Computer Graphics Seminar:

From Multi-Layer Depth Peeling to Efficient Order-Independent Transparency

Manuel Jüngst

2019-06-19

Contents

List of Figures

Abstract

In this text, four distinct techniques for transparency rendering in computer graphics are covered, following a brief overview of existing approaches. In particular, the focus is on so called order-independent transparency (OIT), where no additional work (e.g. object sorting) is required on the application side. While there are multiple different subcategories there, the direction here is further towards approaches that conserve memory and deliver precise results, such as Multi-layer Depth Peeling and Concurrent Linked Lists. The techniques are also ordered chronologically, in order to illustrate how the continuous refinements of the hardware rendering pipeline affected the efficiency of OIT.

1 Introduction

Transparency rendering has been a classically difficult problem in computer graphics, or specifically real-time rendering. Solving it is a key component for achieving cinematic-quality real-time rendering [1]. For conventional opaque geometry, only the surfaces closest to the camera need to be drawn (per pixel). However, we generally do not know which surfaces are closest to the camera, before trying to draw them. Disregarding performance, as long as the closest surface at every pixel is not overwritten by some surface behind it, we get correct results. This can be done by using a *depth buffer*, also known as z-(near)-buffer, which stores the distance to the camera of each surface fragment that was drawn; this information can then be used on subsequent drawing attempts to discard fragments that have a greater camera distance than the one already stored. This *depth test* ensures that no background objects are drawn in front of foreground objects.

This process is greatly complicated for transparent geometry, because there are generally multiple surfaces that all have to be drawn at a given pixel location, and it needs to be done in the correct order as well. The order is important because layers with transparency need to be blended together. For instance, an *almost* completely opaque piece of glass should override the majority of the color that can be seen behind it, so whether it is drawn before or after another surface matters a lot. In this example, a back-to-front blending mode is used, where order matters because it is a non-commutative function.

Only after attempting to render all surfaces, we could know what the correct ordering would have been, assuming we also had a data structure that can hold multiple fragment's camera distance values per pixel location (as many as there can be objects in any line of sight). If we use two rendering passes to accomplish this, we have to send all objects through the rendering pipeline twice, incurring many

(a) Correctly sorted fragments via OIT (b) Alpha blending

Figure 1. **Effect of unordered fragment rendering:** Image (a) shows a mecha model consisting of several layers of transparent fragments, that have been blended in order of distance to the camera. Image (b) shows the result of fragments being blended in the (random) order they arrive. Note how not only the skeleton is drawn over the armor, but even the right skeleton leg is incorrectly drawn in front of the left one.

redundant computations, e.g. geometric transformations and rasterization. Avoiding this by simply sorting on the object (or even polygon) level before rendering is not an ultimate solution, because these can overlap or intersect, meaning one object (or primitive) is generally not entirely in front of another. Further, it requires additional programming effort on the application level.

While there exist some approaches for presorting the geometry, the focus here will be on so called *order-independent transparency* (OIT) methods, that work regardless of the ordering on the object level. This is achieved by doing the sorting on the fragment level, where geometry intersection is also no longer a factor. Figure 1 illustrates the need for transparency sorting with an example of the visual artifacts that otherwise occur.

As we will see, doing this even with just two rendering passes is difficult, because of the unknown memory/space requirements for a data structure to hold the variable (and possibly large) amount of fragments at each pixel. Starting with the classical *depth peeling* algorithm, we will see how the sorting was done initially—and on older hardware. Building on that, multi-layer depth peeling and similar methods leverage increased control over GPU memory to do the sorting increasingly more efficiently. Lastly, a brief overview over more modern and possible future developments is given.

2 Existing Approaches

A detailed overview for various transparency techniques is given in a survey by Maule et al. [2], which is also focused mostly on fragment-based / OIT methods. The first method that was used for OIT (among other tasks) is *A-buffer*, which stores and sorts all fragments in per-pixel linked lists [3]. While not suited for real-time graphics, there have been many works since its publication in 1984, that can be seen as derivatives of this method. These commonly have similar names, such as S-buffer, k-buffer or l-buffer. There are multiple ways in which one can classify such approaches.

One distinction is whether a buffer is *dense*, meaning it exploits sparseness, i.e. the fact that most pixels have many fewer fragments than the largest depth possible. In a simple approach, a very large or unbounded buffer is created to store the variable amount of fragments per pixel, with some unused space at pixels with low fragment counts. A dense buffer compresses this unused space in some sense. Some of these methods will be covered in section 3 (*Order-Independent Transparency Techniques*). One non-dense approach is *Bucket Sort*, which partitions the depth dimension into a number of buckets and (in one variant) places fragments into the bucket corresponding to its depth value [4]. Another is *FreePipe*, that also allocates a fixed amount of memory for each pixel, but looks for the next empty spot to place fragments into [5].

Another distinction is whether an approach gives precise results. Especially towards layers far in the background, an exact ordering often does not make a noticeable difference. This represents another trade-off to get more performance. The two previously mentioned methods can be seen as approximate, since they reserve a fixed amount of memory, that can only correctly handle a fixed amount of fragments. Instead of discarding on overflow, one other way to deal with it is to blend together fragments (i.e. freeing space) while sorting [6]. Besides space limitations, another cause of getting approximate results can be indeterministic behavior of the algorithm, caused by concurrency issues, e.g. multiple fragment threads writing to the same memory location. *Stencil Routed A-Buffer*, as mentioned by Lipowski [7], can lead to artifacts due to undefined processing order [8]. *Multi-layer Depth Peeling* deals with this, but at a significant performance cost [9]. This subset of approximate methods is particularly interesting however, since it is still possible for better hardware synchronization to become available at a later point. Lastly, *stochastic transparency* makes a more deliberate step towards performance in the trade-off [10].

Some approaches suggest hardware modifications that allow for more efficient OIT, e.g. with recirculating [11] or delayed [12] fragments, or a general off-grid fragment storage [13]. Such approaches cannot be relied on for the foreseeable

future, as conventional hardware manufacturers may not choose to implement such architectures.

As a final distinction herein, some methods decide to break free from the limitations of the hardware rendering pipeline, without requiring changes thereof, and use/implement a *software rendering pipeline*, e.g. a CUDA rasterizer [5]. In this instance, it allows to freely access generated fragments in memory. This is difficult to scale up to complex applications, as some functionality provided by the hardware pipeline cannot be used. Alternatively, using Vulkan provides some trade-off between this and the more traditional graphics APIs (e.g. OpenGL), by allowing more fine-grained access to the hardware pipeline, where multi-pass sorting can be done with low overhead [14].

Fragment sorting, or more generally structures that can hold such data, have been used for tasks beyond transparency as well. Some uses are: indirect shadowing [15], scene voxelization [16], translucency and constructive solid geometry (CSG) rendering [17], and collision detection [18].

3 Order-Independent Transparency Techniques

In this section, a small selection of (distinct) OIT techniques are covered, with the goal of providing an introduction to the topic. They are ordered by publication date, which allows to add some important historical context, i.e. showing how features added to the rendering pipelines have been leveraged for increasingly efficient OIT. Being able to separate this context from the methods also allows us to reevaluate them as more flexible control over the hardware emerges in the future.

3.1 Depth Peeling Algorithm (2001)

The starting point here is the classical depth peeling algorithm by Everitt [19], which is one of the earliest methods that could be considered real-time (e.g. achieving above 1 fps on difficult scenes with hardware from 2004 [9]). The core idea behind depth peeling is to extend the standard depth test that uses a z-buffer, that is able to hold one 'layer' of (frontmost) fragments. As a reminder, this layer is constructed in the process of rendering the arbitrarily-ordered scene geometry, where only those fragments are allowed to pass through that are closer to the camera than anything previously seen (in that pass). The aforementioned extension for this then consists of keeping that depth buffer for a further geometry pass, where fragments that are in (or in front of) this captured layer are now also discarded. This happens in addition to the regular depth test, that now gives us the next closest 'layer' of fragments. This step can be repeated as often as desired, always saving

(a) Layer 1 (b) Layer 2 (c) Layer 3

Figure 2. **Depth Peeling:** The progressive removal of layers in a scene with three shapes is shown. We (humans) can imagine how the entire scene looks like from (a), but this first layer only consists of the surfaces that are actually visible. Note that the three layers do not correspond to the three shapes. The removal of the first layer is shown at (b), where everything that was visible in the first layer was removed, e.g. part of the circle was covered by the triangle and thus remains. The third layer (c) has only a small section remaining, that was initially covered by two surfaces. Also note that removed layers are, for the sake of clarity, left at 10% opacity here.

the current layer to subsequently discard any fragments of previous layers. The term 'peeling' encompasses this concept, where one layer of fragments is removed at a time. See fig. 2 for an illustration of depth peeling. Conceptually, it can be envisioned as sending out rays for every pixel, that each capture and destroy the closest surface they hit. In addition to the depth values, the color values (RGBA) of each layer have to be stored somehow, so they can all be blended together (in the now-obtained correct order) to produce the final image.

There are several things to consider when implementing this idea. Most importantly, note how we require two depth tests per pass. One is being done behind the scenes as usual (creating a front layer), and the other is done when we additionally discard fragments belonging to previous layers. However, even up until now, there exists no support for multiple depth tests in e.g. OpenGL and presumably most hardware. So the only option is to *store* the depth buffer of a completed layer, and in the next pass of the fragment shader explicitly decide there whether to discard fragments. At the time of this algorithm however, there was presumably no way to store depth values through the fragment shader, at least not in a way that allows use for future passes. Certainly it was not possible to output into multiple textures/buffers. But the key insight (credited to Rui Bastos) was that (hardware

accelerated) *shadow mapping* represents a depth test. Conventionally it is used to determine which surfaces are visible to a light source, i.e. it saves the depth buffer from a light's perspective in a texture, that can be accessed in a subsequent pass (to determine which fragments are illuminated based on this texture). Shadow mapping solves our problem of being unable to store depth values. All of the differences to the actual depth test can be overcome: The orientation and resolution is fixed to that of the camera, meaning that it stores the same depth values as the standard depth test, with only some invariance issues that can also be solved.

With the main issue of a second depth test solved, there are two more details to consider. For the blending mode, back-to-front is used, which also means that the colors of each geometry pass have to be stored in intermediate textures. At the *end*, they can be drawn in that order as viewport-sized quads. The second question that remains is how that end is even defined. In this algorithm, a fixed number of passes was done, which generates approximate results with an easy stop condition.

3.2 Multi-Layer Depth Peeling via Fragment Sort (2006)

Following the availability of *Multiple Render Targets* (MRT) in 2004 (OpenGL 2.0) [20], more efficient OIT was made possible. These allow the fragment shader to output more than one color/depth value. *Multi-Layer Depth Peeling* is selected here to show one such way to increase OIT efficiency [9]. Being considered a (relatively) dense method here, it does not allocate a very large buffer to store fragment data, nor does it deliver approximate results. Instead, it creates a relatively small buffer that collects multiple depth layers per pass, but generally still requires several passes in total—hence the term 'peeling' remains appropriate. This technique can be seen as a bridge between classical depth peeling and more advanced dense buffers that will be shown in the next subsections.

The basic idea is as follows: We choose some number of MRT that allow us to store some number of layers M. There are some limitations to consider when choosing this number, but this will be elaborated later. For a given render pass, these will store the next M *frontmost* layers, i.e. each pixel in the fragment shader will output M fragments' color and depth values. In the subsequent pass, we discard any fragment in or in front of the previously last layer, as well as fragments further away than the next M layers (which completely replaces the standard depth test, i.e. this also discards fragments that are behind what has already been rendered). Using *front-to-back* blending this time, the M layers of each pass are blended immediately into a texture that ultimately holds the final result (of transparency rendering).

In order to understand the idea, it is important to see how the next M layers are determined, or in other words how the sorting is done. Liu et al. illustrate the

sorting procedure with a single (array)list per pixel [9], that (a) holds all layers, and (b) is completely sorted. This simplified view shows the problem related to the size of M, but from the implementation perspective, this situation is more complicated. As already mentioned, (a) all layers are never stored explicitly, as storage is limited and they are composited after each pass, and (b) the set of next M layers is sorted as a whole, i.e. frontmost, but items within the set need not be sorted immediately; it can be done in the compositing step instead. If we view the storage buffer as a list, we can think of the sorting algorithm as insertion sort, i.e. we take incoming fragments and place them into the appropriate position within that list. Since it is space-limited, and we do not require ordering within the list, it suffices to simply compare against the largest element (depth value), and replace it whenever a new fragment with lower depth is encountered. This way, regardless of the order that fragments are processed, only those that are the next M closest will never be replaced. An example of this algorithm is shown in fig. 3.

The deciding factor in regards to performance of the algorithm is the size of M. If it is very large, the sorting can be achieved in a single pass; if it was 1, we essentially get classical depth peeling. Note that we require read/write access for the MRT—the 'list' needs to be read to find the next position for insertion, and correspondingly be written to. Without proper synchronization, different threads of fragments create *data hazards*, as they can simultaneously write to the same index/address. Concretely, two fragments of the same pixel can read the same list state, and each try to write out their own version of the modified list. However, since both lists can only differ in one place (at the largest depth index), such a collision cannot produce an invalid/undefined state; the result is as if one fragment was simply ignored. The authors Liu et al. call this the atomic commitment assumption. Although it can at first be unintuitive to only require partial sorting within the MRT buffer, this method is resistant to the occurring data hazards. The only thing we need to do is repeat the sorting procedure whenever a concurrency error has been detected. Since there is also always some form of progress in regards to the sorting state, the algorithm is then guaranteed to converge to the correct result. The cost of one or more concurrency errors is having to do at least one repeated geometry pass per peeling step. The probability for a concurrency error depends on M—the more layers we try to sort per geometry pass, the higher the chance for one to occur. So to conclude, the choice for M represents a trade-off between possible amount of layers sorted per geometry pass, and probability of concurrency errors (that significantly reduce the number of layers sorted per geometry pass).

There is one last factor to consider before selecting M: the number of layers that can be stored per render target. The data that we have to store for each fragment is a color and a depth value, both of which can be of variable precision. Common for color values is one byte per channel, i.e. four bytes for RGBA, while depth

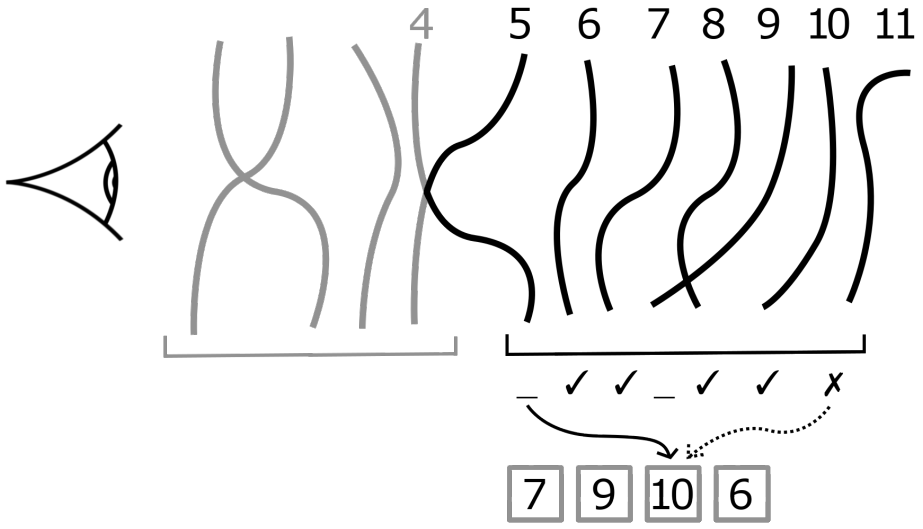


Figure 3. **Multi-layer depth peeling:** A snapshot of the algorithm by Liu et al. is shown [9]. In front of the camera on the left are several layers to be sorted/peeled. A buffer size M of 4 is shown on the bottom, meaning four layers can be peeled per geometry pass. Here we are in the second pass. We saved layer 4, the backmost layer in the previous pass, which we use to remove already processed layers in the front, shown in gray. Layers behind the next four are removed by checking against the currently buffered ones. In this case, layers 7, 9, 10 and 6 trivially went into the buffer as it started off empty. The next layers pass only when they are closer than the farthest layer, which is now 10. Layer 11 failed this check, but layer 5 passes and replaces 10. In the final step, that is not displayed, layer 8 would replace layer 9 to complete the back removal. Note that layer indices are shown here in place of depth values, but the algorithm works the same with the latter.

values are commonly three or four bytes. In their implementation, Liu et al. have MRT with 16 bytes available, meaning with 4+4 byte data per pixel, each can hold two layers/fragments. The term *bucket* is used to differentiate storage locations from MRT, as they are now distinct quantities. Presumably it is advantageous to pack data into a few larger MRT as opposed to more smaller MRT (with the same number of buckets M), e.g. it may reduce concurrency errors, help with caching, or noticeably reduce the number of write instructions. This can be determined experimentally, but in any case, the authors opted to pack $M=8$ into 4 MRT through their experiments. As a final note, there is also a hard limit on the maximum number of MRT possible, at least through the amount of GPU memory available to the

application.

With a good understanding of what the buffer for this technique looks like, the rendering algorithm 1 is fairly simple. As usual in OIT, all opaque objects in the scene can be rendered separately first, with the regular depth test. In addition to the M buffer (to discard layers behind the current M), we have a z-buffer to discard (already processed) layers in front of the current M layers. We then have two loops: The outer loop peels the next M layers until the entire scene is done, and the inner loop repeats the current peeling step until there are no more concurrency errors. Until now we have not covered how either of the exit conditions can be detected; in classical depth peeling there was simply a fixed number of peeling iterations. The tools for this job are *occlusion queries*, that will return the number of fragments rendered (i.e. not discarded) for any number of geometry draw calls. Here we check all transparent objects, in the inner loop. If there was at least one fragment drawn in the first iteration of the inner loop, i.e. the current M peeling pass, we know that there are still layers to peel; otherwise we can conclude the outer loop as the entire scene is finished. The check for concurrency errors functions the same: Assuming we drew something in the first inner loop iteration, we have to go into an additional iteration and query the fragment count again. If it has since then become zero, there were no errors, i.e. not a single new layer made it into the buffer; otherwise we know there were changes and have to repeat. Lastly and to recap, after each inner loop iteration, the (partially) sorted layers in the M buffer are blended in front-to-back order into a collective texture.

There is one caveat with separating the rendering of opaque objects: combining transparent with opaque layers. We need to avoid collecting transparent fragments behind the first opaque fragment, otherwise they will be included in the later blending step, despite not being visible. In other words, the front opaque layer represents the background (i.e. final) color for the transparent layers. But the transparent rendering algorithm has no awareness of opaque objects/fragments. But we can keep the z-buffer from the opaque rendering pass, to then discard transparent fragments that are behind the first opaque fragments. While previously mentioned that the built-in depth test is not used, this applies only to comparisons between transparent fragments, and it can actually be used to now do this opaque discard for us. This is illustrated in fig. 4. So due to this separation, the opaque geometry can not only be rendered traditionally fast with just one pass, but transparent geometry rendering is also sped up via additional early discard opportunities. Since we are using front-to-back blending, the results of the opaque rendering pass have to be kept separately, and can later be combined with the texture that contains all the transparent layers.

The most important steps for the fragment shader have already been mentioned, namely: the front discard by means of z-buffer, and the back discard and simultaneous in-range keep by means of finding and comparing against the backmost

Algorithm 1 Multi-layer Depth Peeling

```
1:  draw opaque geometry with standard depth test
2:  while more layers to peel do
3:      buffer = new MRTbuckets[M]
4:      first_iteration = true
5:      while concurrency errors not excluded do
6:          draw transparent geometry
7:          (fragment shader:) update buffer from current M range
8:          if 0 fragments rendered then
9:              concurrency errors are excluded
10:             if first_iteration then
11:                 no more layers to peel
12:             end if
13:         end if
14:         first_iteration = false
15:     end while
16:     blend current M layers
17:     save farthest layer for future discards
18: end while
```

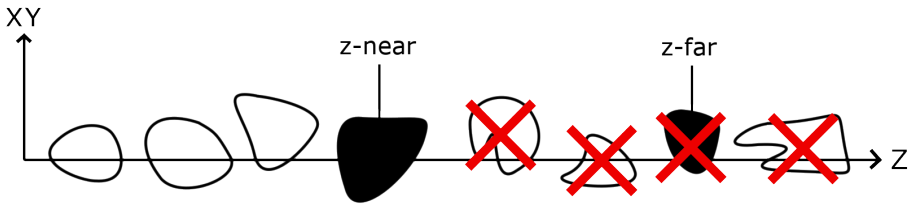


Figure 4. **Opaque rendering:** For increased performance, all opaque objects in a scene with transparency can be rendered conventionally (i.e. not sorted) first. Shown here is a side view on a scene with opaque (black) and transparent (white) objects. The standard z-buffer after the opaque pass contains the object labeled z-near (or more precisely, its fragments). In the transparency pass, we keep this z-buffer and built-in depth test, to prevent processing transparent fragments behind it. Otherwise they will be falsely blended into the other transparent fragments, despite not being visible. For clarity, this is shown with objects, but keep in mind that we actually operate on the fragment level.

fragment in the M buffer. One additional important detail is the discard of already stored fragments, that is required due to the repeat geometry passes in the case of errors. This is done between the two aforementioned steps, by checking for (exact) equality of fragment depths; if this occurs, the fragment is already stored, and we prevent inserting the same layer multiple times to the buffer (as it would otherwise pass the second test). The rest of the shader deals with (un)packing the data from/into the correct formats, e.g. floating point and integer numbers. In the most general sense possible, we have four depth tests in total:

- front discard, i.e. completed layers
- back discard, i.e. layers behind current M, found iteratively
- equality discard, i.e. already within current M contained layers
- opaque discard, i.e. transparent fragments behind first opaque layer

In summary, it is possible to speed up classical depth peeling by a factor of up to M, by utilizing MRT as a buffer to hold fragment data. Data hazards arise when multiple fragment threads try to concurrently write to a buffer, which significantly slows down the theoretical performance of the algorithm; in their experiments the speedup was roughly $2\times$. The authors of [9] envisioned either improved concurrency control or hardware modifications for future improvements.

3.3 Real-Time Concurrent Linked List Construction on the GPU (2010)

The next milestone for OIT techniques was reached in 2010 (OpenGL 4.0), when atomic access to arbitrary memory locations on the GPU was made possible [15], [20]. This allows fragment shaders to not only write out into a fixed buffer (render targets), but any custom data structure. This is especially relevant to dense buffer methods, since we can now store all possible fragments, without having to allocate a very large buffer that keeps most of the space unused (just to catch outlier pixels with a large number of fragments). Atomic counters provide a way to prevent data hazards, as the hardware guarantees to read and modify a value as what is effectively a single operation. The data hazards from section 3.2 occurred as multiple threads were able to read the same state, even though a modification in between those reads was still pending.

The first approach mentioned in section 2 was *A-buffer*, which created linked lists per pixel to store all fragments. *Real-time Concurrent Linked List Construction* is selected here to show one (simple) way of utilizing the new hardware capabilities to improve OIT, by efficiently implementing such an advanced data structure on the GPU [15]. Unlike depth peeling, this approach is generally able to store all fragments in a single geometry pass.

As a reminder, a (singly) linked list consists of nodes, that contain a payload and a pointer to the next node in the list. The payload in this case is a fragment, specifically its color and depth value. This is different than e.g. array lists, that do not require an overhead to locate the next element, because each has a predetermined/fixed location (i.e. $index \cdot element\_size$). To append an element to the list, a new node is created at any available memory location, with its pointer value set to some value that marks the end, and the previously last list element changes its pointer to where this new node was placed.

The basic idea here is that all linked lists for every pixel can be stored in only two buffers (i.e. two allocations): The *head pointer buffer* has a fixed size and contains the locations of each list, i.e. a pointer to the first node of each list. All nodes are stored in a single *node buffer*. Paired with this buffer is an atomic counter variable, that always points to the next free position in the buffer, i.e. it is incremented by the size of a node. Since this read and increment is atomic, no two concurrent threads will ever obtain the same memory location to store their data. See fig. 5 for an example of this idea. Since the node buffer stores all data sequentially, the data is packed densely, e.g. a pixel that has only one fragment will only use up the size of one node. It is however not perfectly dense, since there is generally some unused space at the tail of the buffer. If we knew how many fragments there are in total, we could allocate the precise amount of memory needed, but as it stands, we are required to make some estimate for a good buffer size. This generally also means that at some point an overflow can occur, whenever this estimate falls short.

When implementing the linked list as described above, we encounter one issue: We have to remember the most recent node for each pixel, so that we can update its pointer to a new node. This is easily solved by reversing the list, i.e. each new node now points back to the *previous* node instead, that is now always stored in the head pointer buffer. In other words, the head pointer buffer always points to (i.e. remembers) the last node; an apter name for it is perhaps *tail* pointer buffer. This update is also done with an atomic swap operation, so that the previously last node is simultaneously inserted into the new node's pointer as the list head/tail is updated.

After capturing all fragments with one geometry pass, they still have to be sorted and blended. This can be done in an extra pass with just a viewport-sized quad. In a corresponding fragment shader, each pixel receives one fragment thread, that can process its corresponding linked list from the previous pass. The sorting can be done in any way, but [15] suggests a simple insertion sort (for small lists). Afterwards (but still in the same shader) we can go through the list in order, blend the colors, and output the final color for each pixel. If we had a layer from a prior opaque geometry pass, we can pass it to this shader and include it as the first (or last) fragment to blend. Just like explained in section 3.2, the standard z-

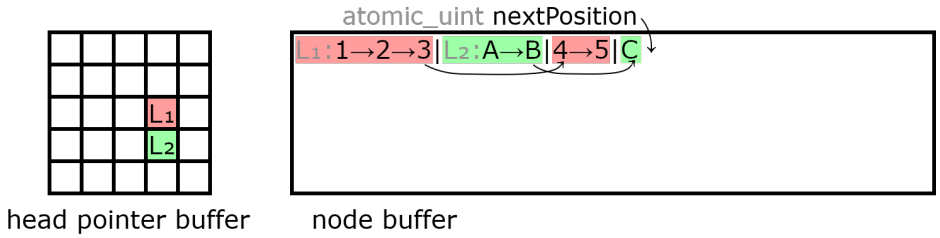


Figure 5. **Linked list buffers:** The left grid shows every pixel for a 5×5 viewport. To capture fragments at every pixel, we create linked lists at the non-empty pixels—here, L_1 and L_2. The head pointer buffer simply holds pointers to the first elements of each list. The actual lists are stored in the node buffer. Here we first stored fragments 1, 2 and 3 of the pixel L_1, then fragments A and B of pixel L_2. Each fragment is stored inside of a node that has a pointer to the next node. As e.g. fragment 4 is inserted, the pointer of the previous node 3 is set to the current position at that time. Since multiple pixels/fragments all want access to the node buffer concurrently, an atomic counter *nextPosition* ensures that no two fragments can obtain the same address to insert a new node. Note the large amount of unused space in the node buffer, that exists because we do not know the total number of fragments, i.e. how much space is required. In this instance, there were only two pixels that even had fragments (so far).

buffer from the opaque pass can also be kept throughout the transparent fragment collection pass.

To briefly recap, atomic variables and random memory access (within allocated buffers) made it possible to implement a concurrent linked list entirely on the GPU, using only two buffers. The main buffer is relatively dense, but overflow can occur. So far, there was no mention of performance. The obvious benefit over the approach by Liu et al. and depth peeling in general is that this approach can get by with a single geometry pass. In the case of an overflow however, a sensible strategy is to allocate a larger buffer and start over in another geometry pass. But since this can be set up to not happen frequently, i.e. by increasing/decreasing buffer size sensibly, the performance comparison largely hinges on whether *one* pass here is significantly slower than *all* passes for multi-layer depth peeling [9]. There are two main causes for performance degradation. The first one arises through the realization of atomic variables, which makes guarantees of, but is not implemented as a single hardware operation. Instead, all but one of the concurrently accessing threads have to be either blocked or keep retrying. This becomes a significant problem here, since *all* fragments require access to the same atomic counter. The

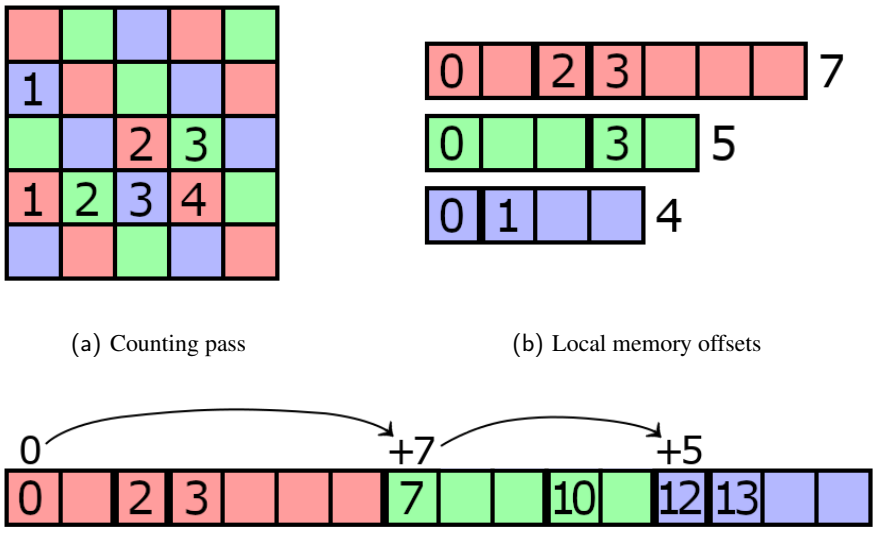
second cause is the memory layout of the fragments, which is random, and thus difficult to cache or optimize. Still, the experimental comparisons by Yang et al. put its performance at the top [15]. However, multi-layer depth peeling was not included; the data available is merely that its speedup over classical depth peeling is between 1.50 and 1.94 in their experiments [9], and that the speedup for Yang et al. over classical depth peeling is between 1.28 and 7.51—with entirely different scenes for both.

3.4 S-buffer: Sparsity-aware Multi-fragment Rendering (2012)

One last OIT approach that is described here is *S-buffer* [17], that illustrates a more sophisticated fragment storage data structure. It similar to, but addresses both weak points of the linked list approach. However this comes with a trade-off of an additional geometry pass, that precisely calculates how much space is required for a buffer to store all fragments; this also makes it more memory efficient. After the *counting pass*, each pixel is assigned a contiguous memory region where its precise array list for fragments resides. In the regular (transparent) fragment collection pass, they are simply added to the lists.

It is important to know how the memory assignment is realized. It can be done with an atomic counter, that is incremented for each pixel by its fragment count. The return values of these increments represent the memory offsets for each list, that can be stored in a viewport-sized texture. This concept existed even prior to the concurrent linked lists one [7]. However, this obviously does not solve the perhaps most important problem of the linked list approach: contention on a single atomic variable. One major feature of S-buffer is the usage of S different atomic counters, in order to alleviate contention. Each pixel is implicitly assigned to one of S groups, based on its position, i.e. all pixels P are distributed alternatingly with the formula: $H(P) = (P.y \cdot width + P.x) \bmod S$. So after the counting pass, all groups create their now local memory offsets in parallel. Afterwards a *prefix sum* is done over all counters C (that now hold the total fragment count for each group) to determine the offset of each group, e.g. $(C_1, C_2, C_3, \ldots) \rightarrow (0, C_1, C_1+C_2, \ldots)$. The memory location for a list within the buffer that holds all fragments is then: local memory offset + group offset. This means at first each group has a contiguous memory region, and within the groups each pixel still has a contiguous region. See fig. 6 for an example of this idea.

To elaborate on how this solves the main problems of the linked lists approach: We alleviate atomic contention by creating an arbitrary amount of counters S, that are each accessed by a lower number of threads. Vasilakis and Fudos found that the ideal amount is roughly 30 in their experiments [17], as going further increases the overhead due to the prefix sum for group offset calculation. Secondly, the



(a) Counting pass

(b) Local memory offsets

(c) Global memory offsets

Figure 6. **S-buffer address calculation:** In order to store all incoming fragments more densely/controlled, this approach takes three additional steps (a)-(c), before going into the transparent fragment collection pass. Another 5×5 viewport is shown in (a), but this time we only store the fragment count at each pixel, which is obtained by an additional geometry pass. The colors correspond to groups S, that are used to alleviate contention of atomic counters; here we have three. The next step (b) shows how the group-internal addresses for each pixel's list are determined: An atomic counter is incremented by the number of fragments at each pixel, e.g. in the red group, the first pixel requires two fragments to store, followed by one, and lastly four. The final atomic counter values are shown at the very right. In the third step (c), we cumulatively add the total fragment counts of each group to place them all into a single buffer. Each pixel keeps track of its own address, where it can later store all incoming fragments from the fragment collection geometry pass.

memory layout of the buffer is more orderly. When e.g. sorting through all stored fragments of a list, they are all adjacent in memory, which makes for better caching and data bus occupancy. Lastly, the buffer is not only more dense in regards to not having any empty space, but we also save the overhead of storing a pointer with each fragment. Note that this denseness is also what allows this approach to

efficiently reduce contention. If multiple atomic counters were used in the linked lists approach, each one would have the problem of unused space at the tail end, as we do not know how much space to allocate. To avoid overflow at every possible partition of the buffer, we would have to allocate generously large buffers, throwing away the denseness property; just one overflow in any of these would likely already require starting over with another pass.

Many implementation details have already been given, so there is not much to gain from going over the algorithm; just a few notes will be given here: The *address passes* (local memory offsets and group offset computation) between the counting and capturing geometry passes are realized entirely in fragment shaders by Vasilakis and Fudos [17]. Also note that not just a few atomic variables are required, but entire textures that can be updated atomically (per pixel). If not for the counting pass (that can also be implemented by add-blending each fragment with a fixed value of 1), then at least in the fragment capturing pass. There, each pixel also has to keep track of its current index in the list (that is also added to the local and global memory offset to obtain a fragment's address), where atomicity is needed due to multiple fragments incrementing a pixel's index. Lastly, the sorting and blending pass at the very end can be realized as in section 3.3.

In summary, we used a counting pass not only to precisely determine the size of a buffer needed to store all fragments, but also to be able to arrange them sensibly in memory. Even though an additional geometry pass is required, the speedup over the linked list approach ends up being roughly $3\times$ (in the experiments of [17]).

4 Conclusion and Development Trends

At this point, we built a basic foundation for the topic of order-independent transparency. We have seen ways of capturing one, multiple, or all layers of transparent fragments at a time, within conventional rendering pipelines, and while obtaining precise results. For further developments in this direction, the reader may refer to e.g. [21] or [22].

Another interesting development is hardware support for *pixel synchronization*, initially developed by Intel (2013) [23] and now also available from NVIDIA [24]. It allows to define critical sections within shader code, where data hazards are removed by ensuring that read-modify-write operations are performed in submission order, and delayed as needed [25]. As mentioned in section 2, there are several OIT techniques which suffer from an undefined processing order of fragments, either causing artifacts or performance loss, including the multi-layer depth peeling described in section 3.2. One approach that already makes use of pixel synchronization is k^+*-buffer*, which is inspired by S-buffer, but also ventures into the category

of approximate OIT [18]. New techniques can surface, that have previously not been explored due to the problematic data hazards.

The future trends go into the directions of being able to do fragment sorting with lower overhead. It can be with a single pass through better synchronization, or in multiple peeling passes but with very minor pipeline overhead/redundancy, e.g. by means of increased pipeline control, such as provided by Vulkan.

References

[1] J. Andersson, Ed., *5 Major Challenges in Real-Time Rendering*, Beyond Programmable Shading course, SIGGRAPH 2012, Aug. 2012 (cit. on p. 2).

[2] M. Maule, J. L. Comba, R. P. Torchelsen, and R. Bastos, "A survey of raster-based transparency techniques", *Computers & Graphics*, vol. 35, no. 6, pp. 1023–1034, Dec. 2011, ISSN: 0097-8493. DOI: 10.1016/j.cag.2011.07.006 (cit. on p. 4).

[3] L. Carpenter, "The a-buffer, an antialiased hidden surface method", in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '84, New York, NY, USA: ACM, 1984, pp. 103–108, ISBN: 0-89791-138-5. DOI: 10.1145/800031.808585 (cit. on p. 4).

[4] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu, "Efficient depth peeling via bucket sort", in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09, New Orleans, Louisiana: ACM, Aug. 2009, pp. 51–57, ISBN: 978-1-60558-603-8. DOI: 10.1145/1572769.1572779 (cit. on p. 4).

[5] ——, "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*, ser. I3D '10, Washington, D.C.: ACM, Feb. 2010, pp. 75–82, ISBN: 978-1-60558-939-8. DOI: 10.1145/1730804.1730817 (cit. on pp. 4, 5).

[6] N. P. Jouppi and C.-F. Chang, "Z3: An economical hardware technique for high-quality antialiasing and transparency", in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, ser. HWWS '99, Los Angeles, California, USA: ACM, Aug. 1999, pp. 85–93, ISBN: 1-58113-170-4. DOI: 10.1145/311534.311582 (cit. on p. 4).

[7] J. K. Lipowski, "Multi-layered framebuffer condensation: The l-buffer concept", in *Computer Vision and Graphics*, L. Bolc, R. Tadeusiewicz, L. J. Chmielewski, and K. Wojciechowski, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, Sep. 2010, pp. 89–97, ISBN: 978-3-642-15907-7 (cit. on pp. 4, 15).

[8] K. Myers and L. Bavoil, "Stencil routed a-buffer", in *ACM SIGGRAPH 2007 Sketches*, ser. SIGGRAPH '07, San Diego, California: ACM, Aug. 2007, ISBN: 978-1-4503-4726-6. DOI: 10.1145/1278780.1278806 (cit. on p. 4).

[9] B. Liu, L. Wei, Y. Xu, and E. Wu, "Multi-layer depth peeling via fragment sort", in *2009 11th IEEE International Conference on Computer-Aided Design and Computer Graphics*, Aug. 2009, pp. 452–456. DOI: 10.1109/CADCG.2009.5246861 (cit. on pp. 4, 5, 7–9, 12, 14, 15).

[10] E. Enderton, E. Sintorn, P. Shirley, and D. Luebke, "Stochastic transparency", in *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '10, Washington, D.C.: ACM, Feb. 2010, pp. 157–164, ISBN: 978-1-60558-939-8. DOI: 10.1145/1730804.1730830 (cit. on p. 4).

[11] C. M. Wittenbrink, "R-buffer: A pointerless a-buffer hardware architecture", in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, ser. HWWS '01, Los Angeles, California, USA: ACM, Aug. 2001, pp. 73–80, ISBN: 1-58113-407-X. DOI: 10.1145/383507.383529 (cit. on p. 4).

[12] T. Aila, V. Miettinen, and P. Nordlund, "Delay streams for graphics hardware", *ACM Trans. Graph.*, vol. 22, no. 3, pp. 792–800, Jul. 2003, ISSN: 0730-0301. DOI: 10.1145/882262.882347 (cit. on p. 4).

[13] G. S. Johnson, J. Lee, C. A. Burns, and W. R. Mark, "The irregular z-buffer: Hardware acceleration for irregular data structures", *ACM Trans. Graph.*, vol. 24, no. 4, pp. 1462–1482, Oct. 2005, ISSN: 0730-0301. DOI: 10.1145/1095878.1095889 (cit. on p. 4).

[14] M. Wellings. (Jul. 2016). Depth peeling order independent transparency in vulkan, [Online]. Available: https://matthewwellings.com/blog/depth-peeling-order-independent-transparency-in-vulkan/ (visited on 05/24/2019) (cit. on p. 5).

[15] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz, "Real-time concurrent linked list construction on the gpu", in *Proceedings of the 21st Eurographics Conference on Rendering*, ser. EGSR'10, Saarbrücken, Germany: Euro-

graphics Association, Jun. 2010, pp. 1297–1304. DOI: 10.1111/j.1467-8659.2010.01725.x (cit. on pp. 5, 12, 13, 15).

[16] E. Eisemann and X. Décoret, "Fast scene voxelization and applications", in *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, ser. I3D '06, Redwood City, California: ACM, Mar. 2006, pp. 71–78, ISBN: 1-59593-295-X. DOI: 10.1145/1111411.1111424 (cit. on p. 5).

[17] A. Vasilakis and I. Fudos, "S-buffer: Sparsity-aware multi-fragment rendering", in *Eurographics 2012 - Short Papers Proceedings, Cagliari, Italy, May 13-18, 2012*, May 2012, pp. 101–104. DOI: 10.2312/conf/EG2012/short/101-104 (cit. on pp. 5, 15, 17).

[18] A. A. Vasilakis and I. Fudos, "K+-buffer: Fragment synchronized k-buffer", in *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '14, San Francisco, California: ACM, Mar. 2014, pp. 143–150, ISBN: 978-1-4503-2717-6. DOI: 10.1145/2556700.2556702 (cit. on pp. 5, 18).

[19] C. W. Everitt, "Interactive order-independent transparency", NVIDIA OpenGL Applications Engineering, Tech. Rep., May 2001 (cit. on p. 5).

[20] Khronos. (Feb. 2019). History of opengl - opengl wiki, [Online]. Available: https://www.khronos.org/opengl/wiki/History_of_OpenGL (visited on 05/29/2019) (cit. on pp. 7, 12).

[21] E. Kerzner, C. Wyman, L. Butler, and C. Gribble, "Toward efficient and accurate order-independent transparency", in *ACM SIGGRAPH 2013 Posters*, ser. SIGGRAPH '13, Anaheim, California: ACM, Jul. 2013, 109:1–109:1, ISBN: 978-1-4503-2342-0. DOI: 10.1145/2503385.2503504 (cit. on p. 17).

[22] J. K. Lipowski, "D-buffer: Irregular image data storage made practical", *Opto-Electronics Review*, vol. 21, no. 1, pp. 103–125, Mar. 2013, ISSN: 1896-3757. DOI: 10.2478/s11772-013-0073-y (cit. on p. 17).

[23] Intel. (Aug. 2013). Opengl extension/intel_fragment_shader_ordering, [Online]. Available: https://www.khronos.org/registry/OpenGL/extensions/INTEL/INTEL_fragment_shader_ordering.txt (visited on 06/07/2019) (cit. on p. 17).

[24] NVIDIA. (Mar. 2015). Opengl extension/nv_fragment_shader_interlock, [Online]. Available: https://www.khronos.org/registry/OpenGL/extensions/NV/NV_fragment_shader_interlock.txt (visited on 06/07/2019) (cit. on p. 17).

[25] M. Salvi, “Advances in real-time rendering in games: Pixel synchronization: Solving old graphics problems with new data structures”, in *ACM SIGGRAPH*, Jul. 2013 (cit. on p. 17).