Vol. [VOL], No. [ISS]: 1–15

Simple and Efficient Mesh Layout with Space-Filling Curves

Huy T. Vo and Claudio T. Silva

Polytechnic Institute of New York University

Luiz F. Scheidegger Facebook, Inc.

Valerio Pascucci SCI Institute, University of Utah

Abstract. We present a simple and efficient algorithm to compute cache-friendly layouts of unstructured geometric data. Coherent mesh layouts minimize cache misses and page faults by laying out vertices, triangles or tetrahedra in a spatially structured manner. Recently, Yoon et al. have shown that it is possible to construct an optimal cache-oblivious mesh layout (COML) for surface and volume data. However, their approach is based on an NP-Hard optimization problem, and is thus very computationally expensive. We present a mesh layout based on space-filling curves that has comparable performance to COML and is orders of magnitude faster to compute. We also discuss extending our algorithm to handle extremely large datasets through an out-of-core approach. Finally, we include an analysis that examines a number of different mesh layouts, highlighting their strengths and weaknesses. Our evaluation indicates that space-filling curve layouts can be an order of magnitude faster and less memory-intensive to compute while, in every application, being able to maintain a performance within 5% of the best layout, including those that are specifically tuned for GPU hardware vertex caches in [Lin and Yu 06, Sander et al. 07].

1086-7651/06 $0.50 per page

1. Introduction

In the past few years, advances in 3D data acquisition technology, as well as improvements in simulation algorithms, have made very large datasets available to the computer graphics community. Currently, the size of these models can vary from a few tens of thousands to hundreds of millions of polygons. Many challenges arise from this notable increase in size and complexity, and recently much attention has been given to the problem of computing high quality memory layouts for geometric datasets. Such layouts aim to minimize the cache miss and page fault penalty incurred by applications. This can be done by defining a cache coherence metric and then optimizing the layout according to this metric. Yoon et al. [Yoon and Lindstrom 06, Yoon et al. 05] have investigated this problem and proposed a cache-oblivious mesh layout (COML) that generates near optimal results for any cache configuration. One of this work's major findings is that by optimizing a dataset's layout in memory, it is possible to improve the performance of many mesh processing applications without modifying the applications themselves. Our technique is similar to COML in the sense that it can also naturally improve the performance of other applications, but it is based on a space-filling curve approach, and is thus much faster to compute.

Space filling curves are well known for their memory coherence characteristics and high spatial locality [Sagan 94]. They are widely used in applications such as terrain [Lindstrom and Pascucci 01] and volume [Pascucci and Frank 01] rendering. However, these curves have not been investigated in the unstructured grid domain, mostly because they do not consider mesh connectivity, and are thus not expected to work well with irregular geometry. Despite this, we have found that since many large models originate from either a 3D scanner or a simulation, their primitives are typically distributed in a quasi-regular fashion. Moreover, if we consider each connected component of a CAD model separately, we can also find a high degree of regularity. Because of this, our technique generates good results even without considering mesh connectivity.

2. Background

The most widely used mesh representation is the face-vertex format where each mesh is represented by two separate arrays: a vertex array V holding vertex geometries ((x, y, z) locations) and a primitive array I holding indices into the vertex array to define mesh faces. Vertices can be shared among faces by specifying the same index in I. In a triangle mesh, for instance, consecutive values in I are taken three at a time (four at a time in the case of a tetrahedral mesh) to indicate the three vertices of every triangle. Thus, for

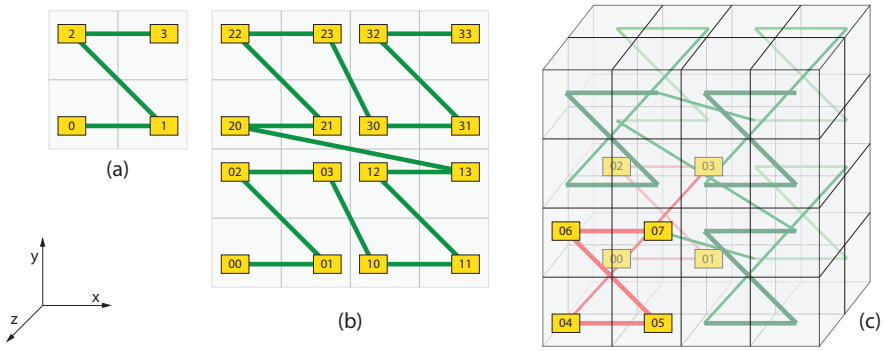

Figure 1. Different iterations of the discrete Morton curve. (a) and (b) denote its traversal order embedded in 2D while (c) illustrates how the curve applies in 3D. Observe how the ordering can be implicitly computed by sorting nodes using a specific labeling in base-4 and base-8 digits (for 2D and 3D space, respectively).

a triangle mesh with N_v vertices and N_t triangles, V and I will contain $3N_v$ floats and $3N_t$ integers, respectively.

Mesh Layouts A mesh layout is a linear ordering of the elements (vertices, triangles or tetrahedra) in the dataset. There are many ways in which these layouts can be computed. Breadth-first (BFS) and depth-first (DFS) traversals are two basic examples of algorithms that can generate layouts. Another approach is spectral sequencing, proposed by Isenburg et. al. [Isenburg and Lindstrom 05], which heuristically finds the first non-trivial eigenvector of the mesh's Laplacian matrix and sorts mesh vertices according to this vector. This layout is most used for minimizing memory consumption in applications where mesh streaming is supported. The work of Yoon et. al. [Yoon and Lindstrom 06, Yoon et al. 05] presents cache-friendly layouts that improve the performance of applications such as view-dependent rendering and isovalue extraction. Unfortunately, their algorithm is based on an NP-Hard optimization problem that is very computationally expensive. Even using a multilevel simplification heuristic, computation times for large datasets can be prohibitive. Both Isenburg et al. and Yoon et al. share the key idea of reordering the vertices in V such that nearby indices in I are likely to point to nearby vertices in V. These layouts tend to reduce the number of cache misses when reading a mesh, and are therefore called cache-friendly layouts. An example of a "cache-unfriendly" layout is a random permutation of the vertex array. Obviously, this causes most adjacent indices to point to distant locations in the vertex array, greatly increasing the number of cache misses.

Space-Filling Curves Space filling curves [Sagan 94] are maps from a 1D interval to a region in n-dimensional space. Intuitively, they can be thought of as curves that traverse all points of the n-dimensional space, thus inducing an order on those points. This ordering has many desirable locality properties [Gotsman and Lindenbaum 94] that make it ideal for cache-friendly layouts. Many algorithms use space-filling curves to compute efficient layouts of two- or three-dimensional regular grids. These layouts increase the performance of image processing [Velho and Gomes 91] and terrain and volume rendering [Lindstrom and Pascucci 01, Pascucci and Frank 01] applications.

Our technique is based on using space filling curves to generate a cache-friendly mesh layout. We present an efficient algorithm to generate this layout that can be easily extended to work in out-of-core mode, thus allowing our technique to deal with extremely large datasets (the largest model with which we tested our system is the Atlas, which has approximately 500 million triangles). Our solution is based on interpreting vertex indices as node locations in a high-resolution regular Octree. These nodes can be sorted to construct an implicit traversal of this tree that corresponds exactly to the underlying curve.

3. Morton Space-Filling Curve Layout

While our technique can work with any space filling curve, we choose to use the 3-dimensional *Morton order* (also known as Z-Order or Bit-Interleaving curve [Hungershöfer and Wierum 02]) by default for both simplicity and efficiency in layout computation. Moreover, as we show in section 4, our results are comparable to other, more expensive, layout schemes including Hilbert curves, which are known to have better locality-preserving behavior. Our implementation can also be easily extended to work with large datasets using an out-of-core approach.

3.1. Indexing algorithm

Our layout algorithm works by assigning to each vertex a position in the Morton order traversal, and then sorting mesh primitives according to this value. Below we summarize the important steps in our technique:

1. Assign an index to each vertex based on its position in the Morton order traversal
2. Sort the primitives based on their vertices' Morton indices
3. Reorder the vertices based on the sorted primitives

Vertex indices We assign a 64-bit unsigned integer index to each vertex on the mesh. When vertices are sorted according to this index, they automatically assume the layout of a Morton space-filling curve. In order to compute an index that has this property, we construct an implicit complete Octree of depth N. The root of the tree is the mesh's bounding box, and the leaf nodes are cells in a $2^N \times 2^N \times 2^N$ regular grid. The index of a vertex is constructed by concatenating N octal digits that represent the octants where the vertex is located in each of the N levels in the Octree (see figure 1 for details). By labelling octants in a specific way, we can ensure that this ordering corresponds to a Morton order curve. Since we only need the dimensions of the bounding boxes of each octant to run this algorithm, no additional data structure is necessary.

In principle, N must be large enough to ensure that each leaf in the Octree is occupied by at most one vertex. Since each octal digit requires exactly three bits, using 64-bit unsigned integers for indices corresponds to choosing $N = \lfloor 64/3 \rfloor = 21$. This is equivalent to a grid with resolution higher than $(2 \times 10^6)^3$, and accounts for even the largest meshes in our evaluation. Algorithm 1 summarizes the Morton order indexing scheme.

Primitive sorting Once all vertices have an index for their position in the Morton order, we sort the primitives of the mesh. These primitives can be triangles for surface data or tetrahedra for volumetric meshes. For each primitive P_i we assign a key K_i, used for sorting. This key is simply the smallest Morton order index among all vertices incident on P_i. It is important to observe that this operation is the only sorting required by our entire algorithm.

Vertex Layout Finally, after the primitives are arranged in their definitive configuration, we rearrange the vertex array. To do this, we traverse the sorted primitive array and reorder the vertices based on their appearance in the sorted primitives. We have chosen to sort the primitives instead of vertices because if we were to sort the vertices and then lay primitives out according to this order, we would need two sorting operations. In our case, we only need one sorting, plus a single pass over the primitive array.

3.2. Out-of-core Implementation

Our technique requires very little additional memory and no complex data structures. Moreover, the algorithm has a very simple control flow: we perform a single pass over the vertex array, a sorting operation over the primitives, and another pass over the vertices. Because of this simplicity, it is very easy to extend our algorithm to work with very large datasets using an out-of-core

```
Function:
Morton_order(float3 vertex, BoundingBox bbox, uint64 N)
index ← 0;
for 0 <= i < N do
    index ← index << 3;
    // Set the octant using Morton order
    if vertex.x > bbox.center.x then
        index ← index + 1
    end
    if vertex.y > bbox.center.y then
        index ← index + 2
    end
    if vertex.z > bbox.center.z then
        index ← index + 4
    end
    // Update the bounding box to the appropriate octant
    for _d ∈ {x, y, z} do
        if vertex._d > bbox.center._d then
            bbox.min._d ← bbox.center._d
        else
            bbox.max._d ← bbox.center._d
        end
    end
end
return index
```

approach.

Below, we summarize the main aspects that need to be modified in order for this out-of-core implementation to work:

1. The only added memory requirement of our algorithm is an array that contains the computed Morton order indices and another array with primitive indices. The Morton order array consists of N_v 64-bit unsigned integers, namely, $8N_v$ bytes. However, since we only do a single pass over the vertices, this array does not need to be kept in-core. It can be written sequentially to non-volatile storage.

2. If the Morton order index array does not fit in main memory, we can construct the primitive index array by performing $\frac{M}{8N_v}$ passes over the primitive data, where M is the total amount of available memory, in bytes.

3. The required sorting step can be replaced by any of a number of existing

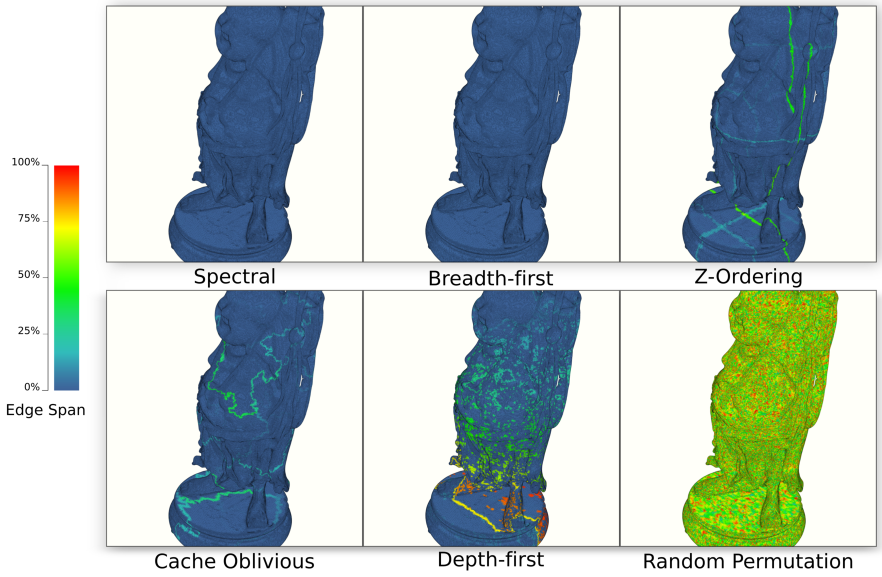

Figure 2. Edge span distribution for different layouts of the Happy Buddha dataset. The colormap is based on the highest span incident on each vertex, and the values are normalized by N_v. In other words, the color on a vertex represents the highest span of its incident edges. Notice how some layout schemes induce particular artifacts on the ordering of the primitives. The cache oblivious layout separates a mesh into locally coherent clusters, while ours (Z-Ordering) breaks it into box-like bricks.

out-of-core sorting solutions.

4. For the last step of the algorithm, we need a bit array of N_v bits that flags vertices which have already been placed in their final position. This bit array has to be kept in-core. Therefore, we need at least $\frac{N_v}{8}$ bytes of main memory. This is a reasonable assumption, however, since for a mesh with a billion vertices, this is equivalent to only 128MB of main memory.

4. Examples and Comparison

There are several ways of evaluating the quality of a given layout. Below, we present the performance of several applications that we have used with our layouts and other alternatives. Moreover, we have also used *edge spans* as a measure of layout quality. The edge span measures the integer separation of two vertices that form an edge. Thus, given $e = (v_0, v_1)$, the span of e

equals the distance between v_0 and v_1 in V. We have observed the edge span distribution of different layouts, and present them in Figure 2.

We have conducted several experimental observations of the performance of the Morton order layout against other popular layouts. We compare our technique with BFS and DFS traversal, cache oblivious layouts, and spectral ordering, as well as the original layout of each dataset. We report the average speedup against the original layout. We ran tests using both surface and volumetric data, running various applications such as simplification, rendering and isovalue extraction. In the case of triangular mesh rendering, we also include additional comparisons to layouts that are optimized specifically for GPU vertex cache, in particular the work of [Lin and Yu 06] and [Sander et al. 07]. We also report computation costs for the layouts. All of the tests were performed on an 8-Core Intel Xeon W5580 3.2GHz, 24GB RAM with an NVIDIA GTX 480.

Dataset	Stats	B/DFS	COML	Lin's	Sander's	Ours
Buddha	Time	0.9s	62s	71s	0.5s	0.4s
1M triangles	Mem	43MB	600MB	350MB	131MB	31MB
Asian Dragon	Time	10s	494s	-	5s	3s
7M triangles	Mem	281MB	5.9GB	-	866MB	199MB
Thai Statue	Time	14s	-	-	7s	4s
10M triangles	Mem	384MB	-	-	1.2GB	275MB
Lucy	Time	55s	-	-	20s	14s
28M triangles	Mem	1.1GB	-	-	3.4GB	771MB
David1mm	Time	202s	-	-	40s	29s
56M triangles	Mem	2.2GB	-	-	6.7GB	1.5GB
St. Matthew	Time	-	-	-	-	239s
372M triangles	Mem	-	-	-	*45GB**	12.7GB
Atlas	Time	-	-	-	-	316s
507M triangles	Mem	-	-	-	*61GB**	17.3GB
Out-of-core (limiting the memory usage to 3GB)						
St. Matthew	Time	-	-	-	-	504s
Atlas	Time	-	-	-	-	701s

Table 1. Layout computation times for triangular meshes. Notice that the Morton order can efficiently process extremely large datasets in both in-core and out-of-core fashion, while alternatives fail above a certain dataset size (**the theoretical amounts of memory required by Sander et al.'s technique*).

Dataset	Tets	BFS/DFS		COML		Ours	
		Time	Mem.	Time	Mem.	Time	Mem.
Blunt Fin	187K	0.4s	9M	8s	115M	0.05s	5M
Heart	359K	1.3s	29M	16s	173M	0.1s	13M
Torso	1M	6s	96M	45s	550M	0.3s	36M
Fighter	1.4M	7s	91M	77s	713M	0.5s	47M
Rbl	3M	14s	281M	165s	2.0G	1.2s	128M
Mito	5M	20s	320M	374s	2.6G	1.7s	183M

Table 2. Layout computation times for volumetric meshes. Our proposed Morton order layout can compute layouts an order of magnitude faster while consuming significantly less resources.

4.1. Layout Computation

Tables 1 and 2 list the computation time and resources needed to build different layouts for surface and volumetric meshes. We were not able to reproduce the COML computation for models larger than the Asian Dragon. Instead, we obtained those meshes directly from the authors of COML [Yoon et al. 05]. For the technique of Lin and Yu, we used the binary that is available directly on the author's website. Unfortunately, the largest mesh that the program can handle is the Buddha model, which is the smallest one in our evaluation. For the method from Sander et al., we were able to obtain the source code from the author, and were thus able to compute layouts on large meshes. As shown by our experiments, our Morton order method is much faster to compute and requires less memory than competing alternatives. For extremely large meshes such as the St. Matthew and Atlas, no method could be applied directly, using 24GB of RAM. In those cases, we also report results for our out-of-core Morton order implementation while limiting main memory to 3GB, a representative amount of memory available on current commodity PCs.

4.2. Effects of Mesh Layouts on Applications

In order to demonstrate the quality of the Morton order layout, we have measured the performance of different applications against our technique, as well as COML and other popular layouts. These applications include interactive rendering, mesh simplification and isovalue extraction.

Rendering We tested interactive frame rates for both surface and volumetric data. For triangle meshes, we used OpenGL with three different ren-

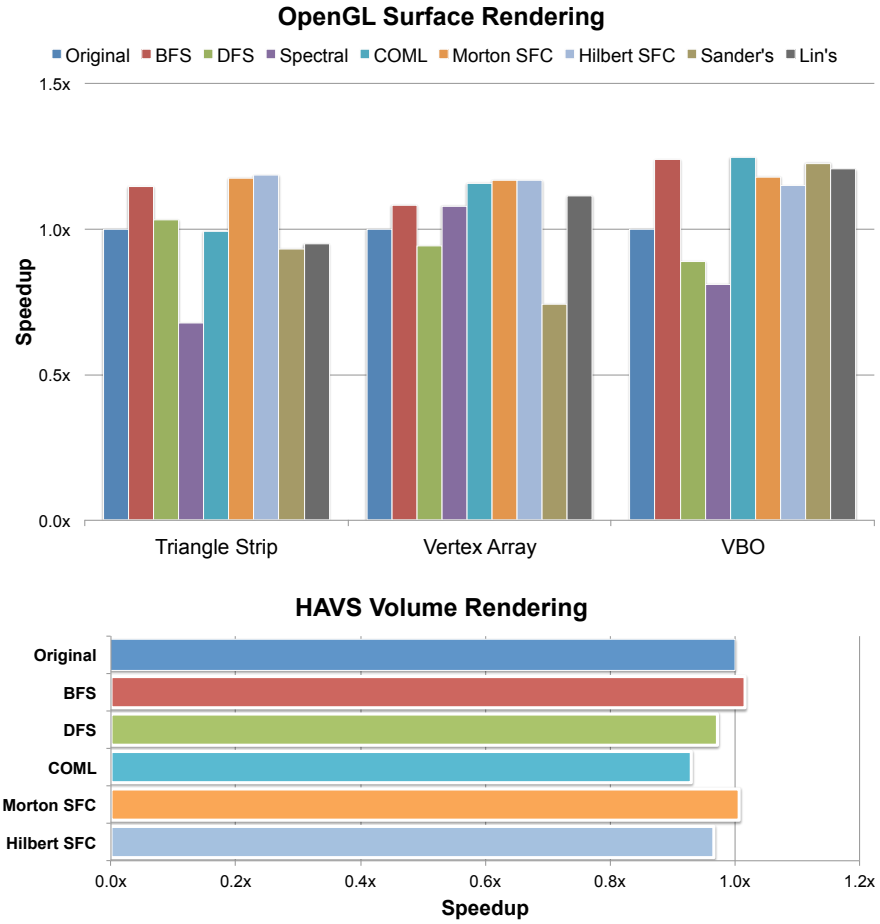

Figure 3. Average frame rates for OpenGL surface rendering and volumetric rendering using HAVS. For triangle meshes, we report the average frame rates using the Happy Buddha, Asian Dragon and Lucy datasets. For volume rendering, we use the Fighter and all smaller datasets.

dering methods: Triangle Strips, Vertex Arrays, and Vertex Buffer Objects (VBOs). Since the largest model that we were able to use with Lin and Yu's method is the Happy Buddha, we limited our model size to the Buddha. Figure 3 presents the results of different layouts. For triangle strips, we used *trimesh2* [Rusinkiewicz] for triangle stripping and rendering. Since both [Lin and Yu 06]'s and [Sander et al. 07]'s techniques are optimized specifically for GPU memory with indexed triangle lists, their layouts have worse per-

formance when rendering triangle strips. On the other hand, they excel in performance when using VBOs, as in this case objects are mapped directly onto GPU memory. Observe that in all cases, space-filling curve layouts are within 5% in performance of the best layout.

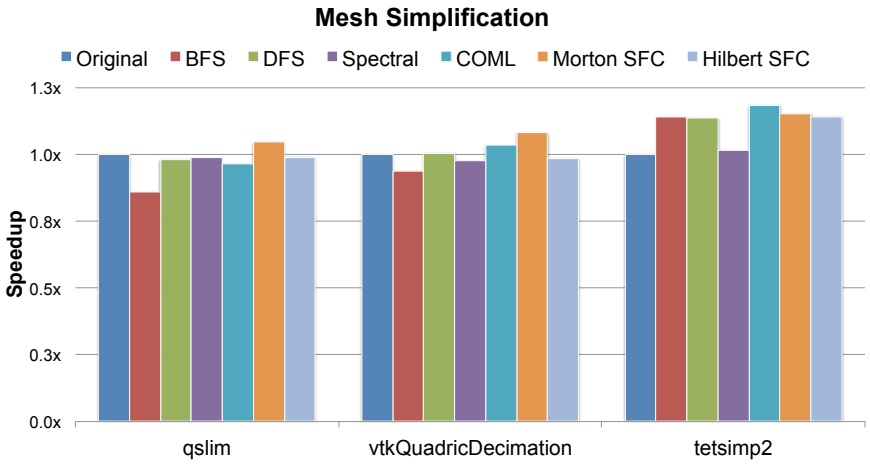

Figure 4. Average simplification time of QSlim, VTK (for surfaces) and tetsimp2 (for tetrahedral meshes).

We used Hardware-Assisted Visibility Sorting (HAVS) [Callahan et al. 05], an unstructured grid volume renderer, to test the performance of tetrahedral mesh layouts. As illustrated in figure 3, the performance gains for volume rendering are less significant. This is expected, since HAVS does not rely on mesh connectivity. Nevertheless, the Morton order layout is consistently among the two best performing methods in our tests.

Simplification Mesh simplification is also a popular processing task against which we tested our layout. We used two different implementations of quadric-based [Garland and Heckbert 97] decimation for surface data: QSlim and the Visualization Toolkit(VTK)'s [Kitware] vtkQuadricDecimation method. For volumetric meshes, we used Vo et. al's Streaming Tetrahedra Simplification [Vo et al. 07]. We report the average speedup to simplify meshes down to 10% of their original resolution. Due to implementation constraints of the simplification methods, the largest meshes we tested were the Lucy (28 million triangles) and Mito (5 million tetrahedra). Figure 4 summarizes these results. In general, the performance gains are marginal. This is due to the fact that quadric-based techniques usually access the mesh in a highly non-structured order. However, the Morton order layout is, again, one of the top two alternatives.

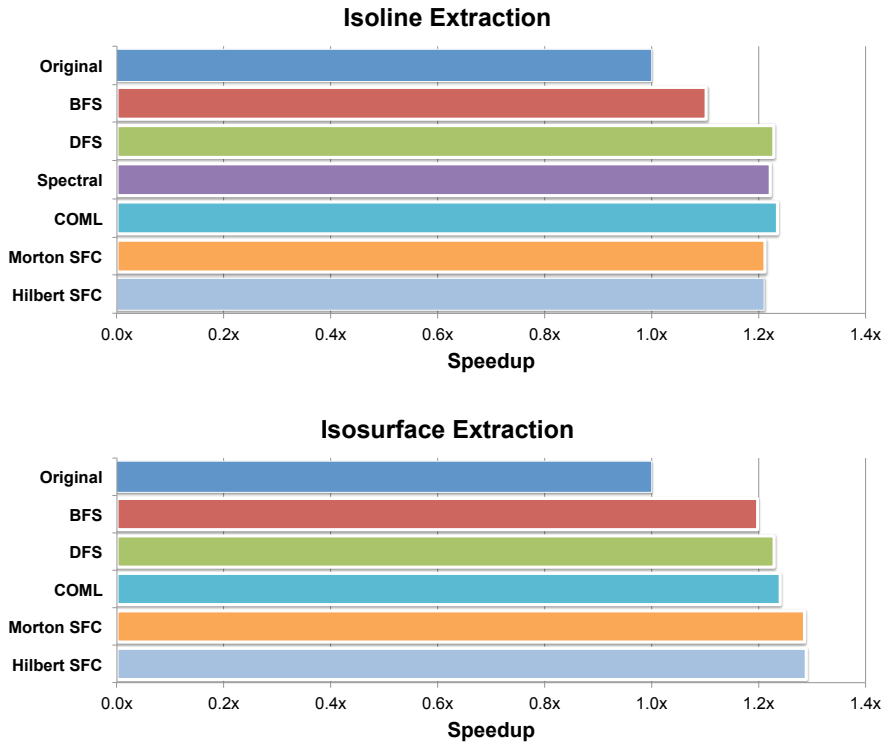

Figure 5. Average isovalue extraction time using VTK's `vtkContourFilter` method. Lucy and Mito were the largest meshes used for this test.

Isovalue Extraction As with simplification, we report the average speedup achieved while extracting isovalues from both surfaces and volumetric datasets. We extracted 100 isovalues using VTK's Marching Cubes [Lorensen and Cline 87] implementation. We embed a simple Euclidean-distance scalar field in meshes that don't already contain scalar data. Figure 5 again shows that Morton order is among the best layouts, though by a slim margin. In this particular test, the DFS also has a comparable quality to both COML and Morton order.

5. Discussion

Our algorithm has many significant advantages over competing alternatives. As we have shown in section 4, its performance is comparable to other, more complex algorithms, such as COML or GPU vertex-cache optimization techniques. Moreover, our Morton order layout is extremely easy to compute.

Formally, our algorithm runs in either $O(N_V + N_I log(N_I))$ or $O(N_V + N_I)$ time, depending on the sorting technique. For our in-core implementation, we use a linear radix sort while our out-of-core implementation relies on a comparison-based external sort. However, we do not provide any bounds on the performance of our Morton order technique. Although extensive tests empirically demonstrate the quality of our layout, we have no theoretical demonstration of its performance. In particular, datasets that have highly irregular geometry distribution may not be improved as much by our technique. For such datasets, a connectivity-based approach, such as DFS, may yield better results. Alternatively, one can always decompose meshes into sections with regular geometric distributions, and construct spatial layouts individually.

Although we have used the Morton order as a basis for mesh layouts, this is not an inherent limitation of our algorithm. We have defined a particular mapping between octal digits and octants in an Octree, to make them correspond to a Morton order. However, any such mapping induces a valid space-filling curve that translates to a mesh layout. In particular, it is easy to obtain a Hilbert curve by reordering the octants. Moreover, other implicit tree structures, such as a Kd-tree, might also be used to generate coherent layouts. A possible avenue for further research is to try and determine whether there exist particularly good spatial tree structures that fit the layout problem. Although this is an interesting question, we have not pursued its solution at this time.

Acknowledgments. We would like to thank the Stanford University Computer Graphics Laboratory and XYZ RGB Inc. for the scanned models. We also thank Jason Shepherd for the Rbl and Mito datasets, NASA for the Langley Fighter dataset and Rob MacLeod at the University of Utah for the Torso dataset. We thank Sung-Eui Yoon, Peter Lindstrom and Pedro Sander for providing their layout source codes. This work was supported in part by an NVIDIA Fellowship, the National Science Foundation awards CNS-1153503, IIS-1153728, OCI-0904631, OCI-0906379, IIS-1045032, IIS-0844572, CNS-0751152, and CCF-0702817, the U.S. Department of Energy BER and ASCR.

References

[Callahan et al. 05] S. P. Callahan, M. Ikits, J. L. D. Comba, and C. T. Silva. "Hardware-assisted visibility sorting for unstructured volume rendering." *IEEE Transactions on Visualization and Computer Graphics* 11:3 (2005), 285–295.

[Garland and Heckbert 97] M. Garland and P. S. Heckbert. "Surface simplification using quadric error metrics." In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 209–216, 1997.

[Gotsman and Lindenbaum 94] C. Gotsman and M. Lindenbaum. "On the metric properties of discrete space-filling curves." *Pattern Recognition* 3:1 (1994), 98–102.

[Hungershöfer and Wierum 02] J. Hungershöfer and J. M. Wierum. "On the Quality of Partitions Based on Space-Filling Curves." In *ICCS '02: Proceedings of the International Conference on Computational Science-Part III*, pp. 36–45. London, UK: Springer-Verlag, 2002.

[Isenburg and Lindstrom 05] M. Isenburg and P. Lindstrom. "Streaming Meshes." In *IEEE Visualization '05*, pp. 231–238, 2005.

[Kitware] Kitware. "The Visualization Toolkit (VTK) and Paraview." http://www.kitware.com.

[Lin and Yu 06] G. Lin and T. P. Yu. "An Improved Vertex Caching Scheme for 3D Mesh Rendering." *IEEE Transactions on Visualization and Computer Graphics* 12 (2006), 640–648.

[Lindstrom and Pascucci 01] P. Lindstrom and V. Pascucci. "Visualization of large terrains made easy." In *IEEE Visualization '01*, pp. 363–371, 2001.

[Lorensen and Cline 87] W. E. Lorensen and H. E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm." In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pp. 163–169, 1987.

[Pascucci and Frank 01] V. Pascucci and R. J. Frank. "Global static indexing for real-time exploration of very large regular grids." In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pp. 2–2, 2001.

[Rusinkiewicz] S. Rusinkiewicz. "trimesh2." http://www.cs.princeton.edu/gfx/proj/trimesh2/.

[Sagan 94] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.

[Sander et al. 07] P. V. Sander, D. Nehab, and J. Barczak. "Fast triangle reordering for vertex locality and reduced overdraw." *ACM Transactions on Graphics* 26.

[Velho and Gomes 91] L. Velho and J. M. Gomes. "Digital halftoning with space filling curves." In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pp. 81–90, 1991.

[Vo et al. 07] H. T. Vo, S. P. Callahan, P. Lindstrom, V. Pascucci, and C. T. Silva. "Streaming Simplification of Tetrahedral Meshes." *IEEE Transactions on Visualization and Computer Graphics* 13:1 (2007), 145–155.

[Yoon and Lindstrom 06] S. E. Yoon and P. Lindstrom. "Mesh Layouts for Block-Based Caches." *IEEE Transactions on Visualization and Computer Graphics* 12:5 (2006), 1213–1220.

[Yoon et al. 05] S. E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. "Cache-Oblivious Mesh Layouts." *ACM Transactions on Graphics* 24:3 (2005), 886–893.

Web Information:

http://vgc.poly.edu/projects/meshlayout/

Huy T. Vo, Polytechnic Institute of New York University, 6 MetroTech Center, Brooklyn, NY, 11201
(hvo@poly.edu)

Received [DATE]; accepted [DATE].