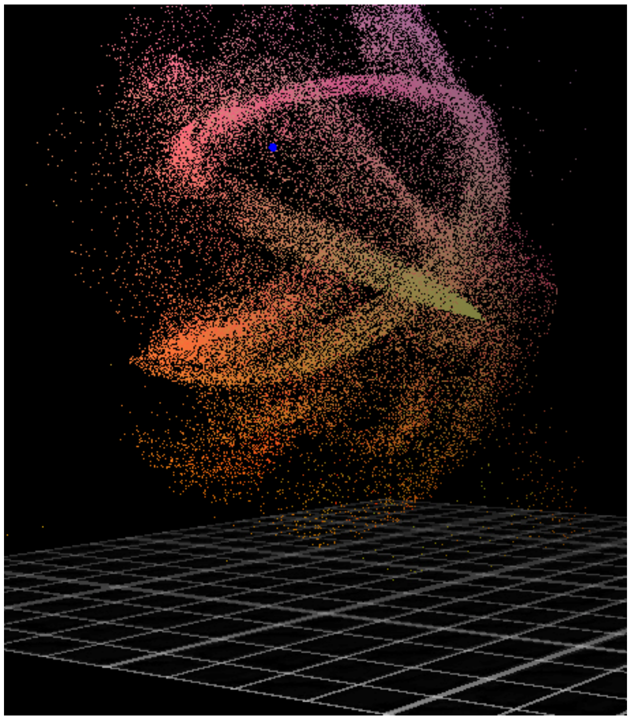
# Vertex Reordering for Cache Coherency: Sorting along Morton-Order Curves



A research paper submitted for
**Departmental Honors**
in the department of
**Computer Science**
at the
**University of Maryland, College Park**

Submitted by
**Jonathan Howard**
Under the advisement of
**Amitabh Varshney**

December 12, 2006

## Table of Contents

## 1. Introduction

In virtually all areas of computer graphics, from Hollywood visual effects to scientific visualization to next-generation video game consoles, there is a growing push toward realism and the ability to process exponentially increasing amounts of data. At the same time, the gap between processor speed and memory speed has been widening, so caches have become increasingly important in reducing latency. While this presents a many-faceted problem, one key to accomplishing this ambitious goal is efficient use of caching to increase performance.

Scientific visualization data is now measured in the hundreds of megabytes, and character models for next-generation game consoles often contain hundreds of thousands of vertices. In these applications, badly ordered vertices could result in significant performance hits due to cache misses each frame. An intelligent vertex ordering minimizes these cache misses and allows applications to take full advantage of the graphics hardware. In some cases, it even opens the possibility for interactivity where, without vertex ordering, it may not have previously been possible.

A completely optimal ordering is computationally intensive to compute, highly dependent on the specific cache properties of each target machine, and therefore impractical. The ideal compromise, therefore, would be an algorithm that finds a close approximation of the optimal ordering, and does so very quickly.

The concept of vertex reordering is not without precedent; multiple techniques have been established for vertex ordering in the past, and a discussion of existing techniques follows in the next section. However, we believe there is still room for improvement. We present an algorithm that orders vertices along a space-filling curve – specifically, using their Morton order – and does so fast enough for real-time applications.

## 2. Existing Techniques

Precedent in this area includes “Triangle order optimization for graphics hardware computation culling” (Nehab, Barczak, and Sander) and “Streaming Meshes” (Isenburg and Lindstrom.) However, the algorithm we will use for comparison is the OpenCCL algorithm published in 2005 by UNC, Chapel Hill in the paper “Cache-oblivious mesh layouts” (Yoon, S., Lindstrom, P., Pascucci, V., and Manocha, D.).

OpenCCL is able to provide a well-optimized mesh ordering, and it does so in the general case, without requiring specific knowledge of the target cache. They claim speedup factors of between 2 and 20 times the original frame rate (though they make no promises for any specific application.)

However, there is one major drawback to OpenCCL: execution time. It takes minutes or even hours for OpenCCL to reorder the vertices of a model of any significant size. According to their documentation, large models must be broken up into spatially coherent chunks of 2000 to 4000 faces. Each individual piece is then run through the algorithm, and some other method must then be used to re-integrate the chunks. This is frustrating for end users at best. Moreover, even the processing time for individual chunks is impractical in a real-time interactive application. Our algorithm presents an alternative with comparable results and greatly reduced processing time.

## 3. Implementation

### 3.1 Chan's Approximate Nearest Neighbor Algorithm

Our method sorts vertices along a Morton-order space-filling curve to achieve comparable cache coherency to that of OpenCCL (as measured by frame rate improvements) and requires much less processing time to do so. The algorithm builds upon code adapted from the paper "A minimalist's implementation of an approximate nearest neighbor algorithm in fixed dimensions," (Chan, 2002.)

The key concept from Chan's code is a new comparison-based sorting method. The obvious method for comparisons along a Morton-order space-filling curve is to evaluate exact distances along the curve for each vertex, and then sort based on the computed values. Instead, Chan uses a clever series of bit-wise operations to determine the *order* in which any two vertices appear on the curve, as opposed to the actual *values* for each vertex. Comparisons that use this bit-wise method are much more efficient than traditional methods of computing Morton order.

### 3.2 Our Algorithm: Morton-Order Sorting

Chan's code was written for approximate nearest neighbor searches and closest pair queries. However, this method of Morton-order comparison can be applied to vertex reordering as well. Our algorithm builds upon Chan's bit-wise operations to perform comparison-based sorting on the vertices of a mesh to obtain a cache-coherent ordering. In our observations, this method achieves comparable results to that of OpenCCL, producing 90% of OpenCCL's performance increase.

In addition, our algorithm does not exhibit the same 2000-4000 face performance limitations of OpenCCL, so our algorithm runs in a fraction of the time. Because of this time speedup, our algorithm presents interesting possibilities in the realm of real-time graphics applications. For instance, OpenCCL might take several hours to reorder a set of isosurface data, making its use impractical in most situations. Our algorithm takes orders of magnitude less time to achieve similar results, so it could feasibly be used to bring such large datasets into the realm of near-interactive rendering.

### 3.3 Real-time Applications

Since our algorithm uses only comparisons between vertices, it does not need to permanently store information about any other vertices in the mesh. Therefore sorting can be amortized over a number of frames without penalty. This presents another interesting possibility: that of real-time vertex reordering at run-time. By amortizing the sorting steps over a number of frames, vertex data can be sorted over time, and frame-rate can be continually increased as the program progresses, without waiting for any pre-processing time whatsoever.

In addition, large particle systems can dynamically sort its particles along a space-filling curve, for cache-coherent lookups into its particle array, provided the system exhibits properties described in the next section. OpenCCL is essentially a pre-processing algorithm, to be completed in one coherent step. Therefore it is not practical for sorting dynamic systems, and is unhelpful when the vertices (particles in this case) are in continuous non-spatially-coherent motion. However, our algorithm, when amortized over time, expands the vertex reordering concept to cover not only static meshes, but dynamically changing data as well.

## 4. Application to Particle Systems

### 4.1 Our Implementation

Since our algorithm works generally on positions in three dimensional space, and the sorting calculations can be amortized over a number of frames, we hypothesized that particle systems might apply our techniques to obtain similar speedups to those observed on mesh data. And indeed, if a particle system is implemented using a C-style array, our Morton order sorting algorithm can be applied using the particles' coordinates, so that operations on the particles are applied in a cache-coherent order. However, our implementation of a Morton-sorted particle system brought to light some restrictions to this application.

Our particle system implementation seeks to take advantage of cache coherence in two ways: the particle array itself, and a three-dimensional array of forces. The system contains an emitter that discharges particles in a random direction, scattering them atop a 100x100 grid. The forces acting within the space are defined by a number of masses, which each apply a gravitational force in the space. The sum of these forces is stored in a 100x100x100 array, or “force field.” Each particle then accesses the force field each frame to determine the forces acting upon it at the time.

With thousands of access into the force field array each frame – and potential for thousands of cache misses – performance is highly dependent on these accesses occurring in a cache-coherent manner. An example image from this implementation is included below, with particles colored according to their position in the force field array. Each system below contains only one force field object (approximately center-screen in both), gravity, and particles moving at a high velocity:

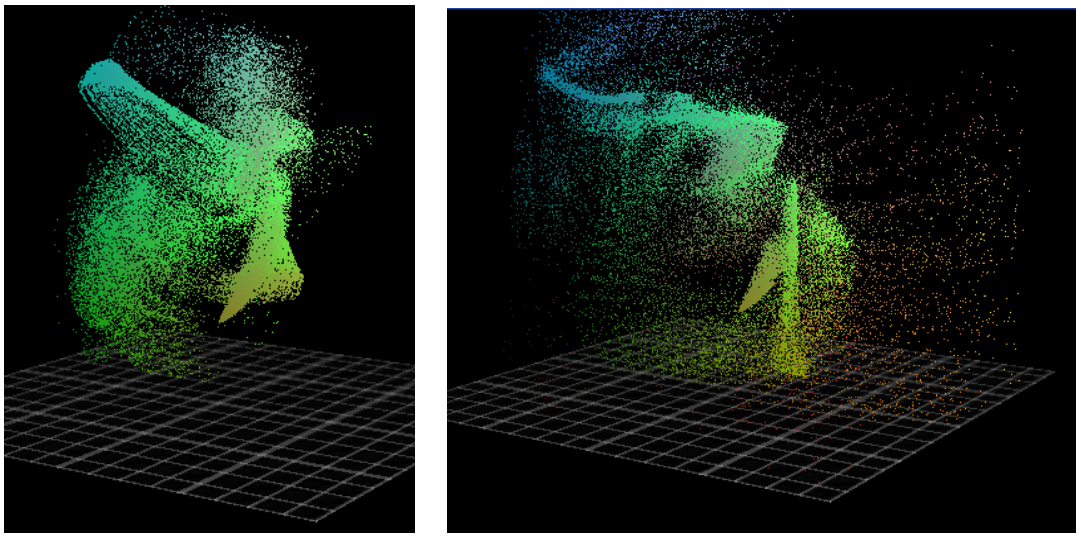


Fig. 1: Particles colored according to their X (red), Y (green) and Z (blue) positions.

4.2 Results: FPS Data

The following data was obtained with the system set to emit 10 particles per frame, each particle having a speed multiplier of 0.5 and a lifetime of 5000 frames, for a total of 50,000 particles when the system reached equilibrium. One “trial” is defined as beginning with the particle system at equilibrium with no sorting, completing one full sort, and then performing an incremental sort until the FPS returns to the original “equilibrium” speed. We performed trials with various degrees of incremental sorting, as shown in the table below:

| **Number of Steps per Incr-emental Sort (x1000)** | | **Initial Stable FPS** | **After Full Sort** | **Sorting Time** | **Init. Incr. FPS** | **Time to Return to Initial FPS** |
|---|---|---|---|---|---|---|
| | **0** | 50 | 60 | < 1 sec | -- | 30 sec |
| | **1** | 65 | 80 | < 1 sec | 70 | 20 sec |
| | **2** | 50 | 60 | < 1 sec | 55 | 5 sec |
| | **4** | 50 | 60 | < 1 sec | 50 | < 1 sec |
| | **8** | 50 | 60 | < 1 sec | 45 | 0 sec |
| | **16** | 70 | 80 | < 1 sec | 40 | 0 sec |

At slow particle speeds, the system maintains its spatial coherence for approximately 30 to 40 seconds, as shown by frame rates above, and as estimated visually below. (In the illustrations below, particles at the beginning of the particle array appear yellow; particles at the end of the array appear blue.)

With no incremental sort adding to computation time, the system does not return to its initial FPS state until approximately 30 seconds after a full sort. It would stand to reason that if this sort were amortized over less than 30 seconds, the system would see an overall increase in performance. However, since the incremental sort must be implemented differently than the full sort to allow for asynchronous execution, and the incremental version is currently too inefficient to keep the system sorted for any significant length of time. Images from the 16000-steps-frame trial are included below to illustrate this point:

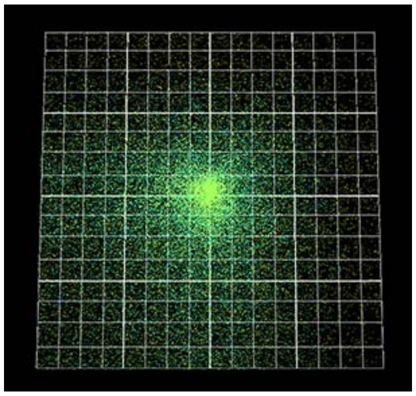


Fig. 1: System at its initial state

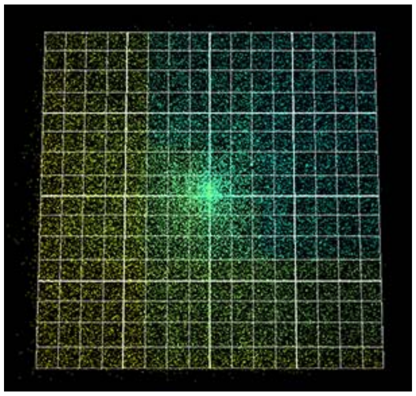


Fig. 2: Just after a full sort. Incremental sort begun.

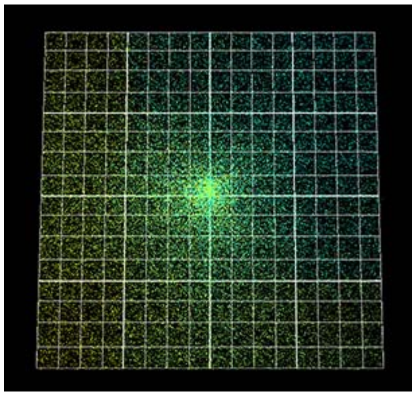


Fig. 3: System 0:30 sec later.

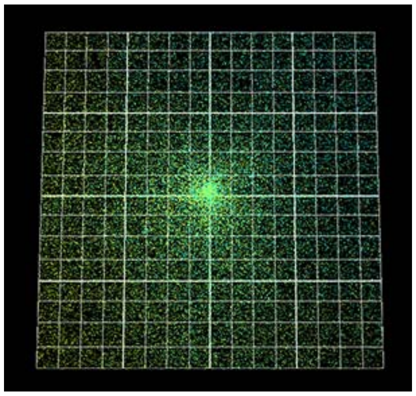


Fig. 4: System 1:00 min later.

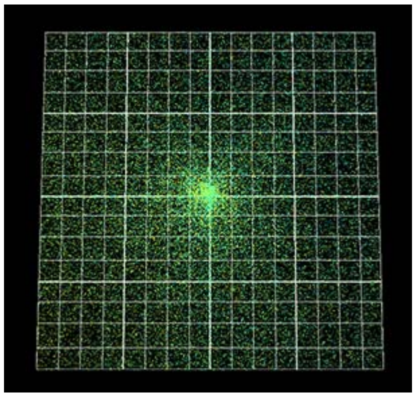


Fig. 5: System 1:30 min later.

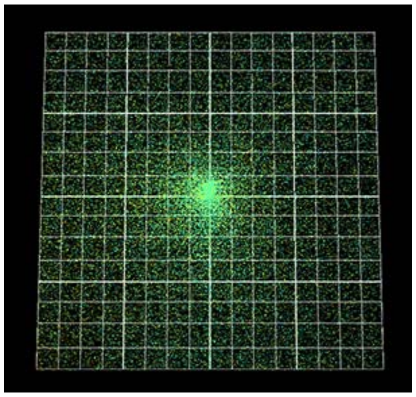


Fig. 6: System 2:30 min later.

## 4.3 Performance Restrictions: Inter-Frame Coherence

Most mesh data can be sorted once to obtain a cache-coherent ordering. However, many particle systems are highly dynamic, meaning that each particle's position on a space-filling curve will change rapidly. Every time the position of a particle changes with respect to its neighbors, the ideal sorting order also changes. Therefore, the more dynamic the system becomes, the more the effect of any sorting is defeated by reordering. The upper limit of this phenomenon would occur with very fast-moving particles, in which every particle changes its relative position on every frame.

The amount of main memory used and the number of array accesses in our implementation would suggest that this system would benefit greatly from cache-coherent processing. However, the particles in our implementation moved in random directions, and did so very quickly, so that each frame exhibited little similarity to the frame before. Under these conditions, there is simply not enough time between frames for the amortized sorting to affect a significant change before the particles reorder again.

## 5. Future Work

The particular characteristics of the particle system we implemented tended to counteract the effects of our sorting algorithm. However, future work may open more possibilities for the algorithm's application to such systems.

As we observed, the full sorting algorithm takes less than one second, whereas its benefits last for 30 seconds. The incremental sort – which was much less efficient – was unable to produce results at a quick enough pace to keep up the full sort's frame rate improvements. However, with a more efficient incremental sorting implementation, the algorithm would exhibit much greater success.

In addition, the algorithm will likely prove more effective in applications where these inter-frame coherence problems are less prominent. For example, the amortized sort would have a longer-lasting affect on performance in particle systems that contain particles with less highly dynamic relative positions, and relative positions change less frequently. It may also prove more effective in applications with much higher memory usage, since performance in those cases may be more tightly bottlenecked by cache coherence.

(Note: since the problem in dynamic particle systems occurs when their low inter-frame coherence counteracts the sorting, ever-increasing processor speeds will also enable more sorting to be done each frame, making this method incrementally more feasible.)

## 6. Acknowledgments

My work on adapting Chan's Approximate Nearest Neighbor algorithm and its implementation in a particle system was done as part of a larger project on vertex reordering for cache coherence at the University of Maryland, College Park. For their periodic assistance, and for allowing me to contribute as an undergraduate, I would like to thank the other members of the team: Amitabh Varshney, David Mount, Derek Juba, Youngmin Kim, and Rob Patro.

## 7. References

- Nehab, D., Barczak, J., and Sander, P. V. 2006. **Triangle order optimization for graphics hardware computation culling.** In Proceedings of the 2006 Symposium on interactive 3D Graphics and Games (Redwood City, California, March 14 17, 2006).
- Martin Isenburg, Peter Lindstrom, **"Streaming Meshes,"** vis, p. 30, IEEE Visualization 2005 ( VIS'05), 2005.
- Yoon, S., Lindstrom, P., Pascucci, V., and Manocha, D. 2005. “**Cache-oblivious mesh layouts.”** ACM Trans. Graph. 24, 3 (Jul. 2005), 886893.
- Sung-Eui Yoon, Peter Lindstrom, **"Mesh Layouts for Block Based Caches,"** IEEE Visualization 2006 (VIS'06), 2006.