In Proceedings SPIE Conference on Visualization and Data Analysis, pages 225–235, 2003.

LOD-based Clustering Techniques for Efficient Large-scale Terrain Storage and Visualization

Xiaohong Bao and Renato Pajarola

Department of Information and Computer Science
University of California Irvine, CA 92697-3425

ABSTRACT

Large multi-resolution terrain data sets are usually stored out-of-core. To visualize terrain data at interactive frame rates, the data needs to be organized on disk, loaded into main memory part by part, then rendered efficiently. Many main-memory algorithms have been proposed for efficient vertex selection and mesh construction. Organization of terrain data on disk is quite difficult because the error, the triangulation dependency and the spatial location of each vertex all need to be considered. Previous terrain clustering algorithms did not consider the per-vertex approximation error of individual terrain data sets. Therefore, the vertex sequences on disk are exactly the same for any terrain. In this paper, we propose a novel clustering algorithm which introduces the level-of-detail (LOD) information to terrain data organization to map multi-resolution terrain data to external memory. In our approach the LOD parameters of the terrain elevation points are reflected during clustering. The experiments show that dynamic loading and paging of terrain data at varying LOD is very efficient and minimizes page faults. Additionally, the preprocessing of this algorithm is very fast and works from out-of-core.

Keywords: terrain data visualization, level-of-detail, data clustering, large-scale terrain data

1. INTRODUCTION

Interactive visualization of very large scale terrain data in scientific visualization, Geographic Information System (GIS), simulation and training applications is a hard problem. Because the grid digital terrain elevation models are not only too large to be rendered in real-time but also exceed physical main memory capacity, traditional in-memory multi-resolution triangulation and rendering techniques do not provide a sufficient solution. Algorithms relying only on virtual memory and the operating system's paging mechanism do not sufficiently take into account spatial as well as level-of-detail (LOD) relations in a multi-resolution triangulation framework. For rendering large terrain from out-of-core, the multi-resolution data structure and rendering algorithm themselves must provide an efficient paging of LOD data from disk.

As main contributions of this paper we propose novel clustering algorithms and data structures to map terrain data to secondary memory (disk blocks) such that dynamic loading/paging of elevation data is very efficient and minimizes the number of page faults (I/Os). The multi-resolution triangulation framework we use is a hierarchical quadtree based terrain triangulation method.

The paper is organized as follows: related work is discussed in Section 2, in Section 3 our clustering techniques are introduced, the physical data file storage structure is given in Section 4, in Section 5 we present experimental results, and Section 6 concludes the paper.

2. RELATED WORK

In this section, we describe related work in the area of multi-resolution triangulation and visualization from external memory.

Further author information: (Send correspondence to Xiaohong Bao)
Xiaohong Bao: E-mail: xbao@ics.uci.edu
Renato Pajarola: E-mail: pajarola@ics.uci.edu

2.1. Multi-resolution triangulation

Many mesh simplification and multi-resolution triangulation methods have been developed over the last decade. We refer to the literature for overviews on general mesh simplification and multi-resolution modeling[1–3] and only point out closely related multi-resolution terrain triangulation methods here.

For grid-digital terrain data sets hierarchical quadtree[4,5] or bin-tree[6–9] based multi-resolution triangulation methods have shown to be exceptionally efficient. These methods all define the same class of triangulations on a regular rectilinear grid of points but differ in algorithms, data structures and error metrics used to efficiently generate and render an adaptively triangulated terrain surface. We adopt the notion of a quadtree based triangulation[5] that uses the dependency relation[4] to guarantee crack-free conforming LOD triangulations. Figure 1 shows the basic recursive triangulation of the quadtree hierarchy.

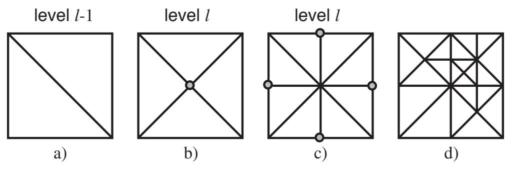

Figure 1. Two-stage subdivision of the hierarchical quadtree triangulation shown in a), b) and c). An adaptive conforming triangulation example shown in d).

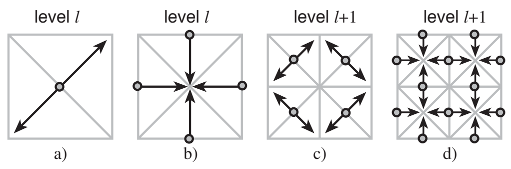

Figure 2. Parent-child dependency shown in a) and c), neighboring vertex dependency shown in b) and d).

Figure 2 shows the dependency relations that have to be satisfied when generating an adaptive triangulation. If a vertex is selected for a particular LOD triangulation then also the vertices pointed to by the dependency graph have to be selected to guarantee a conforming triangulation. Note that these dependencies always propagate from finer to coarse grid resolutions.

The object space error metric δ used in this paper is the vertical distance between corresponding vertices in the original and the simplified terrain surface. For view-dependent triangulation and rendering an image-space error metric ρ is used which is a perspective projection of δ.

2.2. Visualization from external memory

While efficient out-of-core algorithms and data structures have been studied in the literature[10,11] and applied to some extend in scientific visualization (see special issue journal[12]), only little work has been done for multi-resolution terrain rendering from external memory. In[5] a multi-resolution terrain quadtree is maintained in an object-oriented database for efficient out-of-core access, however, the physical clustering is mostly left to the database system. In[13] the terrain data is partitioned and clustered into square tiles without taking any LOD information into account. Furthermore, this approach does not cluster data into fixed size disk blocks. In[14] the multi-resolution terrain triangulation hierarchy is linearized into an array and a memory-mapped file mechanism (supported by the operating system) is used to provide efficient out-of-core access. The main drawbacks are that the terrain data is only clustered on disk with respect to the linearization of the triangulation hierarchy and that the storage cost is comparatively high.

3. CLUSTERING ALGORITHMS

In this section, we first present overall terrain data query process, then propose a clustering algorithm with its optimizations.

3.1. Query Process

The process of querying the elevation data in this paper is as follows (see Figure 3, 4): When the view-dependent parameters, such as viewer position, screen space error, etc., are given, the query process will be started with the quad-tree root n_0. For each node to be checked, we will test (i) whether the area the node represents is visible; (ii) whether the screen space error of the node is more than or equal to the screen space error threshold. If both (i) and (ii) are met, the center vertex of the node will be selected (see Figure 5). Then the other vertices of this node, named as west, south, east and north, will be checked in the same way. When the selected vertices are determined, all vertices on which the selected vertices depend will be selected mandatorily. Figure 2 shows the dependency relations that have to be satisfied when generating an adaptive triangulation. If a vertex is selected for a particular LOD triangulation then also the vertices pointed to by the dependency graph have to be selected to guarantee a conforming triangulation. Note that these dependencies always propagate from finer to coarse grid resolutions.

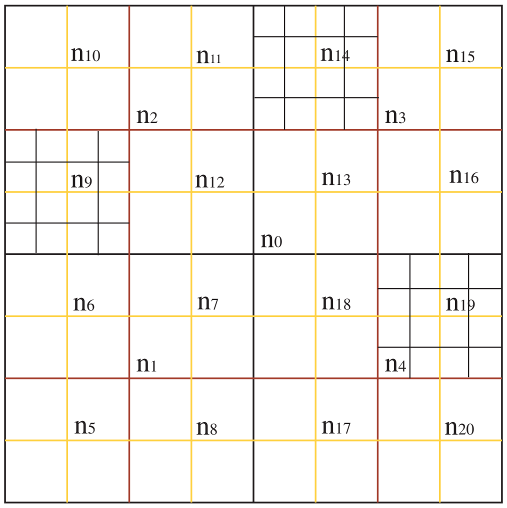

Figure 3. The grids of the elevation data. They are represented with a quad-tree structure in Figure 4. The vertices labeled here are the center vertices of quad-tree nodes. See Figure 5 for the quad-tree node structure.

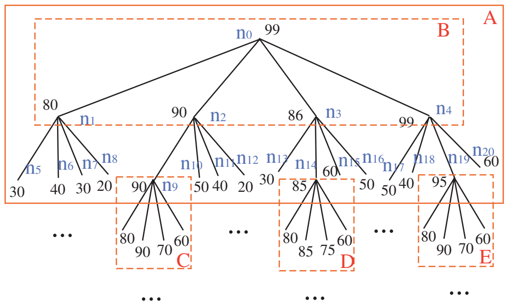

Figure 4. The quad-tree representation of the elevation data in Figure 3. Clustering of quadtree nodes with associated object space error δ, and page capacity of 21 nodes. Set *A* contains 21 nodes representing a complete three-level subtree of the quadtree hierarchy. Sets *B*, *C*, *D* and *E* together represent a more flexible clustering of 20 nodes.

If the node is selected, then all of its four children will be checked in the same way. The whole query process will be ended when there isn't any node in the quad-tree that meets (i) and (ii) together. The query process of the other rendering algorithms may not be exactly as same as the above. But condition (i) and (ii) are exactly the same.

From the above, we can see that in interactive terrain visualization, whether or not a vertex or triangle is rendered not only depends on its spatial location with respect to the current view frustum (*space constraint*) but also is determined by its approximation error and a given image- or object-space error threshold (*LOD constraint*). Considering the adaptive triangle mesh refinement, it is very likely that not all vertices within a query window (i.e. the view frustum) are required for rendering. For view-dependent terrain triangulation and rendering only vertices with a projected image-space error ρ larger than a user defined threshold are needed. To achieve a conforming triangulation additional vertices may have to be selected (triangulation *dependency constraint*).

If we want to reduce terrain access time from external memory we must avoid loading too many vertices which are not used for rendering. To satisfy the space constraint terrain data can be organized in a spatial index structure. However, within the spatial region of interest vertices with projected image-space error ρ below the given threshold should not be loaded from disk. Previously this was only addressed by ordering terrain data by level within the hierarchy. Unfortunately the error of a vertex cannot be expressed by its level information precisely. Furthermore, ρ is view-dependent and cannot be used for clustering vertices. However, ρ is strongly related to the object space approximation error δ which can be precomputed. The LOD constraint can thus be satisfied partially by grouping vertices on disk with respect to δ.

As shown in Figure 4, suppose each page can hold 21 quadtree nodes. We can cluster vertices in set *A*, a small balanced sub-quadtree, into one page (tiling strategy). Thus the error variation among vertices is ignored. Another way is to cluster sets *B*, *C*, *D*, *E* into one page of the same size. This scheme not only aims at spatial clustering but also limits the error variation within one page. Accessing all vertices with error greater than 70, the first approach loads 8 pages (page of *A* and 7 more pages just below A) while the second method only loads one page (containing the relevant sets *B*, *C*, *D*, *E*).

Based on the above observations we propose clustering techniques for quadtree based multiresolution terrain triangulation incorporating spatial location, δ values and dependency relation to minimize disk access times and page faults. The object space error metric δ used in this paper is the vertical distance between corresponding vertices in the original and the simplified terrain surface. For view-dependent triangulation and rendering an image-space error metric ρ is used which is a perspective projection of δ.

3.2. Basic clustering technique (C_{basic})

Let us first define some notations that we will use in the remainder of the paper.

Node is a quadtree node storing five vertices (*Center*, *West*, *South*, *East* and *North*) as shown in Figure 5.

Evenness E is a measure of variation of object space error δ among a set of nodes or vertices.

Primitive element (PE) is a set of nodes with similar object space errors, satisfying some evenness threshold. Each PE is a small complete quadtree, and for storage efficiency we define the minimal PE to be a two-level quadtree consisting of five nodes. The size of PE is flexible and is determined by the Evenness.

Linker is the pointer from a parent PE's leaf node to the child PE's root node (see also Figure 7). *Intra-linker* connect two PEs within the same page, and *inter-linker* connect PEs between different pages.

Page is virtual page in this paper. Its size could be different from or equal to the real physical page size.

Page height is the total number of pages to be visited from the root to load a set of nodes or vertices.

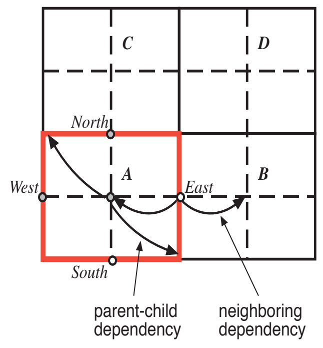

Figure 5. A, B, C, and D are quadtree nodes. Each node is associated with 5 vertices: *Center*, *West*, *South*, *East* and *North*

Intuitively, given an evenness threshold ε we can partition the complete terrain quadtree into a set of PEs and map this set to secondary memory pages. However, this mapping cannot be done arbitrarily. The idea of this mapping is to minimize the page height as well as the total page number for the entire data set. The following issues have to be considered:

(i). What is the optimal size of a PE? We have defined the lower bound above (minimal PE), The upper bound is the page size.

(ii). How can the page height be minimized for loading a set of vertices in a given region at a given LOD?

(iii). How can the terrain quadtree be mapped to a minimum number of pages?

Studies have shown that optimizing (ii) and (iii) is NP-complete even if all PEs have the same size. It can be reduced to the bin-packing problem easily.

To address point (i) we grow PEs according to an evenness threshold ε. Starting with a minimal PE we grow level by level until the evenness E of the leaf nodes of the current PE exceeds ε or the size of the PE exceeds the current page size. Based on the actual traversal of the multiresolution terrain quadtree we introduce the following heuristic for point (ii). If a node is visited and selected for triangulation its children and neighbors will have the highest possibility to be selected too. Therefore, we should merge PEs which are close in space and LOD into the same page. To address point (iii), if the PE stored in a page cannot be grown anymore and the remaining free page memory is larger than a minimal PE, we start a new PE in the same page. This strategy guarantees that the wasted memory per page is not larger than the size of a minimal PE (a negligible fraction for reasonable page sizes).

```
Generate-Primitive-Element(quad-tree t)
{
  if (there is no room in the current page for a new primitive element)
    return (the current page is full);
  Node n = Get-Primitive-Element-Root(t) ; //start a new PE.
  if ($n = NULL$)
    return (the whole quad-tree is finished);
  Construct a minimum PE c rooted at n;
  store $n's$ unprocessed siblings into the priority queue $Q_{page}$; //the potential roots for the next new PEs.
  while (Check-Next-Level(t, c))
    //the current primitive element is growing
    expand the PE c to the next level;
  store $c's$ children into the priority queue $Q_{page}$; //the potential roots for the next new PEs.
  return c ;
}
```

```
Get-Primitive-Element-Root(quad-tree t)
{
  if (the priority queue $Q_{page}$ is not empty)
    return (dequeue($Q_{page}$)) ;
  If there is(are) node(s)in t which is not processed
    return the first node searched by Depth-first-search;
  else
    return NULL;
}
```

```
Check-Next-Level(quad-tree t, PE c)
{
  if (the current level of c is leaf level)
    return false ; //stop the current PE.
  if (there is no space in current page for the next level)
    return false ; //stop the current PE.
  if (the next level is leaf level)//load leaf level anyway.
    return true ;
  if (evenness of the next level of c meets requirement)
    return true ; //grow to the next level.
  return false ; //stop the current PE
}
```

Figure 6. Pseudo-code for generating Primitive Elements

Each time before generating a new PE, the current disk page is checked whether it has enough room for a new PE or not. If not, the current page is finished and a new empty page becomes the current page to be filled. A new minimal PE is generated starting at an unprocessed node from a priority queue, or if it is empty, from the set of unprocessed quadtree nodes. Siblings of this PE root node are pushed into the priority queue Q_{page}. The current PE is grown level by level until the current page is full, the leaf level of the quadtree is reached, or the evenness threshold ε is exceeded. Upon completion of a PE its child nodes are pushed into Q_{page}.

The priority queue Q_{page} stores the potential root nodes for new PEs within the same page. Q_{page} is ordered based on the nodes' object space errors in relation to the current page's first PE. This strategy assures that PEs in the same page are not only close in space but have also little difference in object space error. Upon completion of a page this priority queue is emptied.

Furthermore, to preserve memory locality for efficient retrieval it is desirable that consecutive pages on disk store vertices and nodes which are closely related in space and LOD. We implement this by top-down depth-first traversal of the terrain quadtree to get the root node of the first PE of each page. Another advantage of the depth-first traversal is that we do not need to hold the entire quadtree in main memory. Instead, only the nodes located along the same path from the root node to the current page are maintained in main memory, making this a real out-of-core pre-process.

Finally, let us define the evenness E used above for PE generation. We define absolute E_{abs} and relative evenness E_{rlt} as follows:

$$E_{abs} = \frac{(\delta_{max} - \delta_{min})}{\delta_{max}} \quad (1)$$

$$E_{rlt} = (\delta_{node} - \delta_{child}) - (\delta_{root} - \delta_{node}) \quad (2)$$

For a PE, δ_{max} and δ_{min} are the max/min object space errors of all unprocessed nodes just below of the PE. δ_{node} is the object space error of one node below the current PE, and δ_{child} is the min object space error of the four children of this node. δ_{root} is the root node object space error of the current PE.

E_{abs} measures the evenness among the nodes of the next level below the current PE. If E_{abs} is equal to or less than a given threshold ε then the current PE is grown by one more level. The relative evenness E_{rlt} measures whether a node in the next level is closer to its children or closer to the root of the current PE in terms of object space error. If E_{rlt} of all nodes just below the current PE is equal to or less than zero then the current PE is grown by one more level.

3.3. Optimized clustering technique (C_{opt})

From the experiments we observed that the triangulation dependency constraints are also very important in selecting vertices. The *parent-child dependency* enforces that when a vertex from a child node is selected also vertices from the parent node are selected, and this dependency is accounted for in the above clustering technique. However, the *neighboring dependency* that propagates vertex selection to sibling nodes is not incorporated. We introduce a subtle change in the clustering strategy here to improve clustering by grouping sibling nodes.

As shown in Figure 7 a minimal PE now includes four sibling nodes, each being the root of a small two-level quadtree. Growing of PEs with this optimized clustering method C_{opt} is similar to the basic method C_{basic} discussed above, all child nodes of the current leaf nodes of the PE are added level by level.

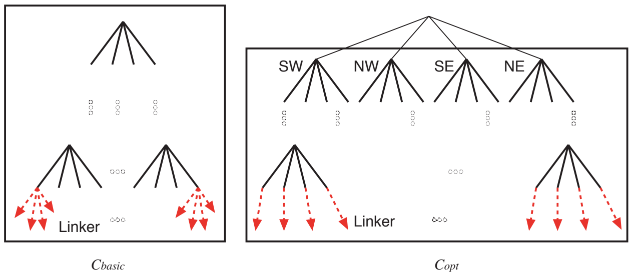

Figure 7. Logical clustering structure of similar size PEs according to strategies C_{basic} and C_{opt}. In C_{basic} a PE is a small complete quadtree with n levels with each leaf node having 4 linkers pointing to the respective child PEs. In C_{opt} a PE contains 4 sub-quadtrees of $n-1$ levels of sibling nodes with each leaf node having only 1 linker to the PE containing its 4 children.

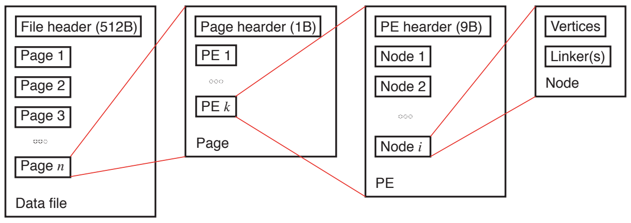

Figure 8. Logical storage layout of data file. A Node contains at least the vertices *West*, *Center* and *North* (and also *South, East* for boundary nodes). PEs contain 4 (C_{basic}) or 1 (C_{opt}) linker elements.

There are two advantages to the C_{opt} clustering strategy. First, it reduces the number of linkers between PEs to 1/4, thus reducing storage cost significantly. Second, in 50% of the cases the neighboring dependency relation points to within the same PE, thus reducing I/O cost for loading PEs to resolve triangulation dependencies.

4. STORAGE STRUCTURE

As shown in the Figure 7, each PE contains one (C_{basic} method) or four (C_{opt} strategy) small complete quadtrees. Each PE stores information on its spatial location (its SW corner), number of quadtree levels it covers, and boundary type (touches none, the East, South, or both boundaries). Furthermore, a PE stores the maximum object space error and a bounding box of all its descendant vertices. This strategy helps to save a lot of disk space while ensuring the traingle mesh accuracy. The reason is that we amplify the bounding box of each descentant slightly. On the other hand, this amplification is not too much, because the variation is bounded by the evenness variation of the PE. Otherwise many unnecesary vertices will be introduced into triangle mesh. Conservative bounding boxes of nodes are dynamically computed at rendering time based on their position in the quadtree within a PE (only geographical extent, elevation range is kept constant for all nodes within the same PE due to the small variation).

Each node in the leaf level of a PE contains 4 linker elements for C_{basic}, respectively 1 for C_{opt}. The leaf nodes of the entire terrain quadtree do not store any linker pointers. For I/O efficiency, the object space error and a bounding box of the child PE are both stored with the *inter-linker* element in the parent PE. Therefore, pages are never loaded in vain only to check for LOD or visibility. On the other hand, the *intra-linkers* only store a pointer to the child PE because accessing the child PE within the same page does not cause any additional I/O cost.

The nodes of each PE are mapped to the disk page ordered by level. Considering redundancy, only three vertices (*Center*, *West* and *North*) of each node are actually stored with the node itself (see also Figure 5). The other vertices *East* and *South* are stored in sibling nodes. Only at the boundary of the terrain data set we actually store all vertices with a node to avoid degenerate nodes along the boundary. Since selection of any non-center vertex causes both adjacent nodes to be loaded due to dependency constraints as shown in Figures 2 and 5, this storage layout will eliminate any vertex storage redundancy and does not increase I/O cost. For example, in Figure 5 if the *East* vertex V of node A is to be loaded then also the *Center* vertex of node B is required and thus node B is loaded which actually stores the initially requested vertex V (as its *West* vertex).

Each vertex only stores the terrain elevation value and its object space error δ.

The entire data file consists of a file header and a set of disk pages as shown in Figure 8. The file header contains the following information: the page size, the total number of quadtree levels of the entire terrain data set, the resolution of the elevation values, the geographical range of the terrain data, the elevation values of the bounding box corners, the information about the top-level quadtree root node, and the four pointers to the child PEs of the root node for C_{basic} (or one pointer for C_{opt}). Each page stores the number of PEs as the only meta information.

5. PERFORMANCE EVALUATION

The main objectives of the conducted experiments are to show the effectiveness of our clustering techniques. For this we implemented a simple terrain viewer that simulates a series of user operations such as changing the LOD parameter or viewing area. The viewer operates on our external memory terrain data structure and dynamically loads elevation data from disk as required by the LOD or viewing parameters. While not optimized for speed, the triangulation and rendering algorithm incorporates simple view-dependent vertex selection as outlined in the following section.

For comparison, we also implemented the quadtree-based indexing scheme and file mapping of[14] (with embedded *white quadtree*). Additionally, we implemented a hierarchical tile partitioning with tiles of $(2^k+1)\times(2^k+1)$ vertices and $(2^k)^2$ child nodes similar to the VGIS approach.[15]

The following performance aspects are studied:

(i). Final file sizes of entire data set;

(ii). Number of page faults (total I/O cost);

(iii). Time of loading selected pages from external memory to main memory;

(iv). Vertex utility rate, which is the ratio of the number of selected and rendered vertices to the number of the total loaded vertices.

5.1. Experimental environment

The hardware configuration is a one-processor 1.5GHz Pentium IV PC with the Microsoft Windows 2000 operating system and 1.5GB of main memory. The source terrain data used for our statistical experiments is $(2^{13}+1)\times(2^{13}+1)$ (67M vertices).

Elevation data and object space errors δ are 4-byte values, and the bounding box occupies 8 bytes. In our implementation of a tiled terrain partitioning and the simulation of the approach presented in[14] each vertex occupies 16 bytes (compared to 20 bytes in[14]).

The basic algorithm we use for terrain mesh generation and rendering is the recursive top-down vertex selection algorithm from.[5] Our implementation is not optimized for rendering speed but rather focused on efficient I/O and dynamic terrain loading from external memory. The viewer application incorporates view-dependent vertex selection that includes view-frustum culling according to the bounding box information of quadtree nodes, and an screen projection error threshold τ. From the geometric approximation error δ, the image-space error is computed by

$$\rho = \frac{\delta}{\|p-e\|_2}, \tag{3}$$

with p being the vertex position and the viewpoint position e. For back-tracking we use $\rho = \frac{\delta}{\|p-e\|_2 - r}$ at *Center* vertices of nodes with bounding sphere radius r.

We implemented our approaches with clustering strategies C_{basic} and C_{opt} as discussed in Section 3. They are represented by PE1 and PE2 respectively in the following figures. With *LP* we denote the embedded quadtree layout and indexing proposed in,[14] and *Tile* denotes a regular hierarchical terrain partitioning. All approaches are implemented in the same framework using the same main memory rendering algorithm and only differ in the indexing and physical data layout on disk. We believe this provides a fair comparison of page faults and other I/O measures such as data locality or vertex utility rate.

5.2. Experimental results

5.2.1. Data file size

In Table 1 we compare the file sizes resulting from different partitioning and clustering techniques with respect to different page sizes. For the hierarchical Tile approach the file size greatly depends on the page size because the tile size is limited by the disk page size, and because the occupancy rate is related to the combination of page and tile size. A high occupancy rate of 90.3% can only be achieved with a large page size of 5K that stores a 17×17 tile per page. The file size of the LP approach is independent of the page size and only depends on the size of vertex elements (16 bytes) and the occupancy rate. In comparison, the approach in[13] requires 50 bytes per vertex (the 70 bytes reported in the paper include some additional information not used in here).

The file sizes of our clustering techniques and storage layout depend to some (minor) extent on the actual page size. That is because of variations in the sizes of PEs and the number of linkers. Furthermore, since the wasted memory per page is at most the size of a minimal PE the occupancy rate is very high even for small pages and increases with page size.

Page size(bytes)	LP	Tile	C_{basic}	C_{opt}
1024	1.33GB	N/A	0.711GB	0.518GB
2048	1.33GB	2.00GB	0.662GB	0.5GB
5120	1.33GB	1.25GB	0.606GB	0.576GB

Table 1. File sizes of different clustering approaches for varying page sizes. Our clustering techniques C_{basic} and C_{opt} used an absolute evenness threshold of $\varepsilon = 0.2$.

From Table 1 we can observe that the file sizes of our approach are significantly smaller than the other two approaches despite some added extra information such as linker elements. Besides the high occupancy rate and compact representation of PE, the main reason for the space savings is that bounding box information is only recorded for each PE rather than for every single vertex. As mentioned in Section 4 bounding boxes of nodes are dynamically computed from the PE bounding box only at run-time for efficient back-tracking of the top-down vertex selection algorithm. Note that the error introduced by this conservative bounding box measure is very little as shown below.

First, for the same screen projection error threshold τ more triangles are generated by our techniques because of the conservative bounding box computation. Second, the ratio of extra triangles is no more than 2.5%. Thus the significant space savings of 50% or more come only at the expense of a very slight increase of the number of rendered triangles per frame.

5.2.2. Static I/O

Figure 9 shows the number of page faults for a static view frustum under different screen projection error thresholds τ. The viewpoint is fixed at 20000 meters altitude with a FOV of 45°. This test measures how many pages have to be loaded from external memory to generate an adaptive triangulation satisfying the image-space tolerance τ. Our C_{opt} clustering strategy achieves best results with fewer page faults for any threshold τ and saves almost 50% page faults when τ is small, compared to LP. The LP approach achieves similar results as C_{basic} for larger threshold values $\tau > 0.05$. But C_{basic} performs much better than LP for small tolerances $\tau < 0.05$.

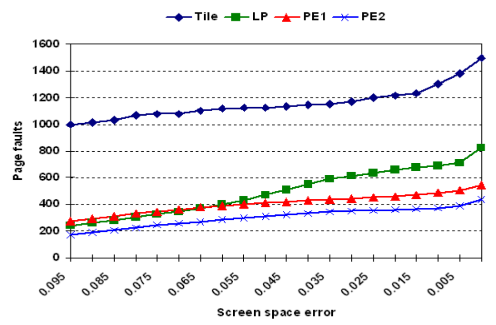

Figure 9. Page faults for different screen projection error thresholds τ to render the fixed view.

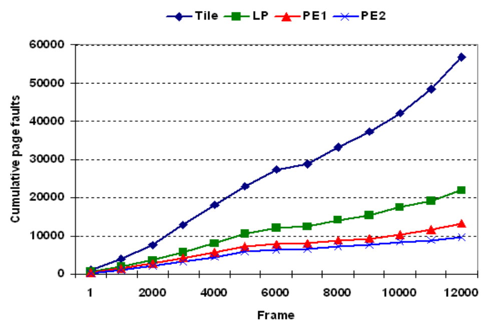

Figure 10. Accumulated number of page faults for a fly-through test with 12000 frames and $\tau = 0.015$.

5.2.3. Dynamic I/O

In this section we report tests of dynamic access to the terrain data based on a fly-through sequence along a route consisting of 12000 rendered frames in total. The height of the moving viewpoint is 20000 meters, with a FOV of 45°.

Figure 10 shows the cumulative number of page faults for each of the 12000 frames of a fly-through test. We can observe that also for continuously changing viewpoints our clustering techniques show constantly better page fault performance and C_{opt} has about half as many page faults as LP.

Figure 11 shows the cumulative page faults for different image-space error tolerances τ. It can be seen that the relative advantages of our clustering techniques are largely maintained for varying thresholds, only for large values of τ the LP approach reaches the same or better efficiency as C_{basic}.

In Figure 12 we measured the loading times of pages from external memory for the different clustering techniques for a fly-through test. Assuming that fixed sized pages require constant loading time it is to be expected and shown in Figure 12 that the cumulative loading time statistic is strongly related to the cumulative page faults of Figure 10. We can observe that C_{opt} and also C_{basic} are both more than two times faster than the LP technique. Compared to the page faults shown in Figure 10, this improved I/O efficiency is achieved by a good physical memory locality of the elevation data on disk. From a different viewpoint we can say that C_{basic} loads 13274 pages within 10 seconds, and C_{opt} loads 9677 pages within 7.9 seconds. In contrast, the LP method only loads 7200 pages in 10 seconds or 5473 pages in 7.9 seconds.

From Figure 13 we observe that utility rate of total loading vertices by using C_{opt} or C_{basic} is higher than that of others when $\tau < 0.04$. After that, the utility rate of LP is slightly better than that of C_{opt} or C_{basic}.

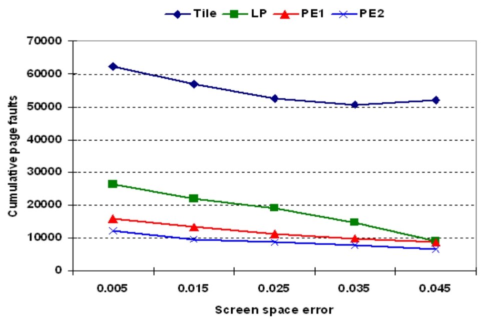

Figure 11. Accumulated number of page faults for different screen projection thresholds τ.

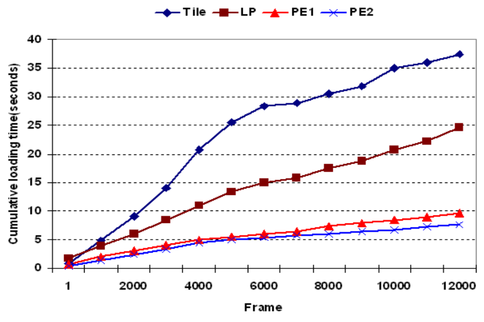

Figure 12. Accumulated loading times for a fly-through test with 12000 frames and $\tau = 0.015$.

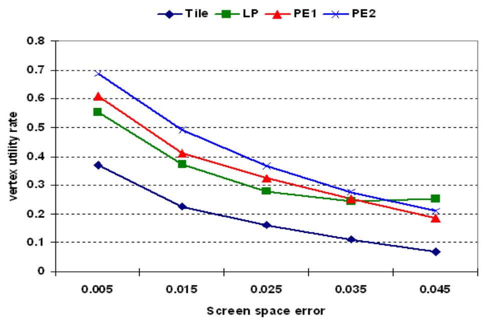

Figure 13. Utility rate of loaded vertices during a fly-through test for different screen projection error thresholds τ.

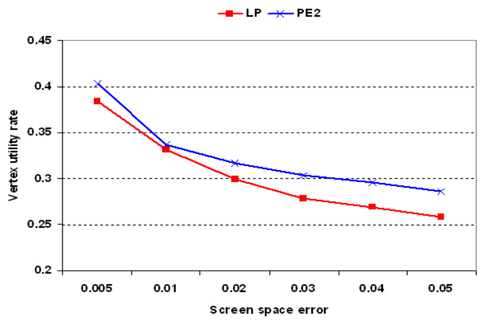

Figure 14. utility rate of loaded vertices during a fly-over test for different screen space error threshold τ (*With extra spatial constraint to C_{opt}*).

To check the pure performance of our clustering strategy, we mandatorily limit the vertex number per page with C_{opt} not exceeding that with the Lindstrom's, even though with C_{opt}, each page is able to hold more vertices, then the page faults could be reduced further. The moving viewpoint is at 1000 meters altitude with a FOV of 45°. And the view direction is close to parallelling to the X-Y plane (ground). Figure 14 shows that the vertex utility rate with C_{opt} and the Lindstrom's algorithm with the different screen space errors under the above constraint along the same fly-over route. On average, the vertex utility rate with C_{opt} is still around 3% higher. And up to 15% vertex loading can be avoided with this strategy.

6. CONCLUSION

In this paper we present novel clustering algorithms (C_{basic} and C_{opt}) and data structures to support interactive large scale terrain visualization from out-of-core. Our clustering techniques allow efficient paging of terrain data at different LOD from disk with minimal number of page faults since our clustering techniques are more closely related to space and LOD constrained access patterns. Furthermore, with our clustering techniques, the data file has better data locality. It helps to reduce the loading time more. In most cases, by using C_{basic} or C_{opt} approach, the total page faults (I/O) can be reduced to less than 20% (compared to the hierarchical tile structure) or less than 50% (compared to the LP algorithm). And the total data loading time can be reduced to less than 30%. Since C_{basic} and C_{opt} approaches can reduce the total page faults so greatly, more benefits will be gained when the large scale terrain visualization applications are distributed.

REFERENCES

1. P. S. Heckbert and M. Garland, "Survey of polygonal surface simplification algorithms." SIGGRAPH 97 Course Notes 25, 1997.

2. P. Cignoni, C. Montani, and R. Scopigno, "A comparison of mesh simplification algorithms," *Computers & Graphics* **22**(1), pp. 37–54, 1998.
3. D. P. Luebke, "A developer's survey of polygonal simplification algorithms," *IEEE Computer Graphics & Applications* **21**, pp. 24–35, May/June 2001.
4. P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner, "Real-time, continuous level of detail rendering of height fields," in *Proceedings SIGGRAPH 96*, pp. 109–118, ACM SIGGRAPH, 1996.
5. R. Pajarola, "Large scale terrain visualization using the restricted quadtree triangulation," in *Proceedings IEEE Visualization 98*, pp. 19–26,515, 1998.
6. M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein, "Roaming terrain: Real-time optimally adapting meshes," in *Proceedings IEEE Visualization 97*, pp. 81–88, 1997.
7. L. Balmelli, S. Ayer, and M. Vetterli, "Efficient algorithms for embedded rendering of terrain models," in *Proceedings IEEE Int. Conf. on Image Processing ICIP 98*, pp. 914–918, 1998.
8. L. Velho and J. Gomes, "Variable resolution 4-k meshes: Concepts and applications," *Computer Graphics Forum* **19**(4), pp. 195–214, 2000.
9. W. Evans, D. Kirkpatrick, and G. Townsend, "Right-triangulated irregular networks," *Algorithmica* **30**, pp. 264–286, March 2001.
10. L. Arge, *Efficient External-Memory Data Structures and Applications*. PhD thesis, Department of Computer Science, University of Aarhus (Denmark), 1996.
11. J. Abello and J. S. Vitter, *External Memory Algorithms*, American Mathematical Society, Providence, R.I., 1999.
12. K.-L. Ma, "Large-scale data visualization," *IEEE Computer Graphics & Applications* **21**, pp. 22–23, July-August 2001.
13. H. Hoppe, "Smooth view-dependent level-of-detail control and its application to terrain rendering," in *Proceedings IEEE Visualization 98*, pp. 35–42, Computer Society Press, 1998.
14. P. Lindstrom and V. Pascucci, "Visualization of large terrains made easy," in *Proceedings IEEE Visualization 2001*, pp. 363–370, Computer Society Press, 2001.
15. P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, and N. Faust, "An integrated global GIS and visual simulation system," Tech. Rep. GVU Technical Report 97-0, Georgia Tech Research Institute, 1997. http://www.gvu.gatech.edu/gvu/virtual/VGIS/.