fig. 11. Equalized histogram evaluation. The depth histogram is a binary vector, with each entry corresponding to one depth value. A value 0 means that there is no fragment mapped to that entry, whereas a value 1 means that at least one fragment is mapped to that depth range. For each incoming fragment, its z value is used to set a bit indicating its presence in the depth histogram. At the end of the geometry pass, the equalized histogram is computed with the depth values that are set to 1 in the depth histogram.

4. Hybrid methods

The methods discussed so far compute transparency by sorting geometry primitives (before rasterization) or raster fragments. In such methods only one of these two approaches is explored. On the other hand, hybrid methods compute transparency using both approaches. In this section, we review two hybrid methods.

4.1. Image-space queries

The image-space queries (ISQ) algorithm [16] describes an approach for ordering non-interpenetrating geometry models. ISQ is similar to the primitive-sorting techniques described in Section 2.2, since it computes a BTF order of primitives before rasterization. However, instead of sorting primitives according to the distance to the eye in object space, ISQ uses occlusion queries in image-space to compute a correct BTF ordering among primitives. Cycles in the visibility ordering can be generated, but they are identified and resolved using occlusion queries over the triangles of objects involved in cycles.

The primitive rendering order is computed by sorting pairs of primitives using their normalized depth value. The comparison uses the occlusion query capability of GPUs by simply rendering two primitives, and evaluating the smallest depth to define the triangle closer to the viewpoint. Sorting all pairs of primitives in this fashion gives a visibility graph to render in BTF ordering.

ISQ rasterizes n primitives, by pairs, to perform visibility comparison, which takes $O(n \log n)$ rendering operations. Once the primitive list is sorted, it is rasterized in $O(n+WH)$ operations. Since most objects are already sorted due to temporal coherence between frames, the algorithm uses the list from previous frames as input for the next frame sorting, therefore execution is expected to run in linear complexity. Memory requirements are minimal, only the standard color and rendering buffers.

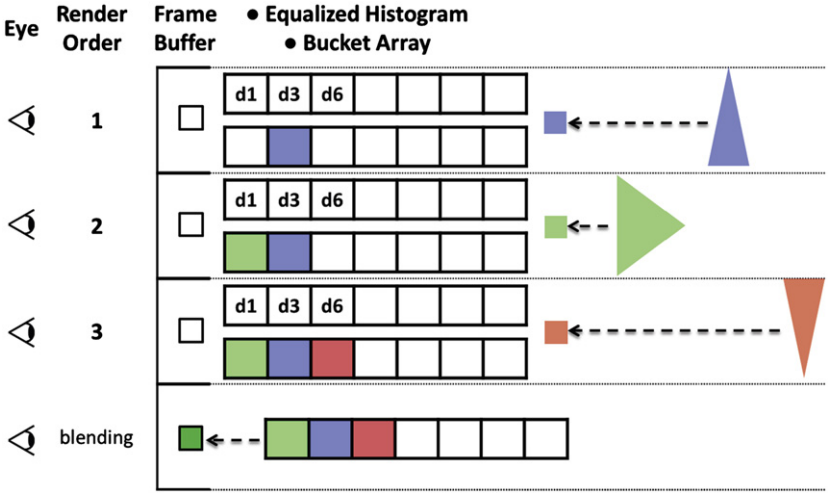

Fig. 12. Color evaluation for a pixel using BS. The incoming fragments are mapped by the equalized-depth histogram into the correct buckets by testing their depth values. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

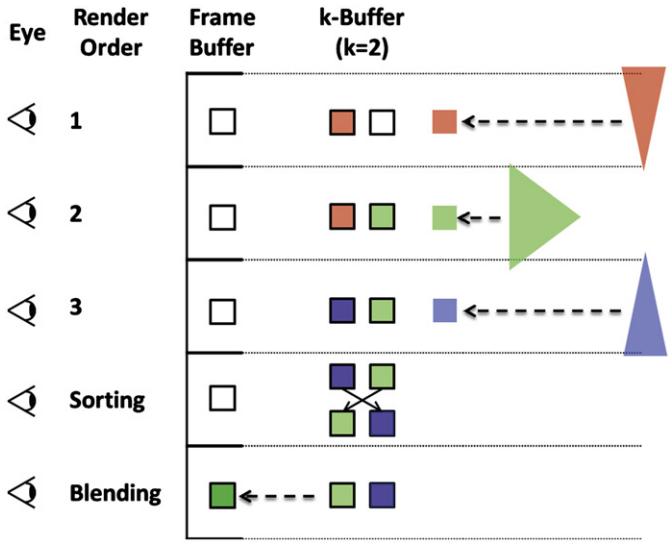

Fig. 13. Color evaluation for one pixel using a k-buffer with two entries per pixel. While the buffer has free entries, fragments are stored in *incoming* order. In case of overflow, the incoming fragment is tested and combined with the z-nearest entry. In a post-rasterization phase, remaining entries are sorted and blended. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4.2. The k-Buffer

A technique to render unstructured grids was described by Callahan et al. [17] and later generalized by Bavoil et al. [18] to handle multi-fragment effects (including transparency). The core of the technique is the k-buffer, a fixed-size fragment buffer that holds up to k fragments per pixel. The technique first uses an approximate sorting in object space, which is not guaranteed to produce the correct ordering, but it allows for fragments to be generated during rasterization in a nearly sorted fashion. This important nearly sorted property allows the sorting to be concluded in image-space with a k-buffer, which only needs to have as many entries as necessary to fix the ordering of samples. Fragments are composited when the k-buffer becomes full (Fig. 13), to make room for other incoming samples, and after all fragments are generated (which requires the k-buffer entries to be flushed).

The k-buffer handles interpenetrating geometry, since fragment-sorting is involved. The final image quality depends on how well object-sorting reduces the number of out-of-order fragments. In situations resulting in poor fragment-sorting, the number of entries in the k-buffer might be smaller than necessary, which might introduce artifacts due to out-of-order blending. Also, in its proposition, the algorithm was prone to artifacts caused by read-modify-write hazards during k-buffer updates. This can be overcome today with atomic operations, at the expense of increased processing time due to the cost incurred by such operations.

Sorting is done in object space with an external radix sort in $O(m)$ operations. Geometry is rasterized in $O(n+WH)$ and fragments are stored into k slots, with possible blending (in overflow cases), costing $O(WHd \log f)$ operations and $O(WHkp)$ bytes. Final sorting of k elements takes $O(WHf \log f)$ and $O(WHf)$ blending operations.

5. Depth-sorting-independent methods

The methods described in this section do not require sorting prior to blending fragments. Instead, they use special compositing equations to blend fragments in incoming (unsorted) order. This approach can lead to incorrect transparency results, when the fragments to be blended have very distinct colors and transparencies.

The weighted sum method [19] blends incoming fragments in arrival order using Eq. (1):

$$C_{dst} = \sum(\alpha_{src} \times C_{src}) + C_{bg} \times \left(1 - \sum \alpha_{src}\right) \quad (1)$$

where C_{dst} is the resulting color, α_{src} and C_{src} are, respectively, the opacity and the color of the incoming fragments, and C_{bg} is the background contribution. For low alpha values, this approximation is acceptable, but increasing alpha values might lead to overly dark or overly bright results.

The weighted average [14] also employs a single-pass blending approach using Eqs. (2)–(4):

$$C = \frac{(\sum(RGB)\alpha)}{(\sum \alpha)} \quad (2)$$

$$\alpha_{dst} = \sum \left(\frac{\alpha}{N}\right) \quad (3)$$

$$C_{dst} = \frac{C\alpha \times ((1-(1-\alpha)^N))}{\alpha + C_{bg} \times (1-\alpha)^N} \quad (4)$$

where RGB represents the three color channels, α is the opacity, N is the number of fragments for the evaluated pixel, and C_{bg} represents the background contribution.

Sorting-independent methods are fast, single-pass, and do not need additional buffers, since fragments are merged upon generation. They are suitable for particle rendering due to the high quantity of particles commonly simulated, and the acceptable error tolerance for this effect. Both methods compute transparency in one geometry pass $O(n+WH)$, which generates $O(WHd)$ fragments blended using $O(WHd)$ operations. In terms of memory, the basic color and depth buffers $O(WHp)$ bytes are sufficient.

6. Probabilistic methods

This section describes techniques that approximate transparency using probabilistic approaches. The main advantage of the stochastic and probabilistic methods is the low processing cost, since pixels are evaluated in a fixed number of geometry passes and fragments are not sorted. The main drawback of these methods is the noise associated to using random sampling. One approach to reduce noise is to increase the number of samples, which increases memory consumption or processing time.

6.1. Stochastic transparency

Stochastic transparency [20] is a stochastic solution for transparency which does not require sorting. The algorithm treats transparency as the number of samples covered by the fragment color. For example, an $\alpha = 0.5$ means, roughly, 50% of the pixel samples being covered by a fragment.

This method implements screen-door transparency [21] with a random sub-pixel stipple pattern. The original screen-door technique uses a bitmask to select pixels that are not colored, producing holes in the object rasterization proportional to its transparency. The stochastic algorithm uses the hardware MSAA to apply screen-door transparency at samples instead of pixels. Samples are later blended to define the pixel colors, and an example is given in Fig. 14.

A pixel color is defined by s samples, each with α probability of receiving the color from an incoming fragment, where α is the fragment opacity. On average, the number of samples updated with the fragment color is $r = s \times \alpha$. The first pass in the algorithm renders opaque geometry and the background, while transparent

objects are handled in subsequent passes. The second rendering pass uses shaders to accumulate the opacity at each pixel, and depths at each sample. These values are used to approximate fragment visibility, in a third rendering pass, without sorting.

The main drawback of stochastic transparency is the noise introduced by the stochastic sample selection scheme, which can lead to flickering in dynamic scenes. The noise can be attenuated by increasing the number of samples. The advantage is the flexibility to improve quality or performance by choosing the number of samples to be used.

This approach uses three geometry passes with $O(n+WHs)$ operations to compute transparency, generating $O(WHd)$ fragments. For each fragment, it tests s samples that may receive the incoming color from the fragment in $O(WHs)$ operations. After rasterization, samples are merged in $O(WHs)$ operations. Memory consumption is given by the samples used per pixel: $O(WHsp)$ bytes.

6.2. Silhouette-opaque transparency

The silhouette-opaque method (SOT) [22] is a screen-door approach defined in image-space (similar to [20]). Each triangle in the original scene is discretized in smaller primitives. The mesh discretization (screen-door construction) computes a sample distribution per triangle, according to its area in screen-space. To compute this area, the closest point to the viewpoint in the triangle is identified, and the triangle is rotated around this point until its normal becomes parallel to the viewing direction and facing the viewer. The triangle is rasterized to a temporary super-sampled framebuffer using current camera parameters, and samples are defined by the number of pixels covered by the triangle. Thus, sampling does not depend on triangle orientation.

After a distribution is defined, some samples are removed based on the α value of the triangle so that the number of samples is proportional to the triangle opacity. Fig. 15 illustrates the entire process, which has to be repeated for every frame. The drawback of SOT is the limitation to scene size imposed by the number of primitive samples to be generated, as well as the noise incurred by the probabilistic sampling. The advantage is the fast rendering due to the single geometry pass and lack of sorting.

The k point samples from each one of the n triangle primitives are rasterized in $O(kn+WHS)$ operations. This generates $O(WHS)$ fragments in a S-super-sampled image, which is resized by a box filter in $O(WHS)$ operations. Memory consumption is given by the samples taken per pixel: $O(WHSp)$ Bytes.

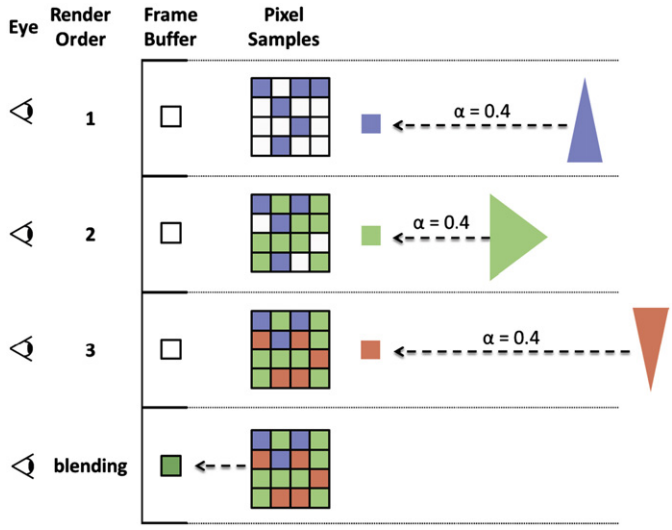

Fig. 14. Stochastic evaluation of one pixel color. Fragment coverage uses the fragment α as estimate of fragment visibility. A random distribution is applied to avoid using the same bitmask. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

7. Summary comparison

In this section we present a comparison among methods with respect to image quality, performance, and memory consumption. To allow for direct comparison, we first summarize in Table 4 the complexity analysis discussed for each method in the previous sections.

We continue the comparison among methods in this section by looking at the type of rendering artifacts the methods can generate, an important aspect to be considered when selecting which method to use. We follow this discussion with an estimate of performance using a hypothetical scene that pushes the limits in terms of transparency computation, and present an illustrative chart that compares performance and quality of all methods.

7.1. Correct transparency rendering

The correct computation of transparency discussed in the methods that we survey is associated to processing *all* transparency layers in *depth-sorted order* with respect to the viewer. Out-of-order blending of fragments might generate incorrect results, as explained in Section 1.

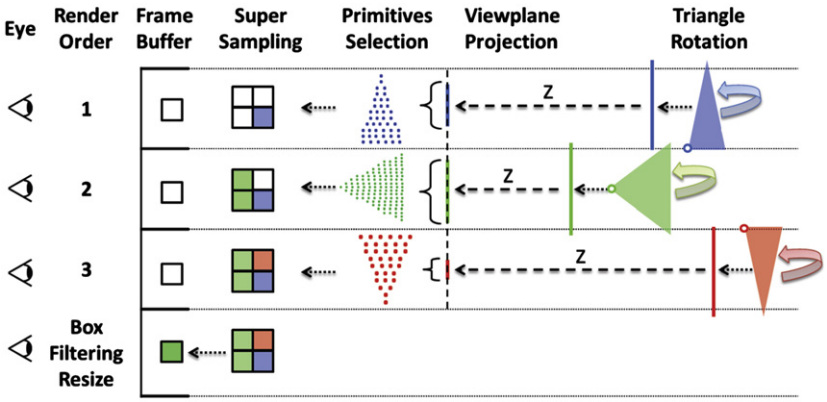

Fig. 15. SOT color evaluation for one pixel: The triangle is rotated to be parallel to the viewing plane and projected from the distance of the nearest point of the triangle. The projected area is used to estimate the primitives (geometric points) size and quantity, which approximates the amount of light passing through the object. The triangle is renderized into a super-sampled texture, which is resized by a filtering box to compose the pixel color. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 4
Complexity comparison. Columns: methods taxonomy; memory consumption (in bytes); geometry passes (count the required number of traversals over the geometry); pre-sorting (sort complexity in object space); raster operations (count operations to rasterize geometry primitives); fragment-sorting (sort complexity in image-space); and blending (counts blending operations to combine colors). N/A = not applicable.

Category	Technique	Memory consumption	Computational cost				
			Geometry passes	Pre sorting	Raster operations	Fragment sorting	Blending
Geometry	Object [23]	$O(WHp)$	1	$O(m \log m)$	$O(n+WH)$	N/A	$O(WHd)$
	Primitive [23]	$O(WHp)$	1	$O(n \log n)$	$O(n+WH)$	N/A	$O(WHd)$
Fragment							
Buffer based	A-buffer [3]	$O(WHdp)$	1	N/A	$O(n+WH)$	$O(WHd \log d)$	$O(WHd)$
	Z^3 [4]	$O(WHkp)$	1	N/A	$O(n+WH)$	$O(WHd \log f)$	$O(WHd)$
	FIFO [6,7]	$O(WHdp)$	1	N/A	$O(n+WH)$	$O(WHd^2)$	$O(WHd)$
	Stencil R. [8]	$O(WHkp)$	$\lceil l/k \rceil$	N/A	$O(n+WH)$	$O(WHd \log k)$	$O(WHd)$
	FreePipe [9]	$O(WHkp)$	1	N/A	$O(n+WH)$	$O(WHf \log f)$	$O(WHf)$
	Linked lists [10]	$O(WHdp)$	1	N/A	$O(n+WH)$	$O(WHd \log d)$	$O(WHd)$
Depth peeling	VPM [11]	$O(WHp)$	l	N/A	$O(n+WH)$	*implicit*	$O(WHd)$
	iOIT [12]	$O(WHp)$	$\leq l$	N/A	$O(n+WH)$	*implicit*	$O(WHd)$
	Bucket Sort [15]	$O(WHkp)$	2	N/A	$O(n+WH)$	$O(WHd)$	$O(WHk)$
Hybrid	ISQ [16]	$O(WHp)$	1	$O(n \log n)$	$O(n+WH)$	N/A	$O(WHd)$
	k-Buffer [17,18]	$O(WHkp)$	1	$O(m)$	$O(n+WH)$	$O(WHd \log f)$	$O(WHf)$
Depth-sorting independent	W. sum [19]	$O(WHp)$	1	N/A	$O(n+WH)$	N/A	$O(WHd)$
	W. average [14]	$O(WHp)$	1	N/A	$O(n+WH)$	N/A	$O(WHd)$
Probabilistic	Stochastic [20]	$O(WHsp)$	3	N/A	$O(n+WHs)$	N/A	$O(WHs)$
	SOT [22]	$O(WHSp)$	1	N/A	$O(kn+WHS)$	N/A	$O(WHS)$

The algorithms that can perform correct transparency computations (given enough time and memory) are: A-buffer, FIFO, linked lists, VPM and iOIT. A-buffer and linked lists are the fastest in this set, since they only need one geometry pass to generate and store the transparent fragments for each pixel before blending. On the other hand, FIFO solutions do not associate the fragments to their pixels, requiring more passes over the fragments to properly sort them. This has a higher computation cost compared to A-buffer and linked lists. The disadvantage of these algorithms (A-buffer, linked list and FIFO) is the amount of memory required to store all fragments before processing them (sorting and blending).

Memory is not a limitation for VPM and iOIT, since they accumulate transparent layers in multiple passes over the geometry, at the expense of a high computational cost. The iOIT has the advantage of processing layers in FTB order, which allows for early termination when pixels saturate, which is not possible in VPM due to its BTF processing.

Correct transparency is costly in both memory and computation aspects. It is appropriate for offline rendering, where image quality surpasses computational cost.

7.2. Artifacts in transparency rendering

Rendering artifacts appear when pixels are colored incorrectly, usually because of out-of-order arrival of fragments, overflow, or insufficient sampling. Artifacts can range from plausible results to completely incorrect pixel colors. Depending on the application, artifacts may be tolerable, especially when comparing the performance gain against the methods that guarantee correct results. We discuss in more details the types of artifacts that can arise.

7.2.1. Artifacts due to out-of-order arrival

Methods that compute transparency, by blending fragments in incoming order, require geometry-sorting before rendering to order fragments by depth. Causes for out-of-order arrival are approximate sorting and interpenetration of primitives, leading to pixels with incorrect colors and possible flickering. ISQ, object and primitive-sorting, weighted sum and weighted average are prone to these artifacts.

In object-sorting algorithms, the low computational cost is due to the low granularity of the sorting, which leads to triangles possibly being rasterized out-of order. Increasing the granularity, by sorting at the triangle level, reduces the out-of-order of fragments, but also increases the computational cost. However, interpenetration cases can still occur. Both problems appear in ISQ and require handling overlapping objects, leading to performance loss.

Scenes with few and simple transparent geometry can take advantage of the high performance and low memory consumption of these algorithms. When the scene complexity grows and interpenetrations appear, the quality of the resulting image can decrease rapidly. A special case is when the transparency algorithm does not require any kind of sorting, like in weighted sum and weighted average. However, correct results can only be obtained for a few restrictive cases. For the general case, blending fragments out-of order generates incorrect results, usually presenting itself as overly light or overly dark regions.

Blending fragments in incoming order is the fastest way to perform transparency. However, due to the number of artifacts, it is only recommended for high performance-dependent applications, such as particle simulations.

7.2.2. Artifacts due to overflow

Overflow occurs when the memory reserved to store partial information is insufficient. This happens in buffer-dependent algorithms when the number of generated fragments is greater than expected, resulting in different problems. If the farthest fragments are lost, or blended in out-of-order, the errors may be minimal, depending on their contribution and the depth complexity of the scene. If this happens to the nearest fragments, artifacts may be severe. Z^3, k-buffer, bucket sort, FreePipe and stencil-routed algorithms may present this kind of artifact.

Z^3, k-buffer and bucket sort do not discard fragments, but might merge fragments in incorrect depth order. Z^3 stores additional depth information to alleviate the incorrect blending, which still might not be enough to handle the case in which an incoming fragment is distant from the previous fragments. Bucket sort cannot guarantee that more than one fragment does not fall in the same bucket, which might generate out-of-order blending. K-Buffer uses prior sorting in object space to generate fragments

in nearly sorted order, which allows the use of a smaller k-buffer. However, there is no guarantee on which size to use.

Such artifacts can generate incorrect blended colors, which may generate flickering when the camera moves. Z^3 and k-buffer can be parameterized to store more fragments, thus increasing computational cost and memory consumption. Bucket sort is limited by the number of render targets, which makes it hard to handle complex scenes with irregular depth distribution. FreePipe and stencil-routed discard fragments when overflow occurs. This can cause severe artifacts if the lost fragments are the frontmost ones. The stencil-routed limitation is the 8-bit word of the stencil buffer, unable to address more than 2^8 layers, which might lead to loss of fragments. FreePipe also can be parameterized to store more fragments to address these problems.

Scenes with few transparent layers can be handled efficiently by these algorithms. However, for scenes with irregular distribution of transparent layers across, fixed-size buffers might lead to either memory waste or overflow.

7.2.3. Artifacts due to insufficient sampling

Transparency computation can be formulated as a sampling problem. In this approach, insufficient sampling might lead to incorrect colors and aliasing, while random sampling might lead to flickering.

Both SOT and Stochastic algorithms have these artifacts. SOT is designed to render opaque silhouettes of transparent objects by simulating light passing through more material. Probabilistic sampling introduces noise, which is reduced using super sampling. Stochastic transparency uses MSAA as transparency, but requires several samples to estimate the correct fragment contribution. It can be parameterized to use more samples, which incurs in higher computational cost and memory consumption.

7.3. Hypothetical scene analysis

We estimated the performance of the algorithms in different rendering contexts aiming at providing a comparative analysis with respect to image quality and performance. We use three rendering contexts to stress different demands of quality and performance, which are based on the proposal of Foley and Feiner [2]: offline rendering (which demands high image quality), interactive rendering (which enables user visualization), and real-time rendering (which enables user interaction with the rendering).

We created a hypothetical scene that can be parameterized to reflect the different rendering contexts described above. Table 5 describes the values used to parameterize the scene within each context. Offline rendering has the highest quality demand, thus we estimate a 4096×2160 window, with detailed meshes and high-quality parameterization. Interactive rendering has an average demand, thus we estimate a 1920×1080 window, with balanced parameterization to allow real-time visualization. Real-time rendering has the highest FPS demand, thus the parameterization prioritizes performance, with reduced geometry and values for buffer size and number of samples.

We estimated the number of operations required by each method using the following equation:

$$op = geo \times (raster + frag_{sort}) + pre_{sort} + blend \quad (5)$$

Table 5
Hypothetical scene parameterization.

Application	$W(p)$	$H(p)$	m	n	l	d	k	s
Offline	4096	2160	33	500 M	200	20	l	l
Interactive	1920	1080	33	50 M	200	20	d	d
Real-time	1920	1080	33	5 M	200	20	3	16

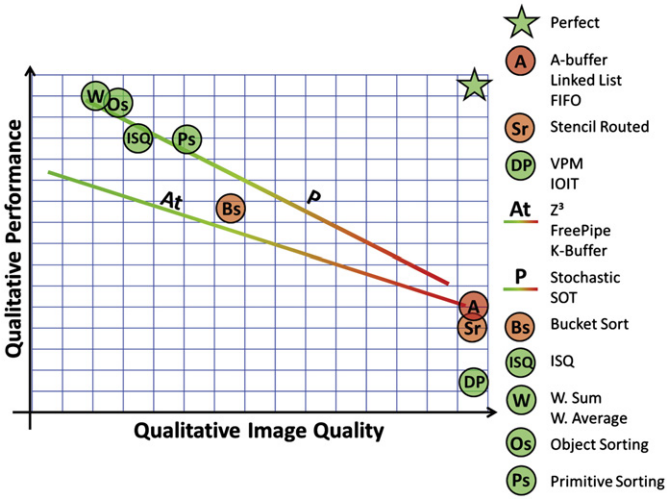

Fig. 16. Comparative chart of techniques regarding image quality, performance, and memory consumption. The graph displays performance with respect to image quality, with colors indicating memory consumption (green = low, red = high). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

where *op* is the approximated number of operations, *geo* is the number of geometry passes (which multiplies the raster and fragment-sorting operations), and the cost for pre-sorting and blending operations.

We use the complexity analysis given in Table 4 as the basis to estimate the number of operations required by each method using Eq. (5). Techniques were grouped and an approximation plot of the qualitative performance and qualitative image quality is given in Fig. 16.

8. Conclusion

Transparency is a required effect in a large set of applications, from computer games to visualizations tools, and realistic rendering. The universe of techniques proposed to solve transparency in raster-based pipelines is vast, as well as the difference in image quality and performance.

The main bottleneck is the fragment-sorting step, required to compute correct depth-ordering for blending. Some techniques avoid sorting and describe stochastic approaches, but without sorting the results are unpredictable. However, those methods present the best cost-benefit, providing convincing results with real-time performance. By presenting several techniques and discussing their advantages and disadvantages, we hope this survey is comprehensive and motivates new research in the field.

Acknowledgments

The work of Marilena Maule is sponsored by a scholarship by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES). The work of João L. D. Comba is sponsored by grants from Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq): 200498/2010-0, 305770/2008-0 and 569239/2008-7.

References

[1] Porter T, Duff T. Compositing digital images. SIGGRAPH Comput Graph 1984;18(3):253–9.

[2] Foley JD, van Dam A, Feiner SK, Hughes JF. Computer graphics: principles and practice. 2nd ed Boston, MA, USA: Addison-Wesley Longman Publishing Co, Inc.; 1990.
[3] Carpenter L. The a-buffer, an antialiased hidden surface method. In: SIGGRAPH'84: proceedings of the 11th annual conference on computer graphics and interactive techniques. New York, NY, USA: ACM; 1984. p. 103–8.
[4] Jouppi NP, Chang C-F. Z3: an economical hardware technique for high-quality antialiasing and transparency. In: HWWS'99: proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on graphics hardware. New York, NY, USA: ACM; 1999. p. 85–93.
[5] Amor M, Bóo M, Padrón EJ, Bartz D. Hardware oriented algorithms for rendering order-independent transparency. Comput J 2006;49:201–10, doi:10.1093/comjnl/bxh159.
[6] Mark WR, Proudfoot K. The f-buffer: a rasterization-order FIFO buffer for multi-pass rendering. In: HWWS'01: proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on graphics hardware. New York, NY, USA: ACM; 2001. p. 57–64.
[7] Wittenbrink CM. R-buffer: a pointerless a-buffer hardware architecture. In: HWWS'01: proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on graphics hardware. New York, NY, USA: ACM; 2001. p. 73–80.
[8] Myers K, Bavoil L. Stencil routed a-buffer. In: SIGGRAPH'07: ACM SIGGRAPH 2007 sketches. New York, NY, USA: ACM; 2007. p. 21. URL 〈http://doi.acm.org/10.1145/1278780.1278806〉.
[9] Liu F, Huang M-C, Liu X-H, Wu E-H. Freepipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In: I3D'10: proceedings of the 2010 ACM SIGGRAPH symposium on interactive 3D graphics and games. New York, NY, USA: ACM; 2010. p. 75–82.
[10] Yang J, Hensley J, Grün H, Thibieroz N. Real-time concurrent linked list construction on the gpu. Comput Graphics Forum 2010;29:1297–304.
[11] Mammen A. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. IEEE Comput Graph Appl 1989;9(4):43–55.
[12] Everitt C. Interactive order-independent transparency. Technical Report, 2001.
[13] Williams L. Casting curved shadows on curved surfaces. SIGGRAPH Comput Graph 1978;12:270–4.
[14] Bavoil L, Myers K. Order independent transparency with dual depth peeling. URL 〈http://developer.ownload.nvidia.com/SDK/10/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf〉; 2008.
[15] Liu F, Huang M-C, Liu X-H, Wu E-H. Efficient depth peeling via bucket sort. In: HPG'09: proceedings of the conference on high performance graphics 2009. New York, NY, USA: ACM; 2009. p. 51–7.
[16] Govindaraju NK, Henson M, Lin MC, Manocha D. Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In: I3D'05: proceedings of the 2005 symposium on interactive 3D graphics and games. New York, NY, USA: ACM; 2005. p. 49–56.
[17] Callahan SP, Ikits M, Comba JLD, Silva CT. Hardware-assisted visibility sorting for unstructured volume rendering. IEEE Trans Visualization Comput Graphics 2005;11(3):285–95.
[18] Bavoil L, Callahan SP, Lefohn A, Comba aLD, Silva Jo CT. Multi-fragment effects on the gpu using the k-buffer. In: I3D'07: proceedings of the 2007 symposium on interactive 3D graphics and games. New York, NY, USA: ACM; 2007. p. 97–104.
[19] Meshkin H. Sort-independent alpha blending. URL 〈http://www.houmany.com/data/GDC2007_Meshkin_Houman_SortIndependentAlphaBlending.ppt〉; 2007.
[20] Enderton E, Sintorn E, Shirley P, Luebke D. Stochastic transparency. In: I3D'10: proceedings of the 2010 ACM SIGGRAPH symposium on interactive 3D graphics and games. New York, NY, USA: ACM; 2010. p. 157–64. URL 〈http://doi.acm.org/10.1145/1730804.1730830〉.
[21] Mulder JD, Groen FCA, van Wijk JJ. Pixel masks for screen-door transparency. In: Proceedings of the conference on visualization 1998, VIS'98. Los Alamitos, CA, USA: IEEE Computer Society Press; 1998. p. 351–8.
[22] Sen O, Chemudugunta C, Gopi M. Silhouette-opaque transparency rendering. In: Comput Graphics Imaging; 2003. p. 153–8.
[23] de Berg M, van Kreveld M, Overmars M, Schwarzkopf O. Computational geometry: algorithms and applications. 2nd ed., Springer-Verlag; 2000.