Out-of-Core Progressive Lossless Compression and Selective Decompression of Large Triangle Meshes *

Zhiyan Du Pavel Jaromersky Yi-Jen Chiang Nasir Memon

Abstract

In this paper we propose a novel *out-of-core* technique for *progressive* lossless compression and *selective* decompression of 3D triangle meshes larger than main memory. Most existing compression methods, in order to optimize compression ratios, only allow *sequential* decompression. We develop an integrated approach that resolves the issue of so-called *prefix dependency* to support *selective* decompression, and in addition enables I/O-efficient compression, while maintaining high compression ratios. Our decompression scheme initially provides a global context of the entire mesh at a coarse resolution, and allows the user to select different *regions of interest* to further decompress/refine to **different** levels of details, to facilitate out-of-core multiresolution rendering for interactive visual inspection. We present experimental results which show that we achieve fast compression/decompression times and low memory footprints, with compression ratios comparable to current out-of-core *single resolution* methods.

1. Introduction

Although there has been a significant amount of research on graphics compression, most of these techniques require the entire input mesh plus the additional data structures to reside in main memory, which is a severe limitation to their applicability. In particular, for those gigantic datasets that need compression the most, such limitation is clearly a major obstacle. Previously, there were a few out-of-core compression algorithms that work well for 3D polygonal meshes larger than main memory [8, 9, 11], but they only provide *single-resolution* (i.e., *non-progressive*) compression. Clearly, a *progressive* representation of a mesh is much more desirable to achieve interactive visual inspection, even for smaller datasets that can fit in main memory [15].

In this paper we propose a novel *out-of-core* technique for *progressive* lossless compression and *selective* decompression of large 3D triangle meshes. An important feature of our algorithm is that we resolve the issue of so-called *prefix dependency*. Most existing compression techniques compress the current data item using the information of the items compressed before, in order to optimize compression ratios. Therefore, each item i depends on the previous items p in the compressed data stream, and we must decompress all items p before we can decompress item i. We call this condition *prefix dependency*. Due to prefix dependency, such compression techniques only support *sequential* decompression as opposed to *selective* decompression, which is more desirable in mesh applications. In particular, in the context of progressive compression for general triangle meshes, the existing approaches are *in-core* and they are mostly constrained by prefix dependency: to obtain some interested portion of the mesh at a particular level of detail, *all* previous levels of the *entire* mesh must be decompressed first. This scheme would not work in the out-of-core setting, since at some (refined) level the main memory would not be large enough to hold the entire mesh for further decompression. A potential method to support selective decompression would be to partition the mesh into self-contained sub-meshes and compress them *independently*, but we need a more clever approach to achieve high compression ratios. Recently, Kim et al. [12] developed a multiresolution compression

*Research supported by NSF grants CCF-0093373 and CCF-0541255. The authors are with the CSE Department, Polytechnic Institute of New York University, Brooklyn, NY, USA. Email: {zdu,jpavel}@cis.poly.edu; {yjc,memon}@poly.edu.

technique that supports *random accessible* decompression, but it does not support I/O-efficient computation. Yoon and Lindstrom [19] gave an out-of-core random accessible compression algorithm, but it is a *single resolution* approach mainly for *geometry computing* applications such as isocontouring and mesh re-ordering rather than for interactive visual inspection. Cai et al. [2] proposed an out-of-core progressive compression method for *isosurfaces*, but it requires the input triangle mesh to be converted into a regular-grid volume data to be encoded, and the decoding has to be followed by an isosurface extraction step to *reconstruct* the triangle mesh; therefore the method is *lossy*. Moreover, selective decoding is *not* addressed in [2]. Gobbetti et al. [7] presented an out-of-core multiresolution compression approach for terrain rendering, but it requires a special mesh structure where all patches share the *same regular triangulation connectivity* (and thus there is *no* issue of prefix dependency). Our new algorithm, which is out-of-core, multiresolution, lossless, and supports selective decompression for general 3D triangle meshes, tries to complement these techniques and fill in the gap in the literature.

Our approach is based on the in-core progressive lossless compression algorithm of Gandoin and Devillers [6], which utilizes a space-partitioning tree (a k-d tree). We propose an integrated solution to support I/O-efficient computation and to resolve the issue of prefix dependency,[1] while maintaining high compression ratios. Our decompression scheme initially provides a global context of the entire mesh at a coarse resolution, and allows the user to select different *regions of interest* to further decompress/refine to **different** levels of details, to facilitate out-of-core multiresolution rendering for interactive visual inspection. The experiments demonstrate the efficacy of our new algorithm. In particular, we can compress the large St. Matthew dataset I/O-efficiently with only 107MB of memory footprint, while achieving a compression ratio comparable to those of state-of-the-art out-of-core *single resolution* (lossless) methods [9, 11, 19], albeit our approach is multiresolution and allows selective decompression (see Sec. 4).

2. Previous Related Work

There has been a significant amount of work on mesh compression. As a comprehensive review of these techniques is not a focus of this paper, we refer to the excellent survey [1] for details.

The in-core compression approach that is most closely related to our work is the progressive method [6]; we review it in Sec. 2.1. We remark that the follow-up technique [16] typically improves the bit rates of [6] by 10-20% in the geometry coding (which dominates the connectivity coding cost) but is more complex, and thus we opt to use the simpler, and yet still bit-rate efficient method [6] as our basis. As for out-of-core compression, so far the results are mainly for *single resolution* [8, 9, 11, 19] (except for the isosurface method [2]). In [8] a special treatment is given on the *borders* of mesh partitions. However the borders are *duplicated* for independent compression of each partition, and it is necessary to encode extra data for "gluing" different copies of the same border vertex. This scheme does not work for our case of multiresolution compression, as duplicating *multiple versions* of *border triangles* (that lie across subtrees in our case) and their gluing information would be too costly in compression ratios. Instead, we only keep *one copy* for each border triangle and *propagate* it across different subtrees at its different stages of compression, which is a major technical component of our algorithm.

As mentioned in Sec. 1, the compression techniques that can support *random accessible* decompression include [12, 19, 7]; the list additionally includes [3], which is an in-core single-resolution method. Moreover, there has been an extensive work on out-of-core multiresolution, view-dependent rendering (with the main focus *not* on compression); we refer to [4, 18] and references therein for these results.

2.1. Our Basis: In-Core Progressive Compression

We now review the in-core progressive compression approach [6], which is the basis of our algorithm. Initially, each vertex coordinate of the input mesh is quantized into a b-bit integer. The quantization bounding box is a $2^b \times 2^b \times 2^b$ uniform grid with each vertex lying in the cell center. Now consider building an octree

[1] In [6] the authors mention that it is possible to refine selected parts of the mesh, however the prefix dependency is *not* discussed.

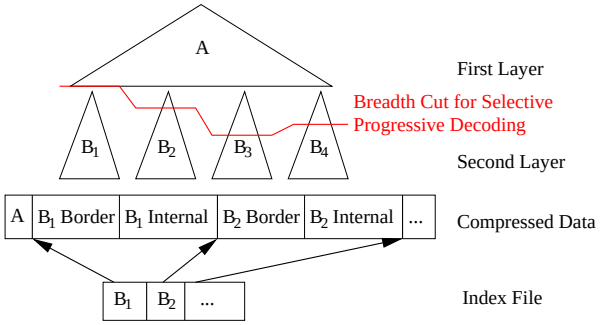

Figure 1: The two-layer k-d tree and the structure of the compressed stream. We also show an example of a breadth cut for selective progressive decompression.

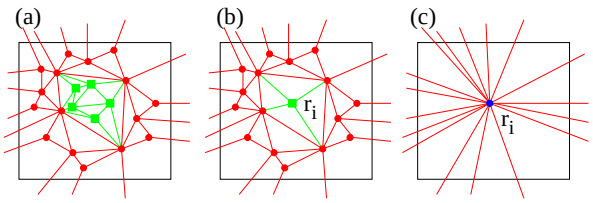

Figure 2: Simplification of the mesh in subtree B_i. The circle vertices are *border vertices* and the square ones are *internal vertices*. The root cell center is r_i.

on the *non-empty* cells as leaves; going up one level is a quantization of one fewer bit in each coordinate, where the (up to 8) child vertices are collapsed to the parent cell center, with the connectivity simplified accordingly. This scheme is similar to the *vertex clustering* method widely used for connectivity simplification (e.g., [14]). Although it may cause some artifacts such as seams and non-manifolds, from the viewpoint of graphics compression, this approach has a nice property that the *geometry* (i.e., *quantization precision*) and the *connectivity* are *simultaneously* simplified naturally.

The method [6] actually uses a *k-d tree* T rather than an octree, replacing each octree level by three levels of (binary) k-d tree, to make the change in quantization precision (as well as connectivity) smoother, one dimension at a time. This also makes the encoding of *geometry information* simple: At the root, we explicitly record the total number n of vertices; next, we explicitly record the total number n_1 of vertices lying in the left-child cell but *not* the number n_2 in the right child, since n_2 can be derived by $n_2 = n - n_1$. In this way, we (explicitly or implicitly) record the number of vertices lying in each node. Due to the regular structure, the explicitly recorded numbers are enough for the geometry information. We construct the compressed stream of the *geometry code* by traversing T top-down in a breadth-first order, and encoding the explicitly recorded numbers by an arithmetic code.

The connectivity information is added at the leaves of the tree T, where for each vertex we store the incident edges and triangles. To encode connectivity, we first traverse T from bottom up and simplify the mesh connectivity as described above. When merging two child vertices into the parent vertex, we store at the parent some necessary connectivity information for the reverse operation to reconstruct the mesh. After simplification is done, we traverse T top-down in a breadth-first order and encode the information for connectivity reconstruction. The connectivity encoding is further enhanced by using some predictive techniques; see [6] for more details.

3. Our Approach

3.1. Overview

We split the k-d tree T of [6] into *two layers*, with the top tree A in the first layer and a set of subtrees $B_1, B_2, \cdots$ in the second layer; each leaf of A is the root r_i of some subtree B_i (see Figure 1). The tree A has L levels where L is a user-specified parameter. Typically we make A stop at the p-bit precision of each coordinate for some value p, and the corresponding L is $L = 3p$. We assume that A, as well as each individual B_i, can fit in main memory. This is typically true for large scanned datasets (e.g., [13]) in practice.

To address the *prefix dependency*, we design our compression stream as shown in Figure 1: The first part is the code for A, followed by the code for the subtrees $B_1, B_2, \cdots$ one by one in that canonical order. For each B_i, we separate its *border portion* (which connects *beyond* B_i), from its *internal portion* (which

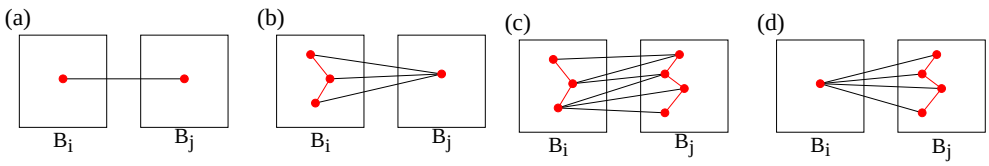

Figure 3: Connectivity between borders of two subtrees B_i and B_j.

only connects *within* B_i), and encode them *separately*, with the border portion first (see Figure 1). We also have an *index file* to indicate the starting position of each B_i. Note that each subtree can be *independently* encoded/decoded for its internal portion, and the prefix dependency is only on the border portions connecting among related subtrees. With this approach, we can achieve I/O-efficient encoding/decoding and support selective decoding, with high compression ratios.

During decompression, we first decompress A completely, providing a coarse version of the mesh at the leaf level of A. It is important to stress that while A is decoded completely (to get the best possible mesh quality from A, which is already coarse), the subtrees B_i can be selected and decoded to *any of their own* desired levels (e.g., B_1 not decoded, B_2 decoded 3 levels down, B_3 7 levels down, etc.), to support progressive adaptivity (see the breadth cut in Fig. 1). When the user selects which subtrees B_i to decompress and to what level of details, for the *prefix-dependent* subtrees that B_i connects to, we only need to decompress the *border* portions and can skip the internal portions (using the indices in the index file). As mentioned, the *internal portions* of different subtrees are *independent* of each other and can be independently decoded progressively to their *own desired levels*. In this way, we can significantly reduce the decompression time and the main memory space needed, to facilitate I/O-efficient *selective* decompression and visualization.

A main focus of our algorithm is how to compress the mesh of each subtree B_i. We first progressively simplify the *internal vertices* of B_i (which only connect within B_i) to a single point r_i (the center of the root cell), and then progressively simplify the *border vertices* (which connect beyond B_i) to this point r_i (see Fig. 2 (a) to (b) to (c)), so that during decompression we can first progressively decode the border and possibly skip the internal part (recall the simplification/(de-)compression scheme in Sec. 2.1). Note that compression and decompression are in the *same* order, which must be the *reverse* order of simplification. Therefore, if we want to decompress the internal vertices, the border vertices must be first *fully* decompressed (see Fig. 2 from (c) to (b)), but then the internal vertices can be decompressed progressively to the *desired level only* (Fig. 2 from (b) to (a) but we can stop in between).

The major technical challenge is how to deal with the *border triangles* that lie across subtrees. Suppose subtrees B_i and B_j share some border triangles (see Fig. 3(c)), where B_i is compressed *before* B_j and thus a prefix-dependent subtree of B_j. As pointed out in Sec. 2, employing the idea of [8] to duplicate the border triangles needs to store the duplicate(s) and the extra information to *glue* copies of the same vertex, and thus the scheme has too much coding cost in multiresolution compression. So we only keep *one* copy for each border triangle.

Now we consider the process of progressively compressing/decompressing the borders. Observe that B_i and B_j can each be fully refined and/or fully simplified (to a single point), resulting in four "end cases" between B_i and B_j, with two "end versions" each (see Fig. 3(a)-(d)). These are the base cases that we need to support. Since B_i is compressed before B_j and the decoder must follow the same order, during decompression we start from Fig. 3(a) and first progressively decode B_i fully to obtain (b) in the figure. We then continue to progressively decode B_j fully to obtain (c). Finally, if we do not need B_i in its full version, we can progressively *collapse* vertices of B_i back (in the reverse order of decoding/refinement), eventually obtaining (d), at *no extra* coding cost. Of course, from (c) we can collapse vertices in B_i to some desired level and also collapse vertices in B_j to another desired level, to obtain connectivity between

any intermediate versions of B_i and of B_j, at *no extra* coding cost. In this way, we can achieve very good coding efficiency while covering all possible connectivities. Observe that this scheme imposes a *prefix dependency* of B_j on B_i: decompressing B_j depends on fully decoding B_i first, which in turn depends on fully decoding its prefix-dependent subtrees, and so on, and we need to resolve all these dependencies first.[2] As for compression, it is important to see that since we only keep *one* copy for a border triangle, the connectivity is *carried over* as we *propagate* the triangle across the subtrees. How to propagate border triangles at their various stages (as in Fig. 3) is a major technical component of our algorithm. We describe our technique in more details below.

3.2. Out-of-Core Progressive Compression

We present our out-of-core progressive compression algorithm. We assume that the input mesh is in the form of a triangle soup; for an indexed mesh we first perform an out-of-core *pointer de-referencing* [17] via a few external sortings to obtain a triangle soup. We remark that the form of *streaming mesh* [10] is also easy to use for our technique, and yet triangle soup provides a self-complete information for each triangle, making our task of propagating border triangles a little bit easier to describe.

After an initial quantization, we proceed the compression in two phases. In the first phase, we construct the first-layer tree A, distribute the input triangles to the leaves of A, simplify the coarse mesh of A and compress it. In the second phase, we process the leaf cells of A and compress their subtrees $B_1, B_2, \cdots$ one at a time *in that canonical order*, with *prefix dependency* the main consideration.

3.2.1. First Phase of Compression

In this phase, we scan through the triangles one by one. For each triangle t, we locate the leaf cells of the tree A containing the three vertices of t. In the process, we *incrementally* construct the tree A: if the leaf u containing the current vertex does not exist, we grow A by adding the missing nodes in the path from the root to u. Recall that A stops at level L. For each current triangle t, we assign it to a *single* leaf of A, and classify it as an *internal* or *border* triangle of that leaf. If all vertices lie in the same leaf, t is *internal* and is assigned to that leaf; otherwise, t is *border* and is assigned to the leaf with the *smallest* node ID. For a border triangle t, we also add the corresponding connectivity information, connecting between two leaves of A that contain vertices of t.

When we scan and distribute the triangles to the leaves of A, the distributed triangles collectively can exceed the main memory size and need to be written to disk. We use the strategy of [5] to perform the same process twice: once we count how many triangles are assigned to each leaf without actually putting them out so that the starting position for each leaf in the file is known, and in the second time we actually write out the triangles.

Now we proceed to compress the first-layer mesh corresponding to A. Recall from Sec. 2.1 that we record the number of vertices lying in each node cell of A for geometry coding. Instead of the top-down process in Sec. 2.1, we use a bottom-up process: we assign number 1 to each leaf of A (since the leaf has one (collapsed) vertex in the coarse mesh, the cell center), and for each internal node we assign its number as the sum of the two child numbers (treating a null child as having number 0). In the same bottom-up process, we also simplify the connectivity progressively. Finally, we perform a top-down breadth-first traversal on A and complete the geometry as well as the connectivity encoding.

When the compression is finished, the structure of the mesh for the second phase of compression is described solely by the subtrees B_i and their neighbors. It is no longer necessary to keep the tree A (and the coarse mesh) in main memory and we can free such memory.

[2] We use a plain queue to find all subtrees whose borders need to be decoded, sort these subtrees by increasing subtree IDs, and then decode their borders in that order.

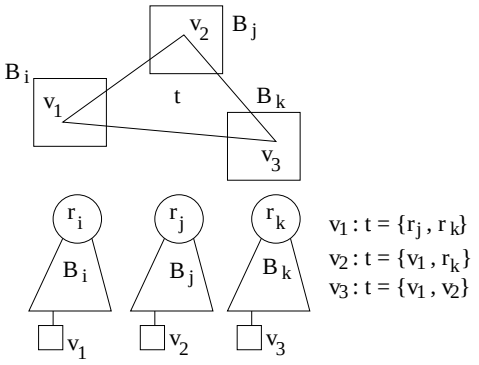

Figure 4: Processing a border triangle t with three vertices lying in B_i, B_j and B_k, with $i < j < k$ (i.e., compression order is B_i then B_j then B_k).

Figure 5: First-layer mesh of Lucy with different depths of the first-layer tree A. Quantization precision (p value): 6, 7, 8 from left to right.

3.2.2. Second Phase of Compression

In the second phase of compression, we process the second-layer subtrees $B_1, B_2, \cdots$ in that canonical order (see Fig. 1) one at a time. For each B_i, after loading to main memory the triangles assigned to B_i, with the *border* and *internal* triangles kept separately, we perform the following tasks: **(1)** process the border triangles; **(2)** process the internal triangles; **(3)** simplify the mesh of B_i; **(4)** encode the mesh of B_i; **(5)** free resources. We now describe the details of these tasks.

Task 1: Processing the Border Triangles

In this step, we scan through the border triangles one at a time, add *border vertices* to B_i, incrementally construct the portion of the k-d tree B_i that correspond to its border vertices, and add the connectivity information to these vertices. Recall that we define a vertex lying in B_i to be *border* if it has an edge connecting outside of B_i, and *internal* if all its edges connect to vertices lying in B_i (see Fig. 2). In the process, we also add border vertices to the *neighboring* subtrees B_j's, $j > i$ (i.e., B_j will be processed *later* than B_i), grow the border-vertex portion of the k-d trees B_j's, and add the connectivity information to these border vertices.

Let $t = (v_1, v_2, v_3)$ be the current border triangle being processed. Recall that t has its three vertices located in different second-layer subtrees, and is assigned *only* to the subtree with the smallest ID. Suppose that v_1 lies in B_i, v_2 in B_j, and v_3 in B_k, with $i < j < k$, meaning that the compression order is B_i first then B_j and then B_k (see Figure 4); note that t belongs to B_i and thus i is the smallest among i, j, k. For each second-layer subtree, we have a local *border vertex array*, created when needed in the first time, to keep the border vertices, where for each border vertex we also include its coordinates in the array. When processing t, we locate v_1 to a leaf of the k-d tree B_i (while incrementally grow B_i), and add v_1 to the border vertex array of B_i. We also add the connectivity information due to t: we store the information of t in v_1. Since the connectivity coder of [6] needs the position of the neighboring vertices to perform prediction, the connectivity stored in v_1 has to be able to get the positions of v_2 and v_3. However, decoding needs to follow the same encoding order; at the time B_i is decoded, B_j and B_k are *not* decoded/refined yet and each is still a single root node r_j (r_k), with v_2 collapsed to the cell center of r_j and v_3 to the cell center of r_k. Therefore, in v_1 we store $t = \{r_j, r_k\}$ (see Fig. 4), meaning that $t = (v_1, r_j, r_k)$.

Moreover, we *propagate* triangle t to both B_j and B_k so that the information of t is available when later we compress B_j and B_k. Similar to the above process, we locate v_2 to a leaf of the k-d tree B_j (while incrementally grow B_j), add v_2 (including its coordinates) to the border vertex array of B_j, and store the connectivity of t to v_2, now recording $t = \{v_1, r_k\}$, where v_1 is recorded by

(a) an encoding of the node ID of the leaf of the k-d tree B_i locating v_1 (for the connectivity as well as the

geometry of v_1), and
(b) an index of v_1 in the border vertex array of B_i (for the coordinates of v_1, to be used by the encoder later when we process B_j).
(We remark that (b) is redundant of (a) and will *not* be encoded into the compression stream. We include (b) here just to make the task of releasing resources easier; see Task 5 below). Observe that this recording of t (with (b) skipped) reflects the scenario when B_j is decoded, where B_i has been refined but not B_k. We then perform a similar process for v_3 in B_k, now recording $t = \{v_1, v_2\}$ (see Figure 4).

Observe that due to this propagation process, if the current subtree B_i has $i > 1$, then some of its border vertices may have already been propagated from previous subtrees, before we start working on B_i. Such propagation is only to *later* subtrees but never to previous subtrees, so when we finish processing the border triangles assigned to B_i, its border vertices are finalized. The k-d tree B_i constructed so far corresponds to the border vertices. Also, the border vertex array may contain duplicated copies of the same vertex, each for a different triangle sharing the vertex.

Task 2: Processing the Internal Triangles
We scan the internal triangles of B_i one at a time. For each current triangle t, we locate the three vertices of t to leaves of the k-d tree B_i while incrementally growing B_i, and add the connectivity information. If any *new* vertex is created, then it is an internal vertex and is added to the local *internal vertex array* for B_i. Note that since only newly created vertices are added to the array, there is no duplication in the array.

Task 3: Simplifying the Mesh
Recall that simplification is done by going level by level up in the k-d tree B_i, and that encoding/decoding will be done in the reverse order, top down. Since in decompression we want to decompress the border vertices first and then the internal vertices (so that we can skip the internal vertices if not needed), the simplification must be performed reversely, internal vertices first and then border vertices (recall from Fig. 2). To simplify the internal vertices, we go up from the leaves of B_i that are internal vertices, until finally we collapse all internal vertices to a single point at the cell center of the root. In the process, we mark all the nodes traversed as *internal* so that later we can visit them during internal-vertex encoding. We then simplify the border vertices together with this cell center, going up from the border-vertex leaves until all vertices are collapsed to the root cell center. Again we mark all the nodes traversed as *border* to be used later during border-vertex encoding. Note that a tree node can be marked *both* internal and border (when it is a common ancestor of an internal and a border leaves). Essentially, the tree nodes marked *internal* make an "internal" k-d tree, and similarly for a "border" k-d tree; the k-d tree B_i is just the union of the two k-d trees. We keep two separate vertex-number counts on the tree nodes, one for the "internal" k-d tree and the other for the "border" k-d tree, for encoding the vertex geometry as well as the tree structures of the two trees.

Task 4: Encoding the Mesh
Now encoding is easy. We first encode the border vertices of B_i by a breadth-first traversal on the border k-d tree, and then we encode the internal vertices of B_i by a breadth-first traversal on the internal k-d tree. Note that in the process we encode both geometry and connectivity, where the connectivity can of course connect between border and internal vertices, among others.

Task 5: Releasing Resources
After encoding the mesh of B_i, we want to release the main memory resources that are no longer needed. The pieces of information that are still needed are related to the *border vertices*, which will be referenced from other second-layer subtrees later (other information can be released right away). Referring to Figure 4 for example, we see that v_1 of B_i will still be needed when later we process B_j and B_k, and when we finish with B_j, v_2 of B_j will still be needed later when we process B_k. Now we discuss when to release the border vertices of a subtree. Suppose B_m is a *neighboring* subtree of B_i, i.e., there is a border triangle connecting them. We call B_m a *prefix neighbor* of B_i if $m < i$ and a *suffix neighbor* if $m > i$. Clearly, the border vertices of B_i can be released when its largest-ID suffix neighbor has been processed. In general, for each subtree we store the subtree IDs of its prefix and suffix neighbors. When we finish processing the current

Mesh	Dawn	Night	Lucy	David	St.Matthew
# V (M)	3.43	11.05	14.03	28.18	186.84
# T (M)	6.59	21.57	28.06	56.23	372.77
Ply (MB)	134	447	508	1,127	7,475
time (m)	1.58	5.03	6.7	15.37	68.6
size (MB)	113	370	482	965	6400

Table 1: Mesh statistics. The lower part shows the time to quantize (to 16 bits) and convert to triangle soup, and the resulting size.

Mesh	bit rate (bpv)			compression time (min) / memory footprint		
	p=7	p=8	in-core	p=7	p=8	in-core
Dawn	24.00	25.99	20.77	2.1	2.53 / 72MB	2.77 / 1.6GB
Night	19.87	21.42	17.84	6.07	7.23 / 83MB	9.4 / 4.8GB
Lucy	19.81	21.2	18.12	8.47	9.37 / 68MB	13.13 / 6.0GB
David	15.74	16.73	14.61	14.05	15.08 / 51MB	833 / 12.7GB
St.Matthew	12.41	12.94	N/A	92.35	125.03 / 107MB	N/A

Table 2: Compression results. Initial quantization is 16 bits per coordinate. The memory footprint for p=7 is similar to p=8 and is not shown. Our approach ran under **0.5GB** of RAM and the in-core method ran under **12GB** of RAM.

subtree B_i, we check to see if any of its prefix neighbors can be released. This is done by checking them one by one, and see if B_i is the largest-ID suffix neighbor of any of them.

4. Experimental Results

We have implemented our technique in C/C++ and ran our experiments on two Dell Precision PCs; they have exactly the same configuration except for the RAM sizes (0.5GB vs. 12GB): two 3GHz Intel Xeon CPUs, Nvidia Quadro FX 4500 graphics, 300GB SCSI 10K rpm disk, and a 64bit Linux OS. We show in Table 1 the datasets used[3], which we initially quantized to 16 bits for each vertex coordinate and converted to triangle soup via out-of-core pointer de-referencing [17].

In order to evaluate our algorithm, we want to compare with its in-core counterpart [6]. To this end, we took our program and set the first-layer tree A to have $L = 48$ levels, i.e., letting tree A in main memory to be the entire k-d tree; we call this program `inc` and ours `ooc`. Note that `inc` still keeps the input triangle soup *on disk* and the only in-core working set to process the input is the space to hold one triangle, so that the entire main memory is devoted to tree A. For `inc`, this way is even a bit more space-efficient than using streaming mesh [10] since there is no need to keep unfinalized vertices. In the following, both `ooc` and `inc` ran on the same triangle soup as inputs, under RAM sizes 0.5GB and 12GB respectively.

To see how many levels the tree A should have in `ooc`, we set L to be 18, 21, 24 (corresponding to quantization precision of p bits per coordinate at the leaf level of A, with $p = 6, 7, 8$; recall that $L = 3p$) and ran it on Lucy. We see that smaller value of L (p) made better compression time and ratio (7m32s, 8m29s, 9m22s for $p = 6, 7, 8$ with bit-rates 18.98, 19.81, 21.2 bpv), but worse image quality (see Fig. 5). We recommend to use $p = 8$, and possibly $p = 7$ for better compression if we are willing to tolerate a worse quality in the global first-layer mesh.

Next, we compared the compression performance of `ooc` (with $p = 7, 8$) and `inc`, and show the results in Table 2. We see that our compression ratios are worse (on an average 11% worse for $p = 7$ and 19%

[3]The datasets are courtesy of the Stanford Graphics Lab. For St. Matthew the ply size shown is the total of the original 12 self-complete indexed sub-files without removing the duplicated vertices.

Mesh	Dawn	Night	Lucy	David	St.Matthew
Uniform LOD (0.5GB RAM) /Varying LOD (0.5GB RAM) /In-core (12GB RAM)					
time (m)	0.03/0.01/0.92	0.11/0.08/2.48	0.4/0.35/3.25	0.73/0.62/5.47	2.5/2.38/ N/A
# tri (M)	0.30/0.26/6.59	0.47/0.33/20.57	0.71/0.41/27.6	1.68/0.52/54.68	3.36/1.09/ N/A
mem (MB)	79/78/829	105/92/2662	134/103/3379	253/112/6144	468/297/ N/A

Table 3: Decompression results (where $p = 8$). We show our selective decompression with uniform and varying LODs, compared with the in-core method.

worse for $p = 8$) due to the support of selective decompression. However, our memory footprint was quite small—only 107MB for St. Matthew, while `inc` had a large footprint for tree A—about 10 times the mesh size, which is not too surprising for multiresolution data structures given that the incident triangles and incident edges are explicitly stored with each vertex in the structure. For David, `inc` already resulted in thrashing, since the memory access of traversing tree A is quite random. This shows that our two-layer scheme is very effective: we can greatly reduce the memory footprint even with the same implementation for the incidence information.

For selective decompression, in principle we can select the subtrees manually or automatically by view-dependent level-of-detail (LOD) techniques [15]. Since our focus in this paper is on compression/decompression, we defer automatic selection to future work. Currently, our implementation supports manual selection of subtrees via a mouse click, where the LOD can either vary gradually (*varying* LOD) or stay the same (*uniform* LOD) when moving away from the clicked subtree through neighboring subtrees up to some degrees of neighboring. We show the results in Table 3, where the most detailed level was set to 16 bits of precision per coordinate (except for St. Matthew uniform LOD (14 bits); for St. Matthew varying LOD it was still set to 16 bits). For varying LOD, the clicked subtree, its immediate neighboring subtrees (the degree-1 neighbors) and the neighbors of degree-1 neighbors (the degree-2 neighbors) had the same tree depth (at 16 bits of precision); the next-degree (degree-3) neighbors were at tree depth two levels up, the degree-4 neighbors were at tree depth two more levels up, and so on, until 18 subtrees were selected. We remark that decompressing the first-layer tree A was always less than 1 second. Also, in the in-core approach, the encoder needs to keep the entire tree but the decoder only keeps the current level, and thus the memory footprint for decoder was only about half of that for encoder. Note that with the original in-core approach, it is necessary to decompress the whole mesh up to the specified LOD. The number of decompressed triangles is also important, since a lower number results in faster and smoother interaction with the decompressed mesh. We show the corresponding images of our selective decompression in Figure 6.

Comparison with other out-of-core methods

Although our compression ratio is (moderately) worse than `inc`, it is comparable to the state-of-the-art *out-of-core, single-resolution* approaches. Taking our $p = 8$ result for St. Matthew (Table 2), we are 21% worse (12.94 vs. 10.67 bpv) than [9] (which neither preserves layout order nor supports selective decompression), 9% worse (12.94 vs. 11.82 bpv) than the order-preserving approach [11] (which does not support selective decompression), and 77% better (12.94 vs. 22.9 bpv) than the order-preserving, random-accessible technique [19] (which also supports transparent mesh access and high cache utilization), while our method is multiresolution and supports selective decompression.

References

[1] P. Alliez and C. Gotsman. Recent advances in compression of 3D meshes. In *Advances in Multiresolution for Geometric Modelling*, pages 3–26, 2005. Springer-Verlag.

[2] K. Cai, Y. Liu, W. Wang, H. Sun, and E. Wu. Progressive out-of-core compression based on multi-level adaptive octree. In *Proc. ACM Virtual Reality Continuum and its Applications*, pages 83–88, 2006.

Figure 6: Selective decompression (p = 8). Datasets (left to right): David and St. Matthew. Each dataset from left to right: uniform LOD (global and zoom-in views), and varying LOD (zoom-in view).

[3] S. Choe, J. Kim, H. Lee, S. Lee, and H.-P. Seidel. Mesh compression with random accessibility. In *Proc. Israel-Korea Bi-National Conference*, pages 81–86, 2004.

[4] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Batched multi triangulation. In *Proc. IEEE Visualization*, pages 207–214, 2005.

[5] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External memory management and simplification of huge meshes. In *IEEE Transactions on Visualization and Computer Graphics*, 2003.

[6] P.-M. Gandoin and O. Devillers. Progressive lossless compression of arbitrary simplicial complexes. In *ACM SIGGRAPH 2003 Proceedings*, pages 372–379, 2002.

[7] E. Gobbetti, F. Marton, P. Cignoni, M. Di Benedetto, and F. Ganovelli. C-BDAM—compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum*, 25(3), 2006. Special Issue for Eurographics '06.

[8] J. Ho, K. Lee, and D. Kriegman. Compressing large polygonal models. In *Proc. Visualization*, pages 357–362, 2001.

[9] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. In *Proc. SIGGRAPH 2003*.

[10] M. Isenburg and P. Lindstrom. Streaming meshes. In *Proc. IEEE Visualization*, pages 231–238, 2005.

[11] M. Isenburg, P. Lindstrom, and J. Snoeyink. Streaming compression of triangle meshes. In *Proc. Symposium on Geometry Processing*, pages 111–118, 2005.

[12] J. Kim, S. Choe, and S. Lee. Multiresolution random accessible mesh compression. *Computer Graphics Forum*, 25(3), 2006. Special Issue for Eurographics '06.

[13] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginztion, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital Michelangelo project: 3D scanning of large statues. In *SIGGRAPH 2000, Computer Graphics Proceedings*, pages 131–144, July 2000.

[14] P. Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *Proc. Sympos. Interactive 3D Graphics*, pages 93–102, 2003.

[15] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan-Kaufmann, 2002.

[16] J. Peng and C.-C. J. Kuo. Geometry-guided progressive lossless 3d mesh coding with octree (OT) decomposition. *ACM Trans. Comput. Graph.*, 24(3):609–616, 2005. Special Issue for SIGGRAPH 2005.

[17] C. Silva, Y.-J. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics, 2002. Tutorial Course Notes, IEEE Visualization 2002. http://cis.poly.edu/chiang/Vis02-tutorial4.pdf.

[18] S.-E. Yoon and P. Lindstrom. Mesh layouts for block-based caches. *IEEE Trans. Vis. Comput. Graph.*, 12(5):1213–1220, 2006. Special Issue for Visualization '06.

[19] S.-E. Yoon and P. Lindstrom. Random-accessible compressed triangle meshes. *IEEE. Trans. Vis. Comput. Graph.*, 13(6):1536–1543, 2007. Special Issue for Vis '07.