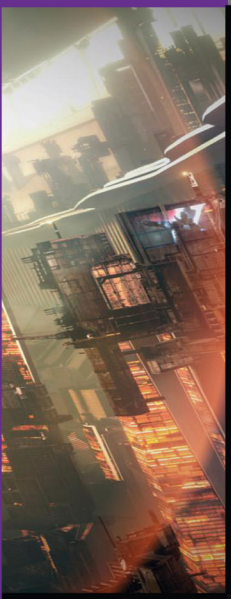
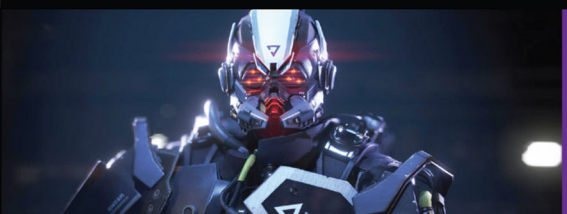
Advanced Rendering Techniques

CRC Press
Taylor & Francis Group
AN A K PETERS BOOK



GPU Pro^5



Edited by Wolfgang Engel

GPU Pro^5

This page intentionally left blank

GPU Pro^5

Advanced Rendering Techniques

Edited by Wolfgang Engel

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN A K PETERS BOOK

Cover art: Screenshots from *Killzone: Shadow Fall* by Guerrilla Games. Courtesy of Michal Valient.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20140227

International Standard Book Number-13: 978-1-4822-0864-1 (eBook - PDF)

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

This page intentionally left blank

Acknowledgments

The *GPU Pro: Advanced Rendering Techniques* book series covers ready-to-use ideas and procedures that can help to solve many of your daily graphics-programming challenges.

The fifth book in the series wouldn't have been possible without the help of many people. First, I would like to thank the section editors for the fantastic job they did. The work of Wessam Bahnassi, Marius Bjørge, Carsten Dachsbacher, Michal Valient, and Christopher Oat ensured that the quality of the series meets the expectations of our readers.

The great cover screenshots were contributed by Michal Valient from Guerrilla Games. They show the game *Killzone: Shadow Fall*.

The team at CRC Press made the whole project happen. I want to thank Rick Adams, Charlotte Byrnes, Kari Budyk, and the entire production team, who took the articles and made them into a book.

Special thanks go out to our families and friends, who spent many evenings and weekends without us during the long book production cycle.

I hope you have as much fun reading the book as we had creating it.

—Wolfgang Engel

P.S. Plans for an upcoming *GPU Pro 6* are already in progress. Any comments, proposals, and suggestions are highly welcome (wolfgang.engel@gmail.com).

This page intentionally left blank

Web Materials

Example programs and source code to accompany some of the chapters are available on the CRC Press website: go to http://www.crcpress.com/product/isbn/9781482208634 and click on the "Downloads" tab.

The directory structure closely follows the book structure by using the chapter number as the name of the subdirectory.

General System Requirements

- The DirectX June 2010 SDK (the latest SDK is installed with Visual Studio 2012).
- DirectX9, DirectX 10 or even a DirectX 11 capable GPU are required to run the examples. The chapter will mention the exact requirement.
- The OS should be Microsoft Windows 7, following the requirement of DirectX 10 or 11 capable GPUs.
- Visual Studio C++ 2012 (some examples might require older versions).
- 2GB RAM or more.
- The latest GPU driver.

Updates

Updates of the example programs will be posted on the website.

This page intentionally left blank

I Rendering

Real-time rendering is not only an integral part of this book series, it is also an exciting field where one can observe rapid evolution and advances to meet the ever-rising demands of game developers and game players. In this section we introduce new techniques that will be interesting and beneficial to both hobbyists and experts alike—and this time these techniques do not only include classical rendering topics, but also cover the use of rendering pipelines for fast physical simulations.

The first chapter in the rendering section is "Per-Pixel Lists for Single Pass A-Buffer," by Sylvain Lefebvre, Samuel Hornus and Anass Lasram. Identifying all the surfaces projecting into a pixel has many important applications in computer graphics, such as computing transparency. They often also require ordering of the fragments in each pixel. This chapter discusses a very fast and efficient approach for recording and simultaneously sorting of all fragments that fall within a pixel in a single geometry pass.

Our next chapter is "Reducing Texture Memory Usage by 2-Channel Color Encoding," by Krzysztof Kluczek. This chapter discusses a technique for compactly encoding and efficiently decoding color images using only 2-channel textures. The chapter details the estimation of the respective 2D color space and provides example shaders ready for use.

"Particle-Based Simulation of Material Aging," by Tobias Günther, Kai Rohmer, and Thorsten Grosch describes a GPU-based, interactive simulation of material aging processes. Their approach enables artists to interactively control the aging process and outputs textures encoding surface properties such as precipitate, normals and height directly usable during content creation.

Our fourth chapter, "Simple Rasterization-Based Liquids," is by Martin Guay. He describes a powerful yet simple way of simulating particle-based liquids on the GPU. These simulations typically involve sorting the particles into spatial acceleration structures to resolve inter-particle interactions. In this chapter, Martin details how this costly step can be sidestepped with splatting particles onto textures, i.e., making use of the rasterization pipeline, instead of sorting them.

—Carsten Dachsbacher

This page intentionally left blank

1

Per-Pixel Lists for Single Pass A-Buffer

Sylvain Lefebvre, Samuel Hornus, and Anass Lasram

1.1 Introduction

Real-time effects such as transparency strongly benefit interactive modeling and visualization. Some examples can be seen Figure 1.1. The rightmost image is a screenshot of our parametric Constructive Solid Geometry (CSG) modeler for 3D printing, IceSL [Lefebvre 13]. Modeled objects are rendered in real time with per-pixel boolean operations between primitives.

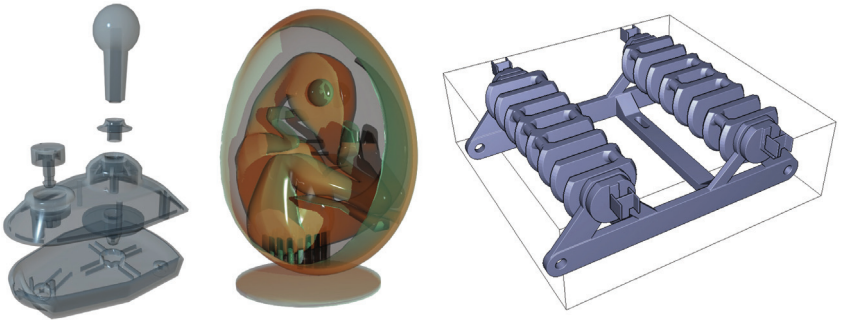


Figure 1.1. Left: Joystick model rendered with the Pre-Open A-buffer technique described in this chapter, on a GeForce Titan. 539236 fragments, max depth: 16, FPS: 490. Middle: Dinosaur in Egg, rendered with transparent surfaces and shadows using two A-buffers. Right: A robot body modeled with 193 solid primitives in boolean operations (CSG), rendered interactively with the Pre-Open A-buffer technique (modeler: IceSL). [Joystick by Srepmup (Thingiverse, 30198), Egg Dinosaur by XXRDESIGNS (Thingiverse, 38463), Spidrack by Sylefeb (Thingiverse, 103765).]

These effects have always been challenging for real-time rasterization. When the scene geometry is rasterized, 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 that are farther away than the current closest, using the Z-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, a list of fragments is gathered for each pixel. Second, the lists are sorted by depth and rendering is performed by traversing the lists, either accumulating opacity and colors (for transparency effects), or applying boolean operations to determine which fragment is visible (for rendering a CSG model). 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. Most techniques for fragment accumulation implement a form of A-buffer [Carpenter 84]. 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*, as visualized in Figure 1.3 (top-right). For a review of A-buffer techniques for transparency we refer the reader to the survey by Maule et al. [Maule et al. 11].

In this chapter we introduce and compare four different techniques to build and render from an A-buffer in real time. One of these techniques is well known while the others are, to the best of our knowledge, novel. We focus on scenes with moderate or sparse depth complexity; the techniques presented here will not scale well on extreme transparency scenarios. In exchange, their implementation is simple and they integrate directly in the graphics API; a compute API is not necessary. All our techniques build the A-buffer in a single geometry pass: the scene geometry is rasterized once per frame.

A drawback of storing the fragments first and sorting them later is that some fragments may in fact be unnecessary: in a transparency application the opacity of the fragments may accumulate up to a point where anything located behind makes no contribution to the final image. Two of the techniques proposed here afford for a conservative early-culling mechanism: inserted fragments are always sorted in memory, enabling detection of opaque accumulation.

The companion code includes a full implementation and benchmarking framework.

1.1.1 Overview

An A-buffer stores a list of fragments for each pixel. Sorting them by increasing or decreasing depth are both possible. However, the sorting technique that we describe in Section 1.3 is easier to explain and implement for decreasing values as we walk along the list. Adding to that, early culling of negligible fragments

is possible for transparency rendering only when the fragments are sorted front-to-back. In order to meet both requirements for the described techniques, we consistently sort in decreasing order and obtain a front-to-back ordering by inverting the usual depth value of a fragment: if the depth z of a fragment is a `float` in the range $[-1, 1]$, we transform it in the pixel shader into the integer $\lfloor S(1-z)/2 \rfloor$, where S is a scaling factor (typically $2^{32}-1$ or $2^{24}-1$).

Our techniques rely on a buffer in which all the fragments are stored. We call it the *main buffer*. Each fragment is associated with a cell in the main buffer where its information is recorded. Our techniques comprise three passes: a Clear pass is used to initialize memory, then a Build pass assembles a list of fragments for each pixel and finally a Render pass accumulates the contribution of the fragments and writes colors to the framebuffer.

The four techniques differ along two axes. The first axis is the scheduling of the sort: when do we spend time on depth-sorting the fragments associated with each pixel? The second axis is the memory allocation strategy used for incrementally building the per-pixel lists of fragments. We now describe these two axes in more detail.

1.1.2 Sort Strategies

We examine two strategies for sorting the fragments according to their depth. The first one, Post-sort, stores all the fragments during the Build pass and sorts them only just prior to accumulation in the Render pass: the GLSL shader copies the pixel fragments in local memory, sorts them in place, and performs in-order accumulation to obtain the final color.

The second strategy, Pre-sort, implements an insertion-sort during the Build pass, as the geometric primitives are rasterized. At any time during the rasterization, it is possible to traverse the fragments associated with a given pixel in depth order.

Both strategies are summarized in Table 1.1.

Each has pros and cons: In the Pre-sort method, insertion-sort is done in the slower global memory, but the method affords for early culling of almost invisible fragments. It is also faster when several Render passes are required on the same A-buffer, since the sort is done only once. This is for instance the case when CSG models are sliced for 3D printing [Lefebvre 13].

Pass	**Rasterized geometry**	Post-sort	Pre-sort
Clear	fullscreen quad	clear	clear
Build	scene triangles	insert	insertion-sort
Render	fullscreen quad	sort, accumulate	accumulate

Table 1.1. Summary of the Post-sort and Pre-sort sorting strategies.

In the Post-sort method, sorting happens in local memory, which is faster but limits the maximum number of fragments associated with a pixel to a few hundred. Allocating more local memory for sorting more fragments increases register pressure and reduces parallelism and performance.

1.1.3 Allocation Strategies

In addition to the scheduling of the sort, we examine two strategies for allocating cells containing fragment information in the main buffer. The first one, Lin-alloc, stores fragments in per-pixel linked-lists and allocates fresh cells linearly from the start of the buffer to its end. Since many allocations are done concurrently, the address of a fresh cell is obtained by atomically incrementing a global counter. Additional memory is necessary to store the address of the first cell (head) of the list of fragments of each pixel. Section 1.2 details the Lin-alloc strategy.

The second strategy that we examine, Open-alloc, is randomized and somewhat more involved. To each pixel p we associate a pseudo-random sequence of cell positions in the main buffer: $(h(p,i))_{i\geq 1}$, for i ranging over the integers. In the spirit of the "open addressing" techniques for hash tables, the cells at positions $h(p,i)$ are examined by increasing value of i until an empty one is found. A non-empty cell in this sequence may store another fragment associated with pixel p or with a different pixel q. Such a *collision* between fragments must be detected and handled correctly. Section 1.3 details the Open-alloc strategy.

The combination of two allocation strategies (Lin-alloc and Open-alloc) with two schedules for sorting (Post-sort and Pre-sort) gives us four variations for building an A-buffer: Post-Lin (Sections 1.2.1 and 1.2.2), Pre-Lin (Section 1.2.3), Post-Open (Section 1.3.2) and Pre-Open (Section 1.3.3).

Section 1.4.1 details how fragments are sorted in local memory in the Render pass of the Post-sort method. Some memory management issues, including buffer resizing, are addressed in Section 1.5, and information about our implementation is given in Section 1.6. In Section 1.7, we compare these four variations, as implemented on a GeForce 480 and a GeForce Titan.

1.2 Linked Lists with Pointers (Lin-alloc)

The first two approaches we describe construct linked lists in each pixel, allocating data for new fragments linearly in the main buffer. A single cell contains the depth of the fragment and the index of the next cell in the list. Since no cell is ever removed from a list, there is no need for managing a free list: allocating a new cell simply amounts to incrementing a global counter `firstFreeCell` that stores the index of the first free cell in the buffer. The counter `firstFreeCell` is initialized to zero. The increment is done atomically to guarantee that every thread allocating new cells concurrently does obtain a unique memory address. A second array,

called `heads`, is necessary to store the address of the head cell of the linked list of each pixel.

Having a lot of threads increment a single global counter would be a bottleneck in a generic programing setting (compute API). Fortunately, GLSL fragment shaders feature dedicated counters for this task, via the `ARB_shader_atomic_counters` extension. If these are not available, it is possible to relieve some of the contention on the counter by allocating K cells at once for a list (typically $K = 4$). To obtain such a paged allocation scheme, the thread atomically increases the global counter by K and uses a single bit in each head pointer as a local mutex when inserting fragments in this page of K cells. The technique is described in full detail by Crassin [Crassin 10], and is implemented in the accompanying code (see `implementations.fp`, function `allocate_paged`). In our tests, the dedicated counters always outperformed the paging mechanism. However, if a generic atomic increment is used instead then the paging mechanism is faster. We use a single dedicated atomic counter in all our performance tests (Section 1.7).

We now describe the two techniques based on the Lin-alloc cell allocation strategy: Post-Lin and Pre-Lin.

1.2.1 Building Unsorted Lists (Post-Lin)

The simplest approach to building an unsorted list of fragments is to insert new fragments at the head of the pixel list. A sample implementation is provided in Listing 1.1.

In line 7, a cell position `fresh` is reserved and the counter is incremented. The operation must be done atomically so that no two threads reserve the same position in the buffer. It is then safe to fill the cell with relevant fragment data in lines 8 and 9. Finally, indices are exchanged so that the cell `buffer[fresh]` becomes the new head of the list.

Later, in Section 1.4.1, we describe how the fragments associated with each pixel are gathered in a thread's local memory and sorted before rendering.

```
1 const int gScreenSize     = gScreenW * gScreenH;
2 atomic_uint firstFreeCell = 0;
3 int heads[gScreenSize];
4 LinkedListCell_t buffer[gBufferSize];
5
6 void insertFront(x, y, float depth, Data data) {
7  const int fresh = atomicCounterIncrement(firstFreeCell);
8  buffer[fresh].depth = depth;
9  buffer[fresh].data  = data;
10 buffer[fresh].next  = atomicExchange(&heads[x+y*gScreenW], fresh);
11 }
```

Listing 1.1. Insertion at the head of a linked list.

```
atomic_uint firstFreeCell = gScreenSize;
Data     databuf[gBufferSize];
uint64_t buffer[gBufferSize + gScreenSize];

uint64_t pack(uint32_t depth, uint32_t next) {
 return ((uint64_t)depth << 32) | next;
}

void insertFrontPack(x, y, uint32_t depth, data) {
 const int fresh = atomicCounterIncrement(firstFreeCell);
 databuf[fresh-gScreenSize] = data;
 buffer[fresh] = atomicExchange(buffer+x+y*gScreenW,
                                pack(depth, fresh));
}
```

Listing 1.2. Insertion at the head of a linked list with packing.

1.2.2 Packing depth and next Together

In order to facilitate the understanding of later sections and render the exposition more uniform with Section 1.3, this section introduces specific changes to the buffer layout. We illustrate this new layout by describing an alternative way to build unsorted linked-lists.

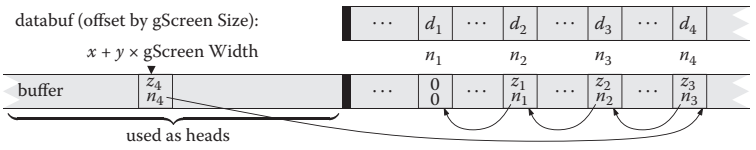
The following changes are done: First, all fragment data except `depth` are segregated in a specific data buffer, that we call `databuf`. Second, the `depth` and the `next` fields are packed in a single 64-bits word. Third, the main buffer is enlarged with as many cells as pixels on screen. These additional cells at the beginning of the buffer are used just like the `heads` array in Listing 1.1.

Listing 1.2 shows the new insertion procedure. Two observations should be kept in mind:

- We must follow the `next` index $n-1$ times to access the depth of the nth fragment, and n times to access its other data.
- Notice the new initial value of `firstFreeCell` and the offset needed when accessing the fragment data.

We keep this buffer layout throughout the remainder of the chapter.

The following diagram illustrates the position of four fragments $f_i, i = 1, 2, 3, 4$, inserted in this order, with depth z_i and data d_i, associated with a pixel with coordinates (x, y). Observe how each cell of the main buffer packs the depth z_i of a fragment and the index n_i of the next item in the list. Note that with this layout the index n_i of the fragment following f_i never changes.



```
1 uint_32_t getNext(uint64_t cell) {
2   return cell; // extract least significant 32 bits
3 }
4
5 void insertSorted(x, y, uint32_t depth, Data data) {
6   const int fresh = atomicCounterIncrement(firstFreeCell);
7   buffer[fresh]   = 0; // 64-bits zero
8   memoryBarrier(); // make sure init is visible
9   databuf[fresh]  = data;
10  uint64_t record = pack(depth,fresh), old, pos;
11  pos = gScreenW * y + x; // start of the search
12  while((old=atomicMax64(buffer+pos, record)) > 0) {
13    if( old > record ) { // go to next
14      pos = getNext(old);
15    } else { // inserted! update record itself
16      pos = getNext(record);
17      record = old;
18 } } }
```

Listing 1.3. Insertion-sort in a linked list.

1.2.3 Building Sorted Lists with Insertion-Sort (Pre-Lin)

It is also possible to perform parallel insertions at any position inside a linked list, and therefore, to implement a parallel version of "insertion-sort." General solutions to this problem have been proposed. In particular, our approach is inspired by that of Harris [Harris 01], albeit in a simplified setting since there is no deletion of single items. A sample implementation is provided in Listing 1.3.

The idea is to walk along the linked list until we find the proper place to insert the fragment. Contrary to the implementation of Harris, which relies on an atomic compare-and-swap, we use an atomic max operation on the cells of the main buffer at each step of the walk (line 12). Since the `depth` is packed in the most significant bits of a cell (line 10), the `atomicMax` operation will overwrite the fragment stored in the buffer if and only if the new fragment depth is larger. In all cases the value in the buffer prior to the max is returned in the variable `old`.

If the new fragment has smaller depth (line 13) then the buffer has not changed and the new fragment has to be inserted further down the list: we advance to the next cell (line 14).

If the new fragment has a larger depth (line 15) then it has been inserted by the `atomicMax`. At this point the new fragment has been inserted, but the remainder of the list has been cut out: the new fragment has no follower (line 7). We therefore restart the walk (line 16), this time trying to insert `old` as the next element of `record` (line 17). That walk will often succeed immediately: the `atomicMax` operation will be applied at the end of the first part of the list and will return zero (line 12). This single operation will merge back both parts of the list. However there is an exception: another thread may have concurrently inserted more elements, in which case the walk will continue until all elements have

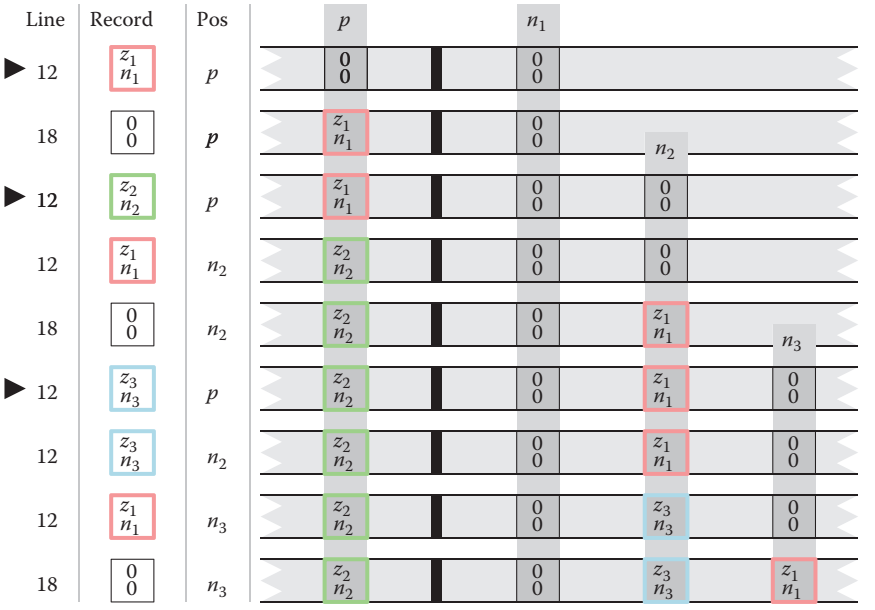


Figure 1.2. Insertion of three fragments into the list of pixel p. Their respective depths are z_1, z_2 and z_3 with $z_2 > z_3 > z_1$. The triangles on the left indicate the start of the insertion of a new fragment. Each line is a snapshot of the variables and main buffer state at each iteration of the `while` loop at lines 12 or 18 of Listing 1.3.

been properly re-inserted. Figure 1.2 illustrates the insertion of three fragments associated with a single pixel p.

Compared to the approach of [Harris 01] based on a 32-bit atomic compare and swap, our technique has the advantage of compactness and does not require synchronization in the main loop. In particular the loop in Listing 1.3 can be rewritten as follows:

```
1   while((old=atomicMax64(buffer+pos, record)) > 0) {
2     pos    = getNext( max(old,record) );
3     record = min(old,record);
4   }
```

Please refer to the accompanying source code for an implementation of both approaches (file `implementations.fp`, functions `insert_prelin_max64` and `insert_prelin_cas32`).

1.3 Lists with Open Addressing (Open-alloc)

In the previous section, a cell was allocated by incrementing a global counter, and each cell in a list had to store the index of the next cell in that list. This is the traditional linked-list data structure.

In this section, we describe a different way to allocate cells in the main buffer and traverse the list of fragments associated with a given pixel. This technique frees us from storing the index of the next cell, allowing more fragments to fit in the same amount of memory. It does come with some disadvantages as well, in particular the inability to store more that 32 bits of data per fragment.

We start with a general introduction of this allocation strategy and then introduce the two techniques based on it, Post-Open and Pre-Open.

1.3.1 Insertion

For each pixel p, we fix a sequence of cell positions in the main buffer, $(h(p,i))_{i\geq 1}$ and call it a *probe sequence*. The function h is defined as

$$h(p,i) = p + o_i \mod H,$$

or, in `C` speak, `(p + offsets[i]) % gBufferSize`.

where $H =$ `gBufferSize` is the size of the main buffer. The sequence $(o_i)_{i\geq 1}$ should ideally be a random permutation of the set of integers $[0..H-1]$, so that the probe sequence $(h(p,i))_{i\geq 1}$ covers all the cells of the main buffer. We call $(o_i)_{i\geq 1}$ the *sequence of offsets*. In practice this sequence is represented with a fixed-length array of random integers, which we regenerate before each frame. The fragments associated with pixel p are stored in the main buffer at locations indicated by the probe sequence. When a fragment covering pixel p is stored at position $h(p,i)$, we say that it has *age* i, or that i is the age of this stored fragment.

There are two interesting consequences to using the probe sequence defined by function h. First, note that the sequence of offsets is independent of the pixel position p. This means that the probe sequence for pixel q is a translation of the probe sequence for pixel p by the vector $q-p$. During the rasterization, neighboring threads handle neighboring pixels and in turn access neighboring memory locations as each is traversing the probe sequence of its corresponding pixel. This *coherence* in the memory access pattern eases the stress of the GPU memory bus and increases memory bandwidth utilization. It was already exploited by García et al. for fast spatial hashing [García et al. 11].

Second, assuming that H is greater than the total number of screen pixels, then the function h becomes invertible in the sense that knowing $h(p,i)$ and i is

enough to recover p as

$$p = h(p, i) - o_i \mod H,$$

or, in `C` speak, `(hVal + gBufferSize - offsets[i]) % gBufferSize`.

Let us define $h^{-1}(v, i) = v - o_i \mod H$. The function h^{-1} lets us recover the pixel p, which is covered by a fragment of age i stored in cell v of the main buffer: $p = h^{-1}(v, i)$. In order to compute this inverse given v, the age of a fragment stored in the main buffer must be available. Hence, we reserve a few bits (typically 8) to store that age in the buffer, together with the depth and data of the fragment.

When inserting the fragments, we should strive to minimize the age of the oldest fragment, i.e., the fragment with the largest age. This is particularly important to ensure that when walking along lists of fragments for several pixels in parallel, the slowest thread—accessing old fragments—does not penalize the other threads too much. This maximal-age minimization is achieved during insertion: old fragments are inserted with a higher priority, while young fragments must continue the search of a cell in which to be written.

We define the *load-factor* of the main buffer as the ratio of the number of fragments inserted to the total size of the main buffer.

Collisions. A collision happens when a thread tries to insert a fragment in a cell that already contains a fragment. Collisions can happen since the probe sequence that we follow is essentially random. When the main buffer is almost empty (the load-factor is low), collisions rarely happen. But as the load-factor increases, the chance of collisions increases as well. The open addressing scheme that we have just described works remarkably well even when the load-factor is as high as 0.95.

A collision happens when a thread tries to insert a fragment f_p covering pixel p at position $h(p, i)$ for some i, but the cell at that position already contains a fragment f_q for some other pixel q. We then have $h(p, i) = h(q, j)$ and solve the collision depending on the value of i and j:

- If $j = i$, then $q = p$. The fragment f_q covers the same pixel p; we keep it there and try to insert fragment f_p at the next position $h(p, i+1)$. Alternatively, as in Section 1.3.3, we might compare the depths of both fragments to decide which one to keep at that position and which one to move.

- If $j \neq i$, then pixels p and q are different pixels. In that case, we store the fragment with the largest age in that cell and continue along the probe sequence of the other fragment. More precisely, if $i > j$ then the older fragment f_p replaces f_q in the main buffer and the insertion of the younger fragment f_q is restarted at age $j + 1$, i.e., at position $h(q, j + 1)$ in the main buffer. Note that the value q is not known in advance and must be computed as $q = h^{-1}(h(p, i), j)$. If $i < j$, then fragment f_q does not move

and the search for a free cell for f_p proceeds at age $i + 1$ in the probe sequence of pixel p.

This *eviction* mechanism, whereby an "old" fragment evicts a younger fragment, has been demonstrated to effectively reduce the maximum age of the fragments stored in the main buffer, over all pixels. This property was discovered by Celis and Munro in their technique called *Robin Hood Hashing* [Celis et al. 85].

1.3.2 Building Unsorted Lists (Post-Open)

In this section, we give the full details of the construction of unsorted lists of fragments using the allocation scheme described above.

In the rest of this chapter, we assume that a cell of the main buffer occupies 64 bits, which lets us use atomic operations on a cell, and that the age of a fragment is stored in the most significant bits of the cell:

MSB	age: 8 bits	empty: 24 bits	data: 32 bits	LSB

In this way, the eviction mechanism described above can be safely accomplished using a single call to `atomicMax`.

We use an auxiliary 2D table A that stores, for each pixel p, the age of the oldest fragment associated with p in the main buffer. Thus, $A[p]$ indicates the end of the list of p's fragments; from which we can start the search for an empty cell for the new fragment f_p to be inserted.

The insertion procedure is shown in Listing 1.4. It increments a counter `age` starting from $A[p] + 1$ (line 2) until it finds an empty cell at position $h(p, \texttt{age})$

```
1  void insertBackOA(p, depth, data) {
2    uint     age  = A[p] + 1;
3    uint64_t record = OA_PACK(age,depth,data);
4    int      iter = 0;
5    while (iter++ < MAX_ITER) {
6      uvec2    h   = ( p + offsets[age] ) % gBufSz;
7      uint64_t old = atomicMax(&buffer[h], record);
8      if (old < record) {
9        atomicMax(&A[p], age);
10       if (old == 0) break;
11       uint32_t oage = OA_GET_AGE(old);
12       p    = (h + gBufSz - offsets[oage]) % gBufSz;
13       age  = A[p];
14       record = OA_WRITE_AGE(old, age);
15     }
16     ++age;
17     record = record + OA_INC_AGE;
18 } }
```

Listing 1.4. Insertion in a list with open addressing.

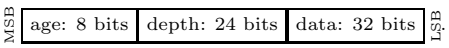
in which the `record` is successfully inserted (line 10). The `record` is tentatively inserted in the buffer at line 7. If the insertion fails, the insertion proceeds in the next cell along the probe sequence (lines 16 and 17). If it succeeds, the table A is updated and if another fragment f' was evicted (`old != 0`), the pixel q covered by f' is computed from the age of f' (line 11) and function h^{-1} (line 12). The insertion of f' continues from the end of the list of fragments for pixel q, given by $A[q] + 1$.

The macro `OA_PACK` packs the age, depth and data of a fragment in a 64-bits word. The age occupies the 8 most significant bits. The macro `OA_WRITE_AGE` updates the 8 most significant bits without touching the rest of the word. Finally, the constant `OA_INC_AGE = ((uint64_t)1<<56)` is used to increment the age in the packed `record`.

1.3.3 Building Sorted Lists with Insertion-Sort (Pre-Open)

In this section, we modify the construction algorithm above so as to keep the list of fragments sorted by depth, by transforming it into an insertion-sort algorithm.

Let f_p be the fragment, associated with pixel p, that we are inserting in the main buffer. When a collision occurs at age i with a stored fragment f'_p associated with the *same* pixel p, we know that both fragments currently have the same age. Therefore, the `atomicMax` operation will compare the cell bits that are lower than the bits used for storing the age. If the higher bits, among these lower bits, encode the depth of the fragment then we ensure that the fragment with largest depth is stored in the main buffer after the atomic operation:



Further, it is possible to show that during the insertion of fragment f_p at age i, if a collision occurs with a fragment f_q with $h(q, j) = h(p, i)$, then $i \leq j$. Thus, the insertion procedure will skip over all stored fragments that are not associated with pixel p (since $i \neq j \Rightarrow q \neq p$) and will correctly keep the fragments associated with p sorted by decreasing depth along the probe sequence of p. The interested reader will find more detail and a proof of correctness of the insertion-sort with open addressing in our technical report [Lefebvre and Hornus 13].

Thus, we obtain an insertion-sort with open addressing simply by packing the depth of the fragment right after its age and always starting the insertion of a fragment at the beginning of the probe sequence. A sample implementation is given in Listing 1.5.

1.4 Post-sort and Pre-sort

In this section we discuss details depending on the choice of scheduling for the sort. We discuss the sort in local memory required for Post-Lin and Post-Open, as well as how to perform early culling with Pre-Lin and Pre-Open.

```
1 void insertSortedOA(p, depth, data) {
2   uint     age  = 1;
3   uint64_t record = OA_PACK(age,depth,data);
4   int      iter = 0;
5   while (iter++ < MAX_ITER) {
6     uvec2    h   = ( p + offsets[age] ) % gBufSz;
7     uint64_t old = atomicMax(&buffer[h], record);
8     if (old < record) {
9       atomicMax(&A[p], age);
10      if (old == 0) break;
11      age = OA_GET_AGE(old);
12      p    = (h + gBufSz - offsets[age]) % gBufSz;
13      record = old;
14    }
15    ++age;
16    record = record + OA_INC_AGE;
17 } }
```

Listing 1.5. Insertion-sort with open addressing.

1.4.1 Post-sort: Sorting in Local Memory

In the Post-sort method, the Build pass accumulates the fragments of each pixel in a list, without sorting them. The Render pass should then sort the fragments prior to accumulating their contributions. This is done in a pixel shader invoked by rasterizing a fullscreen quad. The shader for a given pixel p first gathers all the fragments associated with p in a small array allocated in local memory. The array is then sorted using bubble-sort, in a manner similar to [Crassin 10]. Insertion-sort could also be used and benefit cases where the transparent fragments are rasterized roughly in back-to-front order.

In contrast to the Post-sort techniques, the Pre-sort approaches perform sorting during the Build pass. This allows for early culling of unnecessary fragments, as described in the next section.

1.4.2 Pre-sort: Early Culling

The Pre-sort method has the unique advantage of keeping the fragments sorted at all times. In a transparency application, when a fragment is inserted in a sorted list it is possible to accumulate the opacity of the fragments in front of it in the list. If this opacity reaches a given threshold, we know that the color of fragment f will contribute little to the final image and we can decide to simply discard it. This early culling mechanism is possible only when the lists of fragments are always sorted, and it provides an important performance improvement as illustrated in Section 1.7.

1.5 Memory Management

All four techniques use the main buffer for storing fragments. We discuss in Section 1.5.1 how to initialize the buffer at each new frame. All implementations assumed so far that the buffer is large enough to hold all incoming fragments. This may not be true depending on the selected viewpoint, and we therefore discuss how to manage memory and deal with an overflow of the main buffer in Section 1.5.2.

1.5.1 The Clear Pass

With the Lin-alloc strategy, the beginning of the main buffer that stores the heads of the lists has to be zeroed. This is implemented by rasterizing a fullscreen quad. The global counter for cell allocation has to be initially set to `gScreenSize`. In addition, when using the paged allocation scheme with the Pre-Lin method, an additional array containing for each pixel the free cell index in its last page has to be cleared as well.

With the Open-alloc strategy the entire main buffer has to be cleared: the correctness of the insertion algorithm relies on reading a zero value to recognize a free cell. The array A used to store the per-pixel maximal age has to be cleared as well.

Figure 1.3 shows a breakout of the timings of each pass. As can be seen, the Clear pass is only visible for the Open-alloc techniques, but remains a small percentage of the overall frame time.

1.5.2 Buffer Overflow

None of the techniques we have discussed require us to count the number of fragments before the Build pass. Therefore, it is possible for the main buffer to overflow when too many fragments are inserted within a frame. Our current strategy is to detect overflow *during* frame rendering, so that rendering can be interrupted. When the interruption is detected by the host application, the main buffer size is increased, following a typical size-doubling strategy, and the frame rendering is restarted from scratch.

When using linked lists we conveniently detect an overflow by testing if the global allocation counter exceeds the size of the main buffer. In such a case, the fragment shader discards all subsequent fragments.

The use of open addressing requires a slightly different strategy. We similarly keep track of the number of inserted fragments by incrementing a global counter. We increment this counter *at the end* of the insertion loop, which largely hides the cost of the atomic increment. With open addressing, the cost of the insertion grows very fast as the load-factor of the main buffer nears one (Figure 1.4). For this reason, we interrupt the Build pass when the load-factor gets higher than 10/16.

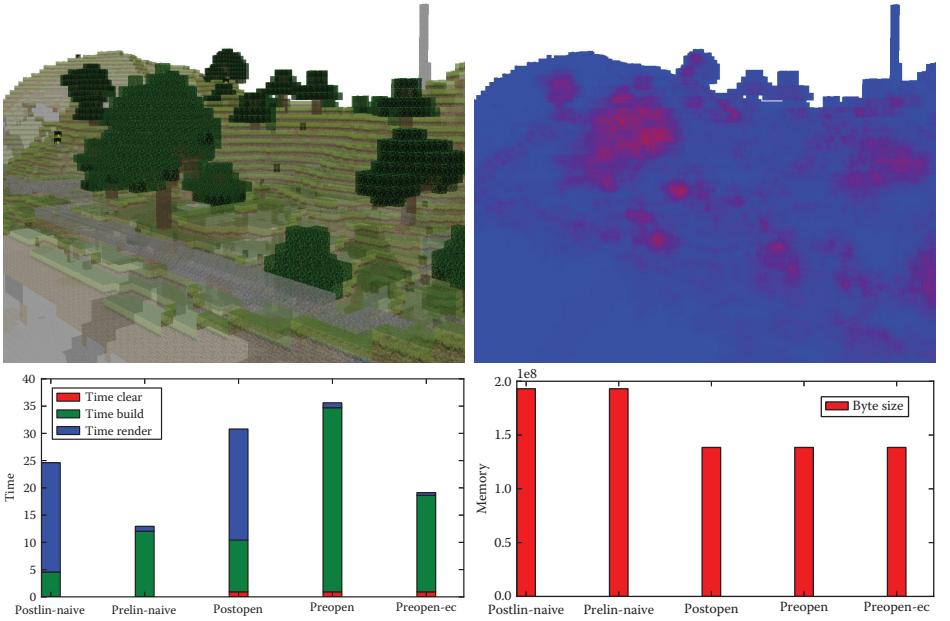


Figure 1.3. The lost empire scene, modeled with Minecraft by Morgan McGuire. The top row (left) shows the textured rendering, with 0.5 opacity (alpha from textures is ignored). The trees appear solid due to a large number of quads. The top row (right) shows a color coded image of the depth complexity. Full red corresponds to 64 fragments (average: 10.3, maximum: 46). The left chart gives the timing breakout for each pass and each technique. The CLEAR pass (red) is negligible for LIN-ALLOC techniques. POST-SORT techniques are characterized by a faster BUILD (green) but a significantly longer RENDER (blue) due to the sort. `preopen-ec` uses early culling, strongly reducing the cost of BUILD (threshold set to 0.95 cumulated opacity). The right chart shows the memory cost of each technique, assuming the most compact implementation. *Load-factor: 0.4.*

1.6 Implementation

We implement all techniques in GLSL fragment programs, using the extension `NV_shader_buffer_store` on NVIDIA hardware to access GPU memory via pointers. We tested our code on both a GeForce GTX 480 (Fermi) and a GeForce Titan (Kepler), using NVIDIA drivers 320.49. We designed our implementation to allow for easy swapping of techniques: each different approach is compiled as a separate DLL. Applications using the A-buffer use a common interface abstracting the A-buffer implementation (see `abuffer.h`).

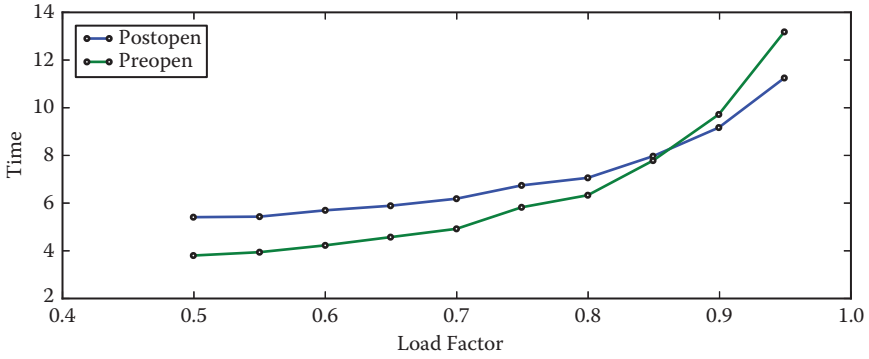


Figure 1.4. Frame time (ms) versus load-factor for open addressing techniques. Note the significant performance drop as the load-factor increases. *GeForce Titan, 320.49, 2.5M fragments, average depth complexity: 2.9.*

An important benefit of the techniques presented here is that they directly fit in the graphics pipeline, and do not require switching to a compute API. Therefore, the BUILD pass is the same as when rendering without an A-buffer, augmented with a call to the insertion code. This makes the techniques easy to integrate in existing pipelines. In addition all approaches require fewer than 30 lines of GLSL code.

Unfortunately implementation on current hardware is not as straightforward as it could be, for two reasons: First, the hardware available to us does not natively support `atomicMax` on 64 bits in GLSL (Kepler supports it natively on CUDA). Fortunately the `atomicMax` 64 bits can be emulated via an `atomicCompSwap` instruction in a loop. We estimated the performance impact to approximately 30% by emulating a 32 bits `atomicMax` with a 32 bits `atomicCompSwap` (on a GeForce GTX480). The second issue is related to the use of atomic operations in loops, inside GLSL shaders. The current compiler seems to generate code leading to race conditions that prevent the loops from operating properly. Our current implementation circumvents this by inserting additional atomic operations having no effect on the algorithm result. This, however, incurs in some algorithms a penalty that is difficult to quantify.

1.7 Experimental Comparisons

We now compare each of the four versions and discuss their performance.

1.7.1 3D Scene Rendering

We developed a first application for rendering transparent, textured scenes. It is included in the companion source code (`bin/seethrough.exe`). Figure 1.3 shows a 3D rendering of a large scene with textures and transparency. It gives the timings breakout for each pass and each technique, as well as their memory cost.

1.7.2 Benchmarking

For benchmarking we developed an application rendering transparent, front facing quads in orthographic projection. The position and depth of the quads are randomized and change every frame. All measures are averaged over six seconds of running time. We control the size and number of quads, as well as their opacity. We use the `ARB_timer_query` extension to measure the time to render a frame. This includes the Clear, Build, and Render passes as well as checking for the main buffer overflow. All tests are performed on a GeForce GTX480 and a GeForce Titan using drivers 320.49. We expect these performance numbers to change with future driver revisions due to issues mentioned in Section 1.6. Nevertheless, our current implementation exhibits performance levels consistent across all techniques as well as between Fermi and Kepler.

The benchmarking framework is included in the companion source code (`bin/benchmark.exe`). The python script `runall.py` launches all benchmarks.

Number of fragments. For a fixed depth complexity, the per-frame time is expected to be linear in the number of fragments. This is verified by all implementations as illustrated Figure 1.5. We measure this by rendering a number of quads perfectly aligned on top of each other, in randomized depth order. The number of quads controls the depth complexity. We adjust the size of the quads to vary the number of fragments only.

Depth complexity. In this experiment we compare the overall performance for a fixed number of fragments but a varying depth complexity. As the size of the per-pixel lists increases, we expect a quadratic increase in frame rendering time. This is verified Figure 1.6. The technique Pre-Open is the most severely impacted by the increase in depth complexity. The main reason is that the sort occurs in global memory, and each added fragment leads to a full traversal of the list via the eviction mechanism.

Early culling. In scenes with a mix of transparent and opaque objects, early culling fortunately limits the depth complexity per pixel. The techniques Pre-Open and Pre-Lin both afford for early culling (see Section 1.4.2). Figure 1.7 demonstrates the benefit of early culling. The threshold is set up to ignore all fragments after an opacity of 0.95 is reached (1 being fully opaque).

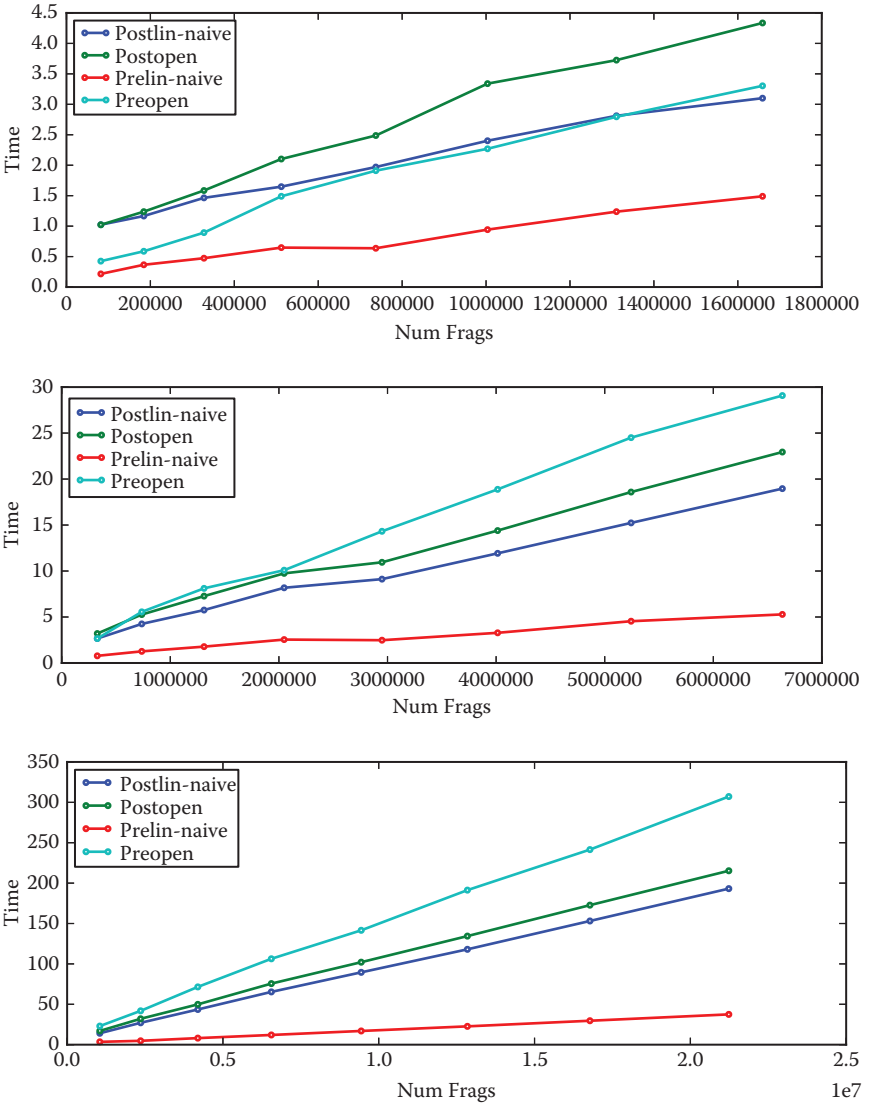


Figure 1.5. Frame time (ms) versus number of fragments. From top to bottom, the depth complexity is 5, 20, and 63 in all pixels covered by the quads. Increase in frame time is linear in number of fragments.

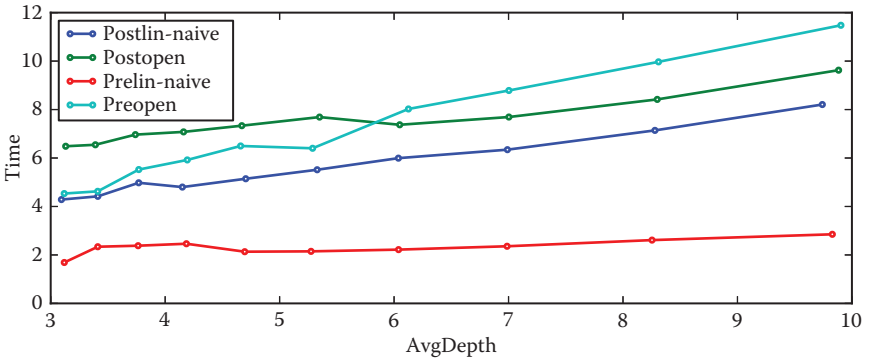


Figure 1.6. Frame time (ms) versus average depth complexity. *GeForce Titan, driver 320.49, load-factor: 0.5, 2.5M fragments.*

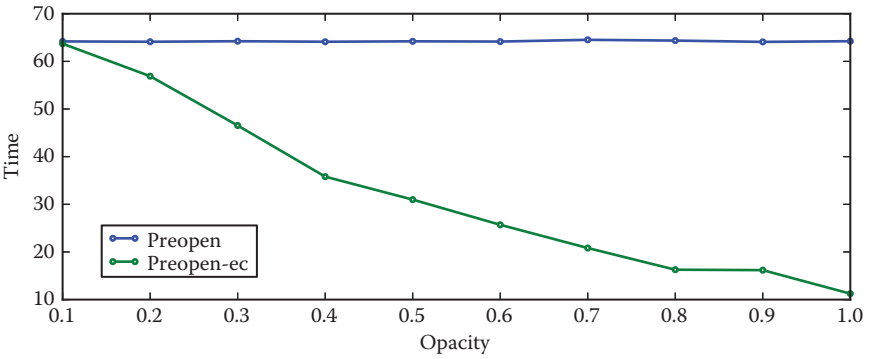


Figure 1.7. Frame time versus opacity for Pre-Open with and without early culling. Early culling (green) quickly improves performance when opacity increases. *GeForce Titan, driver 320.49, load-factor: 0.5, 9.8M fragments*

1.8 Conclusion

Our tests indicate that Pre-Lin has a significant advantage over other techniques, while the Open-alloc cell allocation strategy falls behind. This is, however, not a strong conclusion. Indeed, all of these methods, with the exception of Post-Lin, are penalized by the emulation of the atomic max 64 bits. More importantly, the implementation of the Open-alloc techniques currently suffers from unnecessary atomic operations introduced to avoid race conditions.

The Lin-alloc cell allocation strategy strongly benefits from the dedicated increment atomic counters. Our tests indicate that without these, the Build

performance is about three times slower (using paged allocation, which is then faster), making PRE-OPEN competitive again. This implies that in a non-GLSL setting the performance ratios are likely to differ.

Finally, the POST-SORT techniques could benefit from a smarter sort, bubble-sort having the advantage of fitting well in the RENDER pass due to its straightforward execution pattern. Using a more complex algorithm would be especially beneficial for larger depth complexities. However, increasing the number of fragments per-pixel implies increasing the reserved temporary memory. This impedes performance: for the rendering of Figure 1.3, allocating a temporary array of size 64 gives a RENDER time of 20 ms, while using an array with 256 entries increases the RENDER time to 57 ms. This is for the exact same rendering: reserving more memory reduces parallelism. In contrast, the PRE-SORT techniques suffer no such limitations and support early fragment culling.

For updates on code and results please visit http://www.antexel.com/research/gpupro5.

1.9 Acknowledgments

We thank NVIDIA for hardware donation as well as Cyril Crassin for discussions and feedback on GPU programming. This work was funded by ERC ShapeForge (StG-2012-307877).

Bibliography

[Carpenter 84] Loren Carpenter. "The A-buffer, an Antialiased Hidden Surface Method." *SIGGRAPH* 18:3 (1984), 103–108.

[Celis et al. 85] Pedro Celis, Per-Åke Larson, and J. Ian Munro. "Robin Hood Hashing (Preliminary Report)." In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, pp. 281–288. Washington, DC: IEEE, 1985.

[Crassin 10] Cyril Crassin. "OpenGL 4.0+ A-buffer V2.0: Linked lists of fragment pages." http://blog.icare3d.org/2010/07/opengl-40-abuffer-v20-linked-lists-of.html, 2010.

[García et al. 11] Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. "Coherent Parallel Hashing." *ACM Transactions on Graphics* 30:6 (2011), Article no. 161.

[Harris 01] Timothy L. Harris. "A Pragmatic Implementation of Non-blocking Linked-Lists." In *Proceedings of the 15th International Conference on Distributed Computing, DISC '01*, pp. 300–314. London: Springer-Verlag, 2001.

[Lefebvre and Hornus 13] Sylvain Lefebvre and Samuel Hornus. "HA-Buffer: Coherent Hashing for Single-Pass A-buffer." Technical Report 8282, Inria, 2013.

[Lefebvre 13] Sylvain Lefebvre. "IceSL: A GPU Accelerated Modeler and Slicer." In *Distributed Computing: 15th International Conference, DISC 2001, Lisbon, Portugal, October 3–5, 2001, Proceedings*, *Lecture Notes in Computer Science 2180*, pp. 300–314. Berlin: Springer-Verlag, 2013.

[Maule et al. 11] Marilena Maule, João Luiz Dihl Comba, Rafael P. Torchelsen, and Rui Bastos. "A Survey of Raster-Based Transparency Techniques." *Computers & Graphics* 35:6 (2011), 1023–1034.

This page intentionally left blank

2

Reducing Texture Memory Usage by 2-Channel Color Encoding

Krzysztof Kluczek

2.1 Introduction

In modern games, textures are the primary means of storing information about the appearance of materials. While often a single texture is applied to an entire 3D mesh containing all materials, they equally often represent individual materials, e.g., textures of walls, terrain, vegetation, debris, and simple objects. These single-material textures often do not exhibit large color variety and contain a limited range of hues, while using a full range of brightness resulting from highlights and dark (e.g., shadowed), regions within the material surface. These observations, along with web articles noticing very limited color variety in Hollywood movies [Miro 10] and next-gen games, coming as far as the proposal of using only two color channels for the whole framebuffer [Mitton 09], were the motivation for the technique presented in this chapter.

The method presented here follows these observations and aims to encode any given texture into two channels: one channel preserving full luminance information and the other one dedicated to hue/saturation encoding.

2.2 Texture Encoding Algorithm

Figure 2.1 presents the well-known RGB color space depicted as a unit cube. Each source texel color corresponds to one point in this cube. Approximating this space with two channels effectively means that we have to find a surface (two-dimensional manifold) embedded within this unit cube that lies as close as possible to the set of texels from the source texture. To simplify the decoding algorithm, we can use a simple planar surface or, strictly speaking, the intersection of a plane with the RGB unit cube (right image of Figure 2.1). Because we have already decided that *luminance* information should be *encoded losslessly* in a separate channel, the color plane should pass through the RGB space's origin

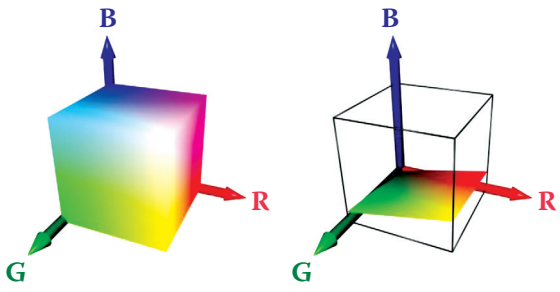


Figure 2.1. RGB color space as unit cube (left) and its intersection with a plane (right).

of zero luminance (black). Therefore, the simplified color space for the 2-channel compression is defined by a single three-dimensional vector—the plane normal.

2.2.1 Color Plane Estimation

Fitting a plane to approximate a set of 3D points is a common task and various algorithms exist. In order to find the best plane for color simplification we have to take the following preparatory steps.

First, we have to remember that RGB pixel color values in most image file formats do not represent linear base color contribution. For the purpose of this algorithm, we want to operate in the linear RGB color space. Most common file formats provide values in sRGB space [Stokes 96]. While being internally more complex, this representation can be approximated with gamma 2.2, i.e., after raising RGB values to power of 2.2 we obtain approximately linear light stimuli for red, green, and blue. We can approximate this with a gamma value of 2, which allows a simple use of multiplication and square root for conversion between sRGB and approximate linear RGB spaces. Strictly speaking, we will then be operating in a RGB space with a gamma of 1.1. While this slight nonlinearity will have only a minor impact on the estimation and the encoding, it is important to use the same simplified gamma value of 2 during the conversion back to the sRGB space after decoding for the final presentation to avoid change in the luminance levels.

After (approximately) converting color values to the linear RGB space, the other thing we have to remember is the fact that the hue perception is a result of the relation between the RGB components and is not linear. To correctly match hues as closely as possible, we could ideally use a perceptually linear color space (e.g., L*a*b*, explained in [Hoffmann 03]). However, this results in a much more costly decoding stage and thus we will limit ourselves to the linear RGB color space, accepting potential minor hue errors. Still, to minimize the impact of not operating in a perceptually correct linear RGB space, we can apply non-

uniform scaling to the space before estimating the plane. This affects the error distribution across the RGB channels, allowing some hues to be represented more closely at the cost of others. The result of this non-uniform scaling is that as RGB components shrink, their influence on the color plane shrinks, because distances along the shrunk axis are shortened. Due to the hue perception's nonlinearity, it is not easy to define the scaling factors once for all potential textures, and in our tests they were set experimentally based on the sample texture set. First we tried the RGB component weights used in the luminance computation (putting most importance on G and barely any on B), but experiments showed that some material textures are better represented when the estimation is done with more balanced weighting. To achieve acceptable results for various textures, we used an experimentally chosen weight set of 1/2 for red, 1 for green and 1/4 for blue, which lies between the classic luminance component weights and the equally weighted component average. Fortunately, the perceived difference in pixel hues after the encoding changes is barely noticeable with these scaling factors. Still, the scaling factors may be used to improve texture representation by fine tuning them separately for each texture.

With the two above operations, the whole initial pixel color processing can be expressed as

$$\begin{aligned} r'_i &= r_i^\gamma w_r, \\ g'_i &= g_i^\gamma w_g, \\ b'_i &= b_i^\gamma w_b, \end{aligned}$$

where γ is the gamma value used to transition from the input color space to the linear color space, and w_r, w_g and w_b are the color component importance weights.

Having taken into account the above considerations, the color of every texel represents a single point in 3D space. The optimal approximating color plane will be the plane that minimizes the sum of squared distances between the plane and each point. Because the plane is assumed to be passing by the point (0,0,0), we can express it by its normal. In effect, the point-plane distance computation reduces to a dot product. Note that since we are using the RGB space, the vector components are labeled r, g, and b instead of the usual x, y, and z:

$$d_i = N \cdot P_i = n_r r'_i + n_g g'_i + n_b b'_i.$$

The optimal plane normal vector is the vector, which minimizes the point-plane distances. Such problems can be solved using least squared fit method that aims to minimize sum of squared distances. The approximation error we want to minimize is expressed as

$$err = \sum_i d_i^2 = \sum_i (N \cdot P_i)^2 = \sum_i (n_r r'_i + n_g g'_i + n_b b'_i)^2,$$

which after simple transformations becomes

$$err = n_r^2 \left(\sum_i r_i'^2 \right) + n_g^2 \left(\sum_i g_i'^2 \right) + n_b^2 \left(\sum_i b_i'^2 \right) + 2n_r n_g \left(\sum_i r_i' g_i' \right) + 2n_r n_b \left(\sum_i r_i' b_i' \right) + 2n_g n_b \left(\sum_i g_i' b_i' \right).$$

For minimalistic implementation, we can use the above equation to compute all six partial sums depending only on the texel colors. Then we can use a brute force approach to test a predefined set of potential normal vectors to find the one minimizing the total approximation error. Because each test is carried out in linear time, costing only several multiplications and additions, this approach is still tolerably fast.

The final step after finding the optimal color plane is to revert the color space distortion caused by the color component weighting by scaling using the reciprocal weights. Because the plane normal is a surface normal vector, the usual rule of non-uniform space scaling for normals applies and we have to multiply the normal by the inverse transpose of the matrix we would use otherwise. While the transposition does not affect the scaling matrix, the matrix inversion does and the final scaling operation is using non-reciprocal weights again:

$$N' = N \left(\begin{bmatrix} 1/w_r & 0 & 0 \\ 0 & 1/w_g & 0 \\ 0 & 0 & 1/w_b \end{bmatrix}^{-1} \right)^T = N \begin{bmatrix} w_r & 0 & 0 \\ 0 & w_g & 0 \\ 0 & 0 & w_b \end{bmatrix}.$$

As all subsequent computation is typically done in the linear RGB space, we do not have to convert into sRGB (which would be nonlinear transform anyway).

2.2.2 Computing Base Colors

The important parameters for the encoding and the decoding process are the two base colors. The color plane cutting through the RGB unit cube forms a triangle or a quadrilateral, with one of the corners placed at the point (0,0,0). The two corners neighboring the point (0,0,0) in this shape are defined as the base colors for the planar color space, as shown on Figure 2.2. Every other color available on the plane lies within the angle formed by the point (0,0,0) and the two base color points. Because the color plane starts at (0,0,0) and enters the unit cube, the base color points will always lie on the silhouette of the unit cube, as seen from the point (0,0,0). To find the base colors, we can simply compute the plane intersection with the silhouette edges, resulting in the desired pair of points. We have to bear in mind that the plane can slice through the silhouette vertices, or even embed a pair of silhouette edges. Therefore, to compute the points we can

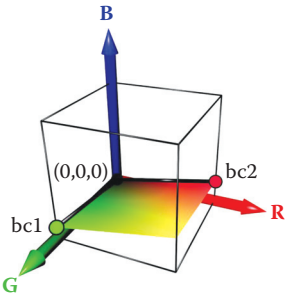


Figure 2.2. Base colors on the color plane.

use an algorithm, which walks around the silhouette computing the two points in which the silhouette crosses the plane.

The key observation now is that we can represent a hue value as the angle to the vectors spanning the plane or, alternatively, using a linear interpolation between the two base colors. In order to compute the final color, we only have to adjust the luminance and perform any required final color space conversions.

2.2.3 Luminance Encoding

The luminance of the color being encoded is stored directly. After colors have been transformed into the linear RGB space, we can use a classic equation for obtaining perceived luminance value derived from the sRGB to XYZ color space transformation in [Stokes 96]:

$$L = 0.2126 \cdot R + 0.7152 \cdot G + 0.0722 \cdot B.$$

Because the weighting coefficients sum up to 1, the luminance value ranges from zero to one. Since the luminance has its own dedicated channel in the 2-channel format, it can now be stored directly. However, as luminance perception is not linear, we are using a gamma value of 2 for the luminance storage. This is close enough to the standard gamma 2.2 and gives the same benefits—dark colors have improved luminance resolution at the cost of unnoticeably reduced quality of highlights. Also the gamma value of 2 means that luminance can simply have its square root computed while encoding and will simply be squared while decoding.

2.2.4 Hue Estimation and Encoding

To encode the hue of the color, we have to find the closest suitable color on the approximating plane and then find the proportion with which we should mix the base colors to obtain the proper hue. The hue encoding process is demonstrated

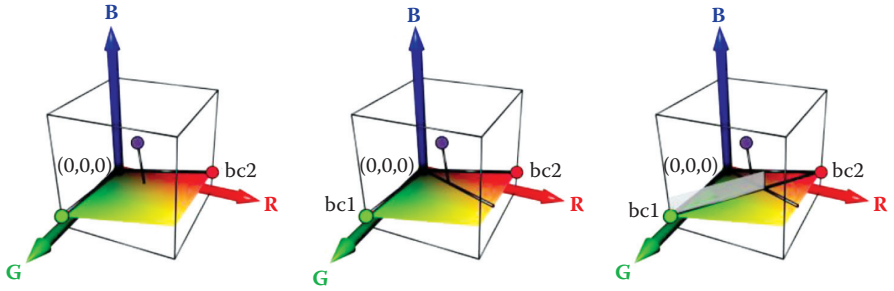


Figure 2.3. Hue encoding process

in Figure 2.3 and can be outlined as follows:

1. Project the color point in the linear RGB space onto the color plane.
2. Compute the 2D coordinates of the point on the plane.
3. Find a 2D line on plane passing through (0,0,0) and the point.
4. Find the proportion in which the line crosses the 2D line between the base color points, i.e., determine the blend factor for the base colors.

The first step is a simple geometric operation. From the second step on, we have to perform geometric operations on 2D coordinates embedded within the plane. Having the two base color points A and B, we can compute the 2D coordinate frame of the plane as

$$F_x = \frac{A}{\|A\|} \quad F_y = \frac{B - (F_x \cdot B)F_x}{\|B - (F_x \cdot B)F_x\|}$$

and then compute 2D coordinates of any point within the plane using the dot product:

$$(x_i, y_i) = (P_i \cdot F_x, P_i \cdot F_y)\,.$$

Please note that we do not actually need the explicit RGB coordinates of the point on the plane nearest to the color being encoded, but only its 2D coordinates within the plane, x_i and y_i. As both the original point and the point projected onto the plane will have the same 2D coordinates, we can skip step 1 in the outlined algorithm completely. The projection onto the plane is a side effect of the reduction to only two dimensions.

The problem of computing the base color blend factor for hue, when considering the points embedded within the color plane, is now reduced to the problem of intersection of two lines: a line connecting both base color points and a line

```
float3 texture_decode( float2 data, float3 bc1, float3 bc2 )
{
  float3 color = lerp( bc1, bc2, data.y );
  float color_lum = dot( color, float3(0.2126,0.7152,0.0722) );
  float target_lum = data.x * data.x;

  color *= target_lum / color_lum;
  return color;
}
```

Listing 2.1. Two-channel texture decoding algorithm.

passing through the origin and the point on the plane being encoded. This gives us the following line-line intersection equation:

$$A + t(B - A) = sP.$$

Solving this linear equation for t gives us the result—the base color blend factor resulting in a hue most closely matching the hue of the encoded point. This blend factor is then simply stored directly in the second channel, completing the 2-channel encoding process.

2.3 Decoding Algorithm

The decoding algorithm is simple and best described by the actual decoding shader code in Listing 2.1.

First, the base colors `bc1` and `bc2`, which are passed as constant data, are blended with a blend factor coming from the second channel of `data`, resulting in a color having the desired hue, but wrong luminance. This luminance is computed as `color_lum`. Next, we compute the desired luminance `target_lum` as a value of first channel of `data` squared (because we stored the luminance with gamma 2). As the resulting color is in a linear color space, we can adjust the luminance by simply dividing the color by the current luminance and then multiplying it by the desired one. If needed, we can of course convert the computed color to a nonlinear color space for presentation purposes.

2.4 Encoded Image Quality

Figures 2.4, 2.5, and 2.6 show examples of the encoding and decoding process. The example textures are taken from the CGTextures texture library and were selected because of their relatively rich content and variety.

Figure 2.4 presents a 2-channel approximation result of a dirt texture with grass patches. Both dirt and grass are reproduced with slight, but mostly unnoticeable differences in color. As the method is designed with limited-color material

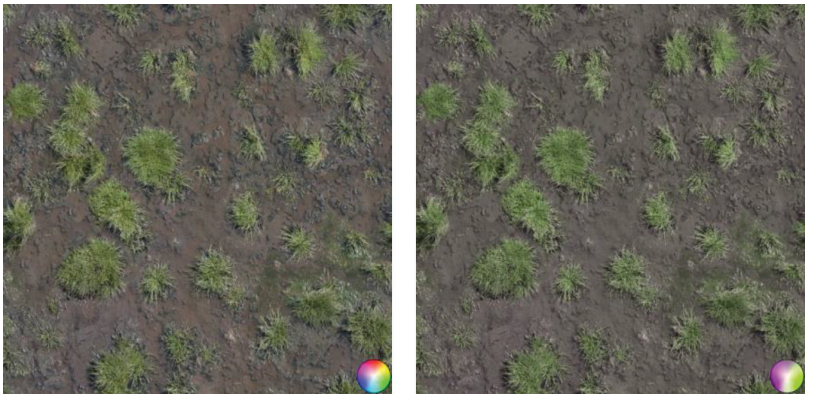


Figure 2.4. Grass and dirt texture example. Original image (left) and result after the encoding/decoding process (right).

textures in mind, the color probe added on the image is of course severely degraded, but clearly shows that the estimation algorithm picked the green-purple color plane as fitting the image best. These extreme colors may not be used directly on the texture, but we should remember that all colors resulting from blending green and purple are available at this stage and this includes colors with reduced saturation in the transition zone. Because of the separate treatment of pixel luminance, the luminance values are unaffected except for processing and storage rounding errors.

Figures 2.5 and 2.6 show two examples of textures with mixed materials. This time, the estimation process has chosen a blue-yellow for the first and a

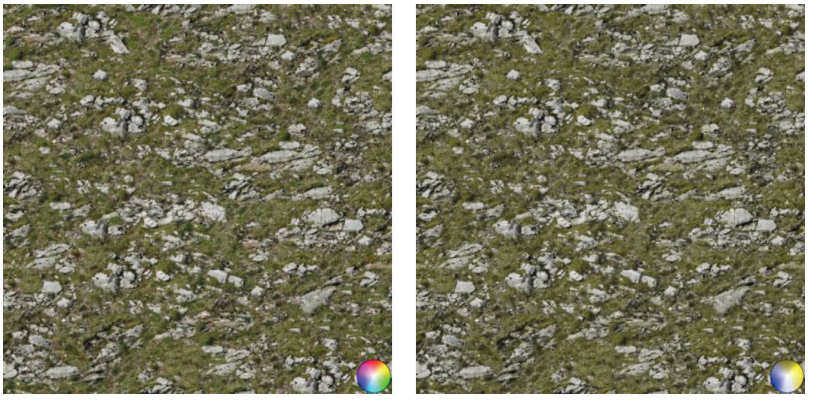


Figure 2.5. Rock and stone texture example. Original image (left) and result after the encoding/decoding process (right).

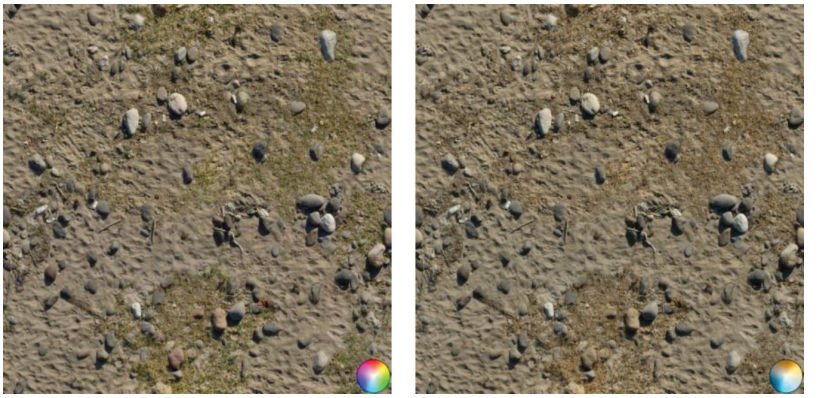


Figure 2.6. Sand, dead grass, and rocks texture example. Original image (left) and the decoded result after 2-channel compression (right).

teal-orange plane for the second image. While the stone and grass texture mostly remains unaffected, the sand, grass, and stones texture required finding a compromise resulting in some grass discoloration and smaller off-color elements changing color completely.

2.5 Conclusion

The encoding and decoding methods presented in this chapter allow storing textures with low color variety using only two texture channels. Apart from the obvious savings, this opens additional possibilities. For example, considering that most texture sampler implementations support 4-channel textures, the two remaining channels can be used for other purposes, e.g., storing x and y components of the material normal map, resulting in a compact material representation with just a single texture image. Even if not using this feature, the fact that the proposed 2-channel color encoding relies on a luminance-hue decomposition allows custom texture compression algorithms. We can assign higher priority to luminance information during texture compression, accumulating most of the compression error in hue, to changes to which the human eye is less sensitive, increasing the overall compressed image quality. We should also note that the proposed encoding scheme can be used directly with existing mip-mapping solutions, because averaging luminance-values and hue-blend factors is a good approximation of averaging color values. We should only be aware that the luminance values are stored with a gamma of 2 and may require a custom mip-map chain generation if we require fully linear color processing in the whole lighting pipeline.

Bibliography

[Hoffmann 03] Gernot Hoffmann. *CIELab Color Space*. http://docs-hoffmann.de/cielab03022003.pdf, 2003.

[Miro 10] Todd Miro. *Teal and Orange—Hollywood, Please Stop the Madness*. http://theabyssgazes.blogspot.com/2010/03/teal-and-orange-hollywood-please-stop.html, 2010.

[Mitton 09] Richard Mitton. *Two-Channel Framebuffers for Next-Gen Color Schemes*. http://www.codersnotes.com/notes/two-channel, 2009.

[Stokes 96] Michael Stokes, Matthew Anderson, Srinivasan Chandrasekar, and Ricardo Motta. *A Standard Default Color Space for the Internet—sRGB*. http://www.w3.org/Graphics/Color/sRGB, 1996.