EUROGRAPHICS 2003 / P. Brunet and D. Fellner
(Guest Editors)

Volume 22 (2003), Number 3

BDAM – Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization

P. Cignoni[§], F. Ganovelli[§], E. Gobbetti[‡], F. Marton[‡], F. Ponchio[§], R. Scopigno[§]

§ **ISTI CNR** via Moruzzi 1, 56100 Pisa, Italy, *email*: {lastname}@isti.cnr.it
‡ **CRS4**, VI Strada OVEST Z.I. Macchiareddu, C.P. 94, 09010 UTA (CA - Italy) *email*: {lastname}@crs4.it

Abstract

This paper describes an efficient technique for out-of-core rendering and management of large textured terrain surfaces. The technique, called Batched Dynamic Adaptive Meshes (BDAM) , is based on a paired tree structure: a tiled quadtree for texture data and a pair of bintrees of small triangular patches for the geometry. These small patches are TINs and are constructed and optimized off-line with high quality simplification and tristripping algorithms. Hierarchical view frustum culling and view-dependent texture and geometry refinement is performed at each frame through a stateless traversal algorithm. Thanks to the batched CPU/GPU communication model, the proposed technique is not processor intensive and fully harnesses the power of current graphics hardware. Both preprocessing and rendering exploit out-of-core techniques to be fully scalable and to manage large terrain datasets.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture and Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism.

1. Introduction

Interactive visualization of massive textured terrain datasets is a complex and challenging problem: the size of current geometry and texture datasets easily exceeds the capabilities of current hardware. Various dynamic multiresolution models have been proposed to face this problem, usually based on the idea of constructing, on the fly, a coarser adaptively approximated representation of the terrain to be rendered in place of the complete terrain model.

Unfortunately current dynamic multiresolution algorithms are very processor intensive: the extraction of an adequate terrain representation from a multiresolution model and its transmission to the graphics hardware is usually the main bottleneck in terrain visualization. Nowadays, consumer graphics hardware is able to sustain rendering rate of tens of millions of triangles per second, but current multiresolution solutions fall short of reaching such performance. This because the CPU is not able to generate and extract such variable resolution data at the requested rate; moreover these data must be sent to the graphics hardware in the correct format and through a preferential data path. The gap between what could be rendered by the graphics hardware and what we are able to batch to the GPU, is doomed to widen because CPU processing power grows at a much slower rate than GPU's one.

Therefore our goal is to propose a technique that is able to manage massive textured terrain datasets without burdening the CPU and to fully exploit the power of current and future graphics hardware. As highlighted in the short overview of the current solutions for interactive visualization of large terrains (Sec. 2), the techniques based on hierarchy of right triangles are the ones which ensure maximum performance, while TIN based multiresolution solutions reach maximal accuracy for a given triangle count. In this paper we introduce a new data structure that gets the best out of the above approaches. Moreover, our approach efficiently supports the combination of high resolution elevation and texture data in the same framework. Our proposed BDAM [†] technique is based on a paired tree structure: a tiled quadtree for texture data and a pair of bintrees of small triangular patches for the geometry (Sec. 3). These small patches are TINs and are

[†] The sound of a gunshot[16]

constructed and optimized off-line with high quality simplification and tristripping algorithms. A hierarchical view frustum culling and view-dependendent texture/geometry refinement can be performed, at each frame, with a stateless traversal algorithm, which renders a continuous adaptive terrain surface by assembling these small patches (Sec. 4). An out-of-core technique has been designed and tested for constructing BDAMs using a generic high quality simplification algorithm (Sec. 5). The efficiency of the BDAM approach has been succesfully evaluated on a standard terrain benchmark (Sec. 6).

2. Related Work

The problem of rapidly rendering a continuous representation of a given terrain where the resolution adaptively matches with the current viewing position is an active research area. Since first approaches ([8, 21, 13]) many different data structures have been proposed. A comprehensive overview of this subject is beyond the scope of this paper. In the following, we will discuss the approaches that are most closely related to our work. Readers may refer to recent surveys [15, 18] for further details.

From the point of view of the rapid adaptive construction and display of continuous terrain surfaces, two main approaches have been proposed to manage this problem: a) techniques that exploit a regular hierarchical structure to efficiently represent the multiresolution terrain, b) techniques that are based on more general, mainly unconstrained, triangulations.

The most successful examples of the first class of techniques include hierarchies of right triangles (HRT) [5] or longest edge bisection [15], triangle bintree [13, 4], restricted quadtree triangulation [17, 22]. The scheme permits the creation of continuous variable resolution surfaces without having to cope with the gaps created by other regular grid schemes. The main idea shared by of all these approaches is to build a regular multiresolution hierarchy by refinement or by simplification. The refinement approach starts from an isosceles right triangle and proceeds by recursively refining it by bisecting its longest edge and creating two smaller right triangles. In the simplification approach the steps are reversed: given a regular triangulation of a gridded terrain, pairs of right triangles are selectively merged. The regular structure of these operations enables to implicitly encode all the dependencies among the various refinement/simplification operations in a compact and simple way: a simple binary tree together with a smart error tagging of the tree nodes.

The second class of algorithms is based on less constrained triangulations of the terrain (TINs) and includes multiresolution data structures like Multi-Triangulations [21] adaptive merge trees [24], hypertriangulations [1], and the extension of Progressive Meshes [9] to the view-dependent management of terrains [10]. As pointed out and numerically evaluated in [5], TIN outperform right triangles hierarchies in terms of number of triangles / error counts; in other words, using an HRT scheme you need a number of triangles that is much higher (even an order of magnitude) than the one needed by TINs to achieve the same terrain resolution. This is mainly due to the fact that TINs adapt much better to high frequency variations of the terrain. On the other hand, this class of methods requires much more memory and more complicated multiresolution data structures. For this reason, when they are used in real-time environments, they are able to output smaller models in the same frame time, possibly yielding lower quality images.

HRT and TIN techniques also considerably differ in the way they interact with LOD texture management. Very few techniques full decouple texture and geometry LOD management. To our knowledge, the only general approach is the SGI-specific clip-mapping extension[23] and 3DLabs Virtual Textures, which requires, however, special hardware. In general, large scale textures are handled by explicitly partitioning them into tiles and possibly arranging them in a pyramidal structure[3]. Clipping rendered geometry to texture tile domains imposes severe limitations on the geometry refinement subsystem. General TIN approaches are difficult to adapt to this context, and the few systems able to support multiresolution geometry and texture are mostly based on hierarchical techniques.

Our work aims to combine the benefits of TINS and HRT in a single data structure for the efficient management of multiresolution textured terrain data. A first attempt towards this aim was given by Pajarola et al. [19], presenting a technique to build a HRT starting from a TIN terrain. The main idea is to adaptively build a HRT following the TIN data distribution and allowing vertex positions to be not constrained to regular grid positions. Among the other differences, in our proposal the advantages of TINS are much better exploited, because each patch is a completely general triangulation of the corresponding domain.

A common point of all adaptive mesh generation techniques is that they spend a great deal of the rendering time to compute the view-dependent triangulation. For this reason, many authors have proposed techniques to alleviate popping effects due to small triangle counts [2, 10] or to amortize construction costs over multiple frames [13, 4, 9]. Our proposal is, instead, to reduce the per-triangle workload by composing pre-assembled surface patches during run-time. The idea of grouping together sets of triangles in order to alleviate the CPU/GPU bottleneck was also presented in Rustic [20] and in the CABTT [12] data structures. RUSTiC is a extension of the ROAM algorithm in which subtrees of the ROAM bintree are, in a preprocessing phase, statically freezed and saved. The CABTT approach is very similar to RUSTIC, but clusters are dynamically created, cached and reused during rendering. With respect to both CABBT and RUSTIC algorithms our proposal makes explicit the simple edge error

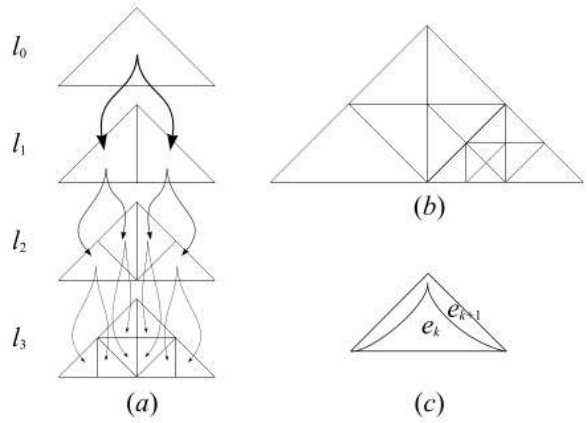


Figure 1: *a) the binary tree representing a Right Triangle Hierarchy, b) an example of a consistent triangulation built by a simple visit of the bintree, c) each triangle of the bintree represents a small mesh patch with error e_{k+1} along the two shortest edges and error e_k elsewhere.*

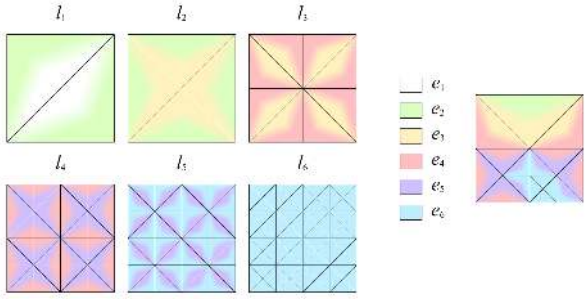


Figure 2: *An example of a BDAM: each triangle represents a terrain patch composed by many triangles. Each error value is marked by a different color; the blending of the color inside each triangle corresponds to the smooth error variation inside each patch.*

property needed for cluster consistency, exploits high quality, fully adaptive triangulation of clusters, cache coherent tri-stripping of clusters for efficient rendering, and multiresolution texturing; finally, it supports out-of-core rendering and construction of huge datasets.

3. Batched Dynamic Adaptive Meshes (BDAM)

Current multiresolution algorithms are designed to use the triangle as the smallest primitive entity. We can synthesize the main idea of our approach as designing a multiresolution approach that uses a more complex primitive: small surface patches (composed of a batch of a few hundreds of triangles). The benefit of this approach is that the per-triangle workload to extract a multiresolution model is highly reduced and small patches can be preprocessed and optimized off line for a more efficient rendering, using, for example, cache coherent triangle strips[11]. In other words, the approach we propose is based on the idea of moving up the grain of multiresolution models from triangles to small contiguous mesh portions.

A hierarchy of right triangles can be coded as a binary tree of triangles (Fig. 1.a); this binary tree representation is the base of ROAM[4] and of many other terrain multiresolution data structures. This is because it can be used to easily extract a consistent set of contiguous triangles which cover a particular region accordin to a given error thresholds (Fig. 1.b). Similarly to the ROAM algorithm, our structure, called Batched Dynamic Adaptive Meshes (*BDAM*), uses a right triangle hierarchy stored as a bintree, to give a high level representation of the data partitioning. On the other hand, we replace single triangles with small mesh patches as the minimal manageable entities to be *batched* (hence the name) to the graphics hardware in the most efficient way. Therefore, each bintree node contains a small chunk (in the range of 256..8k) of contiguous well packed tri-stripped triangles. To ensure the correct matching between triangular patches, we exploit the HTR property that each triangle can correctly connect either to: triangles of its same level; triangles of the next coarser level through the longest edge; and triangles of the next finer level through the two shortest edges. The above property works well for triangles, but, switching from triangles to small patches, the correct connectivity along borders of different simplification level patches is not directly guaranteed. This simple edge error property is exploited, as explained in Sec. 5, to design an out-of-core high quality simplification algorithm that builds each triangular patch so that the error is distributed as shown in figure 1.c: each triangle of the bintree represents a small mesh patch with error e_k inside and error e_{k+1} (the error corresponding to the next more refined level in the bintree) along the two shortest edges. In this way, each mesh composed by a collection of small patches arranged as a correct bintree triangulation still generates a globally correct triangulation.

In Fig. 2 we show an example of these properties. In the upper part of the figure we show the various levels of a HRT and each triangle represents a terrain patch composed by many graphics primitives. Colors correspond to different errors; the blending of the color inside each triangular patch represents the smooth error variation inside each patch as shown in Fig. 1.c. When composing these triangular patches using the HRT consistency rules, the color variation is always smooth: the triangulation of adjacent patches correctly matches.

3.1. The texture quadtree

Terrain textures have dimensions which are typically similar or larger than the corresponding ones of elevation data. Since such a big texture sizes cannot be handled by commodity graphics boards, we need to partition them into chunks before rendering. Total size also generally exceeds typical core memory size, hence the texture management unit has to deal with out-of-core memory handling techniques. These considerations lead to a multiresolution texture management technique similar to the one used for geometry.

For efficiency reasons, textures are however best managed as rectangular tiles, as opposed to the triangular geometric tiles, leading to a tiled texture quadtree instead of the geometry bintree. If the original image covers the same region of the elevation data, each texture quadtree element corresponds to a pair of adjacent geometry bintree elements (Fig. 3), and descending one level in the texture quadtree corresponds to descending two levels in the associated pair of geometry bintrees. This correspondence can be exploited in the preprocessing step to associate object-space representation errors to the quadtree levels, and in the rendering step to implement view-dependent multiresolution texture and geometry extraction in a single top-down refinement strategy.

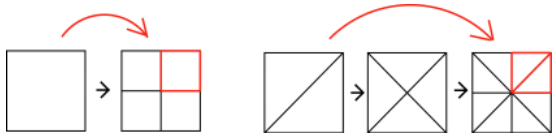


Figure 3: *A texture quadtree element is associated to a pair of adjacent geometry bintree elements. View-dependent refinement is performed using a combined top-down traversal of the texture and geometry trees.*

3.2. Errors and Bounding Volumes

To easily maintain the triangulation coherence we exploit the concept of nested/saturated errors, introduced by Pajarola[17] that allows to extract a correct set of triangular patches with a simple stateless visit of the bintree[17, 14].

Object space embedded error Object-space error is independent from the metric used, and can be computed directly from the finest resolution grid, or incrementally from the patches of the previous level. Once these errors have been computed, a hierarchy of errors (that respect nesting conditions) can be constructed bottom up. Texture errors are computed from texture features, and are embedded in a corresponding hierarchy by using a structure similar to the one used for the geometry errors.

Nested bounding volumes hierarchy Object space errors are view independent, but for the rendering purpose we need a view dependent hierarchy of errors where nesting conditions are still valid. Thus, a tree of nested volumes is also built during the preprocessing, with properties very similar to the two error rules: 1) bounding volume of a patch include all children bounding volumes; 2) two patches adjacent along hypotenuse must share the same bounding volume which encloses both. These bounding volumes are used to compute screen space errors and also for view frustum culling (Fig. 4).

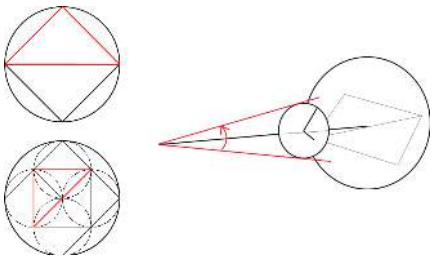


Figure 4: *The nested sphere hierarchy is used for refinement and view culling (left). Screen space error is the quantity that drives view-dependent refinement (right).*

4. Top-down view-dependent refinement and rendering

The goal of the multiresolution rendering component is to efficiently extract and render a textured mesh with a small number of triangles and an associated coarse texture, which should be a good approximation of the original, dense mesh for the given view. The algorithm is based on a combined top-down traversal of the texture and geometry trees, that implicitly guarantees mesh continuity, manages large terrains datasets through out-of-core paging and data layout techniques, and is designed to fully exploit the rendering capabilities of modern graphics accelerators through batched primitive rendering.

4.1. Screen space error

View-dependent refinement is driven by screen space error. Screen space error is derived at run time from a patch bounding volume and its object-space geometry and texture errors; it adopts a monotonic projective transformation to preserve the error nesting conditions. This approach, that supports variable resolution data extraction with a stateless visit of the hierarchy, is similar to [14]. In our case, however, nested errors and bounding volumes are associated to the patch hierarchy, rather than the vertex hierarchy, and we have to combine the geometry error with the texture error. In our current code, we obtain a consistent upper bound on screen space error by measuring the apparent size of a sphere centered at the patch bounding volume point closest to the viewpoint and having radius equal to the maximum between the texture and the geometry object space errors (see figure 4). The refinement condition, once the closest point from the viewpoint is found, requires only one multiplication to check if $error_{sphere} radius > error_{threshold} * distance$.

```
proc refine(V, tex_tree, geo_tree1, geo_tree2)
  B := bounding_volume(geo_tree1)
  if visible(V, B)
    if view_error(V, B, obj_error(tex_tree)) > eps
      for each i in { i_SE, i_SW, i_NW, i_NE }
        refine(V,
          child(tex_tree, i),
          top_grand_child(geo_tree1, geo_tree2, i),
          bottom_grand_child(geo_tree1, geo_tree2, i))
      end for
    else
      bind_texture(tex_tree)
      define_texcoords(tex_tree)
      geo_parent_refine(V, tex_tree, parent(geo_tree1))
      geo_parent_refine(V, tex_tree, parent(geo_tree2))
    end if
  end if
end proc

proc geo_parent_refine(V, tex_tree, geo_tree_parent)
  B := bounding_volume(geo_tree_parent)
  if view_error(V, B, obj_error(geo_tree_parent)) > eps
    geo_refine(V, child(geo_tree, 0))
    geo_refine(V, child(geo_tree, 1))
  else
     enable_clipping(tex_tree)
     geo_render(geo_tree_parent)
     disable_clipping
  end if
end proc

proc geo_refine(V, geo_tree)
  B := bounding_volume(geo_tree)
  if visible(V, B)
    if view_error(V, B, obj_error(geo_tree)) > eps
      geo_refine(V, child(geo_tree, 0))
      geo_refine(V, child(geo_tree, 1))
    else
      geo_render(geo_tree)
    end if
  end if
end proc
```

Figure 5: ***View-dependent refinement.*** *Variable resolution textures and geometry are extracted with a combined stateless visit of texture quadtree and geometry bintrees.*

4.2. Combined texture and geometry traversal

Having defined a criterion for view-dependent refinement, we now summarize the algorithm for top-down selective refinement and on-the-fly textured rendering.

The refinement procedure (Fig. 5) starts at the top-level of the texture and geometry trees and recursively visits the nodes until the screen space texture error becomes acceptable. While descending the texture quadtree, corresponding triangle patches in the two geometry bintree are identified and selected for processing. Once the texture is considered detailed enough, texture refinements stops. At this point, the texture is bound and the OpenGL texture matrix is initialized to define the correct model to texture transformation. Then, the algorithm continues refining the two geometry bintrees until the screen space geometry error becomes acceptable and the associated patch can thus be sent to the graphics pipeline. Each required texture is therefore bound only once, and all the geometry data covered by that square is then drawn, avoiding unnecessary context switches and minimizing host to graphics bandwidth requirement.

Since a one level refinement step in the texture quadtree corresponds to two refinement steps into the geometry bintree, all even geometry levels are skipped during the texture refinement step, therefore possibly missing a correct geometry subdivision. To avoid introducing cracks, after the texture is bound, the algorithm starts geometry refinement from the parent patches of those selected by the texture refinement step, since the error nesting rules ensure that the correct geometry levels cannot be above that level (see figure 6). If parent patches meet the error criterion, they are rendered using clipping planes to restrict their extent to that of the selected texture; otherwise, geometry refinement continues normally, descending in the geometry bintree. In order to load balance the graphics pipeline, clipping geometry outside the current texture domain is done at the pixel level, using fragment kill operations. Rendering twice the same patch at the parent level uses the same number of triangles of the standard solution of forcing a refinement step, but requires half of the graphics memory to store vertex data and it can be implemented in a stateless refinement framework.

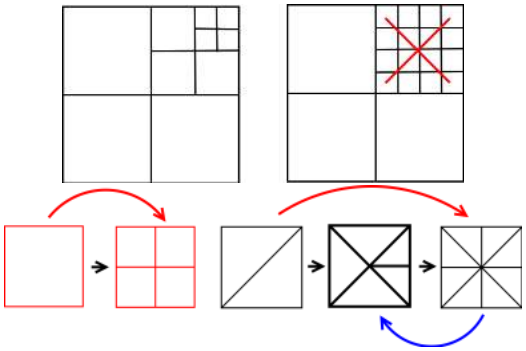


Figure 6: ***Texture continuity.*** *Top: error nesting rules force neighboring texture patches to differ by at most one level. Bottom: to ensure continuity, geometry refinement starts from the even geometry level above the selected texture.*

View frustum culling is easily done as part of the recursive refinement, exploiting the nested bounding volumes. Since a patch bounding volume contains all the geometry of a given subtree, recursion can stop without rendering whenever the bounding volume is detected as invisible. Since the structure is patch based, there is no need to handle the artifacts generated by partially visible triangles typical of vertex based structures [9, 14].

4.3. Memory management

Time-critical rendering of large terrain datasets requires real-time management of huge amounts of data. Moving data

from the storage unit to main memory and to the graphics board is often the major bottleneck.

Host to GPU communication To take advantage of spatial and temporal coherency, it is worth to spend some CPU time to build an optimal rendering representation for each patch, that can be efficiently reused on subsequent frames, instead of using direct rendering each time. This is the only way to harness the power of current graphics architectures, that heavily rely on extensive on board data caching. With BDAM, a memory manager based on a simple LRU strategy explicitly manages graphics board memory. Texture data is used to build OpenGL texture objects, and geometry data is written directly to graphics memory using the OpenGL Vertex Array Range extension. The primitive geometric element is a patch composed of multiple triangles, that is heavily optimized during pre-processing using cache-coherent tristripping. Since we use an indexed representation, the post trasformation-and-lighting cache of current graphics architectures is fully exploited.

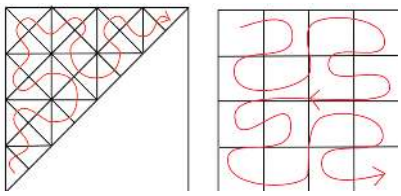


Figure 7: *Texture and Geometry data are stored using space-filling indexing schemes to improve memory coherency.*

Out-of-core paging and data layout Since the BDAM algorithm is designed to work in a stand-alone PC (as opposed to a distributed, network-based solution), we assume that all data is stored on local secondary storage unit. Our approach, similarly to Lindstrom and Pascucci's [14], optimizes the data layout to improve memory coherency and accesses external texture and geometry data through system memory mapping functions. To minimize the number of page faults, data storage order has to reflect traversal order. All data is therefore sorted by level, then by patches. The patch order inside a level is defined through two space filling curves, (one for geometry and one for texture), which achieve good memory locality. Figure 7 illustrates the indexing schemes used for data layout. To improve memory locality further, data files are subdivided into a *structure file* and a *data file*. The structure files contains for each node information on the relative patch error and the location of the data stored into the data file. The data file contains all the information required by the top-down traversal algorithm (error, bounding volume, location of information in the associated data file), while the data file contains the information required for rendering the object (image data in for the textures, connectivity and vertex data for the patches). As we let the operating system manage the external memory (by means of memory mapping) the finite size of the address space introduces a data size limitation. On a 32 bit architecture, 4GB is the address space limit, but operating systems such as Windows or Linux typically reserve less than 2GB for memory mapped segments. Our solution is to overcome this limitation by mapping only a segment of the data file at a time, moving the mapped segment as needed. This only applies to the data files, since structure files are small enough to be always maintained in core.

5. Building a BDAM

In this section we describe how a BDAM can be constructed starting from a regular terrain height field. The technique here presented to build the geometry bintrees is quite general. The simplification algorithm used to generate the various patches could be replaced with any other technique that is able to remove/insert a given number of vertices and to constrain the simplification to avoid removal of some vertices. For example, the approach presented in the next paragraphs could be easily reversed from simplification to refinement and by using the classical greedy Delaunay-insertion algorithm[6].

5.1. High Quality Geometric Simplification

The hierarchical structure described in Section 3 is built bottom-up, level by level. We refer to figure 8 to explain the construction of level l_i from level l_{i+1}. As a first step, all the vertices laying onto triangles' longest edges are marked as non-modifiable, which means that they will still exist in the next level of the hierarchy. As you can see, this marking splits the mesh into a set of square shaped sub-meshes (bold gray lines), each one made of four triangular patches joined along their shortest edges. After a square-shaped sub-mesh is simplified, it is split along a diagonal into two triangular patches. This *splitting diagonal* (lines in black) is taken so that so that each of the two triangular patches corresponds to a triangle at the next level of the bintree hierarchy. The simplification is targeted to always halves the number of vertices of the sub-mesh, thus the size of patches is approximatively constant everywhere in the hierarchy.

In the right part of figure 2 we show an example of a mesh that can be assembled by using the patches contained in the bintree nodes and built following the process described above. Note that each triangular patch has the longest edge of the error (color) of the previous level, hence the error distribution of each patch, shown in figure 2, is respected. The generality of this approach allows to use high quality feature preserving simplification techniques (e.g. quadric based[7] or Delaunay insertion[6]) which produce, with the same number of triangles, a much better terrain approximation than constrained bintree techniques[4].

Placement of the vertices on the border of the quadtree

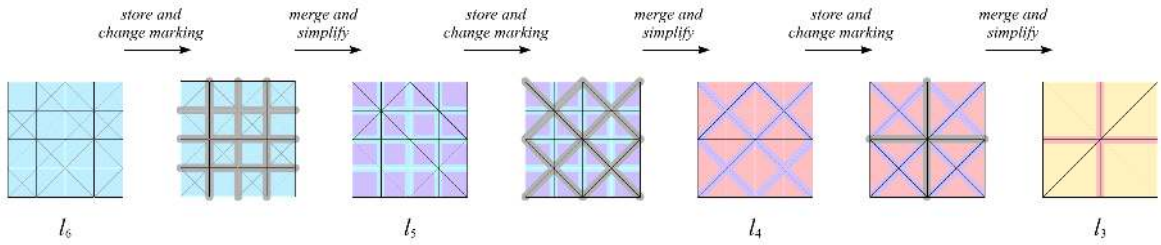


Figure 8: *Construction of a BDAM through a sequence of simplification and marking steps. Each triangle represents a terrain patch composed by many triangles. Colors correspond to different errors as in Fig 2.*

cells needs extra care to ensure compatibility with textures boundaries. Their projection must lie exactly on the boundary of the patch triangular shape resulting from the regular quadtree subdivision of the terrain. This results in a constrain on the simplification of a region: no triangle must cross the splitting diagonal if this diagonal is vertical or horizontal in texture space. This constrain can be dropped when building a BDAM where when texture are not used: the patches resulting from the splitting will have unconstrained zigzag borders instead of straight ones.

During the construction we save each patch into an intermediate format containing vertices, border vertices and faces. Finally we proceed to convert each level into the final format. This consists of reordering all the patches to maximize data locality (as described in section 4.1), building triangle strips and quantizing vertices coordinates to 16 bit, updating errors in order to satisfy the nesting conditions.

5.2. Simplification algorithm

The task of simplifying a surface mesh is a well known problem. We employ an edge collapse simplification driven by the well known quadric error metric [7] with few minor modifications. With respect to the classical formulation our simplification algorithm must respect the constraints on the border vertices and on the diagonal specified in Section 5.1. The preservation of the marked vertices is trivially satisfied by "locking" such vertices and discarding those collapses which would involve them. In the cases in which no new triangle must cross the splitting diagonal, we simply force the vertices laying in the diagonal and involved in an edge collapse to stay in the vertical plane containing the patch's diagonal: this means that the minimization of the quadric error is done only with respect to that plane.

We also introduce two small variations in the process that improve the output mesh quality. The use of *dihedral planes* to build the quadric and a smoothing step at the end of simplification.

Dihedral planes In flat areas, the faces sharing a vertex lay (almost) in the same plane, and in this plane the quadric error value is constant and minimal. This means that the vertex resulting from the collapse of two other vertices could be positioned everywhere in the plane, producing no error. One way to cope with this problem is to add a measure of the quality of the triangles generated by the collapses, such as area and/or aspect ratio. Conversely we prefer to work on the definition of the initial quadric error able to work also in flat areas. We add supplementary planes to build the initial vertex quadrics: for each edge e we add the plane including e and with normal $p = \bar{e} \times n$, where n is the average normal of the two triangles sharing e, as a contribution to the quadric construction of the vertices of e. This contribution is scaled by a factor proportional to the dihedral angle between the two faces sharing the edge e (Fig. 9). In other words, the quadric error associated with a vertex will take account not only of the distance of that vertex from from the planes of the triangles adjacent to it, but also the distance from these dihedral planes. We have simply added a contribute from the lower dimension quadrics computed in the tangent space, which can be easily seen taking a completely flat mesh. In this case the only contribution to the quadrics comes from the dihedral planes, which are all vertical and correspond to the 2D quadric.

Smoothing A final step of Laplacian smoothing is used to improve the triangle quality, with the constraint that a vertex is actually moved into its "smoothed" position if and only if the quadric error associated with the vertex does not worsen over a given threshold.

Evaluating the error The error of the simplified mesh is taken as the maximum vertical difference with the original one. To perform this computation quickly by exploiting graphics hardware, we render under orthographic projection the original and simplified meshes and evaluate the difference among the corresponding depth buffers.

5.3. Out-of-core Texture LOD construction

In the preprocessing step, textures are converted into a hierarchical structure containing the original image at different levels of detail. The quadtree of texture tiles is obtained by a bottom-up filtering process of the original image, optionally followed by a compression to the DXT1 format. To ensure texture continuity, texture tiles at a given level overlap by

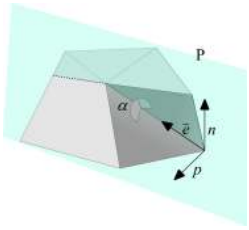


Figure 9: *The plane added by edge e is weighted with the cosine of dihedral angle* α

a one texel border. The object space error of a given texture tile is the size, expressed in original image texels, of the biggest feature that disappears when replacing the original image data with the filtered one. A more pessimistic object space error, used in our current implementation, is just the magnification factor of the given texture tile. As for geometry, error nesting is ensured by a final bottom-up pass that combines all object space errors of neighboring and descendant tiles. The entire process is easy to implement out-of-core, since the filtering and compression process is done independently for each tile, and the connectivity information for error propagation has a negligible size even when explicitly stored.

6. Results

An experimental software library and a terrain rendering application supporting the BDAM technique have been implemented and tested on Linux and Windows NT. The results were collected on a Linux PC with two AMD Athlon MP 1600MHz processors, 2GB RAM, and a NVIDIA GeForce4 Ti4600 graphics board.

The test case discussed in this paper is a terrain dataset over the Puget Sound area in Washington state [‡]. We used a 8,193x8,193 elevation grid with 20 meter horizontal and 0.1 meter vertical resolution. On this terrain, we mapped a 16,384x16,384 RGB texture.

6.1. Preprocessing

The input dataset was transformed by our texture and geometry processing tools. For textures, we used a tile size of 512x512 pixels, which produced a 9 level quadtree and compressed colors using the DXT1 format. Texture preprocessing, including error propagation, took roughly two hours and produced a structure occupying 178 MB on disk. Processing time is dominated by texture compression. For geometry, we generated two 19 levels bintrees, with leaf nodes containing triangular patches of 16x16 vertex side at full resolution and interior nodes with a constant vertex count of 200. Geometry preprocessing, that included optimized tristrip generation, exhibits the following times and memory requirements:

Size	Tris	Time (h:m:s)	Output size	RAM
1K x 1K	2M	6:35	2 x 14MB	9MB
4K x 4K	32M	1:42:33	2 x 196MB	30MB
8K x 8K	128M	6:39:23	2 x 765MB	115MB

For the sake of comparison, Hoppe's view dependent progressive meshes [10], that, like BDAMs, support unconstrained triangulation of terrains, need roughly 380MB of RAM and uses 190MB of disk space to build a multiresolution model of a simplified version of 7.9M triangles of the Puget Sound dataset. Preprocessing times are similar to BDAM times. By contrast, SOAR [15] geometry data structure, which is based on a hierarchy of right triangles, takes roughly 3.4 GB[§] on disk for the processed data set, but is much faster to compute since the subdivision structure is data independent.

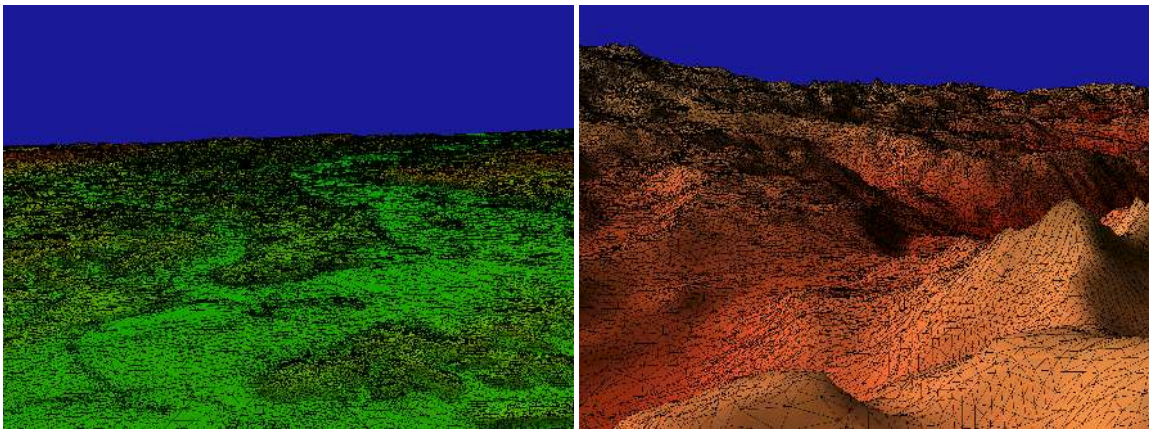
6.2. View-dependent Refinement

We evaluated the performance of the BDAM technique on a number of flythrough sequences over the Puget Sound area. The quantitative results presented here were collected during a 50 seconds high speed fly-over of the data set with a window size of 800x600 pixels and a screen tolerance of 1.0 pixel. The qualitative performance of our view-dependent refinement is further illustrated in an accompanying video, showing the live recording of the analyzed flythrough sequence (Fig. 10). During the entire walkthrough, the resident set size of the application is maintained at roughly 160 MB, i.e. less than 10% of data size, demostrating the effectiveness of out-of-core data management.

Figure 11(a) illustrates the rendering performance of the application. We were able to sustain an average rendering rate of roughly 22 millions of textured triangles per second, with peaks exceeding 25 millions, which are close to the peak performance of the rendering board (Fig. 11(a) left). By comparison, on the same machine, SOAR peak performance was measured at roughly 3.3 millions of triangles per second, even though SOAR was using a smaller single resolution texture of 2Kx2K texels. The increased performance of the BDAM approach is due to the larger granularity of the structure, that amortizes structure traversal costs over many graphics primitives, reduces AGP data transfers through on-board memory management and fully exploits the post-texture-and-lighting cache with optimized indexed triangle strips. The time overhead of BDAM structure traversal, measured by repeating the test without executing OpenGL calls, is only about 20% of total frame time

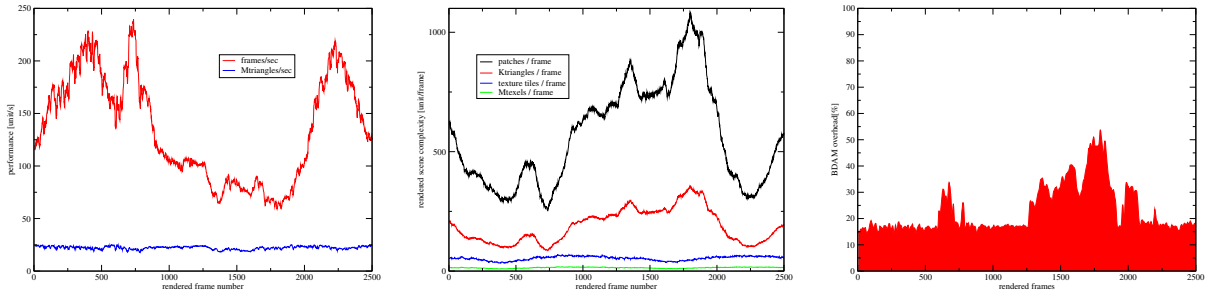
[‡] The dataset at various resolution is freely available from `http://www.cc.gatech.edu/projects/large_models/ps.html` and is now a standard benchmark for terrain rendering applications.

[§] The version of SOAR used in this comparison is v1.11, available from `http://www.cc.gatech.edu/~lindstro/software/soar/`



(a) Frame 400 (b) Frame 1751

Figure 10: ***Selected flythrough frames.*** *Screen space error tolerance set to 1.0 pixels.*



(a) Rendering rates per frame (b) Rendered complexity per frame (c) BDAM overhead per frame

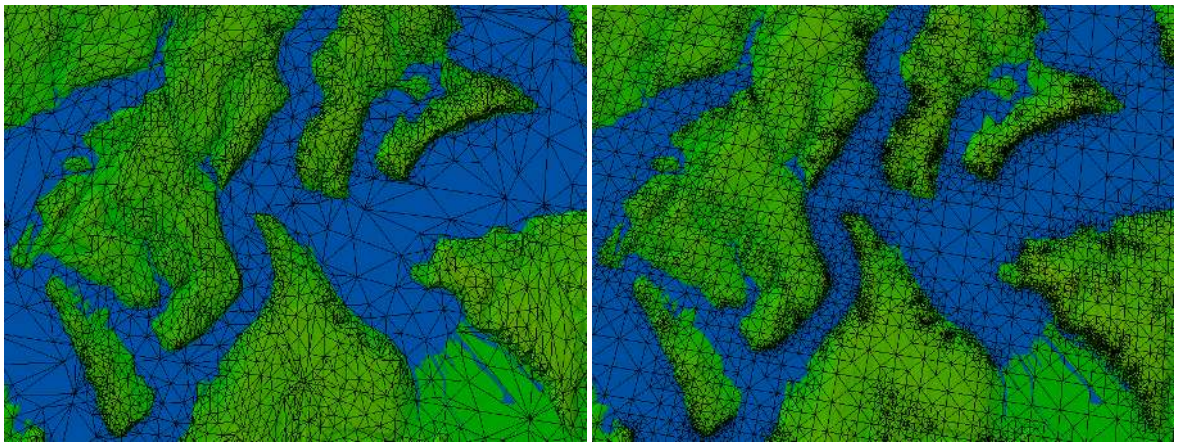
Figure 11: ***Performance Evaluation.***

(Fig. 11(c)), demonstrating that we are GPU bound even for large data sets.

Rendered scene granularity is illustrated in figure 11(b): even though the peak complexity of the rendered scenes exceeds 350K triangles and 20 M texels per frame, the number of rendered graphics primitives per frame remains relatively small, never exceeding 1000 patches and 64 texture blocks per frame. Since we are able to render such complex scenes at high frame rates (60 to 240 Hz for the entire test path, Fig. 11(b)), it is possible to use very small pixel threshold, virtually eliminating popping artifacts, without resorting to costly geomorphing features. Moreover, since TINs are used as basic building blocks, triangulation can be more easily adapted to high frequency variations of the terrain, such as cliffs, than techniques based on regular subdivision meshes (Fig. 12).

7. Conclusions

We have presented an efficient technique for out-of-core rendering and management of large textured terrain surfaces. The technique, called Batched Dynamic Adaptive Meshes (*BDAM*), is based on a paired tree structure: a tiled quadtree for texture data and a pair of bintrees of small triangular patches for the geometry. These small patches are TINs and are constructed and optimized off-line with high quality simplification and tristripping algorithms. Hierarchical view frustum culling and view-dependendent texture and geometry refinement is performed at each frame with a stateless traversal algorithm that renders a continuous adaptive terrain surface by assembling out-of-core data. Thanks to the batched CPU/GPU communication model, the proposed technique is not processor intensive and fully harnesses the power of current graphics hardware. Both preprocessing and



(a) BDAM approximation with 14K triangles

(b) SOAR approximation with 14K triangles

Figure 12: ***Quality Evaluation.*** *TINs can easily adapt to high frequency variations of the terrain such as cliffs, while many subdivision levels are needed for regular subdivision meshes, that spend a large fraction of the triangle budget in edge following.*

rendering exploit out of core techniques to be fully scalable and be able to manage large terrain datasets.

References

1. P. Cignoni, E. Puppo, and R. Scopigno. Representation and visualization of terrain surfaces at variable resolution. *The Visual Computer*, 13(5):199–217, 1997. 2
2. Daniel Cohen-Or and Yishay Levanoni. Temporal continuity of levels of detail in delaunay triangulated terrain. In *IEEE Visualization '96*. IEEE, October 1996. ISBN 0-89791-864-9. 2
3. Jürgen Döllner, Konstantin Baumann, and Klaus Hinrichs. Texturing techniques for terrain visualization. In *IEEE Visualization '00 (VIS '00)*, pages 227–234, Washington - Brussels - Tokyo, October 2000. IEEE. 2
4. M.A. 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*, pages 81–88. IEEE, October 1997. 2, 3, 6
5. William Evans, David Kirkpatrick, and Gregg Townsend. Right triangulated irregular networks. *Algorithmica*, 30(2):264–286, Mar 2001. 2
6. R.J. Fowler and J.J. Little. Automatic extraction of irregular network digital terrain models. *ACM Computer Graphics (SIGGRAPH 79 Proc.)*, 13(3):199–207, Aug. 1979. 6
7. M. Garland and P.S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 209–216. Addison Wesley, August 1997. 6, 7
8. Markus Gross, Roger Gatti, and Oliver Staadt. Fast multiresolution surface meshing. In *Proceedings IEEE Visualization '95*, page 207Ű234. IEEE, Oct 1995. 2
9. H. Hoppe. View-dependent refinement of progressive meshes. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 189–198. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7. 2, 5
10. H. Hoppe. Smooth view-dependent level-of-detail control and its aplications to terrain rendering. In *IEEE Visualization '98 Conf.*, pages 35–42, 1998. 2, 8
11. H. Hoppe. Optimization of mesh locality for transparent vertex caching. In *Computer Graphics Proc., Annual Conf. Series (SIGGRAPH '99), ACM Press*, pages 269–276, 1999. 3
12. Joshua Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *Proceedings IEEE Visualization '02*, pages 259–266. IEEE, Oct 2002. 2
13. 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 *Comp. Graph. Proc., Annual Conf. Series (SIGGRAPH 96), ACM Press*, pages 109–118, New Orleans, LA, USA, Aug. 6-8 1996. 2
14. P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *Proc. IEEE Visualization 2001*, pages 363–370, 574. IEEE Press, October 2001. 4, 5, 6
15. P. Lindstrom and V. Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transaction on Visualization and Computer Graphics*, 8(3):239–254, 2002. 2, 8
16. D. Moench, M. Manley, J. Rubinstein, A. Roy, K. Bruzenak, and D. O'Neil. Phoenix in chaos. *Batman*, 13(502):1–22, 1993. 1
17. R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In H. Rushmeier D. Elbert, H. Hagen, editor, *Proceedings of Visualization '98*, pages 19–26, 1998. 2, 4
18. R. Pajarola. Overview of quadtree based terrain triangulation and visualization. Technical Report UCI-ICS TR 02-01, Department of Information, Computer Science University of California, Irvine, Jan 2002. 2
19. Renato Pajarola, Marc Antonijuan, and Roberto Lario. Quadtin: Quadtree based triangulated irregular networks. In *Proceedings IEEE Visualization '02*, pages 395–402. IEEE, Oct 2002. 2
20. Alex A. Pomeranz. Roam using surface triangle clusters (rustic). Master's thesis, University of California at Davis, 2000. 2
21. E. Puppo. Variable resolution terrain surfaces. In *Proceedings Eight Canadian Conference on Computational Geometry, Ottawa, Canada*, pages 202–210, August 12-15 1996. 2
22. H. Samet. *Applications of Spatial Data Structures*. Addison Wesley, Reading, MA, 1990. 2
23. Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: A virtual mipmap. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 151–158. ACM SIGGRAPH, Addison Wesley, July 1998. 2
24. J.C. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In R. Yagel and G. Nielson, editors, *IEEE Visualization '96 Proc.*, pages 327–334, 1996. 2