HA-Buffer: Coherent Hashing for single-pass A-buffer

Sylvain Lefebvre, Samuel Hornus, Anass Lasram

► **To cite this version:**

Sylvain Lefebvre, Samuel Hornus, Anass Lasram. HA-Buffer: Coherent Hashing for single-pass A-buffer. [Research Report] RR-8282, INRIA. 2013, pp.27. hal-00811585

HAL Id: hal-00811585

https://inria.hal.science/hal-00811585

Submitted on 10 Apr 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HA-Buffer: Coherent Hashing for single-pass A-buffer

Sylvain Lefebvre, Samuel Hornus, Anass Lasram

RESEARCH REPORT
N° 8282
April 2013
Project-Teams Alice

ISSN 0249-6399 ISRN INRIA/RR--8282--FR+ENG

HA-Buffer: Coherent Hashing for single-pass A-buffer

Sylvain Lefebvre, Samuel Hornus, Anass Lasram

Project-Teams Alice

Research Report n° 8282 — April 2013 — 24 pages

Abstract: Identifying all the surfaces projecting into a pixel has several important applications in Computer Graphics, such as transparency and CSG. These applications further require ordering, in each pixel, the surfaces by their distance to the viewer. In real-time rendering engines, this is often achieved by recording sorted lists of the fragments produced by the rasterization pipeline. The major challenge is that the number of fragments is not known in advance. This results in computational and memory overheads due to the necessary dynamic nature of the data-structure. Similarly, many fragments which are not useful for the final image—due to opacity accumulation for instance—have to be stored and sorted nonetheless, negatively impacting performance. This paper proposes a novel approach which records and *simultaneously* sorts all fragments in a single geometry pass. The storage overhead per fragment is typically lower than 8 bits per record, and no pointers are involved. Since fragments are progressively sorted in memory, it is possible to assess during rendering whether a new fragment is useful. Our approach combines advantages of previous approaches at similar levels of performance, and is implemented in a single fragment shader of 24 lines of GLSL.

Key-words: A-buffer, real-time, transparency, spatial hashing

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

HA-Buffer: application du hachage cohérent à la création de A-buffer en une passe

Résumé : Plusieurs applications en synthèse d'image nécessitent le calcul de l'ensemble des surfaces visibles au travers d'un pixel. Citons le dessin correct de surfaces transparentes ainsi que le dessin de modèles CSG. Ces applications nécessite également de trier les surfaces, pour chaque pixel, selon leur distance au point de vue.

Pour les applications en temps-réel, ce sont les fragments produits par l'étape de rasterisation qui sont triés et stockés en mémoire vidéo. Le nombre de ces fragments n'étant pas connu à l'avance, il est nécessaire d'utiliser de coûteuses techniques de gestion de la mémoire. De plus, tous les fragments sont traités même si une fraction non négligeable d'entre eux peut être inutile au dessin de l'image finale (grâce, par exemple, à l'accumulation de l'opacité de plusieurs surfaces combinées).

Nous proposons une technique simple pour trier les fragments d'un même pixel au moment de leur rasterisation, sans utiliser de liste chainée (et donc de pointeur). Puisque la liste des fragments pour un pixel est toujours triée, il est possible de déterminer, au moment de sa rasterisation, si un fragment contribuera ou pas à l'image finale, et de le rejetter le cas échéant. La technique combine les avantages de plusieurs approches existantes pour un niveau de performance similaire. Elle a l'unique avantage d'être très simple à coder : 24 lignes de GLSL.

Mots-clés : A-buffer, temps-réel, transparence, hachage spatial

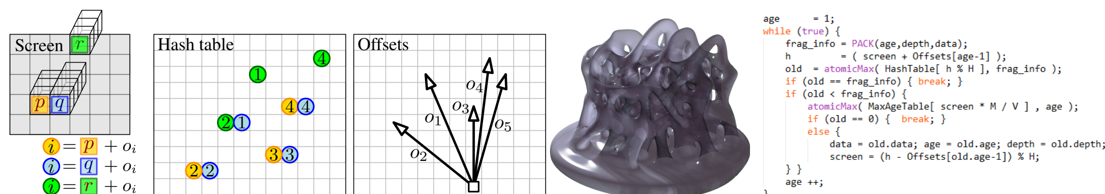


Figure 1: *Left:* The A-buffer stores lists of fragments in each pixel. Instead of lists, we use a single *hash table* wherein each pixel p is assigned a fixed permutation $i = p + o_i$ of locations where to store the fragments covering p. A simple mechanism is used to detect collisions and skip over them. *Middle:* Transparent rendering with translucent shadows (118 FPS, 1024^2 viewport). *Right:* The *complete* algorithm for inserting and *simultaneously* sorting fragments, called from the fragment shader during rasterization.

1 Introduction

Transparency and CSG operations have always been challenging for real–time rasterization. A major difficulty comes from the fact that the on–screen color of a pixel can only be determined after all surfaces influencing the pixel are known and sorted in depth–order.

During rasterization, each triangle generates a number of *fragments*. Each fragment corresponds to a screen pixel – it is a small surface element *potentially* visible through this pixel. In a classical rasterizer only the fragment closest to the viewer is kept: The rasterizer blindly rejects all fragments which are further away than the current closest, using the depth–buffer algorithm.

Instead, algorithms dealing with transparency or CSG have to produce ordered lists of all the fragments falling into each pixel. This is typically performed in two stages: First, lists of fragments sorted by depth are gathered in each pixel. Second, rendering is performed by traversing the lists, either accumulating opacity and colors (transparency), or applying boolean operations to determine which fragment is visible (CSG). The data-structure is recreated at every frame, and therefore has to be extremely efficient and integrate well with the rasterizer. A large body of work has been dedicated to this problem (see Section 2).

Contributions This paper introduces a novel approach which is both efficient and simple to implement in available graphics APIs. In particular, our approach has the following advantages:

- It requires only a single rasterization pass to both store and sort all fragments, without depth–complexity limitations.
- The memory overhead is low, 8 bits per stored fragment in our implementation.
- The implementation is simple, the entire fragment insertion and sort code fitting in less than 24 lines of GLSL (see also Figure 1). Access requires a simple loop without dependent memory reads.
- Our approach enables a *conservative* early rejection of fragments that will have no or little influence on the final image.

This is made possible by recent advances in spatial hashing. Our work introduces a hashing mechanism for depth-sorted fragment lists based on coherent hashing [8]. We exploit properties of this hashing scheme that make it uniquely well suited to the problem of storing and sorting per-pixel fragments.

While the main focus of the paper is on transparency, our technique may be applied to other applications such as CSG and voxelization (in the manner of layered depth images [19]). Our analysis shows that performance is similar to state of the art approaches with unique advantages:

sort and store in a single geometry pass, conservative early culling, simplicity of implementation, arbitrary depth complexity and memory–performance tradeoff.

2 Related work

Most techniques for fragment accumulation implement a form of A-buffer [2]. The A-buffer stores in each pixel the list of fragments that cover that pixel. The fragments are sorted by depth and the size of the list is called the *depth-complexity*. While the A-buffer is primarily designed for anti-aliasing it inspired many similar approaches for transparency, CSG and voxelization.

We review below the approaches for A-buffer construction most related to ours. Since our work is based on spatial hashing, we recall the necessary background in Section 2.5.

2.1 Depth peeling techniques

Depth-peeling [16, 7, 10] is a multi-pass algorithm leveraging the Z-buffer to process all fragments in depth-order. It mimics selection-sort: after a first rasterization of the scene, the Z-buffer contains only the visible fragments, closest to the viewer. A second rasterization is performed, keeping the closest fragments which are farther than the Z-buffer of the previous iteration. This extracts a second layer of fragments. This process is iterated until no fragment remains.

The obvious drawback of the algorithm is that it requires as many rasterization passes as the depth complexity of the scene. An advantage, however, is that fragments are processed in front-to-back order without having to store all the fragments at once. This affords for early culling of fragments that are hidden, for instance, by opacity accumulation.

2.2 Per-pixel fixed-size arrays

A number of approaches pre-allocate fixed-sized buffers per pixel. A per-pixel counter is maintained to accumulate fragments in the buffers. This offers good performance, but imposes a hard limit on the per-pixel depth-complexity for a single rasterization while requiring large amounts of memory. Crassin [4] describes an efficient implementation of this approach on mainstream GPUs.

Stencil routed A-buffer [18] captures up to K fragments per pixel by rendering to a multisampled buffer with multisampling antialiasing disabled. A stencil mask is applied to write only one sample at a time. This approach is limited by the maximum number of MSAA samples.

Several approaches propose a mechanism to deal with buffer overflows: For instance merging fragments with closest z values [12] or relying on stochastic blending [6]. This is possible for effects which support approximations such as transparency but not for CSG or voxelization. Our algorithm is compatible with such approximations, but can also improve them since the *sorted* list of fragments gathered so far is available at all times. In particular, our algorithm can keep fragments up to a given opacity threshold, rather than summarily rejecting supernumerary ones.

FreePipe [15] allocates fixed size arrays and uses an `atomicMin` to sort the fragments as they are inserted. If K is the size of the per-pixel buffers, then only the K closest fragments are available after a single pass. Our approach also provides sorting in a single pass but without limiting the number of fragments per-pixel.

2.3 Per-pixel linked lists

The original A-buffer proposed an implementation based on per-pixel linked lists. A number of recent work describe how to implement this mechanism on modern rasterization hardware.

These techniques proceed in two stages: First, unordered lists of fragments are gathered in a geometry pass. Then the lists are sorted.

Yang *et al.* [20] describe a linked-list A-buffer implementation in GLSL. A first buffer stores a list-head pointer per pixel. A second buffer stores fragment records (next-pointer, depth and data). When a fragment is inserted, a global counter is incremented to allocate the next fragment record. The new record is then linked to the pixel list by updating the head pointer atomically. When the geometry pass is complete, the linked lists are sorted. This technique is improved by allocating small blocks of fragments instead of single elements, thereby reducing contention on the allocation counter at the expense of additional memory usage [5].

A major advantage of these approaches is their simplicity of implementation, and their versatility: There is no other constraint on depth complexity than the maximum global number of fragments. There is a unique geometry pass, although a sort pass is required. In particular, fragments having no impact on rendering can only be detected *after* the sort. The per-fragment pointers incur a memory overhead.

Our approach brings similar advantages but in a single geometry pass (no additional sort), and without pointers. It enables early rejection of fragments. We further compare with these approaches in Section 6.2.

2.4 Count, gather, sort approaches

Another category of work relies on parallel stream processing techniques to allocate just enough memory for storing all the fragments seen from a given viewpoint. A first geometry pass counts the fragments covering each pixel. The obtained set of counters is then converted to prefix sums. These sums indicate where to start storing the fragments of each pixel. A fragment buffer of the required size is allocated and a second geometry pass is used to fill it. An optimized implementation of this technique is described in [17].

This approach gracefully adapts to large variations in the total number of fragments. It requires however two geometry passes as well as an efficient parallel scan implementation (prefix sum). All fragments have to be gathered, disabling conservative early culling.

2.5 Background in parallel hashing

Since our technique is based on parallel spatial hashing we recall here the necessary background to present our work. For the sake of brevity we do not describe spatial hashing techniques outside the scope of this paper [14, 1] but focus on the technique of García *et al.* [8] which inspired our technique.

2.5.1 A brief introduction to coherent hashing

Notations Coherent spatial hashing takes as input a set of integer *keys* $K \in U$, where U is called the universe and is typically the set of 32 bits integer coordinates. Each key is associated with a data record, for instance a 32 bits RGBA color. K is the set of *defined* keys while $U \setminus K$ is the set of *empty* keys. Generally $|K| \ll |U|$. The hashing scheme encodes K in a hash table H and afford for fast random queries to discover whether a key $k \in U$ belongs to K and retrieve its associated data.

The table H stores both keys and data packed together in integer records. Let $\texttt{key}(H[x])$ and $\texttt{data}(H[x])$ be respectively the key and the data record stored in $H[x]$. Storing the keys alongside data is mandatory, and incurs a significant overhead: This doubles the required size of the hash table. In addition, not all entries of H are used: The *load factor* of a hash table is measured by the ratio $\frac{|K|}{|H|}$. Fortunately, coherent hashing reaches load factors of up to 0.99.

Coherent hashing is based on an open addressing mechanism: Each key k is associated with a sequence of possible insertion locations $(h^i(k))_{i\geq 0}$. If the key k is inserted at index $h^i(k)$ in the table, for some $i \geq 0$, we say that the *age* of that key is i and define $\texttt{age}(H[h^i(k)]) = i$. Coherent hashing keeps track of the age of the keys inserted in the hash during construction.

Parallel construction of the hash table The keys and their associated data are inserted in the table in parallel. The thread responsible for key k explores its sequence of insertion locations. At each location two cases occur: 1) the location is empty in which case the key is immediately inserted, or 2) the location is occupied by another key q, in which case the algorithm checks whether to *evict* the key. If q is evicted then it is exchanged with k in the table and q becomes the key to be inserted: the thread resumes with insertion until finding an empty location in the table. Coherent hashing uses the Robin Hood strategy, which consists in evicting keys with a smaller age than the current key. This reduces the *maximum* age of the keys across the entire table to $O(\log N)$ for a table storing N keys [3], thereby ensuring that all inserted keys can be accessed quickly.

Since many threads compete to insert keys, care must be taken to preserve the consistency of the hash table. Coherent hashing relies on an atomic instruction to both test for eviction and exchange the keys if needed. To achieve this, age, key and data are packed together into a 64 bits integer record. The age is stored in the most significant bits (MSB), so that an `atomicMax` operation can be used to both check for eviction and store the record. In this setting, the age is usually small and requires only 3-4 bits from the key and data. The full insertion code of García *et al.* is given below, followed by a short description:

```
insert(uvec3 key, uint data)                              1
  age = 0;                                                2
  while( ++ age < MAX_AGE ) {                             3
    h_k_i    = hash( key , age ) ;                        4
    record   = PACK( age , key, data ) ;                  5
    prev     = atomicMax( & H[ h_k_i ] , record ) ;       6
    if ( record > prev ) {                                7
      if ( EMPTY( prev ) ) ) {                            8
        return SUCCESS;                                   9
      } else {                                            10
        key = KEY( prev );                                11
        age = AGE( prev );                                12
      }                                                   13
    }                                                     14
  }                                                       15
  return FAILURE;                                         16
```

l.3 `MAX_AGE` is an upper bound on the maximal age of a key.
l.5 Age, key and data are packed into an integer record. Age is in the MSB.
l.6 The hash relies on an *atomicMax* operation to automatically evict keys with a smaller age than the current key.
l.7 This tests whether the `atomicMax` inserted the key.
l.8-10 If a key was evicted it becomes the current key. `KEY` and `AGE` extract information from the packed integer record.

Coherent hash function The code described above is generic and could use any hash function. Coherent hashing is obtained by generating the sequence of insertion locations using the following

hash function:

$$h^i(k) = k + o_i \mod |H|$$

where o_i is a predefined random offset that does *not* depend on k. If two neighboring threads process neighboring keys they access coalesced memory addresses at each step i. This greatly improves memory access efficiency during construction and access.

As we explain later this hash function has important properties for us, besides its improved memory access pattern.

Hash access Accessing a key amounts to following its sequence of locations until either it is found or `MAX_AGE` is reached:

```
access(uint key)                                    1
  for (i = 0; i < MAX_AGE ; i ++ ) {                2
    h_k_i    = hash( key , i ) ;                    3
    if ( KEY( H[ h_k_i ] ) == key) {                4
      return DATA( H[ h_k_i ] );                    5
    }                                               6
  }                                                 7
  return null;                                      8
```

3 Overview

Our approach has two main stages: A *gather+sort* stage, where the geometry is rasterized, fragments are stored and sorted. A *render* stage, where the image is produced on screen from stored, depth-sorted fragments.

The critical stage is the gather+sort. We rasterize the scene using a specific fragment shader that shades each input fragment and inserts it in the hash table:

```
void main()                                                     1
  uint   data    = shadeFragment();                             2
  vec3   prj     = v_Pos.xyz / v_Pos.w;                         3
  vec3   pos     = vec3( prj.xy * 0.5 + 0.5,                    4
                    1 - (v_Pos.z+ZNear)/(ZFar+ZNear) );         5
  uvec2 screen = uvec2( pos.xy*V );                             6
  uint   depth   = pos.z*MAX_DEPTH;                             7
  insert( screen, depth, data );                                8
  discard;                                                      9
```

In this shader, `v_Pos` is the transformed and projected surface point, `shadeFragment()` computes the fragment color and opacity as a 32 bits integer, `pos` are the surface point coordinates remapped in $[0,1]^3$. `V` and `MAX_DEPTH` define the screen resolution and depth resolution. Typical values are `V = 1024` (for simplicity we assume a square viewport), `MAX_DEPTH =` 2^{24}. Note that we reverse depth so that large values correspond to fragments closer to the view. The reason for this will become apparent later.

During this pass nothing appears on screen: All fragments are discarded from the frame buffer after being inserted in the hash table.

The second pass is the render stage. A screen-covering quad is rendered with a shader accessing the sorted list of fragments for each pixel and computing the final color:

```
void main()                                                  1
  vec2  pos    = ( u_Pos.xy * 0.5 + 0.5 );                   2
  uvec2 screen_pos = uvec2( pos.xy * V );                    3
  renderFragList( screen_pos );                              4
```

These two shaders produce the final image. No other steps are required.

Section 4 discusses how to implement `insert` using a variation of coherent hashing—this is the entire gather+sort. Section 4.1.4 explains how to retrieve data during the render stage (`renderFragList`). Section 5 presents a modification of the algorithm for early-culling of fragments.

4 The HA-Buffer

This section describes how we modify the coherent hash to obtain an efficient mechanism for *simultaneously* storing and sorting fragments. There are different aspects to our approach:

- We modify the definition of the keys to exploit collisions and form a list of fragments in each pixel (Section 4.1).
- We show that we can avoid storing the keys together with the data, resulting in significant memory savings (Section 4.1.1).
- From the above we deduce a major property of the hash which lets us sort the keys as we insert them.

4.1 Hashing fragments

In order to insert a fragment in the hash table, we need to compute a key from information that describes the fragment (screen coordinates, depth, color).

García *et al.* store and access voxels in the hash table by computing a key from their x, y, z coordinates, for instance with linear addressing. The case of fragments is different: We seek to store and retrieve efficiently a *list* of fragments for each screen pixel.

We propose to exploit collisions to this end: the idea is to form keys (to be hashed) from the screen coordinates only, ignoring the fragment depth. As a consequence, all fragments covering screen coordinates $\mathbf{p}$ collide with each other; they all appear along the access sequence $(h^i(\mathbf{p}))_{i\geq 0}$. This is illustrated Figure 1.

This automatically produces lists of fragments. However, the downside is that the maximum age is at least the depth complexity of the pixel. To avoid incurring a large penalty in all pixels we implement the *maximum age table* mechanism described by García *et al.*: A 2D table covers the screen and records locally the maximum age reached by inserted fragments. This table has a lower resolution than the screen, so as to avoid divergence in neighboring threads during access: one cell of the maximum age table typically covers a block of 8×8 pixels.

This first step already provides hashing for fragment lists. However, the main benefits of using the hash are yet to be described.

4.1.1 Reducing memory overheads

When hashing with screen coordinates, fragments of a same pixel will collide with each other as well as with fragments from other pixels. We therefore still need to store the keys alongside the data, so as to remember by which pixel the fragment was produced.

This is a major overhead, typically 24 bits per fragment for encoding screen coordinates. This overhead is similar to the pointers used by linked-list approaches, making our approach less beneficial from a memory standpoint. Fortunately, we can improve this.

García *et al.* essentially considers the case where $|H| \ll |U|$. However, our scenario is very different. The set of keys is limited to the screen coordinates. Assuming a full-screen rendering, we can expect at least one fragment per screen pixel. As a consequence the size of the hash table is expected to be *larger* than the screen: $|H| > |U|$.

This has an important property: Consider the set of keys that can collide at age i in location $x \in [0, |H|-1]$ of the table: $\{k \in U \mid k + o_i \mod |H| = x\}$. In our case this set contains a *unique* coordinate, which follows from $|H| > |U|$: the screen does not fold onto itself when mapping into H. This coordinate is the unique pixel coordinate that will send a fragment at age i into location x. It is easily computed from the age as $p = x - o_i \mod |H|$. *No other pixel can produce a fragment that maps to x at age i.* We refer to this as the *age equivalence property.*

As a consequence, a fragment inserted in the table belongs to pixel p if and only if $\texttt{age}(H[h^i(p)]) = i$. Thus, we can safely remove from the hash table the pixel screen coordinates. The age is enough to identify the source pixels of the fragments. The code for accessing a fragment covering pixel p becomes:

```
access(uvec2 p)                                   1
  for (i = 0; i < MAX_AGE ; i ++ ) {              2
    x = h( p , i ) ;                              3
    if ( AGE( H[ x ] ) == i) {                    4
      return DATA( H[ x ] );                      5
    }                                             6
  }                                               7
  return null;                                    8
```

Notation For a fragment covering pixel x at depth z inserted in the hash table at index $h^a(x)$ we note $(a, z)_x$ the integer record stored in $H[h^a(x)]$. Only the age, depth and data are stored. For clarity we indicate the source pixel in subscript and ignore the data record. By the age equivalence property, we have:

$$H[h^a(\mathbf{x})] = (a, z)_{\mathbf{q}} \Rightarrow \mathbf{q} = \mathbf{x}$$

We describe next a consequence of the age equivalence property which greatly facilitates sorting of the fragments in depth order.

4.1.2 Sorting fragments by depth

Recall that collisions are resolved with an `atomicMax`. This implements the Robin Hood strategy provided that the age is stored in the most significant bits of the integer records. From now on, we also assume that the depth immediately follows the age in the integer record: $(a, z)_x = {}^{\text{MSB}}\boxed{a}\boxed{z}\boxed{\ldots}{}^{\text{LSB}}$.

A natural question is whether this mechanism is also sorting fragments in decreasing depth-order, *as they are inserted in the hash.* It turns out this is not a trivial question. First, key ages and depths are uncorrelated: a key of any depth can be at any age. Second, many threads are working simultaneously, and some fragments may not be compared with each others due to evictions. We now prove that this in fact works as expected, *i.e.*, that we obtain depth-sorted lists as a by-product of using coherent hashing.

Proof We start with a *monotonicity property*: "*The hash table cells are exclusively updated with the* `atomicMax` *function. This implies that the value of a cell of H can only increase.*"

We then prove Proposition $\mathcal{A}$ = "*At any time, if a cell of the hash table contains $(a, z)_x$ then $\forall b \in [0..a-1]$, $H[h^b(x)] \geq (b, z)_x$.*":

Assume that cell $h^a(x)$ of H contains $(a, z)_x$ in H. If $a = 0$ then proposition $\mathcal{A}$ is vacuously true. If $a > 0$ then consider $b \in [0..a-1]$. The fragment currently stored in $h^a(x)$ has been tested for insertion in all indexes $h^b(x)$: a record $(b, z)_x$ was compared to the value at index $\mathbf{p} = h^b(x)$. Since it was not inserted at, or was evicted from that position, then at that time the integer record stored at index $\mathbf{p}$ was larger than $(b, z)_x$. Because of the monotonicity property stated above, $H[\mathbf{p}]$ still contains a value larger than $(b, z)_x$, which proves proposition $\mathcal{A}$.

Finally, to prove our claim, assume that the table stores two fragments covering the same pixel x with depth z at index $h^a(x)$ and w at index $h^b(x)$, such that $b < a$. From proposition $\mathcal{A}$ it comes: $(b, w)_x = H[h^b(x)] \geq (b, z)_x$, which implies $w \geq z$. □

Simply by encoding the depth after the age in the integer records, *without any change to the insertion algorithm*, we obtain lists of fragments ordered by decreasing depth. To obtain a front-to-back ordering we reverse the depth values so that largest values are closest to the viewer.

Discussion From the point of view of sorting, the algorithm exhibits a complexity of $\Omega(k^2)$ for k fragments in a pixel list. As k increases we can expect this to become inefficient. It is interesting, however, to note that this bears similarity with the sorting scheme of FreePipe [15] and the depth-peeling technique. Depth peeling, although very different in nature, also exhibits such a quadratic behavior for front-to-back enumeration of the fragments covering a pixel.

However, in our approach the unique geometry pass does not only sort but simultaneously hashes the fragment for storing them in memory. Therefore, only considering the sort complexity would be misleading: The cost is amortized by the simultaneous storage of the fragments.

In addition, the resulting algorithm maps very well to embarrassingly parallel architectures: Each thread independently inserts keys through a simple atomic operation, and thanks to the coherent scheme memory is accessed in a coalesced fashion. Elaborate gather and sort techniques tend to make such parallelism more difficult to achieve, to implement and to optimize.

4.1.3 Gather+sort algorithm

The complete pseudo-code for the gather+sort mechanism is:

```
int insert(uvec2 screen,uint depth,uint data)                       1
  age       = 1;                                                    2
  while (true) {                                                    3
    // pack fragment in a 64 bits integer record                    4
    frag_info = PACK(age,depth,data);                               5
    // compute hash location                                        6
    h         = ( screen + Offsets[age-1] );                        7
    // try to insert/evict                                          8
    old   = atomicMax( HashTable[ h % H ], frag_info );             9
    if (old == frag_info) { break; } // duplicate                   10
    else if (old < frag_info) {      // did we insert?              11
      atomicMax( MaxAgeTable[ screen * M / V ] , age );             12
      if (old == 0) {  break; } // old was empty                    13
      else {                    // old was evicted                  14
        data      = old.data;                                       15
        age       = old.age;                                        16
```

```
        depth    = old.depth;                                   17
        // deduce screen position from age                      18
        screen   = (h - Offsets[old.age-1]) % H;                19
      }                                                         20
    }                                                           21
    age ++;                                                     22
  }                                                             23
```

We store the fragment records in 64 bits, from MSB to LSB: 8 bits for the age, 24 bits for the depth, 32 bits for the data (RGBA color). `PACK` forms the 64 bit integer record. H, V and M are respectively the hash table size, the screen size and the max age table size. Note that the stored age starts at 1: We use 0 as a special value to indicate an empty cell in the hash (the hash table is cleared at the start of every frame).

This algorithm is implemented directly in GLSL, using the `NV_shader_buffer_store` extension. The only complication on current hardware is the 64 bits `atomicMax` which is not yet available (it will be available on the GK110 NVidia GPU). We emulate it with a 64 bits `atomicCAS` (Compare And Swap). The resulting GLSL code is only 24 lines of code.

32 bits version Some applications do not require a data record. This is for instance the case of voxelization, where only the fragment depth is necessary. In this case a native 32 bits `atomicMax` can be used, resulting in an increased performance (Section 6.2).

Insertion failures The insertion algorithm may fail in two situations: when the age of an inserted key does not fit on 8 bits, or when the hash table is full.

The maximum reachable value for the age of a key depends on the worst depth-complexity and the load factor (the eviction mechanism of Robin Hood hashing is activated more often at higher load factors). A complex scene could require a depth complexity larger than 256, in which case the age can be encoded on more bits. However, as we describe in Section 5 some applications such as transparency allow for an early-cull mechanism which reduces the need for a large maximum age.

A full hash table would also trigger insertion failures. In addition, performance varies with the hash table load factor (Section 6.2 analyzes this trade-off). We therefore dynamically adapt the size of the hash table. It is initially set to the screen resolution, which provides a reasonable estimate for viewing an object centered on screen. We monitor the number of fragments stored in the hash and dynamically reallocate when reaching a high density (80%). Similarly, we decrease the hash table size if a low density is reached (30%). We always increase/decrease the table size by a the same ratio (25%). To count fragments efficiently, we exploit the clear mechanism. The clear is performed by rendering a quad covering the hash table. An occlusion query monitors the number of drawn fragments during the clear. If the fragment being cleared was previously empty, it is discarded. Otherwise, it is drawn and will be counted by the occlusion query. This counts fragments with a one-frame latency. This mechanism has a very small impact on performance.

Offsets for the hash function Recall that the sequence of insertion positions for key k is $(h^i(k) = (k+o_i) \mod |H|)_{i\geq 0}$ where $(o_i)_{i\geq 0}$ is a sequence of offsets in the hash table: $\forall i, o_i \in H$. The good properties of Robin Hood hashing have been analyzed in a probabilistic setting, as most other hashing schemes. Therefore, in order to ensure a well behaved insertion in table H at all times, it is recommended to regenerate the offset sequence o at random when the hash table is cleared, *i.e.*, before each frame.

In practice the set of defined keys is often cluttered due to spatial coherence. We found that in this situation the set of offsets o_i can be made slightly more efficient. We use iterated farthest point sampling [9] to generate offsets which are as far as possible from each others. In addition, the first offsets have a greater distance between them, making it more likely to find a free slot in the hash table at small age values. We seed this process randomly and generate different offsets at every start of the application – this is unfortunately too slow for regeneration at every frame. A drawback is that biases due to the structure in the offsets are more likely to occur. All renderings in the paper use such offsets, while synthetic test cases use random offsets to avoid bias in measurements.

4.1.4 Traversing the list of fragments covering a given pixel

The GLSL code for traversing the front-to-back list of fragments covering a pixel at position `screen` follows.

```
void renderFragList(uvec2 screen) {                                   1
    // read max age for this pixel                                    2
    uvec2      m      = (screen * M / V) % M;                         3
    uint32_t   maxage = MaxAgeTable[ Maddr(m) ] - 1;                  4
    if (maxage == 0) return;                                          5
    // walk along the insertion sequence                              6
    for (uint n = 0 ; n < maxage ; n ++ ) {                           7
      uvec2 h         = ( screen + Offsets[n] ) % H;                  8
      uint  frag_info = HashTable[ Haddr(h)+1 ];                      9
      // is this fragment originating from the pixel?                 10
      if ( AGE(frag_info) == n+1 ) { // yes                           11
        uint frag_data = HashTable[ Haddr(h) ];                       12
        //// do something with fragments                              13
        // (front-to-back order guaranteed)                           14
      }                                                               15
      if ( AGE(frag_info) <= n ) return;                              16
    }                                                                 17
}                                                                     18
```

`Haddr` and `Maddr` compute a linear address from the 2D coordinates in row-major order. If the condition in line 16 is verified, then proposition $\mathcal{A}$ (Section 4.1.2) guarantees that we have already seen all the fragments covering the query pixel.

5 Early cull mechanism

The algorithm as we described it so far gathers all fragments per-pixel. However, there are some cases were many fragments end up not being used in the final rendering. This a common situation when the accumulation of semi-transparent surfaces results in an almost opaque media: deeper fragments have no impact on the final image.

Few methods have the ability to efficiently detect these cases: In most other approaches the fragments are gathered first and sorted in a separate pass (see also Section 2). Depth peeling offers such a functionality but due to multiple geometry passes is much slower than our approach (see Section 6.2). Fixed-size arrays methods propose mechanisms to deal with overflows, relying on approximate techniques to merge fragments in excess. We could easily implement a similar mechanism, keeping only the K closest fragments. However, what we propose here is different:

Since inserted fragments are sorted in correct front-to-back order in the hash table, our algorithm offers a unique opportunity for *conservative* early culling. That is, we can check during insertion whether a fragment is still useful for the final rendering based on the sorted list of previously inserted fragments. If we determine that the fragment is no longer useful we stop trying to insert it. The decision is conservative: We may insert too many fragments, but we will never improperly reject them.

This is in effect very similar to the early depth cull of the Z-buffer: The efficiency of the mechanism depends on the order with which surfaces will be sent for rasterization. However, in complex scenes it results in a significant savings in the amount of fragments that need to be stored, as illustrated Figure 2.

The GLSL code for early culling follows. It is a simple modification of the earlier insertion algorithm that we specialize here for opacity accumulation. Other applications for which a conservative culling criterion can be devised should benefit from early culling as well.

```
int insert_early_cull(uvec3 key,uint data) {                              1
  float w_accum  = 0.0; // accumulates opacity                            2
  uint  age      = 1u;  // age init                                       3
  uint  iter     = 0;                                                     4
  while (iter++ < MAX_ITER) {                                             5
    // pack age,depth,data in a 64bits record                             6
    uint64_t key_info = PACK(age,key.z,data);                             7
    // compute hash insertion sequence                                    8
    uvec2    l    = ( key.xy + Offsets[age-1] );                          9
    uint     h    = Haddr( l % H );                                       10
    // test for eviction and insert                                       11
    uint64_t old  = atomicMax64( HashTable+h, key_info );                 12
    if (old == key_info) {                                                13
      return 1; // stop (success) on duplicate                            14
    } else if (old <= key_info) { // key was inserted                     15
      // update max age table                                             16
      uvec2 m = (key.xy * M / V)  % M;                                    17
      atomicMax( MaxAgeTable + Maddr(m) ) , age );                        18
      // extract age and depth from record                                19
      uint old_key = uint( old >> uint64_t(32) );                         20
      if (old_key == 0) {                                                 21
        return 1; // stop (success) on empty                              22
      } else {                                                            23
        // reinsert evicted key                                           24
        uint age_prev = age;                                              25
        age      = AGE( old_key );                                        26
        data     = uint( old );                                           27
        // recompute key from age and offset                              28
        key      = uvec3( (l - Offsets[age-1]) % H ,                      29
                          DEPTH(old_key) );                               30
        // check if early cull can be continued                           31
        if (age != age_prev) { // still on same pixel?                    32
          w_accum = 0.0; // no: reset oppacity                            33
          age ++;        // try next age                                  34
        } else {                                                          35
          // yes: keep age; evicted will be tested again                  36
          // at this age at next iteration                                37
        }                                                                 38
```

```
      }                                                      39
    } else {                                                 40
      // we did not evict, continue trying?                  41
      uint old_key = uint( old >> uint64_t(32) );            42
      // only perform early cull if from same pixel          43
      if (AGE(old_key) == age) {                             44
        // accumulate oppacity                               45
        uint     d = uint( old );                            46
        float    w = float(d & 255u) / 255.0;                47
        w_accum   += (1.0-w_accum) * w;                      48
        // test against threshold                            49
        if (w_accum > OpacityThreshold) {                    50
          return 1; // early culling                         51
        }                                                    52
      }                                                      53
    }                                                        54
    // try next age                                          55
    age ++;                                                  56
  }                                                          57
  return 0;                                                  58
}                                                            59
```

The most important part of the algorithm are lines 44-53, which are executed when a fragment f is not inserted. If the age is the same as the stored fragment s (line 44), then both fragments cover the same pixel. Fragment f was not inserted because it is located *after* s along the view ray. We therefore accumulate the opacity of s (lines 46-48) and test whether f will still be visible by comparing the accumulated opacity to a user-defined threshold.

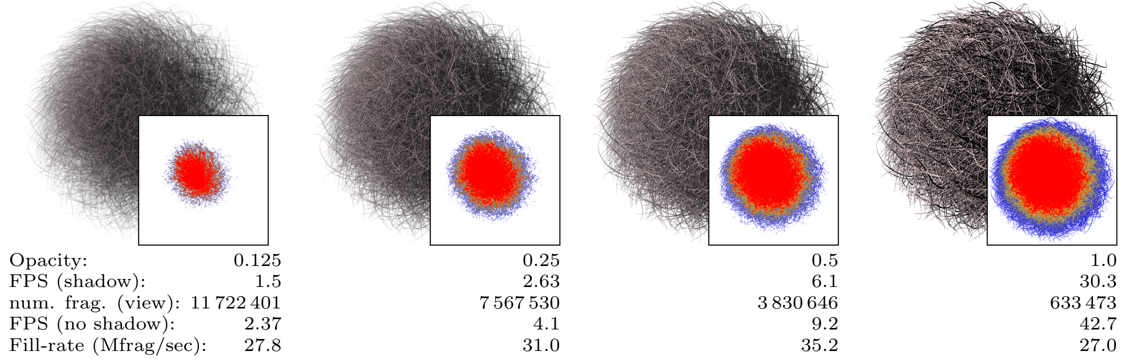
The other change to the base algorithm are lines 32-38, executed after a fragment e has been evicted by the inserted fragment f. It is important to verify whether e covers the same pixel as f. If it does not, the opacity accumulator is reset (line 33) since it is no longer valid. Note that e may not be the first fragment for its pixel, but since we cannot know the accumulated opacity at e we simply set it to 0. This is conservative since opacity monotonically increases. If e and f cover the same pixel, we do not increment the age of e so that the opacity of f can be accumulated in the next iteration.

6 Results

In this section we further analyze the performance of our technique. We focus on transparency and only briefly present results on CSG to illustrate the versatility of our technique.

6.1 Test setup

For performance results we use a NVidia GeForce GTX 680 (Kepler GK104), NVidia driver 310.90 and OpenGL 4.3. The GK104 lacks native support for `atomicMax64`, contrary to the soon to be released GK110. We therefore emulate `atomicMax64` with `atomicCAS64`. Unfortunately, on Kepler this triggered a known GLSL compiler issue regarding race conditions and atomics, preventing proper execution of our shader (the same shader runs correctly on a Fermi GPU). To circumvent this issue we have to insert a useless atomic memory fetch in the main loop of our shader. This has a negative impact on our performance but prevents the compiler from generating the problematic code.



Opacity:	0.125	0.25	0.5	1.0
FPS (shadow):	1.5	2.63	6.1	30.3
num. frag. (view):	11 722 401	7 567 530	3 830 646	633 473
FPS (no shadow):	2.37	4.1	9.2	42.7
Fill-rate (Mfrag/sec):	27.8	31.0	35.2	27.0

Figure 2: The hairball model rendered at 1024^2 resolution and two HA-buffers of size 4096^2 (translucent shadow map + view) at varying opacity. Early-culling directly benefits the frame rate even in this challenging case (opacity threshold: 99%). The insets show the color-coded number of culled fragments. The total number of fragments without culling is 15 420 969 – this is roughly doubled when shadowing is enabled.

This issue only affects the 64 bits version of our code. We therefore use the 32 bits version to estimate the overall performance penalty. Figure 5 measures performance of our reference 32 bits implementation as well as a 32 bits implementation modified to include similar penalties as the 64 bits version: The `atomicMax` is emulated with `atomicCAS` and the same additional atomic memory fetch is added. We find that the ratio between both is 1.37 on average. In the remainder of the paper we report the performance *measured* on our 64 bit implementation and in brackets the projected performance.

6.2 Transparency

Transparency effects directly benefit from the HA-buffer. Once fragments have been inserted in the hash table, it suffices to traverse the lists and blend the fragments in front-to-back order. Let F^i_{rgb} be the color of fragment i and F^i_α its opacity:

$$\begin{aligned} \mathcal{C}^{i+1}_{\text{rgb}} &= \mathcal{C}^i_{\text{rgb}} + F^i_{\text{rgb}}(1 - \mathcal{C}^i_\alpha) \\ \mathcal{C}^{i+1}_\alpha &= F^i_\alpha(1 - \mathcal{C}^i_\alpha) \end{aligned}$$

with $\mathcal{C}^0_\alpha = 0, \mathcal{C}^0_{\text{rgb}} = 0$. A last opaque fragment with any chosen background color $F^{\text{bkg}}_{\text{rgb}}$ is implicitly added. The final pixel color is $\mathcal{C}^{K+1}_{\text{rgb}}$ where K the number of fragments for the pixel.

We optionally add a translucent media effect inside surfaces, tracking in/out pairs of fragments. This effect is visible Figure 3.

Performance vs load factor Our approach offers a tradeoff between performance and memory: Performance varies depending on the hash table load factor. Figure 5 illustrates this tradeoff. The object of Figure 7 is displayed and rotates on screen, with a flat shading (simple alpha blending of constant color fragments). The viewpoint is chosen to fill the hash up to about 95%, and then the hash table size is gradually increased. As can be seen performance slowly decreases as the load factor increases, with an abrupt drop after 80%: the high load factor decreases the probability to find an empty cell in the hash table during fragment insertion.

We next analyze the performance and memory requirements with respect to other techniques.

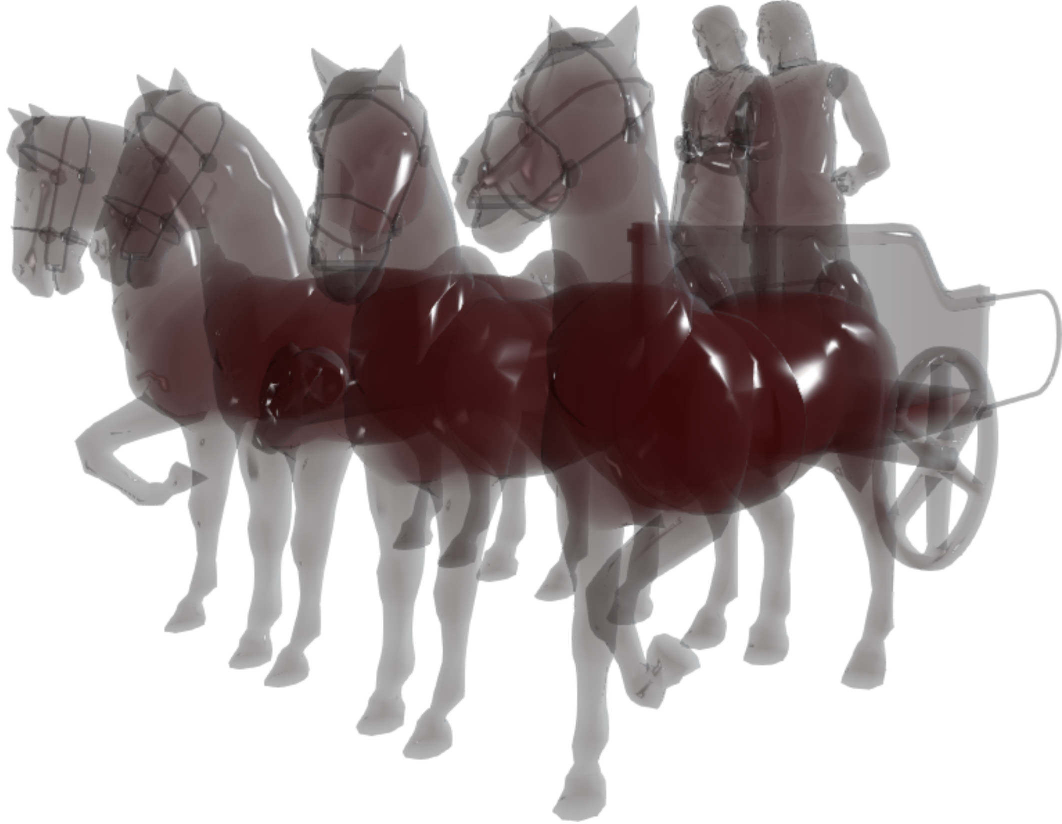


Figure 3: Horses and chariot rendered with our HA-buffer using a GeForce GTX 680 and FXAA. 891929 frag., load: 54.4% (12.6 MiB) max depth: 22, FPS: 216.7.

Analysis: Memory We consider a screen of size V^2, fragments of 56 bits (depth:24 RGBA:32), and a total number of fragments F. Two existing techniques offer compact storage of the fragments (see Section 2): count-gather-sort and linked-lists.

The count-gather-sort approaches only store a per-pixel pointer, but require two geometry passes (count+gather) and a parallel scan on all pixels to complete. The required storage is $32V^2 + 56F$ bits.

The per-pixel linked-lists store one head pointer per-pixel, as well as a pointer per fragment. The required storage is $32V^2 + (56 + 32)F$ bits. Paged linked-lists store one head pointer per-pixel, a per pixel counter (8 bits) as well as several pages for the fragments. Each page contains K fragments and a pointer. The technique requires at least $(32 + 8)V^2 + F/K(56K + 32)$ bits and at most $56(K - 1)V^2$ more.

Using our HA-buffer the storage is the hash table and the maximum age table. Each entry of the max age table covers a block of 8×8 pixels. The age is encoded on 8 bits (could be less in many situations), resulting in 64 bits per fragment in the hash. The storage is a function of the load factor L: $V^2/8 + 64F/L$ bits.

Table 1 summarizes memory occupancy for a variety of settings. As can be seen our technique offers compact storage at high load factors. Of course, this impacts performance and we recommend 80% for memory intensive applications, and to target 50% for best performance.

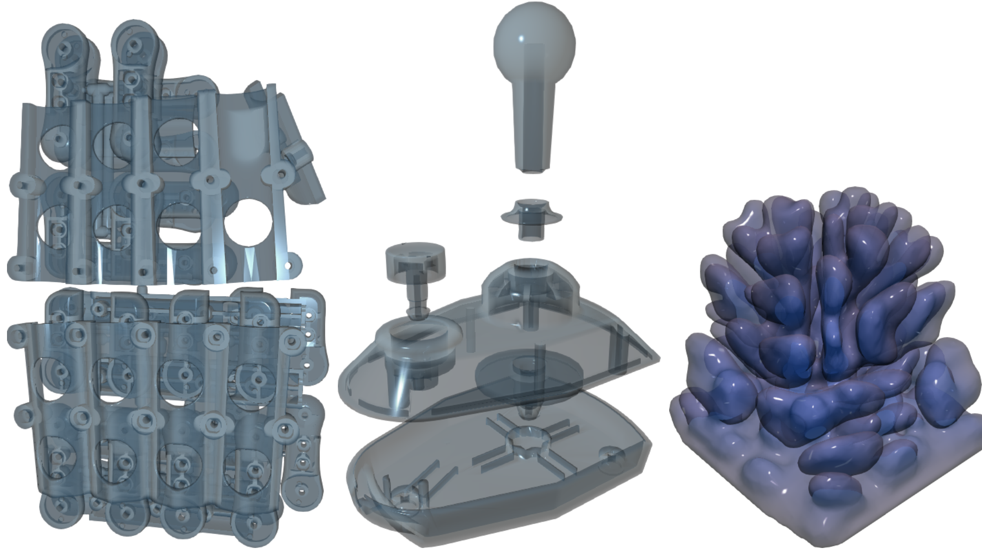


Figure 4: Technical models and intricate shapes rendered with our HA-buffer using a GeForce GTX 680 and FXAA. *Left:* Robotic hand model, 1487942 frag., load: 58.1% (19.6 MiB) max depth: 20, FPS: 91.1 *Middle:* Joystick model, 539236 frag., load: 51.4% (8.06MiB) max depth: 16, FPS: 348 *Right:* Dendrite model, 855296 frag., load: 52.2% (12.6 MiB) max depth: 16, FPS: 202.7

Even at this lower load factor the memory consumption remains comparable to that of other schemes thanks to the low overhead per fragment.

We next discuss performance of these methods.

Analysis: Speed We compare the speed of our technique to several available A-Buffer implementations. In each case we use the original code from the authors, only modified to use a simple flat shading with alpha blending. We measure performance of our 64 bits implementation (without early culling). Results are summarized in table 2 for rendering the view shown in the inset.

Our objective with this analysis is to verify that our technique, despite the simplicity of its algorithm, offers similar levels of performance as the best available approaches.

Technique	$F = 0.85\,\mathrm{M}$	$1.5\,\mathrm{M}$	$10\,\mathrm{M}$
count-gather-sort	9.7	14.0	70.8
linked-list	12.9	19.7	108.9
paged, $K = 4$, lower bound	11.5	16.4	81.3
paged, $K = 4$, upper bound	32.5	37.4	102.3
HA-buffer, $L = 90\,\%$	7.2	12.7	84.8
HA-buffer, $L = 80\,\%$	8.1	14.3	95.4
HA-buffer, $L = 50\,\%$	13.0	22.9	152.6

Table 1: Estimated memory usage (in MiB) of the different A-buffer techniques for a screen size $V = 1024^2$ and a varying number of fragments F.

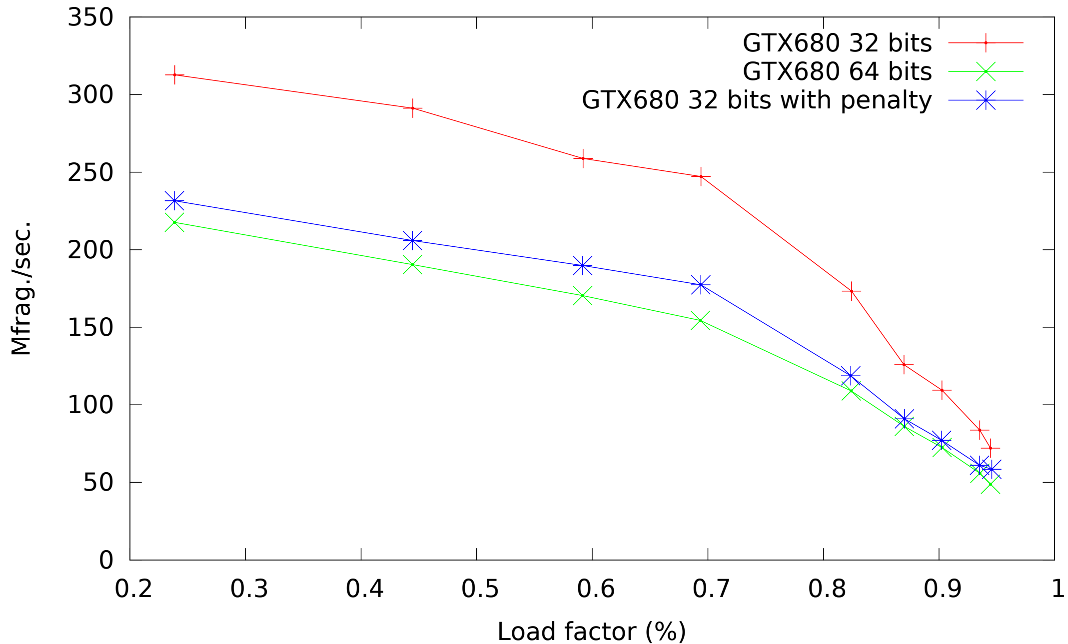
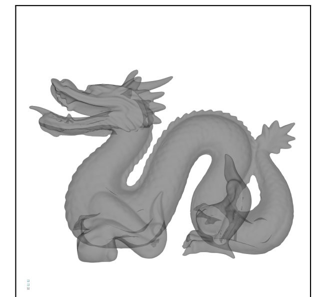


Figure 5: Millions of fragments per second as a function of hash table load factor. Performance is averaged over multiple runs.



It is worth mentioning that performance measures for A-buffers depend on many factors: Balance between geometric complexity and number of fragments, opacity, early culling, etc. Hardware design also has a huge impact: For instance in table 2 the performance of paged per-pixel linked lists increases by 57% when using textures instead of global memory, resulting in the fastest approach. On a Fermi architecture both were equivalent, and slower than our approach.

The table does not include the count-gather-sort of [17] but this paper reports performance in the range of paged per-pixel linked lists (see Figure 7 of [17]) on a benchmark similar to our test scene. We however expect this approach to outperform all others when the fragment complexity is much higher than the geometric complexity. The strengths of this approach are its tight memory allocation and its efficient construction of contiguous lists of pixels via a parallel scan. Its weaknesses are that it always requires two full-resolution geometry passes – making it less efficient when geometric complexity increases for a same number of fragments – and that it cannot reject fragments based on accumulation effects. In contrast we perform all computations in a single geometry pass, and support rejection of fragments based on the current sorted lists of fragments.

The second fastest approach is paged per-pixel linked lists. As shown in table 2, using 64 bits the HA-buffer is slower than paged linked-lists using texture memory – however the expected performance boost of native `atomicMax64` support gives an equivalent performance. We also modified the code of paged linked-list to store only depth in 32 bits textures and compared it to our 32 bits approach which does not suffer from penalties. In this scenario we also achieve a similar frame rate.

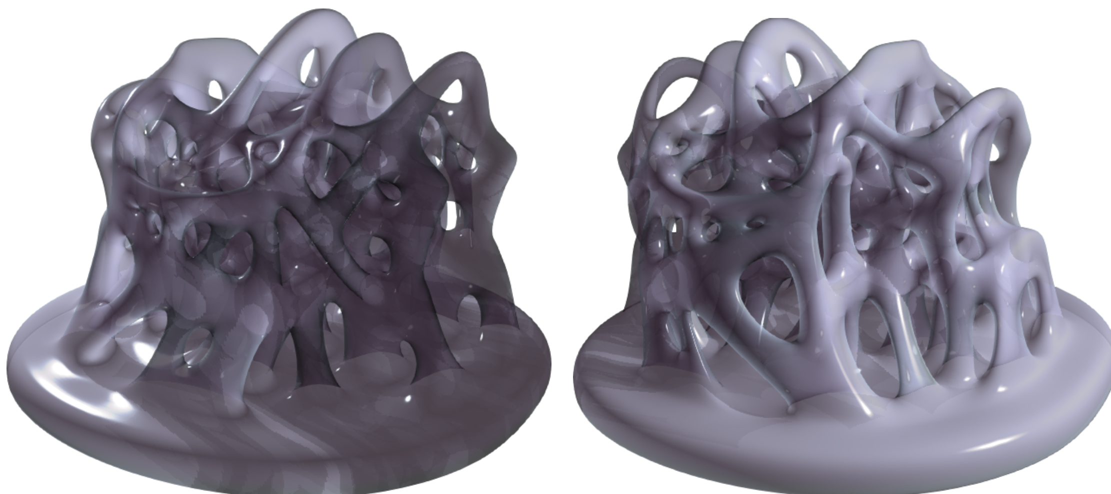


Figure 6: Two viewpoints of a glass sculpture rendered with translucent HA-buffer shadows. (Fixed light, $V = 1024^2 H = 1450^2$, FXAA enabled). *Left:* 1021046 frag., load: 48.6% (16.1 MiB) max depth: 16, FPS: 118.0 (shadows on). *Right:* 1001412 frag., load: 47.6% (16.1 MiB) max depth: 18, FPS: 122.5 (shadows on).

In conclusion our approach exhibits similar performance than state of the art approaches, with similar memory requirements. Note however that there is a speed–memory tradeoff: At compact memory usage our performance is lower, but applications that can afford more memory directly see a performance increase.

Shadows The HA-buffer can be used for translucent shadow effects: A HA-buffer is rendered from the light view and used as a shadow map. Since it contains all fragment colors and opacities ordered by depth, translucent effects may be applied. When the HA-buffer for the view is rendered, the HA-buffer shadow map is used for shading fragments. The HA-buffer shadow map is only updated when the light moves. Figures 2, 7 and 6 illustrates this.

Early culling Figure 2 illustrates early culling on a complex object. It is worth noting that the fill-rate remains constant: This outlines the efficiency of early culling since the FPS proportionally increases with the reduction in number of fragments.

Analysis: Stress test Figure 8 is a stress-test of our approach: We render random quads of a same size at various depth in orthographic projection. The quads are then slowly brought together, until they all exactly stack up. The number of fragments remains constant through this process, but the depth complexity in pixels not only increases but becomes more and more uniform. As expected, once all pixels have a same large amount of fragments performance degrades. One reason is that parallelism cannot hide the cost of long lists by processing many small lists simultaneously: The work load is the same everywhere. The other reason is that the sort complexity is in $\Omega(k^2)$ (Section 4.1.2) and therefore the cost increases significantly with larger number of fragments. Of course, such a catastrophic case is highly unlikely in practice, and as shown Figure 8 using early culling alleviates the issue even in the worst cases.

Technique	Frame rate
Dual depth peeling (8 passes)	140 FPS
Depth peeling (16 passes)	82 FPS
Stencil routed K-Buffer (16 samples)	226 FPS
Paged linked lists (global)	157 FPS
Paged linked lists (texture)	**246** FPS
HA-buffer 50% (64 bits)	187 FPS (257*)
HA-buffer 80% (64 bits)	83 FPS (114*)
Paged linked lists (32 bits, global)	180 FPS
Paged linked lists (32 bits, texture)	290 FPS
HA-buffer 50% (32 bits)	**298** FPS
HA-buffer 80% (32 bits)	181 FPS

Table 2: Performance of different A-buffer techniques measured in our benchmark. The view contains 938174 fragments, and a maximum depth complexity of 10. The model has 871414 triangles (Stanford Dragon model). (*) For the 64 bits version we give in parenthesis projected performance without `atomicMax64` emulation and compiler issue.

6.3 Constructive Solid Geometry

We implement a simple CSG renderer by walking along the ordered lists of fragments, keeping track of the entry/exit points of the primitives. A bit field with one bit per primitive indicates whether we are inside/outside. As we walk along the list of fragments, the bit field and the boolean expression are updated in order to check whether the current fragment is on the boundary of the desired CSG model. A sample CSG rendering is shown Figure 9. The main challenge with CSG is the high depth complexity and the large number of fragments per pixel involved in even simple shapes.

Our implementation is quite naive and different approaches and optimizations should be experimented with, inspired by state of the art work [13, 11].

7 Limitations, future work

As we have seen the main limitation of our approach is related to the increase in complexity when long fragment lists appear in *all* pixels. This is fortunately a rare situation, and it can be alleviated through the use of early culling in transparency applications.

For future work we would like to consider conservative tests regarding CSG. It seems interesting to revisit the works that have been done in the context of depth peeling and see how these would apply to our approach.

8 Conclusion

When looking at the implementation, the hash mechanism we describe seems a simple modification of coherent hashing to deal with fragment lists. However, the important contribution of this work is to show that this mechanism offers unique properties regarding storage and sorting of fragments. It would be easy to miss these if hashing was naively applied to an A-buffer construction.

The good performance of our approach is due to the simplicity of its implementation and the coherence properties of the hash function. We however do not think this is the fastest approach

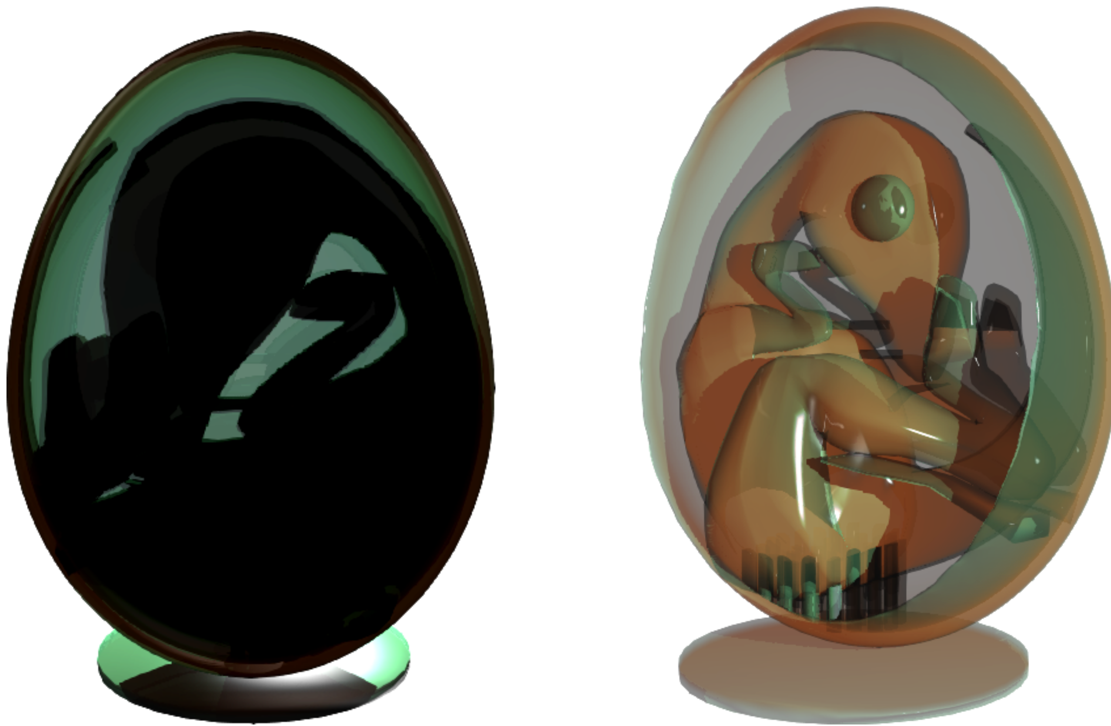


Figure 7: *Left:* The shadow projected by the baby dinosaur on the back-lit egg shell is encoded in a second HA-buffer, used as a shadow map. This back-lit shadow is only possible thanks to the translucent shadow map. View: 744208 frag., load: 71% (8 MiB) max depth: 24, FPS: 119.5 (shadows on). *Right:* Front view with different lighting and opacity. View: 701822 frag., load: 67% (8 MiB) max depth: 20, FPS: 158 (shadows on). (Fixed light, $V = 1024^2, H = 1024^2$, FXAA enabled).

for A-buffer. It will be especially efficient on scenes of moderate fragment complexity with high geometric complexity. Besides performances, we would like to re-emphasize the properties of our approach: A unique geometry pass is immediately followed by the rendering pass, both utilizing a coherent memory access pattern. No separate sorting is necessary and conservative early culling is simple to add. There are no pointers and the access loop does not even involve dependent reads in memory. The construction and access code fit within 50 lines of GLSL code. Together, these properties cover most of the advantages of prior methods. These benefits make the approach versatile: it takes little time to implement a variety of applications or mix several viewpoints with sorted fragments for shadowing effects.

We hope our approach will facilitate the wide adoption of transparency and CSG as standard effects in real-time rendering engines.

Acknowledgments This work was supported by ERC grant ShapeForge (StG-2012-307877). The authors thank NVIDIA for hardware donation.

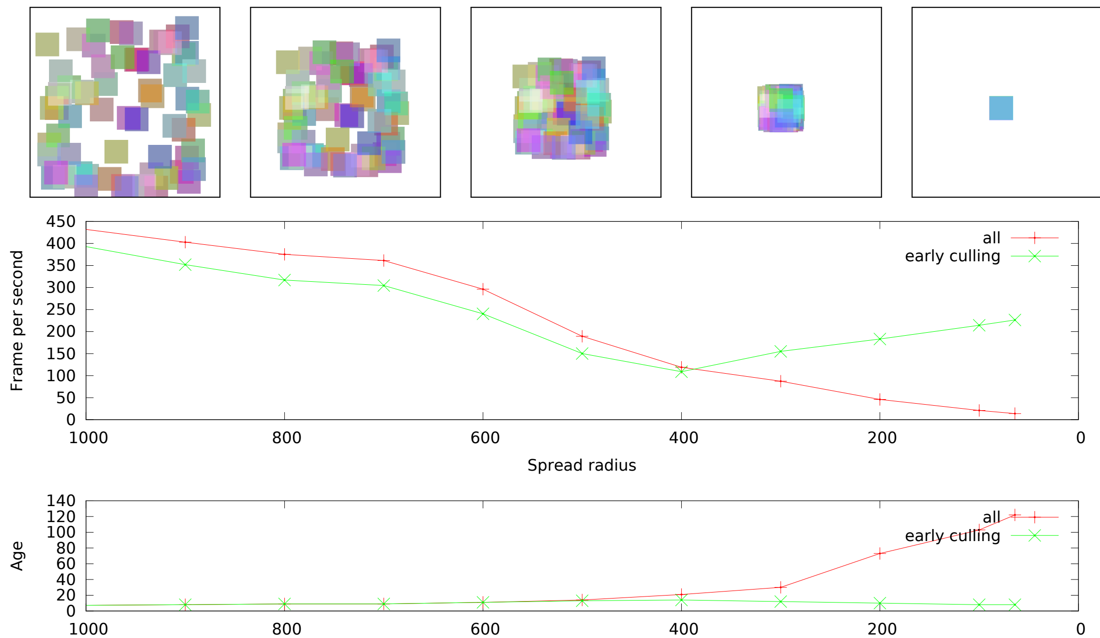


Figure 8: Stress test of our technique $V = 1024^2$, $H = 1280^2$, frag.:1048576, load: 64%. The top row shows screenshots at various times. The squares have 0.5 opacity and are pulled together until exactly stacking, an unfavorable situation for our algorithm. The middle row shows the frame rate, and the bottom row the maximum age in the table. The red curve is our standard scheme. Note the large increase towards the end, as all pixels get the exact same workload. Fortunately, when enabling early culling (green curve) the negative effects are compensated by the large number of culled fragments (threshold for culling: 99% opacity).

References

[1] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the GPU. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2009)*, 28(5), 2009.

[2] Loren Carpenter. The A-buffer, an antialiased hidden surface method. *SIGGRAPH*, 18(3):103–108, 1984.

[3] Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin hood hashing (preliminary report). In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 281–288. IEEE, 1985.

[4] Cyril Crassin. Fast and accurate single-pass A-buffer using OpenGL 4.0+, 2010. `http://blog.icare3d.org/2010/06/fast-and-accurate-single-pass-buffer.html`.

[5] Cyril Crassin. OpenGL 4.0+ A-buffer v2.0: Linked lists of fragment pages, 2010. `http://blog.icare3d.org/2010/07/opengl-40-abuffer-v20-linked-lists-of.html`.

[6] Eric Enderton, Erik Sintorn, Peter Shirley, and David Luebke. Stochastic transparency. In *ACM Symposium on Interactive 3D Graphics and Games*, pages 157–164. ACM, 2010.

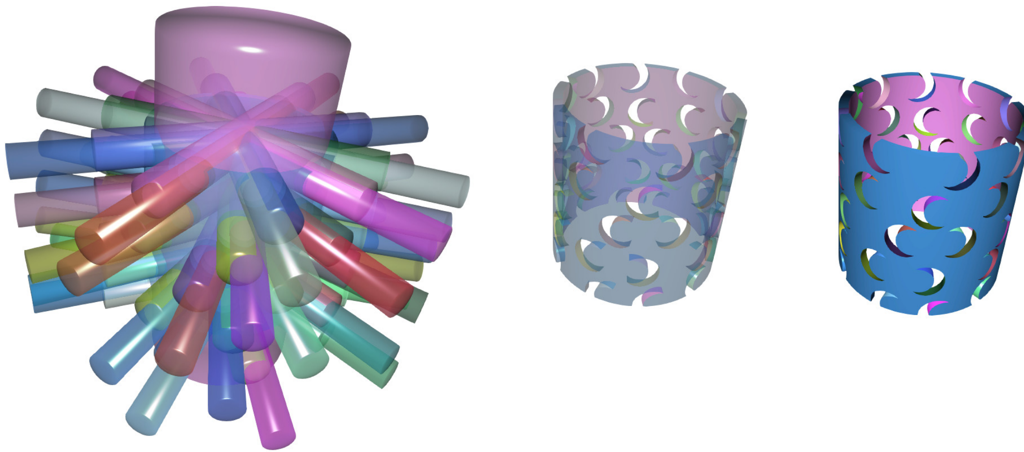


Figure 9: Example of CSG operations with the HA-buffer. *Left:* 52 primitives are involved. *Middle, Right:* The final shape in transparent and opaque versions. View: 2244900 frag., load: 54%, max depth: 52, FPS: 39. ($V = 1024^2, H = 2048^2$, FXAA)

[7] Cass Everitt. Interactive order-independent transparency, 2001. Technical report, NVIDIA Corporation.

[8] Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. Coherent parallel hashing. *ACM Transactions on Graphics*, 30(6):161, 2011.

[9] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[10] John Hable and Jarek Rossignac. Blister: GPU-based rendering of boolean combinations of free-form triangulated shapes. *ACM Transactions on Graphics*, 24(3):1024–1031, 2005.

[11] John Hable and Jarek Rossignac. CST: Constructing solid trimming for rendering BReps and CSG. *Transactions on Visualization and Computer Graphics*, 13(5):1004–1014, 2007.

[12] N.P. Jouppi and C.F. Chang. Z^3: an economical hardware technique for high-quality antialiasing and transparency. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 85–93. ACM, 1999.

[13] Florian Kirsch and Jürgen Döllner. Rendering techniques for hardware-accelerated image-based CSG. *Journal of WSCG*, 12(2):221–228, 2004.

[14] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. *ACM Transactions on Graphics*, 25(3), 2006.

[15] F. Liu, M.C. Huang, X.H. Liu, and E.H. Wu. Freepipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 75–82. ACM, 2010.

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

[17] M. Maule, J.L.D. Comba, R. Torchelsen, and R. Bastos. Memory-efficient order-independent transparency with dynamic fragment buffer. In *Proc. 25th Conference on Graphics, Patterns and Images (SIBGRAPI)*, pages 134–141. IEEE, 2012.

[18] Kevin Myers and Louis Bavoil. Stencil routed A-buffer. In *ACM SIGGRAPH 2007 sketches*. ACM, 2007.

[19] Jonathan Shade, Steven Gortler, Li-wei He, and Richard Szeliski. Layered depth images. In *Proc. SIGGRAPH*, pages 231–242. ACM, 1998.

[20] J.C. Yang, J. Hensley, H. Grün, and N. Thibieroz. Real-time concurrent linked list construction on the gpu. *Computer Graphics Forum*, 29(4):1297–1304, 2010.

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399