Page Array Data Structures for Flexibility and Performance

Neil G. Dickson
SideFX

ABSTRACT

Visual effects impose demanding requirements for data structures and algorithms. They are expected to be flexible enough to support any idea an artist or TD could think of, while being as fast as a custom implementation developed for one purpose. Our solutions are built on page array data structures. Our arrays can represent a wide variety of geometry data, including polygons, and support reference counted page sharing and constant-value page compression for memory efficiency. Our method permits reasonably fast reading and writing in serial or parallel. We can also process data in a page-aware manner for even better performance.

CCS CONCEPTS

• **Computing methodologies** → *Shared memory algorithms*; *Computer graphics*;

KEYWORDS

data structures, parallel computing, computer graphics

ACM Reference Format:
Neil G. Dickson. 2018. Page Array Data Structures for Flexibility and Performance. In *Proceedings of SIGGRAPH '18 Talks.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3214745.3214757

1 INTRODUCTION

Geometry in Houdini is stored in a container called a *detail*, which can contain a heterogeneous collection of *primitives*, e.g. polygons, tetrahedra, and volumes, possibly together. Each primitive has one or more unique *vertices* that reference *points*. Points can be shared between primitives. Each type of element (detail, primitives, vertices, points) can have arbitrary data, known as *attributes*, added at any time. For example, position data could be stored on points while colors are stored on vertices, temperatures are stored on primitives, and a tuple of filenames is stored on the detail.

Most geometry operators in Houdini copy incoming geometry and modify that copy. However, explicit copying would waste memory for anything not modified, so some form of copy-on-write [Reinders et al. 2017] is necessary. Attributes on points or primitives are often not the same value for every element, but are often a single value for large, contiguous spans. To avoid storing many copies of the same value and wasting memory, local constant-compression is necessary. Numeric attribute data can be int/float, 8/16/32/64-bit, and any tuple size. Polygons can have any number of vertices, and can be open (curves) or closed (surfaces). Frequently, there are large contiguous spans of polygons with identical vertex counts (triangles or quads); often those vertices are also sequential.

Despite all of those requirements, reading attribute data and vertex lists of polygons must be fast. Further, multicore architectures make it imperative that we can write to attribute data quickly in parallel.

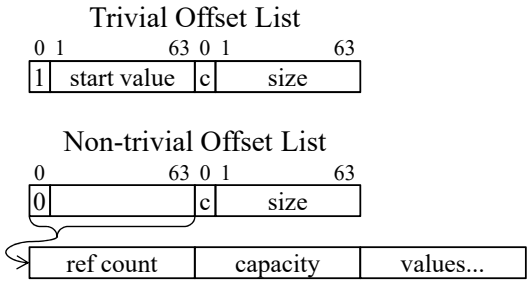

Figure 1: Offset list data structure.

2 DATA STRUCTURES

2.1 Offset List

We frequently need to represent a sequence of integers that is usually contiguous, for example, the list of vertices of a polygon. For the contiguous case, a so-called "*trivial*" offset list, just the start number, the size of the list, and an indication that the list is trivial are needed.

All memory allocators we use allocate memory at even addresses, (for allocations of 8 bytes or more), so bit 0 can be used to indicate whether the list is trivial. If the bit is 1, the following 63 bits store the start number, otherwise the entire 64 bits are an actual pointer to an array of integers. This trick is used later for constant attribute data page pointers. Bit 0 of the following 8 bytes is used to indicate whether a polygon is closed or open, and the remaining 63 bits hold the size of the list.

If an offset list is not trivial, duplicating the data upon copying would be wasteful, and so we use copy-on-write referencing to minimize overhead. We also amortize the cost of appending to lists using a separate capacity from the size, and minimize memory allocations by storing the reference count and the capacity in the same allocation as the values.

2.2 Page Array

Instead of storing a numeric attribute in a single array of contiguous memory, an array of multiple pages is used. Each page usually contains 1024 entries, so that the page number can be determined with a bit shift by 10 and the page offset can be determined by ANDing with 0x3FF. However, when there is a single page, we allow its size to be any power of two that is 1024 or less, since we

SIGGRAPH '18 Talks, August 12-16, 2018, Vancouver, BC, Canada

ACM ISBN 978-1-4503-5820-0/18/08.
https://doi.org/10.1145/3214745.3214757

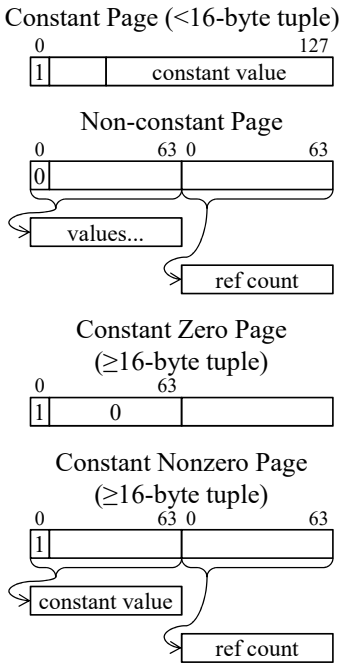

Figure 2: Page table entry for page array data structure.

do not want 10 points to waste 1014 entries. This helps minimize data storage for small geometry or attributes stored on the detail.

Using the same trick as trivial offset lists, a constant-value page is indicated using bit 0 of what would otherwise be a 64-bit pointer to the data in the page table. Pages are reference counted to avoid copying until it is necessary. To ensure that page data is usually aligned to 4KB, we store reference counts for each page in separate 8-byte allocations. We found that storing the reference counts in the same allocation as the page data caused significant memory overhead, due to allocation sizes being slightly larger than common allocator bucket sizes. Each page table entry has 16 bytes, conveniently giving us up to 15 bytes for storing a page's constant value. For constant pages with tuples of 16 bytes or more, a page with a single tuple is allocated, and bit 0 of its pointer is set to 1 to indicate that it is a constant page, but this bit is masked out before reading. Since constant pages for larger tuple sizes are frequently all zero, we have a special case that a null pointer represents an all-zero page, avoiding the allocation of a tuple and of a reference count.

We store an enum indicating the storage type and an integer indicating the tuple size with the pointer to the page table. To avoid having to always re-check the tuple size and storage type, C++ templates for the type and tuple size can be used, with a void type indicating that the type still needs to be checked, and a -1 tuple size indicating that the tuple size still needs to be checked, but any other type or value indicating that the type or tuple size is known at compile time. This allows the type and tuple size to be checked only once at runtime, and in common cases of matching the expected type or tuple size, they are built directly into the compiled code, resulting in fast lookups.

Constant pages and shared pages can sometimes be used to significantly speed up operations. For example, if two pages refer to the same data, their difference is zero, so computing of deltas between two similar models for blending may only need to compare pages with different data. Blending two constant pages only requires blending the two representative values.

The main page array structure is also used for topology attributes, (vertex-to-point and point-to-vertex mappings), as well as for string attributes tracking a string index for each element. Similar shareable pages are used for numeric array attributes, string array attributes, and element groups.

2.3 Polygon Vertex Lists

A similar page array structure can be used, very carefully, to represent polygons, with offset lists of vertex offsets. The size of an offset list structure is 16 bytes, so is similar to a tuple of two 64-bit integers, with the caveat that upon destruction, any allocated array must have its reference count decremented, and upon copying, any allocated array must have its reference count incremented. This is complicated by the page array's own reference counting, so destruction (decrementing offset list reference counts) must occur only when a page's reference count reaches zero, and copying (incrementing offset list reference counts) must occur when hardening a previously-shared page. The most substantial advantage comes from the "constant" page representation. We can reinterpret a constant page to allow a single vertex list to represent an entire page of contiguous vertex lists with the same number of vertices. This adds even more complication, because our constant representation no longer indicates that all elements of the page have the same value, but instead that they can be quickly computed from the single value.

3 CONCLUSIONS

We implemented our original page array structure with copy-on-write semantics for attribute data in Houdini 12.0. The promising results attained by the use of page-sharing and constant-page optimizations led to further optimizations and generalization to polygon and tetrahedron vertex lists in Houdini 16.0. These page array structures have yielded dramatic memory and time savings in Houdini, and any access time overhead relative to a single array is negligible in practice. We plan to apply similar structures to more primitive types in the future, and will continue to use them for more algorithms requiring globally sparse but locally dense data.

REFERENCES

James Reinders, Andy Lin, Joe Longson, Jeff Lait, Florian Zitzelsberger, and Martin de Lasa. 2017. Multithreading for Visual Effects. In *ACM SIGGRAPH 2017 Courses (SIGGRAPH '17)*. ACM, New York, NY, USA, Article 9, 259 pages. https://doi.org/10.1145/3084873.3084891