Depth-Fighting Aware Methods for Multifragment Rendering

Andreas-Alexandros Vasilakis and Ioannis Fudos, *Member*, *IEEE*

Abstract—Many applications require operations on multiple fragments that result from ray casting at the same pixel location. To this end, several approaches have been introduced that process for each pixel one or more fragments per rendering pass, so as to produce a multifragment effect. However, multifragment rasterization is susceptible to flickering artifacts when two or more visible fragments of the scene have identical depth values. This phenomenon is called *coplanarity* or *Z-fighting* and incurs various unpleasant and unintuitive results when rendering complex multilayer scenes. In this work, we develop depth-fighting aware algorithms for reducing, eliminating and/or detecting related flaws in scenes suffering from duplicate geometry. We adapt previously presented single and multipass rendering methods, providing alternatives for both commodity and modern graphics hardware. We report on the efficiency and robustness of all these alternatives and provide comprehensive comparison results. Finally, visual results are offered illustrating the effectiveness of our variants for a number of applications where depth accuracy and order are of critical importance.

Index Terms—Depth peeling, Z-fighting, visibility ordering, multi-fragment rendering, A-buffer

1 Introduction

Current graphics hardware facilitate real-time rendering for applications that require accurate multifragment processing such as Three-dimensional scene inspection for games and animation, solid model browsing for computer-aided design, constructive solid geometry, visualization of self-crossing surfaces, and wireframe rendering in conjunction with transparency and translucency. This is accomplished by processing multiple fragments per pixel during rasterization.

Z-fighting is a phenomenon in Three-dimensional rendering that occurs when two or more primitives have the same or similar values in the Z-buffer (see Fig. 1). Z-fighting may manifest itself through: 1) intersecting surfaces that result in intersecting primitives, 2) overlapping surfaces, i.e., surfaces containing one or more primitives that are coplanar and overlap, 3) nonconvergent surfaces due to the fixed-point round-off errors of perspective projection.

Traditional hardware supported rendering techniques do not treat Z-fighting and render only one of the fragments that possess the same *depth* value. This results in dotted or dashed lines or heavily speckled surface areas. In this context, Z-fighting cannot be totally avoided and may be reduced by using a higher depth buffer resolution and inverse mapping of depth values in the depth buffer [1] or using depth bias [2].

Multifragment capturing techniques are even more susceptible to Z-fighting, because they need to examine all fragments (even those that are not visible) in a certain order (ascending, descending or both) before deciding what to render. Thus, they may encounter multiple Z-fighting triggered liabilities per pixel.

Correct depth peeling techniques are important for a number of coplanarity-sensitive applications (see Fig. 2), from nonphotorealistic rendering (e.g., order-independent transparency [3], styled/wireframe rendering [4]) to shadow volumes, Boolean operations, self-trimmed surfaces [5], and visualization of intersecting surfaces [6].

In this work, we do not treat the numerical robustness/instability that arises due to the finite precision of the Z-buffer and the numerical errors incurred from the transformations applied prior to rendering. However, we introduce image-based *coplanarity*-aware algorithms for reducing (may miss fragments but are usually faster), eliminating (guaranteed to extract all fragments) and/or detecting-highlighting-related flaws in scenes suffering from coplanar geometry. We provide alternatives for both commodity and modern graphics hardware. We further present quantitative and qualitative results with respect to performance, space requirements, and robustness. A short discussion is offered on how to select a variant from the given repertoire based on the application, the scene complexity, and the hardware limitations. Finally, visual output is provided illustrating the effectiveness of our variants over the conventional methods for a number of Z-fighting sensitive applications.

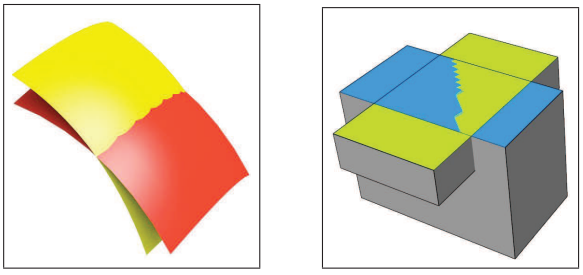
The structure of this paper is as follows: Section 2 offers a brief overview of prior art. Section 3 introduces robust and approximate algorithm variants along with several optimization techniques. Section 4 provides extensive comparative results for all multifragment rendering alternatives. Finally, Section 5 offers conclusions and identifies promising research directions.

2 Related Work

Fragment level techniques work by sorting surfaces viewed through each sample position, avoiding the

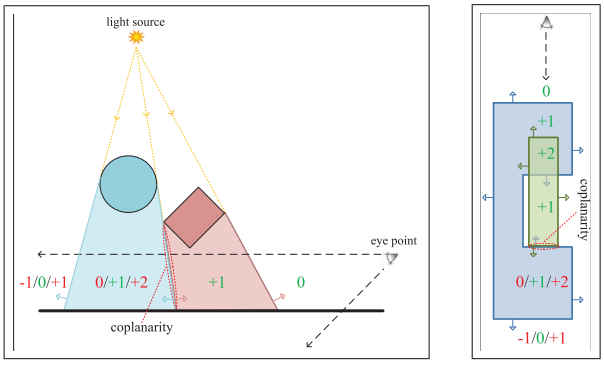
• *The authors are with the Department of Computer Science, University of Ioannina, Ioannina 45110, Greece. E-mail: {abasilak, fudos}@cs.uoi.gr.*

Manuscript received 10 Jan. 2012; revised 19 July 2012; accepted 6 Oct. 2012; published online 15 Oct. 2012.
Recommended for acceptance by W. Wang.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2012-01-0008.
Digital Object Identifier no. 10.1109/TVCG.2012.300.



(a) *Blender* 2.5 (b) *Google SketchUp*

Fig. 1. Illustrating unpleasant effects when rendering (a) intersecting or (b) overlapping surfaces on popular modeling programs.



(a) *Shadow volumes* (b) *CSG*

Fig. 2. Illustrating the values of the popular winding number when ray casting for in/out classifications. Red-painted values highlight erroneous computations (in cases where only one of the two coplanar fragments is successfully captured).

sorting drawbacks that occur in object/primitive sorting techniques [7], [8] (e.g., geometry interpenetration, primitive splitting) or hybrid methods that order the generated fragments by exploiting spatial coherency [9], [10], [11]. These algorithms can be classified in two broad categories, those using *depth peeling* and those employing *hardware implemented buffers*, according to the approach taken to resolve visibility ordering [3].

Given the limited memory resources of graphics hardware, multipass rendering is often required to carry out complex effects, often substantially limiting performance. Probably, the most well-known multipass peeling technique is *front-to-back* (*F2B*) depth peeling [12], which works by rendering the geometry multiple times, peeling off a single fragment layer per pass. *Dual depth peeling* (*DUAL*) [13] speeds up multifragment rendering by capturing both the nearest and the furthest fragments in each pass. Finally, Liu et al. [14] extend dual depth peeling by extracting two fragments per *uniform clustered bucket* (*BUN*). To reduce collisions at scenes with highly nonuniform distributions of fragments, they further propose to *adaptively subdivide depth range* (*BAD*) according to fragment occupation [15] at the expense of extra rendering passes and larger memory overhead.

However, all currently proposed depth peeling techniques cannot deal with fragments of equal depth, thus detecting only one of them and missing the others. A number of solutions have been introduced to alleviate coplanarity issues in depth peeling. Cole and Finkelstein [4] propose *id peeling*, which addresses artifacts where lines obscure other lines by allowing a line fragment to pass only if its index is lower than the highest index at the corresponding pixel in the previous iteration. Despite its accurate behavior, it peels only one fragment per peeling iteration and cannot support rendering of occluded edges. Vasilakis and Fudos [16] extend this work to a multipass scheme achieving robust rendering behavior with the tradeoff of low frame rates. Recently, Busking et al. [6] introduced *coplanarity peeling* extending F2B with in/out classification masking. However, it can only distinguish coplanar fragments between different objects that do not self-intersect.

On the other hand, buffer-based methods use GPU-accelerated structures to hold multifragments (even coplanar) per pixel. The major limitations of this class are first, the potentially large and possible wasted memory requirements due to their strategy to allocate the same memory for each pixel (see Fig. 3a) and second, the necessity of an additional fullscreen postprocessing pass to sort the captured fragments. *K-buffer* (*KB*) [17] and *stencil routed A-buffer* (*SRAB*) [18] increase performance by capturing up to k fragments in a single rendering pass. Read-modify-write hazards (RMWH) during KB updates can be fixed using a multipass variation of this algorithm (*MultiKB*) [19]. Conversely, SRAB avoids RMWH but is incompatible with hardware supported multisample antialiasing and stencil operations. Recently, Yu et al. [20] develop a sorting-free and memory-aware GPU-based k-buffer technique for their hair rendering framework.

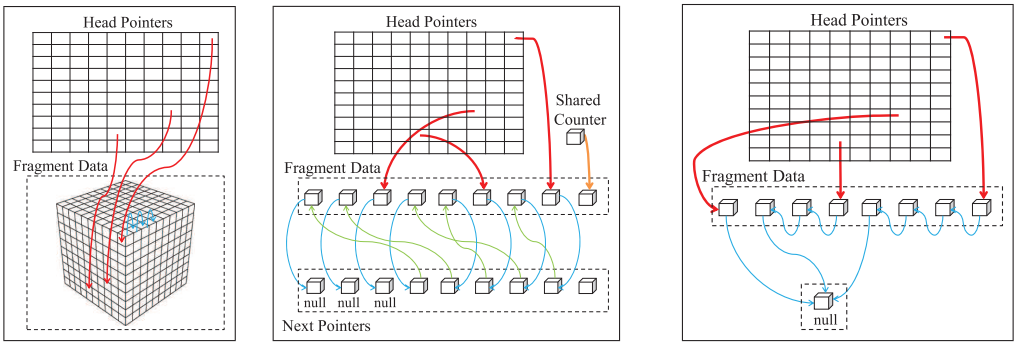
Liu et al. [21] introduce a complete CUDA-based rasterization pipeline (*FreePipe*) maintaining multiple *unbounded* fragments per pixel in real time. To supersede pixel level parallelism, Patney et al. [22] extend the domain of parallelization to individual fragments. However, both methods limit user to switch from the traditional graphics pipeline to a software rasterizer. FreePipe has been realized using modern OpenGL APIs (*FAB*) [23].

To alleviate the memory consumption of fixed-length structures, Yang et al. [24] proposed dynamic creation of *per-pixel linked lists* (*LL*) on the GPU. However, its performance degrades rapidly due to the heavy fragment contention and the random memory accesses when assembling the entire fragment list (see Fig. 3b).

To avoid limitations of constant-size array and linked-lists structures, *S-buffer* (*SB*) [25] organizes linearly memory into variable contiguous regions for each pixel as shown in Fig. 3c. The memory offset lookup table is computed in a parallel fashion exploiting sparsity in the pixel space. However, the need of an additional rendering step results in performance downgrade when compared to FAB.

3 Correcting Raster-Based Pipelines

We have investigated two approaches to treat fragment coplanarity in image space that can be applied to several depth peeling methods. Both approaches can be successfully integrated into the standard graphics pipeline and can take advantage of features such as multisample antialising (MSAA), GPU tessellation and geometry instancing. First, we introduce an additional term to the depth comparison



(a) *Fixed-size arrays* (b) *Linked-lists* (c) *Variable sequential regions*

Fig. 3. A-Buffer realizations: (a) and (c) structures pack pixel fragments physically close in the memory avoiding random memory accesses of (b) when accessing the entire fragment list. However, (a) allocates the same number of entries per pixel resulting at significant waste of storage and possible fragment overflows.

operator. Second, we present an efficient pipeline that can capture multiple coplanar fragments per depth layer by exploiting the advantages of buffer-based techniques. The core methodology for these extensions is explained in detail by applying it to the F2B depth peeling method (shader-like pseudocode is also provided). Then, a brief discussion is provided for applying it to the other depth peeling techniques.

We classify our algorithms based on the fragment hit ratio R_h, also called *robustness ratio* (i.e., the total number of extracted fragments over the total number of fragments). *Robust* algorithms succeed to capture all fragment information of a scene regardless of the coplanarity complexity (i.e., $R_h = 1$). On the other hand, *approximate* algorithms are not guaranteed to extract all fragments (i.e., $R_h \leq 1$). The main advantage of the latter is the superiority of the performance over the robust methods at the expense of higher memory space requirements.

We describe features and tradeoffs for each technique, pointing out GPU optimizations, portability, and limitations that can be used to guide the decision of which method to use in a given setting.

3.1 Robust Algorithms

We introduce two robust solutions for peeling the entire scene through a multipass rendering pipeline. The first one extracts a maximum of two coplanar fragments per iteration, implemented with a constant video-memory budget. Each iteration carries out one or more rendering passes depending on the algorithm. The second technique is able to capture at once all fragments that lie at the current depth layer before moving to the next one using dynamic creation of per-pixel linked lists.

3.1.1 Extending F2B

Overview. The classic F2B method [12] proposed a solution for sorting fragments by iteratively peeling off layers in depth order. Specifically, the algorithm starts by rendering the scene normally with a depth test, which returns the closest per-pixel fragment to the observer. In a second pass, previously extracted fragments are eliminated based on the depth value extracted during the last iteration (pass) returning the next nearest layer underneath. The iteration loop halts either if it reaches the maximum number of iterations set by the user or if no more fragments are produced. Fig. 4 (top row, red colored boxes) illustrates the consecutive color layers when depth peeling a *duck* model in a F2B direction.

Unfortunately, fragments with depth identical to the depth layer detected in the previous iteration are discarded and thus not considered in the underlying application. We introduce a robust coplanarity aware variation of F2B (*F2B-2P*) by adapting the F2B algorithm so as to peel all fragments located at the current depth before moving to the next depth layer. The basic idea of this technique is to use an extra rendering pass to count per pixel the (nonpeeled) coplanar fragments at a specific depth layer. To extract all coplanar fragments, we use the GPU autogenerated primitive identifier (*gl_PrimitiveID* [29]) that is unique per primitive geometric element and is inherited downward to fragments produced by this primitive. To decide, at iteration i, which fragments among the remaining coplanar ones to extract, we store the minimum and maximum identifiers (denoted as id^i_{min} and id^i_{max}, respectively) of these fragments:

$$id^i_{min} = \min\{id_f\}, id^i_{max} = \max\{id_f\}, \forall\, id_f \in \left(id^{i-1}_{min}, id^{i-1}_{max}\right).$$

We define as *nonpeeled* a fragment f that has a primitive identifier (denoted as id_f) in the range of the identifiers determined during the previous step $i-1$. This strategy guarantees that all coplanar fragments will survive since

$$id^1_{min} < id^2_{min} < \cdots < id^2_{max} < id^1_{max}.$$

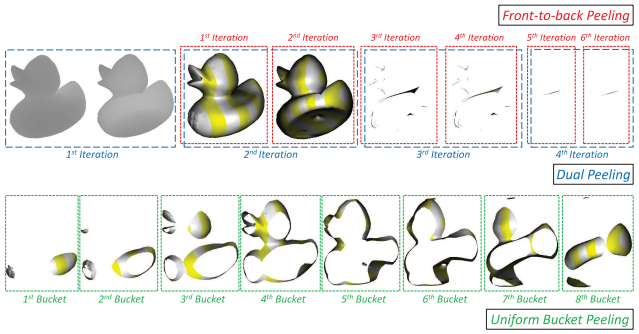


Fig. 4. The color-buffer result of each extracted layer when depth peeling is performed on a duck model using F2B, DUAL (top row), and BUN (bottom row).

Finally, a subsequent rendering pass extracts the fragment information of the corresponding identifier and decides whether the next depth layer underneath should be processed by accessing the counter information. If the counter is larger than 2, we have to keep peeling at the current layer since there is at least one more fragment to be peeled.

GPU implementation. We use one extra color texture (with internal pixel format RGBA_32F) to store the min/max identifiers at the RG channels and the counter at the A channel. Querying and counting for the identifier range and the counter may be performed in one rendering pass using 32-bit floating point blending operations. When computing the output color, two blending operations are used: MAX for the RGB portion of the output color, and ADD for the alpha value. To query the minimum identifier using maximum blending, we store the negative identifier of the primitive.

A second rendering pass is employed to simultaneously extract the fragment attributes and the next depth layer exploiting multirender targets (MRT). Depth testing is again disabled while the blending operation is set to MAX for all components of the MRT. The custom (under blending) min depth test is implemented adapting the idea of the min/max depth buffer of DUAL [13] with the use of a color texture (R_32F). If the counter is less or equal than 2, then we have extracted all information in this layer. We move on to the next one by keeping (blending) the fragments with depth greater than the previously peeled layer. Otherwise, we discard all fragments that do not match the processing depth. The min and max color textures (RGBA_8) are initialized to zero and updated only by the fragments that correspond to the captured identifiers. The algorithm guarantees that no fragment is extracted twice. Initially, we render the scene so as to efficiently capture only the closest depth layer before proceeding with the counter and identifier computation pass.

The details of this method are shown in Algorithm 1, where OUT.xxx denote the output MRT fields, IN.xxx the input texture fields (initialized to zero), and FR.xxx the attributes of each fragment.

Algorithm 1. F2B-2P Depth Peeling

/* 1st Rendering Pass using MAX Blending */
if FR.depth $< -$ IN.depth **then**
  discard ;
end if
OUT.color $_{min} \leftarrow (-$ IN.id $_{min} ==$ FR.id) ? FR.color : 0.0 ;
OUT.color$_{max} \leftarrow (\,$IN.id$_{max} ==$ FR.id) ? FR.color : 0.0 ;
OUT.depth $\leftarrow$ (IN.counter > 2 *or* $-$ IN.depth $\neq$ FR.depth) ? $-$ FR.depth : -1.0;
/* 2nd Rendering Pass using MAX and ADD Blending */
if (IN.counter ≤ 2 *or* FR.id $\in (-$IN.id$_{min}$, IN.id$_{max}))$ *and* ($-$IN.depth $==$ FR.depth) **then**
  OUT.id$_{min} \leftarrow -$FR.id ;
  OUT.id$_{max} \leftarrow$ FR.id ;
  OUT.counter $\leftarrow 1.0$;
else
  discard ;
end if

Discussion. The drawback of this technique is the increase of the rasterization work as compared to the original F2B algorithm by a factor of 2. Moreover, the requirement for per-pixel processing via blending may result to a rasterization bottleneck after multiple iterations.

Pre-Z pass [26] is a general rendering technique for enhancing performance despite the additional rendering of the scene. Specifically, a *double-speed* rendering pass is first employed to fill the depth buffer with the scene depth values by depth testing and turning off color writing. Shading the scene with depth write disabled results on enabling *early-Z culling*; a component which automatically rejects fragments that do not pass the depth test. Therefore, no extra shading computations are required.

We introduce the *F2B-3P* technique, an F2B-2P variant which follows the above pipeline. The idea is to carry out the first rendering pass of F2B-2P in two passes. A double-speed depth rendering pass is performed to compute the (next) closest depth layer. Then, by exploiting early-Z culling, we perform counting and identifier queries by enabling blending, turning off depth writing and changing depth comparison direction to EQUAL. The difference from the second pass of Algorithm 1 is that depth comparisons inside the shader are not needed, thus minimizing the number of texture accesses. Shading is performed in a subsequent pass by matching the fragments of the extracted identifier set without modifying pixel-processing modes (blending or Z-test) of the previous pass. This modified GPU-accelerated version uses the same video memory resources and performs slightly better than its predecessor in some cases despite the cost of the extra rendering pass.

3.1.2 Extending DUAL

Overview. DUAL depth peeling [13] increases performance by applying the F2B method for the F2B and the back-to-front directions simultaneously. Due to the unavailable support of multiple depth buffers on the GPU, a custom min-max depth buffer is introduced. In every iteration, the algorithm extracts the fragment information which match the min-max depth values of the previous iteration and performs depth peeling on the fragments inside this depth set. An additional rendering pass is needed to initialize depth buffer to the closest and the further layers. Fig. 4 (top row, blue-colored boxes) shows that the number of rendering iterations needed is reduced to half when simultaneous bidirectional depth peeling is used.

To handle coplanarity issues raised at both directions, we have developed a variation of DUAL (*DUAL-2P*), which adapts the F2B-2P algorithm for working concurrently in both directions.

Discussion. Developing manually a min-max depth buffer requires turning off the hardware depth buffer. Thus, we cannot benefit from advanced extensions of the graphics hardware in the DUAL workflow (such as the ones used for F2B-3P). DUAL-2P depth peeling as compared to the F2B-2P and F2B-3P variations reduces the rendering cost to half by extracting up to four fragments simultaneously. The burden for providing this feature is that it requires twice as much memory space.

3.1.3 Combining F2B and DUAL with LL

Overview. Yang et al. [24] introduced a method to efficiently construct highly concurrent per-pixel linked lists via atomic memory operations on modern GPUs. The basic idea behind this algorithm is to use one buffer to store all linked-list *node* data and another buffer to store *head* offsets that point the start of the linked lists in the first buffer. A single shared counter (*next*) is atomically incremented to compute the mapping of an incoming fragment, followed by an update of the head pointer buffer to reference the last rasterized fragment (see Fig. 3b).

Although fast enough for most real-time rendering applications, the creation of these lists may incur a significant burden on video memory requirements when the number of fragments to be stored increases significantly. We propose two efficient multipass coplanarity-aware depth peeling methods (*F2B-LL* and *DUAL-LL*) by combining F2B and DUAL with LL. The idea is to store *all* fragments located at the extracted depth layer(s) using linked-list structures. Coplanarity issues can be easily handled using this technique without wasting any memory.

*GPU implementation.*The rendering workflow of F2B-LL consists of two passes: First, a double speed depth pass is carried out enabling Z-buffering. Second, we construct linked lists of the fragments located at the captured depth by changing depth comparison direction to EQUAL and turning off depth writing (which results at early-culling optimizations).

The details of this method are shown in Algorithm 2, where LL.xxx denote the linked list fields, IN.xxx the input texture fields (initialized to zero), and FR.xxx the attributes of each fragment.

Algorithm 2. F2B-LL Depth Peeling

```
/* 1st Rendering Pass using LESS/EQUAL Z-test
   comparison */
1: if FR.depth <= IN.depth then
2:     discard ;
3: end if
/* 2nd Rendering Pass using EQUAL Z-test comparison
*/
1: LL.next ↞ LL.next+1 ;
2: LL.head[LL.next] ← IN.head ;
3: LL.node[LL.next] ← FR.color ;
4: IN.head ← LL.next ;
// where ↞ denotes an atomic memory operation
```

Construction of a min/max depth buffer for DUAL-LL disables depth testing, which results in an increase of the number of texture accesses and per pixel shader computations. In the context of storage, one extra screen image is allocated for the head evaluation of the back layer. To avoid a slight increase of contention due to the extensive attempts of accessing the shared memory area from both front and back fragments, an additional address counter variable for back layers is used ($next_{back}$). Conflicts between front and back fragments are avoided by employing an inverse memory mapping strategy for the fragments extracted in the back-to-front direction. Specifically, we route them starting from the end of the node buffer toward the beginning.

Discussion. The key advantage of these techniques over the rest of the robust methods introduced in this paper is that they can handle fragment coplanarity of arbitrary length per pixel in one iteration. This results in a significant decrease of the rendering workload. Practically, contention from all threads trying to retrieve the next memory address for accessing the corresponding data has been reduced since coplanarity occurs only for a small number of cases as compared to the original LL algorithm.

Despite the fact that order of thread execution is not guaranteed, list sorting is not necessary since all captured fragments are coplanar. Moreover, F2B-LL rendering pipeline is boosted by hardware optimization components. All these lead to efficiently usage of GPU memory and performance increase. Conversely, random memory accesses and atomic updating of *next* counter(s) from all fragment threads may lead to a critical rasterization stall. Finally, the necessity of atomic operations for GPU memory—available only in the state-of-the-art graphics hardware and APIs—makes it nonportable to other platforms such as mobile, game consoles, and so on.

3.1.4 Combining BUN with LL

Overview. Liu et al. [14] presents a multilayer depth peeling technique achieving partially correct depth ordering via bucket sort on the fragment level. To approximate the depth range of each pixel location, a quick rendering pass of the scene's bounding box is initially employed. Fig. 4 (bottom row, green colored boxes) illustrates the peeling output for each bucket for a scene divided into eight uniform intervals.

Despite the accurate depth-fighting feature of the above proposed extensions, their performance is rather limited when the depth complexity is high due to their strategy to perform multiple iterations. Furthermore, as mentioned above, LL may exhibit some serious performance bottlenecks when 1) the total number of generated fragments (storing process) or 2) the number of per-pixel fragments (sorting process) increases significantly. To alleviate the above limitations, we propose a single-pass coplanarity-aware depth peeling architecture combining the features of BUN and LL. In this variation, we uniformly split the depth range of each scene and assign each subdivision to one bucket. Then, we concurrently (in parallel) store all fragment information in each bucket using linked lists.

GPU implementation. Due to the current shader restrictions, we can divide the depth range into $five$ uniformly consecutive subintervals. A *node* buffer (RGBA_8) is used to store all linked-list fragment data from all buckets. We explore a nonadaptive scheme where all buckets can handle the same number of rasterized fragments. The location of the next available space in the node buffer is managed through $five$ global unsigned int address counters ($[next_{b_0}, \ldots, next_{b_4}]$). Each pixel contains $five$ head pointers (R_32UI), one for each bucket, containing the last node ($[head_{b_0}, \ldots, head_{b_4}]$) it processed. Each incoming fragment is mapped to the bucket corresponding to its depth value. The address counter of the corresponding bucket is incremented to find the next available offset at the node buffer. The head pointer of the bucket is lastly updated to point to the previously stored fragment. After the complete storage of all fragments, a postsorting mechanism is carried out in each bucket sorting fragments by their depth.

Discussion. The core advantage of BUN-LL is the superiority in terms of performance over the rest of the proposed methods due to its single-pass nature. BUN-LL is faster than LL and exhibits time complexity comparable to SB and FAB. However, unused allocated memory from empty buckets as well as fragment overflow from overloaded ones may arise for scenes with nonuniform depth distribution.

3.2 Approximate Algorithms

To alleviate the performance downgrade of multipass techniques, we have explored per-pixel fixed-sized vectors [17], [23] for capturing a bounded number of coplanar fragments. The core advantage of this class of methods is the superiority of performance in the expense of excessive memory allocation and fragment overflow.

3.2.1 Combining F2B and DUAL with FAB/KB

Overview. Bounded buffer-based methods store fragment data in a global memory array using a fixed-sized array per pixel (see Fig. 3a). A per-pixel offset counter indicates the next available address position for the incoming fragment. After a complete insertion in the storage buffer, the counter is atomically incremented.

We introduce a solution for combining FAB/KB with F2B and DUAL (*F2B-B*, *DUAL-B*) to partially treat fragment coplanarity. The idea is to adapt the previously described core methodology of linked lists by exploiting bounded buffer architectures for storage.

GPU implementation. Similar to FAB, constant length vectors are allocated to capture the fragment data for each pixel. In the case of DUAL-FAB, we have to allocate two buffer arrays for front and back peeling at the same time. Without loss of generality, we use the same length for both buffers. To support efficiently this approach in commodity hardware, we may employ a KB framework in place of FAB. While KB is restricted by MRT to peel a maximum of eight fragments, data packing may be used to increase the output (and reduce memory cost) by a factor of 4. Note that there is no need for presorting and postsorting, since we peel fragments placed at same memory space (RMWH-free).

The details of combing F2B with KB and FAB are shown in Algorithm 3, where A.xxx is used to define the fixed-size data array, IN.xxx the input textures (initialized to zero), FR.xxx the attributes of each fragment, and TMP.xxx the temporary variables.

Algorithm 3. F2B-B Depth Peeling

```
  /* using KB: F2B-KB */
1: for i = 0 to A.length do
2:     if A[i] == 0 then
3:         A[i] ← FR.color; break ;
4:     end if
5: end for
  /* using FAB: F2B-FAB */
1: TMP.counter ← IN.counter+1 ;
2: IN.counter ↞ (TMP.counter == A.length) ? 0 :
   TMP.counter ;
3: A[TMP.counter−1].color ← FR.color ;
  // where ↞ denotes an atomic memory operation
```

*Discussion.*The major advantage of this idea is that by updating atomically only per-pixel counters no access of shared memory is attempted, which results in significant performance upgrade. Performance is degraded when KB is used due to concurrent updates, but this is a useful option when advanced APIs are not available. SRAB is a promising option but in this context it is ruled out because it cannot support MSAA, stencil and data packing operations. Note that attribute packing except from extra memory requirements requires additional shader computations and imposes output precision limitations on fragment data (32 bit).

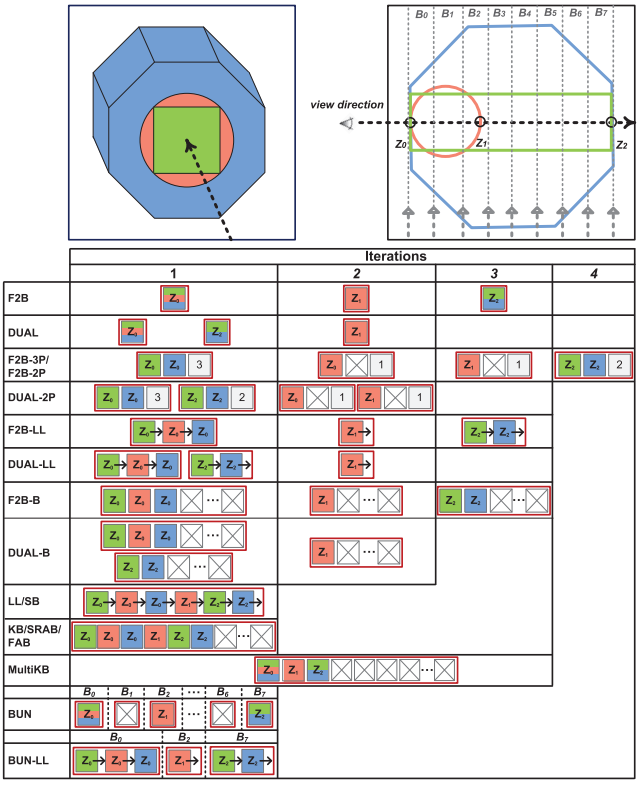


Fig. 5. Overview of peeling results for our proposed methods and their predecessors. Z_0, Z_1, and Z_2 indicate the depth layers captured by ray casting (black dashed line) and $B_0, B_1, \ldots, B_7$ the uniformly distributed buckets. Each column shows the produced output of each method for the corresponding iteration: extracted fragment(s) painted with the color of an object and coplanarity counters. Squares painted with more than one color demonstrate z-fighting artifacts (is undefined which fragment might win the z-test). To distinguish between fragments of the same object, we have included their depth value to their associated square.

A simplified example that illustrates the peeling behavior of the base methods and our proposed extensions is shown in Fig. 5. The scene consists of three objects of different color with the following rendering order: green, coral, and blue resulting in the green having the smallest and blue the largest primitive identifiers. A ray starting from an arbitrary pixel hits the scene at three depth layers, where three and two fragments overlap at the first and the third layer, respectively.

3.3 GPU Optimizations for Multipass Rendering

The previous sections introduced extensions of the multipass depth peeling algorithms to cope with coplanar fragments. In this section, we propose an optimization making use of various features of modern GPUs so as to improve the performance when multipass rendering is performed on multiple objects. Inspired by the *occlusion culling* mechanism [27] (where geometry is not rendered

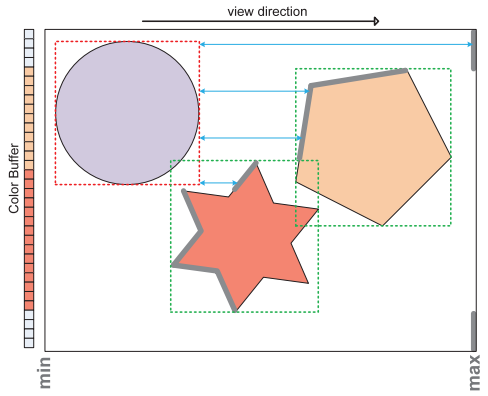


Fig. 6. A sphere is efficiently culled and thus not needed to be rendered for the remaining iterations because its bounding box lies entirely behind the current depth buffer (*thick gray line strips*).

when it is hidden by objects closer to the camera), we propose to avoid rendering objects that are completely peeled from previous iterations. By skipping the entire rendering process for a completely peeled object, we reduce the rendering load of the following rendering passes.

Similarly to occlusion culling, we substitute a geometrically complex object with its bounding box. If the bounding box of the object ends up entirely behind the last captured contents of depth buffer, we may cull this object at the geometry level (see Fig. 6). This is easily realized by hardware occlusion queries. Due to the observation that objects that are culled during a specific iteration will be always culled in the successive ones, we reuse the results of the occlusion queries from previous iterations [28]. This leads to a reduction of the number of issued queries eliminating CPU stalls and GPU starvation.

Finally, we avoid the synchronization cost between the CPU and GPU required to obtain the occlusion query result, be using *conditional rendering* [29]. Note that conditional rendering can also be used to automatically halt the iterative procedure of multipass rendering methods.

4 Results

We present an experiment analysis of our extensions focusing on performance, robustness, and memory requirements under different testing scenarios. For the purposes of comparison, we have developed *F2B2*; a two-pass variation of F2B that uses double speed Z-pass and early Z-culling optimizations. Our methods successfully integrate into the standard graphics pipeline and take advantage of features such as multisample rendering, GPU-based tessellation and instancing. Methods that do not exploit the FAB or the LL structures can be used in older hardware. All methods are implemented using OpenGL 4.2 API and performed on an NVIDIA GTX 480 (1.5 GB memory). The shader source code has been also provided as supplementary material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TVCG.2012.300.

We have applied our coplanarity-aware peeling variants on several depth-sensitive applications (transparency effects, wireframe rendering, CSG operations, self-collision detection, coplanarity detection) demonstrating the importance of accurately handling scenes with z-fighting (see Figs. 12 and 13). We further provide a video demonstrating the rendering quality of one of our robust variants (F2B-FAB) for various coplanarity critical applications. The rest of our robust variants yield similar visual results.

Table 1 presents a comparative overview of all multifragment raster-based methods with respect to memory requirements, compatibility with commodity and state-of-the-art hardware, rendering complexity, coplanarity accuracy, and other features.

TABLE 1
Comprehensive Comparison of Multilayer Rendering Methods and Our Coplanarity-Aware Variants

Algorithm		*Max layers*	*Rendering passes*		*GPU optimizations*			*Video-memory per pixel*		*Peeling accuracy*		*Sorting needed*		*MSAA*
Acronym	*Description*	*Per iteration*		*Total*	*Conditional rendering*	*Double speed z- pass*	*Early z-culling*	*Old API*	*Modern API*	*Handles coplanarity*	*Robustness ratio*	*on primitives*	*on fragments*	
F2B	*Front-to-back depth peeling* [12]	1	1	D	x	x	x	3		x	D/C(Zall)	x	x	√
F2B2	*Two-pass F2B depth peeling*		2	2D	√	√	√							
F2B-2P	*Two-pass Z-fighting free F2B*	2		C(Zall)		x	x	12	10	√	1			
F2B-3P	*Three-pass Z-fighting free F2B*		3	3C(Zall)/2	√	√	√							
F2B-LL	*Z-fighting free F2B using Linked Lists*	C(Z)	2	2D				x	2C+4					
F2B-B	*Z-fighting free F2B using fixed-size Buffers*	K ; 4K						K+3 ; 4K+3	K+4 ; 4K+4		$\sum\{K/C(Zi)\}$; $4\sum\{K/C(Zi)\}$			
DUAL	*Dual Depth Peeling* [13]	2	1	D/2+1	x	x	x	6		x	D/C(Zall)	x	x	√
DUAL-2P	*Two-pass Z-fighting free DUAL*	4	2	C(Zall)/2+1	√			24	20	√	1			
DUAL-LL	*Z-fighting free DUAL using Linked Lists*	C(Zf,Zb)		D				x	2C(Zf,Zb)+6					
DUAL-B	*Z-fighting free DUAL using fixed-size Buffers*	K ; 4K						K+4 ; 4K+4	K+6 ; 4K+6		$\sum\{K/C(Zi)\}$; $4\sum\{K/C(Zi)\}$			
KB	*K-Buffer* [17]	K ; 2K	1	1 to D/K				2K+2 ; 4K+2			K/C(Zall) to 1 ; 2K/C(Zall) to 1	√	√	
MultiKB	*Multipass K-Buffer* [19]		1...K ; 2K	1 to K ; 1 to 2K	x					x				
SRAB	*Stencil Routed A-Buffer* [18]	K	1	1 to D/K	√	x	x	3K+2		√	K/C(Zall) to 1			x
BUN	*Uniform Bucket Peeling* [14]	2K ; 4K	1	D/(2K) to D/2 ; 1				4K+2		x	D/C(Zall)		x	
BUN-LL	*Z-fighting free BUN using Linked Lists*	all	1	1	x			x	3C(Zall)+8 to 15C(Zall)+8		*overflow* to 1		√	
BAD	*Adaptive Bucket Peeling* [14]	4K	4	4				6K+3		x	4K/C(Zall)	x	x	√
FreePipe/FAB	*A-Buffer using fixed-size Arrays* [21,23]	all	1	1				x	2D+2					
LL	*Linked-list based A-Buffer* [24]								3C(Zall)+3	√	*overflow* to 1		√	
SB	*S-Buffer: Sparsity-aware A-Buffer* [25]		2	2					2C(Zall)+3					

K = buffer size (max=8 for all except from FreePipe/FAB), {color, depth} attribute of fragment = {32bit, 32bit}. Video-memory is measured in *mb(=4 bytes)*.

D = max{depth}, C(Z) = [# of coplanar fragments at depth Z] , C(Zall) = $\sum\{C(Zi)\}$, C = max{C(Zi)}, where C(Z) ≥ 1 and C(Zall) ≥ D. *overflow* = 1 - (max_memory/needed_memory).

In A ; B, A denotes the layers/memory/ratio/sorting for the basic method and B for the variation using attribute packing.

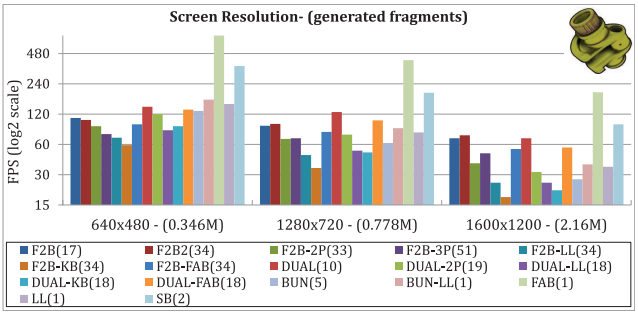


Fig. 7. Performance evaluation in FPS (log_2 scale) on a scene where no fragment coplanarity is present at different rendering dimensions. Our FAB-based extensions exhibit slightly worse performance than their base methods (10 percent in average). Rendering passes carried out for each method are shown in brackets.

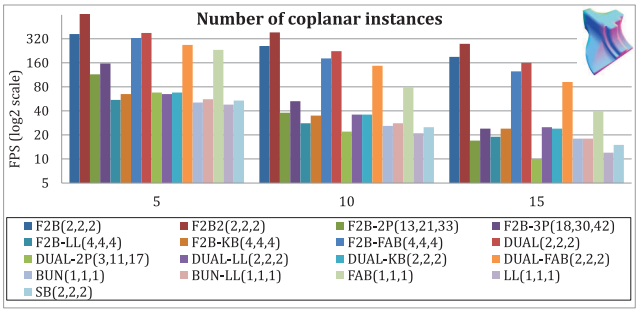


Fig. 8. Performance evaluation in FPS (log_2 scale) on a scene with varying coplanarity of fragments. FAB extensions outperform other proposed alternatives and are slightly affected by the number of overlapping fragments. Rendering passes performed for each method are shown in brackets.

4.1 Performance Analysis

We have performed an experimental performance evaluation of all our methods against competing techniques using a collection of scenes under four different configurations. Except from the first scene that is evaluated under different image resolutions, the rest of the tests are rendered using a $1{,}280 \times 720$ (HD Ready) viewport.

4.1.1 Impact of Screen Resolution

Fig. 7 shows how the performance scales by increasing the screen dimensions when rendering a *crank* model (10 K primitives) whose layers varies from 2 to 17 and *no* coplanarity exist. In general, we observe that our variants perform slightly slower than their predecessors due to the extra rendering passes (around 30 percent in average). Our dual variants perform faster at low resolutions as compared to the corresponding F2B ones because they need half the rendering passes. Similar performance behavior moving from low-to-high screen dimensions is observed between F2B-2P and F2B-3P. GPU optimizations becomes meritorious when image size is increasing rapidly.

FAB and SB are highly efficient in this scenario due to the low rate of used pixels that require heavy postsorting of their captured fragments. DUAL-FAB has the best performance from all proposed multipass variants, which is slightly worst than DUAL (from 6 percent (low resolution) to 18 percent (high resolution)). However, it achieves speed regression by a factor of 2 to 4 as compared to the SB and FAB methods, respectively. This is reasonable since we iteratively render the scene up to 18 times to extract all layers. We further observe that DUAL-2P and DUAL-KB perform quite well in low-screen resolution but exhibit significant performance downgrade in the higher ones. Finally, rendering bottlenecks appear in all LL-based methods when the resolution is increased due to higher fragment serialization.

4.1.2 Impact of Coplanarity

Fig. 8 illustrates performance for rendering overlapping instanced fandisk objects (1.4K primitives). We observe that F2B-3P outperforms F2B-2P and DUAL-2P, enhanced by the full potential of GPU optimizations. Similar behavior is observed for F2B-FAB as compared to its corresponding dual variation. Conversely, DUAL-LL performs better than F2B-LL alleviating the increased fragment contention at high instancing.

FAB extensions exhibit improved performance as compared to constant-pass ones despite of they have to carry out multiple rendering iterations. This is reasonable since these buffers have to sort the captured fragments resulting in a rendering stall. Finally, BUN-LL is slightly superior than LL and SB, but again is not suitable for scenes with high concentration of fragments in small depth intervals.

4.1.3 Impact of High Depth Complexity

Fig. 9 illustrates performance comparison of the constant-pass accurate peeling solutions when rendering three uniformly distributed scenes that consist of high depth complexity: sponza (279K primitives), engine (203.3 K primitives), hairball (2.85M primitives). We observe the superiority of our BUN-LL over the LL and SB methods regardless of the number of generated fragments due to the reduced demands for per-pixel post-sorting of the captured fragments. On the other hand, thread contention in the BUN-LL storing process results at a performance downgrade as compared with FAB when the rasterized fragments are rapidly increased.

4.1.4 Impact of Geometry Culling

Fig. 10 illustrates how the performance scales when our geometry culling is exploited at three representative F2B peeling methods under a set of increasing peeling iterations (similar behavior is observed for the rest variations). The scene consists of three nonoverlapping, aligned at Z-axis,

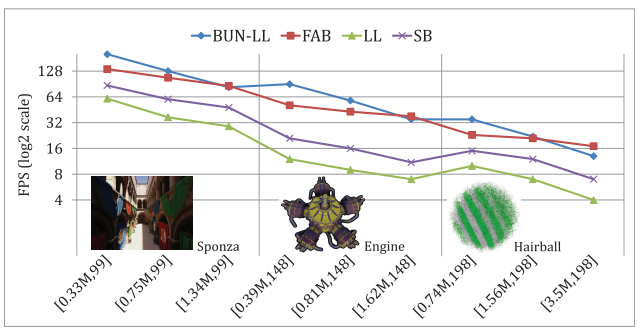


Fig. 9. Performance evaluation in FPS (log_2 scale) on three uniformly distributed scenes with varying number of fragments and high depth complexity (shown in brackets, respectively). Our BUN-LL outperforms the other buffer-based methods when the fragment capacity remains at low levels.

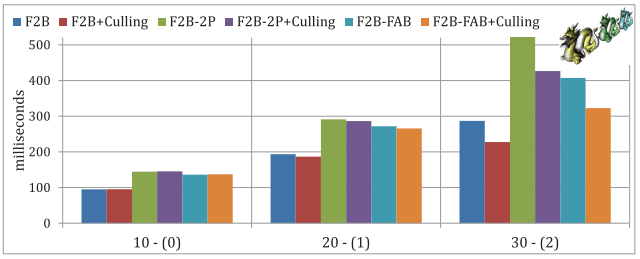


Fig. 10. Performance evaluation in milliseconds after F2B layer peeling a scene without and with enabling our geometry-culling mechanism. The number of completely peeled models for each peeling iteration is shown in brackets.

dragon models (870 K primitives, 10 depth complexity). The scene is rendered from a viewport that the third dragon is occluded by the second one, which is similarly hidden by the first. We observe that all F2B testing methods are exponentially enhanced by the use of our early-z geometry culling process when the number of completely peeled objects is increasing. Note that when we have not completely peeled any instance, the additional cost of our culling process slightly affects performance (0.01 percent).

4.2 Memory Allocation Analysis

Fig. 11 illustrates evaluation in terms of storage consumption for a scene with varying number of generated fragments (defined by the combination of screen resolution, depth complexity, and fragment coplanarity). An interesting observation is the high GPU memory requirements of FAB due to its strategy to allocate the same memory per pixel. BUN-LL, LL, and SB require less storage resources by dynamically allocating storage only for fragments that are actually there. However, it will lead at a serious overflow as the number of the generated fragments to be stored increases rapidly.

On the other hand, our multipass depth peeling extensions outperform the unbounded buffer-based methods even at high coplanarity scenes. We also observe that robust F2B-2P and F2B-3P methods require slightly less storage than the approximate F2B-KB. Video-memory consumption blasts off to high levels, when data packing is employed for correct capturing high fragment coplanarity. Note that methods that exploit the F2B strategy require less memory resources when compared to the dual-direction ones. The same conclusions may be obtained from the formulations of Table 1.

4.3 Robustness Analysis

4.3.1 *Impact of Coplanarity*

From Table 1, we observe that robust variations are able to accurately capture the entire scene regardless of the depth and coplanarity complexities. F2B and DUAL peeling reach their peak when no coplanarity is present. However, robustness is significantly downgraded due to their inability to capture overlapping areas. Multipass bucket peeling and

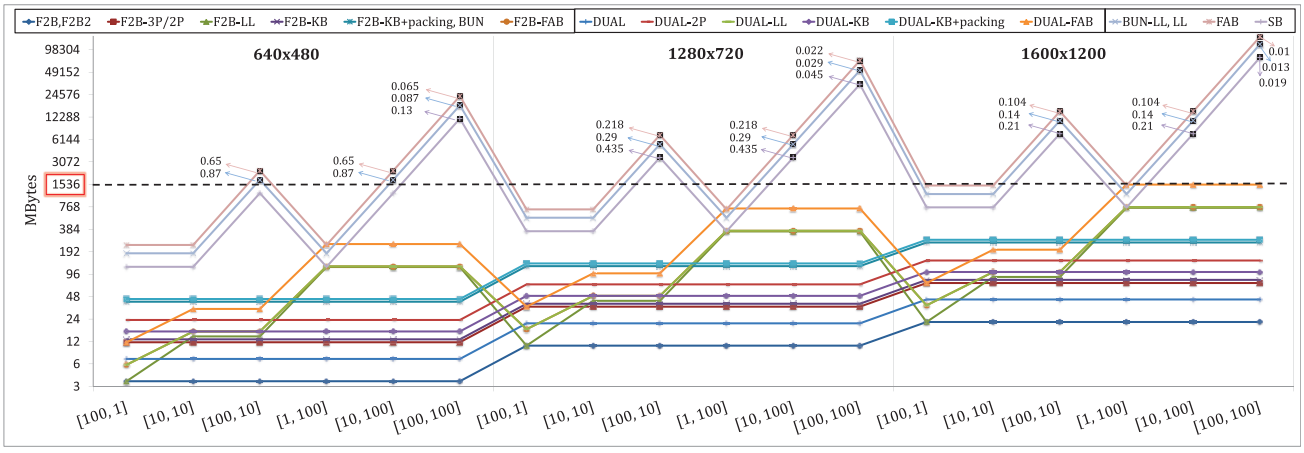
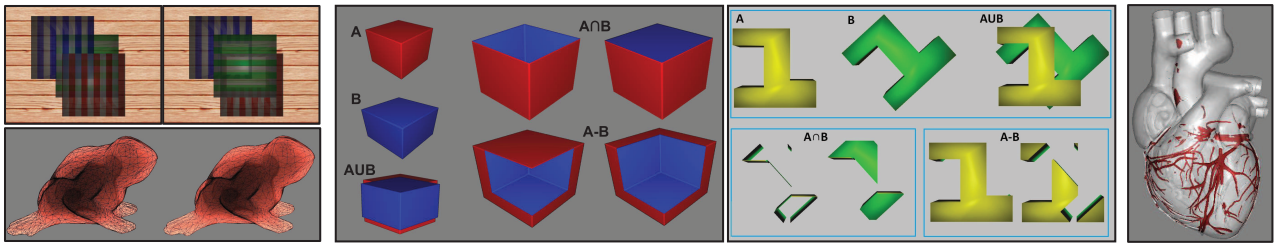


Fig. 11. Robustness comparison based on memory allocation/overflow (log_2 scale) of a scene with varying resolution and [depth, coplanarity] complexity. Our variants does not consume more than the maximum storage of Nvidia GTX 480 graphics card (dashed line). Note the low robustness ratio of the buffer-based solutions due to the memory overflow.



(a) (b) (c)

Fig. 12. Illustrating the image superiority of our extensions over the base methods in several depth-sensitive applications. (a) (top) Order-independent transparency on three partially overlapping cubes with and without Z-fighting, (bottom) wireframe rendering of a translucent frog model with and without Z-fighting. (b) CSG operations rendering without and with coplanarity corrections. (c) Self-collided coplanar areas are visualized with red color.

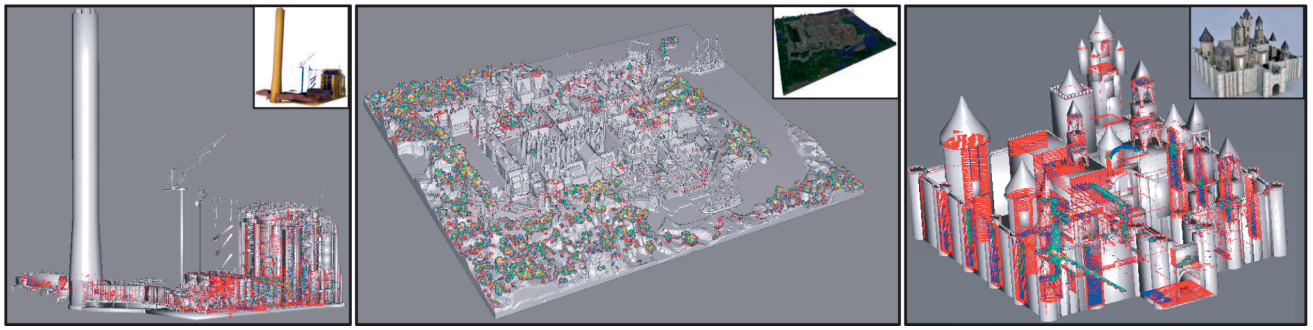


Fig. 13. Image-based coplanarity detector. (left) *Power plant* ($R_h = 0.98, C_p = 0.285$), (middle) *rungholt* ($R_h = 0.9, C_p = 0.48$) and (right) *castle* ($R_h = 0.88, C_p = 0.81$) scenes are visualized based on the total per-pixel fragment coplanarity: $gray = none$, $red = 2$, $blue = 3$, $green = 4$, $cyan = 5$, $aquamarine = 6$, $fuchsia = 7$, $yellow = 8$, $brown = 9$. C_p is the average probability for a pixel p to suffer from fragment coplanarity when rendering with the F2B.

its single-pass packed version present similar behavior. Approximate buffer-based alternatives (maximum peeled fragments: without packing ($K = 8$) and with packing ($K = 32$)) are suitable to correctly handle up to 8 or 32 coplanar fragments. Peeling with KB, MultiKB, and SRAB result at memory overflow (hardware restricted to 8 or 16 if attribute packing is used) failing to capture more fragment information. If the scene is presorted by depth, multiple rendering with these buffers will improve robustness. Finally, BUN-LL, FAB, LL, and SB perform robustly when fragment storage does not result in memory overflow.

4.3.2 Impact of Memory Overflow

Fig. 11 shows the needed storage allocated by the memory-unbounded buffer solutions under a scene with varying number of generated fragments. Without loss of generality, we assume that the percentage of pixels covered on the screen is 50 percent and all pixels have the same depth complexity. Robustness ratio is closely related to memory allocation for these methods (see also Table 1). To avoid memory overflow (illustrated by black markers), we have to allocate less storage than we actually need leading at a significant fragment information loss. BUN-LL, FAB, LL, and SB robustness is significantly downgraded when the number of generated fragments exceeds a certain point. Conversely, we observe that our buffer-based extensions perform precisely, allocating less than the maximum storage of the testing graphics card under all rendering scenarios.

4.4 Discussion

FAB has the best performance in conjunction with robust peeling but comes with the burden of extremely large memory requirements. SB alleviates most of the wasteful storage resources running at high speeds, but cannot avoid the unbounded space requirement drawback. Both methods necessitate per-pixel depth sorting resulting at comparable frame rates with BUN-LL when the number of stored fragments per pixel is high and uniformly distributed.

Multipass peeling with primitive identifiers is the best option when accuracy and memory are of utmost importance. FAB extensions are shown to offer a significant speed up over LL variations with satisfactory approximate (or precise when coplanarity is maintained at low levels) results. However, memory limitations should be carefully considered. When modern hardware is not available, KB variations might be used to approximate scenes with high coplanarity in the entire depth range.

It is preferred to use F2B extensions for handling scenes with low detail under high resolutions. On the other hand, dual extensions performs better when rendering highly tessellated scenes at low screen dimensions.

5 Conclusions and Future Work

Fragment coplanarity is a phenomenon that occurs frequently, unexpectedly and causes various unpleasant and unintuitive results in many applications (from visualization to content creation tools) that are sensitive to robustness. Several (approximate or exact) extensions to conventional single and multipass rendering methods have been introduced accounting for coincident fragments. We have also included extensive comparative results with respect to algorithm complexity, memory usage, performance, robustness, and portability. A large spectrum of multifragment effects have been considered and used for illustrating the detected differences. We expect that the suite of features and limitations offered for each technique will provide a useful guide for effectively addressing coplanarity artifacts.

Further directions may be explored for tackling the problem of coplanarity in rasterization architectures. To reduce bandwidth demand of the rendering operations and increase locality of memory accesses, tiled rendering [30] may be exploited. Determining the set of elements that are not visible from a particular viewpoint, due to being occluded by elements in front of them may affect the performance of the multipass peeling methods [27], [28]. Finally, a hybrid technique [11] is an interesting option that should be investigated further. To this end, one may seek a modified form of peeling which efficiently captures a sequence of layers when coplanarity is not presented followed by on demand peeling of overlapping fragments.

References

[1] "OpenGL SDK 10: Simple Depth Float,"C. NVIDIA, 2008.
[2] R. Herrell, J. Baldwin, and C. Wilcox, "High-Quality Polygon Edging," *IEEE Computer Graphics and Applications*, vol. 15, no. 4, pp. 68-74, July 1995.
[3] M. Maule, J.L. Comba, R.P. Torchelsen, and R. Bastos, "A Survey of Raster-Based Transparency Techniques," *Computers & Graphics*, vol. 35, no. 6, pp. 1023-1034, 2011.

[4] F. Cole and A. Finkelstein, "Partial Visibility for Stylized Lines," *Proc. Sixth Int'l Symp. Non-Photorealistic Animation and Rendering (NPAR '08),* pp. 9-13, 2008.
[5] J. Rossignac, I. Fudos, and A.A. Vasilakis, "Direct Rendering of Boolean Combinations of Self-Trimmed Surfaces," *Proc. Symp. Solid and Physical Modeling (SPM '12),* 2012.
[6] S. Busking, C.P. Botha, L. Ferrarini, J. Milles, and F.H. Post, "Image-Based Rendering of Intersecting Surfaces for Dynamic Comparative Visualization," *Visual Computers,* vol. 27, pp. 347-363, May 2011.
[7] P.V. Sander, D. Nehab, and J. Barczak, "Fast Triangle Reordering for Vertex Locality and Reduced Overdraw," *ACM Trans. Graphics,* vol. 26, no. 3, article 89, 2007.
[8] E. Sintorn and U. Assarsson, "Real-Time Approximate Sorting for Self Shadowing and Transparency in Hair Rendering," *Proc. Symp. Interactive 3D Graphics and Games (I3D '08),* pp. 157-162, 2008.
[9] D. Wexler, L. Gritz, E. Enderton, and J. Rice, "GPU-Accelerated High-Quality Hidden Surface Removal," *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graphics Hardware (HWWS '05),* pp. 7-14, 2005.
[10] N.K. Govindaraju, M. Henson, M.C. Lin, and D. Manocha, "Interactive Visibility Ordering and Transparency Computations among Geometric Primitives in Complex Environments," *Proc. Symp. Interactive 3D Graphics and Games (I3D '05),* pp. 49-56, 2005.
[11] N. Carr and G. Miller, "Coherent Layer Peeling for Transparent High-Depth-Complexity Scenes," *Proc. 23rd ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware,* pp. 33-40, 2008.
[12] C. Everitt, "Interactive Order-Independent Transparency," technical report, Nvidia Corporation, 2001.
[13] L. Bavoil and K. Myers, "Order Independent Transparency with Dual Depth Peeling," technical report, Nvidia Corporation, 2008.
[14] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu, "Efficient Depth Peeling via Bucket Sort," *Proc. First ACM Conf. High Performance Graphics (HPG '09),* pp. 51-57, 2009.
[15] E. Sintorn and U. Assarsson, "Hair Self Shadowing and Transparency Depth Ordering Using Occupancy Maps," *Proc. Symp. Interactive 3D Graphics and Games (I3D '09),* pp. 67-74, 2009.
[16] A.A. Vasilakis and I. Fudos, "Z-Fighting Aware Depth Peeling," *Proc. ACM SIGGRAPH,* 2011.
[17] L. Bavoil, S.P. Callahan, A. Lefohn, J.L.D. Comba, and C.T. Silva, "Multi-Fragment Effects on the GPU using the K-Buffer," *Proc. Symp. Interactive 3D Graphics and Games (I3D '07),* pp. 97-104, 2007.
[18] K. Myers and L. Bavoil, "Stencil Routed A-Buffer," *Proc. ACM SIGGRAPH,* 2007.
[19] B. Liu, L.-Y. Wei, Y.-Q. Xu, and E. Wu, "Multi-Layer Depth Peeling via Fragment Sort," *Proc. 11th IEEE Int'l Conf. Computer-Aided Design and Computer Graphics,* pp. 452-456, 2009.
[20] X. Yu, J.C. Yang, J. Hensley, T. Harada, and J. Yu, "A Framework for Rendering Complex Scattering Effects on Hair," *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games,* pp. 111-118, 2012.
[21] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu, "FreePipe: A Programmable Parallel Rendering Architecture for Efficient Multi-Fragment Effects," *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games,* pp. 75-82, 2010.
[22] A. Patney, S. Tzeng, and J.D. Owens, "Fragment-Parallel Composite and Filter," *Computer Graphics Forum,* vol. 29, no. 4, pp. 1251-1258, 2010.
[23] C. Crassin, "Icare3D Blog: Fast and Accurate Single-Pass A-Buffer," 2010.
[24] J.C. Yang, J. Hensley, H. Grn, and N. Thibieroz, "Real-Time Concurrent Linked List Construction on the GPU," *Computer Graphics Forum,* vol. 29, no. 4, pp. 1297-1304, 2010.
[25] A.A. Vasilakis and I. Fudos, "S-Buffer: Sparsity-Aware Multi-Fragment Rendering," *Proc. Eurographics Conf.,* pp. 101-104, 2012.
[26] E. Persson, "Depth in-Depth," technical report, ATI Technologies, Inc., 2007.
[27] D. Sekulic, "Efficient Occlusion Culling," *GPU Gems,* pp. 487-203, Addison-Wesley, 2004.
[28] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer, "Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful," *Computer Graphics Forum,* vol. 23, no. 3, pp. 615-624, 2004.
[29] M. Segal and K. Akeley, "The OpenGL Graphics System: A Specification of Version 3.3 Core Profile," 2010.
[30] S. Tzeng, A. Patney, and J.D. Owens, "Efficient Adaptive Tiling for Programmable Rendering," *Proc. Symp. Interactive 3D Graphics and Games,* p. 201, 2011.

Andreas-Alexandros Vasilakis received the BSc and MSc degrees from the Department of Computer Science, University of Ioannina, Greece, in 2006 and 2008, respectively. He is currently working toward the PhD degree in the same department. His research interests include character animation, GPU programming, and multifragment rendering.

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