Batch-Based Visualization of Large Multiresolution Polygonal Models

Jean Pierre Charalambos*and Josep Vilaplana†

November 23, 2005

Abstract

In this report we study some techniques for the visualization of large multiresolution polygonal models, some of which do not fit into main memory. We focus our attention in batch-based techniques. The main idea is to represent the model, a given height field or a mesh with no particular topological genus, in a recursive tree-like structure; and to link patches, composed of a few set of triangles, to each of their nodes. To render the model, patches which could reside in secondary memory, are batched to the graphics processor unit (GPU) according to a simply stateless top-down traversal of the structure, guided by a view-dependent error mechanism.

1 Introduction

Triangles are the most popular drawing primitive in computer graphics. Since in many applications surfaces are represented by triangle meshes, they have become the standard in all graphics libraries, and special purpose hardware. Detailed triangle meshes are typically obtained from range scanners, or by extracting isosurfaces from high resolution datasets in volume visualization. Storage, interactivity, and real time rendering of huge meshes may easily require memory and processing resources beyond the power of state-of-the art computers. Furthermore, it has been seen that the size of triangle meshes grows faster than RAM capacity does. Therefore, adopting a multiresolution model of the mesh, where it is possible to adapt the resolution of the mesh according to some specified user needs, is a basic issue.

Multiresolution models have been studied for a long time and they can be classified into two groups: *tree-like models* and *historical models*, [22]. We will only deal with the former. For a thorough review the reader may refer to [22].

1.1 Tree-like models

A tree-like model encodes a nested subdivision of an initial mesh (understood as the coarsest) through a recursive refinement process: at each step of the refinement, a subpart of the mesh is refined independently into a local submesh,

*charalam@lsi.upc.edu
†josep@lsi.upc.edu

through a decomposition rule that is characteristic of each specific model. The hierarchy of the meshes is represented as a tree where each node is a local mesh, and each arc corresponds to a refinement operation, [22].

According to the assumed memory model of the underlying computer system, tree-like visualization techniques could be classified into two groups: *in-core* and *out-of-core visualization algorithms*. In the former case, it is supposed that all data fit into internal memory, and thus it suffices to assume a single level of main memory. In the latter case, when dealing with data sets of sizes exceeding main memory[1], communication between the fast internal memory and the slow external memory is often the bottleneck, and thus it is necessary to distinct between them (the previous assumption of a single level of main memory is now meaningless). Moreover, under this circumstances a more realistic measure of the efficiency of an algorithm is the number of I/O-operations performed between internal memory and disk. Thus, the goal of out-of-core algorithms (also known as *external memory* algorithms) is to minimize the number of I/O-operations.

In the other hand it is also possible to classify the tree-like visualization techniques according to what the obtained set of meshes represent: they could stand for approximations of the original model, or they could represent *containers* holding small triangles meshes approximating the original model. The former approach, which we will refer as *single multiresolution visualization framework*, has been studied for a long time and is generally better suitable for graphics applications running under a basic hardware configuration. The later approach, which we will refer as *batch multiresolution visualization framework*, has recently been adopted to take advantage of the capabilities found in modern GPU's, [3, 4]. It is important to stress that this classification is orthogonal to the one stated previously, i.e., under each framework it is possible to implement in-core techniques as well as out-of-core techniques. Throughout this report we will mainly focus our attention in the batch visualization framework for in-core techniques, remarking some of the issues needed to be taken into account when dealing with out-of-core techniques.

1.1.1 Single Multiresolution Visualization Framework

Under this approach the vertices of the local meshes encoded in the hierarchy correspond to vertices of the approximated mesh (see for example the hight field encoded as *hierarchy of right triangles* in figure 1). During execution time the model is updated from hierarchy traversals by the CPU: each primitive is first extracted from those nodes meeting a view-dependent approximation error, and then sent to the GPU to be rendered.

In a typical out-of-core implementation the tree-like structure is kept in main memory and the geometry is extracted from disk through system memory mapping functions, [15].

The subdivision process is carried out until the resolution of leaf node meshes represent the original high resolution of the model, or a well enough approximation of it. Whilst this subdivision process is typically performed off-line, i.e.,

[1]The need for an interactive visualization of very large meshes comprising hundreds of millions of polygons arises in many computer graphics applications, such as 3D scanning [13], geometric modelling [24], numerical simulation [16], terrain visualization [3] and 3D object reconstruction from a stack of 2D images, [1, 26].

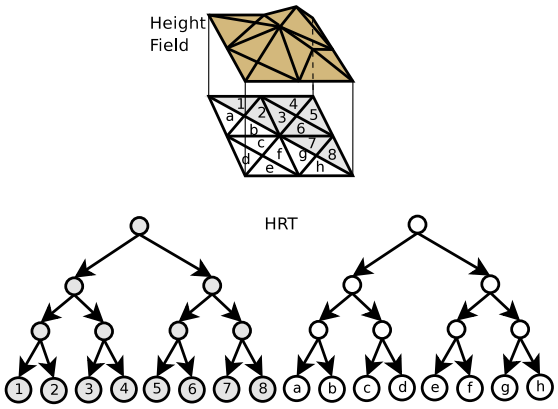

Figure 1: A height field encoded as a HRT.

as a preprocess; the extraction and visualization of a mesh representing a well approximation of the model according to the user point of view, is performed during run time.

1.1.2 Batch Multiresolution Visualization Framework

In the above framework when dealing with a large model, the CPU becomes the bottleneck, i.e., the application is CPU-bound. The number of triangles are too big for the CPU to be able to extract them from the hierarchy and to feed the GPU at a rate to fully harness it. Moreover, since CPU's processing power grows at a much slower rate than GPU's, this gap is doomed to widen.

Modern GPU's possesses two important properties: 1. Their memory could be directly manipulated through *vertex programs*; and, 2. The optimal representation for rendering are triangle strips (more precisely, *cache coherent index strips*). Thus, to fully harness their power, the level of granularity of the original multiresolution model should be moved from single triangles to small contiguous mesh portions, referred as *triangle patches* (or, simply *patches*)[2], [3, 4]. Some properties of the approach are the following:

1. Patches are associated to nodes of the tree-like hierarchy, i.e., the region encoded within the node is not used anymore as a render primitive, instead it now holds a pointer to a patch holding a sub-mesh of the model found somewhere within the region. Under this approach, the nodes of the tree-like hierarchy could be regarded as containers, i.e., each one of them contains a single triangle patch. See figure 8 (the reader may contrast it with figure 1).

2. To keep the implicit multiresolution encoded scheme found in the recursive structure itself, a given patch must reflect the resolution of the node referencing it. Hence, patches need to be simplified according to their level in the hierarchy. This process is performed off-line and bottom-up.

[2]Since this small triangle meshes should generate globally correct surface triangulations (see sections 5 and 6), they are referred as *triangle patches,* or simply *patches*, [3, 4]

3. Patches are preoptimized off-line for rendering. Each triangle patch is processed to obtained a set of triangle strips whose optimal size is determined according to the underlying GPU.

4. In out-of-core techniques patches are efficiently stored in secondary memory. During execution time only the patches needed to be rendered are loaded into main memory. The task is left to operating system through the use of system memory mapping functions.

5. During execution time patches are batched to the GPU for rendering through a vertex program.

In [3, 4] it is reported that under a similar approach they were able to be GPU bounded. In this report we will thoroughly review this approach. In part I we will deal with the suitable subdivision processes adopted for batch rendering; in part II we will review all the steps comprising the visualization process; and in part III we close the report with the conclusions.

Part I
Review of Subdivision Processes

To our knowledge there has only been used two subdivision processes for batch rendering: *regular grid subdivision* (section 2), [10, 21]; and *longest-edge bisection* (section 3), [3, 4]. If the task were to approximate a given model, only in the later approach we would be able to extract different conforming resolution meshes, i.e., meshes without cracks. However, in the case of batch rendering, to obtain different conforming resolution meshes it suffices to constrain the patch simplification process, independently of the conformity of the container mesh itself, see section 5. For this reason the regular grid subdivision process has been used for batch rendering. In the case of the longest edge bisection procedure, in [3, 4] it is claimed that by ensuring the conformity of the container mesh it is possible to obtain a better simplification quality, section 5. Thus, we will thoroughly review how to extract conforming resolution meshes in the longest edge bisection procedure.

2 Regular Grid Subdivision

In 2D the process consists in the recursive quaternary subdivision of a square into quadrants, obtained by splitting the square with two orthogonal lines through its center, [22]. The process is typically encoded within a *quadtree* spatial structure (a quadtree representing a grid and a surface is depicted in figure 2: the 2D grid is depicted in figure 2 part a.; and the surface, in part b.). Starting the process with a box it is easy to generalize the process to 3D. In this case the spatial structure obtained is an *octree*.

The hierarchies produced by this process can not be refined locally, i.e., an extracted mesh is conforming only if all of the selected nodes lie in the same level. See the cracks in the quadtree surface found in figure 2, part b.

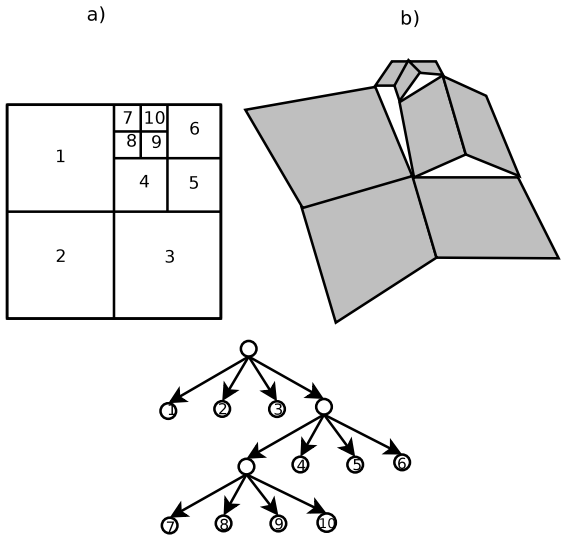

Figure 2: A quadtree structure: a) A 2D grid; and, b) A surface with cracks (T-Vertices).

3 Longest Edge Bisection (LEB)

The set of multiresolution meshes in a LEB hierarchy are obtained from a recursive subdivision process carried out from an initial right isosceles triangle or a tetrahedron, depending whether the process is performed in 2D or 3D, respectively. In this section we will first deal with the refinement process, and its encoding mechanism. We will also review some concepts relating *diamonds* which are necessary to state the rules to extract, from the hierarchy, different conforming resolution meshes. Each of the extracted conforming resolution meshes is used to approximate a given original high resolution mesh in a view-dependent manner.

3.1 Refinement Process

The refinement process is carried out in triangle (tetrahedron) by triangle (tetrahedron) basis by a subdivision operation that continuously adds vertices to the mesh. This operation consists in splitting a triangle (tetrahedron) at the midpoint of its longest edge e; thus, creating two children triangles (tetrahedra) and introducing a new vertex in the mesh. To avoid introducing cracks in the mesh, each time a triangle (tetrahedron) is split all neighbour triangles (tetrahedra) sharing edge e must also be carefully split, see section 3.1.1. The nodes in the hierarchy corresponding to the two children triangles (tetrahedra) are referred as *sibling nodes*, the longest edge as the *split edge*, and its midpoint as the *split vertex*, [8]. The process begins with a small set of triangles (tetrahedra) which vertices represent the coarse base mesh, e.g, a box divided by *two* right isosceles triangles adjacent along their hypotenuse, a cube divided into *six* tetrahedra around a major diagonal. The subdivision process finishes when all children

triangles (tetrahedra) represent the original full resolution of the model (or a well enough approximation).

3.1.1 Subdivision Rules of Neighbour triangles (tetrahedra)

Since the insertion procedure is carried out in a triangle (tetrahedron) by triangle (tetrahedron) basis, when splitting a triangle (tetrahedron) special care must be taken to avoid introducing cracks in some of the edges found in other triangles (tetrahedra). Let P and Q be two triangles (tetrahedra) with split vertices e and f, respectively. Furthermore, suppose that e lies on an edge in Q, and P is to be split. To decide how to split Q we have identified two cases:

1. e and f lies on the same edge. Therefore, we simple split Q. See figure 3, part a.

2. e and f lies on different edges. Therefore, we need to: 1. Split Q; 2. Split the child of Q which split vertex now matches e; and, 3. Repeat the procedure taking Q as the base triangle (tetrahedron) to be split. See figure 3, parts b., and c.

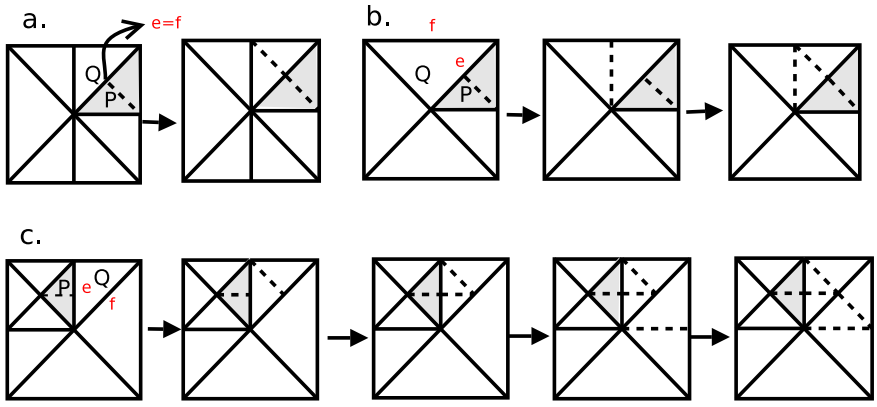

Figure 3: Avoiding the appearance of cracks (T-Vertices) in neighbours of a split polyhedron. Cases: a) e and f lies on the same edge; and, b) and, c) e and f lies on different edges.

3.1.2 Bintrees

If, when refining the mesh, each triangle (tetrahedron) is related to the two smaller triangles (tetrahedra) it gets divided into, the subdivision process can simply be encoded as a bintree (one bintree for each triangle/tetrahedron in the coarse base mesh). In the literature found in the 2D case, the meshes produced by this subdivision scheme have received several names, such as *hierarchies of right triangles* (HRT), 4-k meshes [25], right-triangulated irregular networks [7], or restricted quadtree triangulations [18]. For a 2D example, see figure 4 part a.

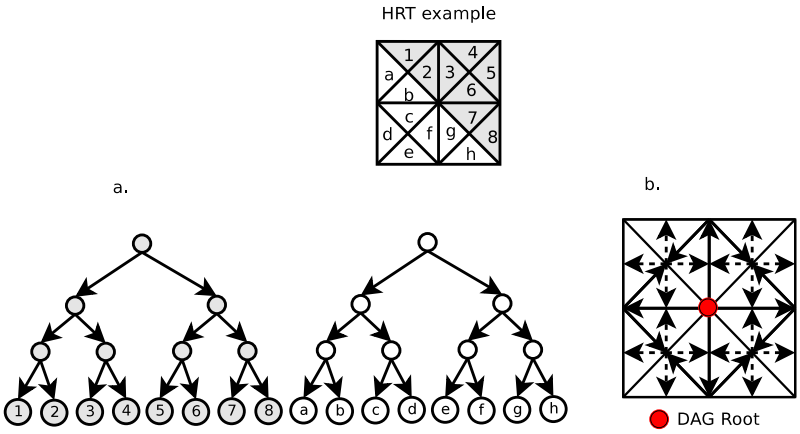

Figure 4: A HRT mesh: a) The mesh represented as a forest of bintrees; and, b) The same mesh represented as DAG of diamonds.

3.2 Diamonds and Diamond DAG's

All triangles (tetrahedra) sharing their split edge may be clustered together to form a *diamond*, i.e., a group of triangles (tetrahedra) sharing their longest edge. A parent-child diamond relationship may be understood in the following sense: the parents of a given diamond D, are the diamonds that must be split to create D's triangles (tetrahedra).

Since each diamond can be uniquely identified by their split edge or split vertex, the mesh can also be represented as a *directed acyclic graph* (DAG) of diamonds using its split vertices, [15]. Each directed edge (i, j) maps from parent diamond split vertex i to child diamond split vertex j.

In the 2D case, it is easy to see that each diamond has 2 parents and 4 children. See figure 4, part b.

To characterize the 3D DAG of diamonds we need to study the subdivision process in more detail. In this report we will follow the notation used in [8]. The subdivision of a tetrahedron may be described by a *level* and a *phase*, with 3 phases at each level. After one refinement the phase is incremented by one. After three refinements, the level is incremented by one. Since all tetrahedra within a diamond are of the same phase/level, we can refer to the diamond phase/level without ambiguity. The 3D diamond DAG parent/child relationships are completely defined according with the information found in table 1. The three tetrahedra phases are illustrated in figure 5.

Phase	Tetrahedra (Phase, Level)	Parents (P., L.)	Children (P, L.)
0	6(0,L)	3(2,L-1)	6(1,L)
1	4(1,L)	2(0,L)	4(2,L)
2	8(2,L)	4(1,L)	8(0,L+1)

Table 1: Diamond DAG Structure.

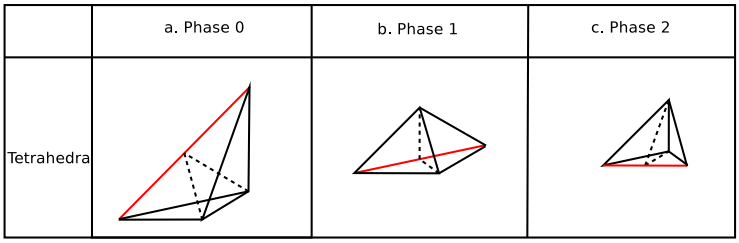

Figure 5: Tetrahedra phases.

3.3 Conforming Multiresolution Meshes

We will now see how to extract different conforming meshes from the hierarchy produced by the subdivision process outlined before.

The idea is to recursively traverse the hierarchy (the forest of bintrees) before reaching the nodes comprising the mesh to be extracted: those which meet a view-dependent approximation error, [15]. It is worth noticing that to perform this task it is possible to visit the hierarchy top-down or bottom-up. A significant disadvantage found in the later mode is that its computational complexity depends on the size of the highest encoded resolution mesh, whereas in the former, it is linear in the size of the approximated mesh, [15]. In this report we will only study the details needed to be taken into account when performing a top-down traversal of the bintrees to extract a resolution mesh.

An important property found in the hierarchies produced by the longest edge procedure, is that they can be refined locally without having to maintain the entire mesh at the same resolution, [15], i.e., for an extracted mesh to be conforming, not all of the selected nodes must lie in the same level. In figure 6, two 2D extracted conforming meshes at different levels-of-detail are depicted. It is important to stress that not all visits of the bintrees produce a conforming mesh, see for example the crack (T-Vertex) in figure 6.

We now state the conditions to extract conforming variable resolutions meshes from the hierarchy of triangles (tetrahedra) produced by the longest edge bisection procedure outlined before:

Let C and D be two diamonds, being D the parent of C. Suppose that triangle (tetrahedron) t belongs to D. Then t can correctly connect either to:

1. Any triangles (tetrahedra) belonging to D. See figure 7, part a.

2. Triangles (tetrahedra) found in neighbour diamonds of the same level. See figure 7, part a.

3. Any triangle (tetrahedron) belonging to C, but which parent triangle (tetrahedron) is not in D. In figure 7, part b.

This conditions could be regarded as the rules for correctly connect two triangles (tetrahedra) among levels of the hierarchy: the first and second rules state the conditions for triangles (tetrahedra) belonging to the same level; the third, for triangles (tetrahedra) belonging to different levels. Observe that neighbour triangles (tetrahedra) could never differ in more than one level.

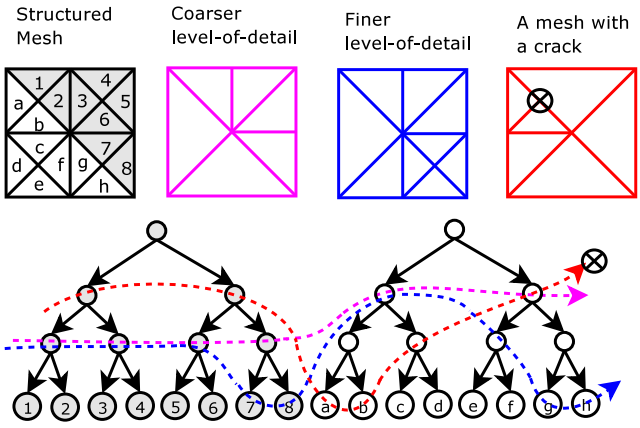

Figure 6: A multiresolution mesh encoded with two bintrees. The dotted lines represent the set of nodes comprising two conforming meshes at different resolutions, and one non conforming mesh.

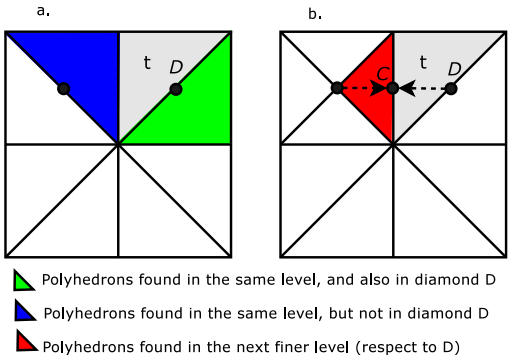

Figure 7: Connectivity rules for polyhedrons found in the multiresolution mesh. The blue polyhedron is the parent of the red polyhedron.

Part II
General Framework for Batch Rendering

As mentioned so far, to keep the implicit multiresolution encoded scheme found in the recursive structure itself, a given patch must reflect the resolution of the node referencing it. Hence, patches need to be simplified according to their level in the hierarchy, e.g., together all leaf node patches should represent the original full resolution model.

The patch simplification process is performed off-line, bottom-up, beginning with leaf nodes patches, which, in turn, are obtained as a result of the construction phase, see section 4. Parent node patches are built out by merging children node patches and simplify them at the parent node level, taking into account the correct connectivity along their borders, section 5. Since the patch simplification phase is carried out as a preprocess, patches can be fully optimized off-line, e.g., taking into account topology compression, mesh simplification, and tri-stripification routines. While the recursive structure is always kept in main memory, in an out-of-core implementation triangle patches reside in secondary memory storage. During execution time only the patches needed to be rendered are loaded into main memory using system memory mapping functions, section 6.

4 Construction of the Hierarchy

The original model mesh is processed to produce: 1. The recursive multiresolution structure; and, 2. The initial triangle patches associated at their leaf nodes. There exists two possible ways to guide the subdivision process: 1. By defining the number of levels comprising the hierarchy, [21, 3]; or, 2. By defining a maximum number of triangles allowed at leaf node patches, [4].

4.1 Computation of the Hierarchy from a Target Number of Levels

In this case we always obtained a *complete* tree-like structure and to generate it we use algorithm 1. In the first loop we simply split the structure until the desired number of levels is reached. In the second loop we recursively call function `inserTriangle1`() (given by algorithm 2) to insert the triangles at leaf nodes.

Since the tree-like structure representing a height field is always complete, this approach is clearly suitable to visualize them. It suffices to calculate the depth at which parent leaves would contain the desired number of triangles. In the case of a regular grid subdivision an implementation is found in [10], while in the LEB subdivision case, an implementation is found in [3] (see figure 8)[3].

[3]It is worth noticing that each patch of triangles could be regarded as a TIN (a Triangular Irregular Network). Since TIN's are an important class of meshes used for view dependent refinement, [15], the approach applied to terrain visualization could be understood as the combination of the best of both representations: whilst HRT's are simpler, TIN's have the

Algorithm 1 `constructHierarchy1`(r, L, n)

Require: The list of vertices, L; a pointer to the root of the recursive structure, r; and the target number of levels, n.

Ensure: The recursive structure, and the triangle patches associated at their leaf nodes.

```
while number of levels < n do
  splitAll(r)
end while
while L is not empty do
  t = readNextTriangle(L)
  insertTriangle1(r, t)
end while
```

Algorithm 2 `insertTriangle1`(p, t)

Require: A pointer to a structure node, p; and a triangle, t.

Ensure: The triangle patches associated at the leaf nodes of the recursive structure.

```
if p is a Leaf Node then
  append(t)
else
  c =localeChild(p, barycenter(t))
  insertTriangle1(c, t)
end if
```

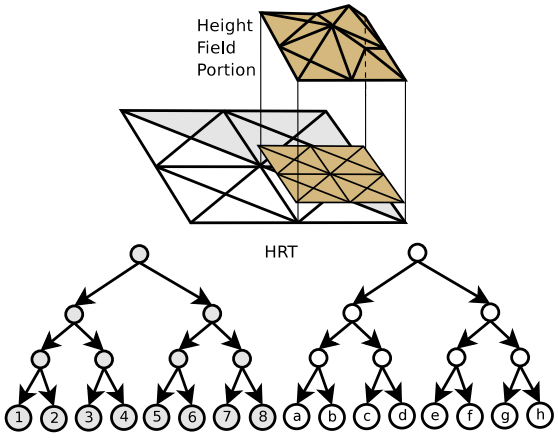

Figure 8: A height field encoded as a HRT. Each leaf node in the HRT contains four triangles.

To our knowledge, in the case of meshes of arbitrary topology the only approach implementing this method is in the case of a regular grid subdivision and it is found in [21]. In this work it is reported that in deciding the size of the patch there is a trade-off between memory usage and the effectiveness of the simplification process: smaller patches take up less memory, but there is little gain in simplify them (see section 5). In the other hand, since smaller patch sizes increase its total number, the renderer could also be compromised. Finally, it is also reported in this work that the *optimum* size of leaf patches should be determined *empirically*.

4.2 Computation of the Hierarchy from a Target Number of Triangles Allowed at Leaf Nodes

In this case the structure is computed by the recursive insertion of triangles, see algorithm 3. Each time a given triangle t is inserted, we locate the leaf that contains its barycenter and, if the number of triangles already contained in it does not exceed the maximum ($maxT$), we simply insert the new one in the associated triangle patch. Otherwise, we need to, 1. Split the hierarchy; 2. Relocate the triangles found in the previous leaves into the new ones; and 3. Recursively called the insertion function in t, see algorithm 4.

Algorithm 3 `constructHierarchy2`$(r, L, maxT)$

Require: The list of vertices, L; the maximum number of triangles allowed at a node, $maxT$; and a pointer to the root of the recursive structure r.
Ensure: The recursive structure, and the triangle patches associates at their leaf nodes.

```
while L is not empty do
  t = readNextTriangle(L)
  insertTriangle2(r, t, maxT)
end while
```

It is worth mentioning that in the case of a LEB hierarchy, the function `split`(r) should take into account the subdivision rules defined in section 3.1.1. The LEB case is suitable for representing meshes of arbitrary topological genus and its framework is due to [4].

5 Simplification

The simplification process takes as its input the hierarchy, together with the initial triangle patches associated at their leaf nodes. The process is performed off-line and bottom-up. Parent node patches are built out by merging children node patches and simplify them at the parent node level. By adjusting the degree of simplification, it is easy to keep the number of triangles at parent nodes near to the number of triangles found at leaf node patches, see algorithm 5.

potential to represent the surface with fewer triangles, [3].

Algorithm 4 `insertTriangle2`$(p, t, maxT)$

Require: A pointer to a structure node, p; a triangle, t; and the maximum number of triangles allowed at a given node, $maxT$.

Ensure: The recursive structure which root node is p, and the triangle patches associated at the leaves located at its descendant children nodes.

```
if p is a LeafNode then
  if number of vertices in node ≤maxT then
    append(t)
  else
    r =getRootPtr(p)
    split(r)
    L' =getVertexList(p)
    while L' is not empty do
      t' = readNextTriangle(L')
      c =localeChild(p, barycenter(t'))
      insertTriangle2(c, t', maxT)
    end while
    c =localeChild(p, barycenter(t))
    insertTriangle2(c, t, maxT)
  end if
else
  c =localeChild(barycenter(t))
  insertTriangle2(c, t, maxT)
end if
```

Algorithm 5 `simplifyHierarchy`(L, l, r)

Require: The list of vertices, L; the target number of levels in the hierarchy, l; the pointer to the root of the recursive structure r; and, optionally, the patch data repository, R.

Ensure: The recursive structure, and the triangle patches associated at their leaf nodes.

```
while level >1 do
  for each node in level do
    if node is not marked then
      mergeRelatedNodes()
      simplify(ε)
      computeAssociatedData()
      storePatchData(R)
      markSiblingNodes
    end if
  end for
  level= level − 1
end while
```

Remarks

Function `simplify`(ϵ)

To simplify the model, the function `simplify`(ϵ) could take one of two parameters: the maximum *error* allowed at the patch after its simplification, or the target *number* of triangles that should reside in the patch after its simplification[4]. In any case, the function should guarantee the correct connectivity along patch borders.

Since in the case of a regular grid subdivision the merging of neighbour nodes is unconstrained (i.e., neighbour subgrids could differ in any number of levels, see section 2), it suffices to lock the vertices at each subgrid. Since after simplifying the whole hierarchy some of the vertices of the original model remains fixed up to the top level, the quality of this procedure is doubtful. In this case the procedure `mergeRelatedNodes` simply consists in stitching the regular subgrids of sibling nodes at their parent level.

In the case of a LEB hierarchy we need to constraint the process according to the set of rules for extracting variable resolution meshes defined in section 3.3. Observe that borders of a patch contained in a triangle (tetrahedron) could be *internal* or *external*, i.e., through internal borders the triangles of the patch get connected with other triangles contained in triangles (tetrahedra) of the same diamond; through external borders, they get connected to triangles contained in triangles (tetrahedra) of neighbour diamonds. Therefore, to guarantee connectivity among internal borders the mesh contained in a diamond should be simplified as a single unit, while the triangles laying in external borders need to be kept fixed. Given a diamond D at the parent level, the procedure `mergeRelatedNodes` consists in merging into a single mesh each sibling pair of triangles (tetrahedra) in the children level whose parent triangle (tetrahedron) belongs to D, i.e., a given patch corresponds to a triangle (tetrahedron), but the simplification process is performed in a diamond by diamond basis. Afterwards data is distributed among triangles (tetrahedra) comprising diamond D (see next remark). This procedure could be regarded as an implementation of the three rules for the correct extraction of variable resolution meshes defined in section 3.3: by handling internal borders we guarantee rule 1, by handling external borders we guarantee rules 2, and 3. In [4] it is claim that these constraints have little effect on overall simplification quality, since constrained vertices alternate from diamond-internal to diamond-external borders throughout the hierarchy, and are locked only in diamond-external state.

Any mesh simplification algorithm that supports the definition of constraints (in the sense stated above) could be used to implement function `simplify`(ϵ). Particularly, the simplification algorithm implemented in [21, 4] is the *quadric metric for simplifying meshes with appearance attributes*, [11].

Associated Data

Each node in the hierarchy (i.e., a triangle/tetrahedron or a 3D regular subgrid) contains all of the following:

[4]The original high resolution of the mesh is kept among leaf node patches, i.e., leaf node patches are not simplify.

- The *patch data* that contains the geometry of the submesh represented as triangle strips.

- The *bounding spheres* that are used for view frustum culling of the patch, see section 6.

- The *bounding cone of normals* that are used for back face culling of the patch, see section 6.

In addition, in the case of a LEB hierarchy we also need to include the *saturated model space error*, and the *bounding spheres of the neighbour*. Both are used for the computation of the screen space errors, see section 6.

Function `storePatchData`(R)

In an out-of core implementation each patch is stored externally on a patch repository in a compressed representation from which a version optimized for efficient rendering can be rapidly extracted, [4]. In [4] it is reported that for disk storage, topology is compressed using a mesh encoding scheme preserving stripification, [12].

6 Rendering

To render the object during execution time, the hierarchy is recursively traversed top-down every frame using algorithm 7. With function `refineNode`() (see algorithm 7) we check whether a given node should be rendered, culled or refined.

Algorithm 6 `renderHierarchy`($r, tol, viewPoint, R$)

Require: A pointer to the root of the recursive structure, r; a screen space tolerance measured in pixels, tol; the user point of view, $viewPoint$; and the patch data repository, R.
  for each `Frame` **do**
    `refineNode`($r, tol, viewPoint, R$)
  end for

The refinement process works as follows: we first perform view frustum culling, and back face culling tests to check whether the node is visible or not. For view frustum culling we check whether or not the node bounding sphere partially lies in the view frustum; for back face culling we test whether or not the direction of the view port lies in the bounding cone of normals of the node. If the node is not visible we simple cut the entire branch. If it is visible we check whether or not the resolution of the node represents a good approximation of the model by measuring its screen space error according to the user point of view. If the node represents a good representation of the model, the correspondent patch is retrieved from the repository and batched to the graphics hardware (otherwise, we recursively refine it). See algorithm 7. A thorough discussion about error handling follows below.

Algorithm 7 `refineNode`$(p, tol, viewPoint, R)$

Require: A pointer to a structure node, p; a screen space tolerance measured in pixels, tol; the user point of view, $viewPoint$; and the patch data repository, R.

```
if p isWithinViewFrustum and p isNotBackFace then
  if screenSpaceError(p, viewPoint) ≤ tol then
    renderNode(p, R)
  else
    while c ≠ 0 do
      c =localeNextChild(p)
      refineNode(c, tol, viewPoint, R)
    end while
  end if
else
  cullBranch(p)
end if
```

Object Space and Screen Space Errors

Since a given patch associated to a node in the hierarchy represents an approximation of the region enclosed in the correspondent container, a measure of the accuracy of the approximation (independently of the user point of view) is usually associated to it. This measure is known as the *object space error* and its commonly measure using the *quadric error metric*, [14]. In the other hand, to extract a mesh in a view-dependent manner we need to compute the block's *screen space error*, i.e., the projection of the object space error into screen space, according to the user point of view. In the case of a LEB hierarchy we need also to guarantee the extraction of a conforming variable resolution mesh. This could be done using a *saturation technique*.

Saturation Technique

Since the view-dependent error mechanism is responsible for guiding the refinement process, it seems natural that it also serves to guarantee the extraction of a conforming mesh from a LEB Hierarchy representation. We will see how to implement a mechanism that is at the same time simple and efficient in the following sense:

1. *Simplicity*: In order to guarantee the rules stated in section 3.3, when measuring errors along the hierarchy (the forest of bintrees) we can state the following two simple conditions on the result:

   (a) Errors should be equal for all the triangles (tetrahedra) within a given a diamond.

   (b) Errors should decrease monotonically when descending the hierarchy.

   Whilst the first condition guarantees the first rule defined in section 3.3, the second condition guarantees rules 2, and 3: neighbour diamonds should never differ in more than one level. In the literature, these two conditions are referred as *nested conditions,* [18, 15].

2. *Efficiency*: In terms of efficiency, when measuring the error at a given node it would be best to guarantee the nested conditions without needing to bother about neighbour diamonds, i.e., with a simple *stateless* top-down traversal of the bintrees, [18, 15].

In [3] as well as in [4], object space errors are first computed in a triangle (tetrahedron) by triangle (tetrahedron) basis, and then *saturated* to make them respect the nesting conditions stated above. This saturation process is performed bottom-up in a diamond by diamond basis, and the saturated object space errors are computed as follows:

1. For any diamond leaf, we simple choose the maximum value between the object space errors of their comprising triangles (tetrahedra).

2. For parent diamonds, we choose the maximum value between the saturated object space errors of all their children diamonds, and the object space errors of their own comprising triangles (tetrahedra).

To extract a conforming mesh according to the user point-of-view, screen space errors must also be saturated. In [3, 4] a consistent upper bound of the *saturated screen space errors* is obtained by measuring the apparent size of a sphere with diameter equal to the saturated object space errors, and centered at the *saturated bounding sphere* point closest to the viewpoint (see figure 9, part b.). The saturated bounding sphere hierarchy is calculated with the following recursive rules (see figure 9, part a.):

1. For diamond leaves we calculate the smallest enclosing ball of its comprising triangles (tetrahedra).

2. For parent diamonds we calculate the smallest enclosing ball of the enclosing balls of their children diamonds.

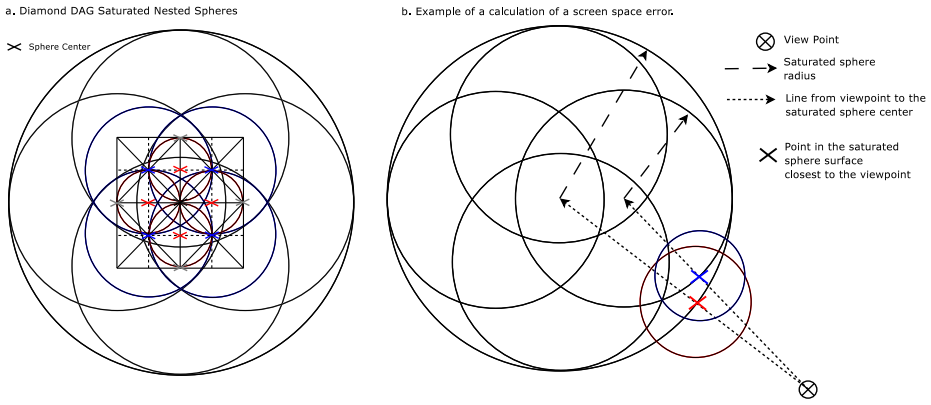

Figure 9: Saturated screen space errors.

Since saturated data, i.e., bounding sphere and object space errors, are associated at each tetrahedron, when extracting a conforming mesh the nested

conditions are ensured with a simple *stateless* top-down traversal of the bintrees.

Part III
Summary and Conclusions

1. A tree-like model encodes a nested subdivision of an initial mesh through a recursive refinement process: at each step of the refinement, a subpart of the mesh is refined independently into a local submesh, through a decomposition rule that is characteristic of each specific model. The hierarchy of the meshes is represented as a tree where each node is a local mesh, and each arc corresponds to a refinement operation.

2. According to the assumed memory model of the underlying computer system, tree-like multiresolution visualization techniques have been traditionally classified into two groups: *in-core* and *out-of-core visualization algorithms*. Whilst in the former case it is supposed that the whole model fits into main memory; in the latter it is supposed that it does not, and thus different parts of the model need first to be fetched into main memory before rendering, i.e., the system main memory could be regarded as memory cache for secondary memory. In the other hand, for those models that do not fit into the GPU memory (and given the fact that modern GPU's provide programmability abilities), the GPU memory itself could be regarded as a cache for the system main memory, i.e., those parts of the model already residing in GPU memory, do not need to be fetched from the system memory.

3. We have referred throughout this report to the methods dealing with latency times and software overhead costs related to the transfer of geometric primitives to the GPU[5], as *batch-based* multiresolution visualization techniques. Batch techniques have recently been adopted to avoid CPU/GPU communication bottlenecks by taking advantage of the capabilities found in modern GPU's: a. Their memory could be directly manipulated through *vertex programs*; and, b. The optimal representation for rendering are triangle strips (more precisely, *cache coherent index strips*).

4. Conceptually, batch techniques could be understood as a transition of the level of granularity of the original multiresolution model from single triangles to small contiguous mesh portions. Some properties of the approach are the following:

   (a) *Patches* are associated to nodes of the tree-like hierarchy, i.e., the region encoded within the node is not used anymore as a render primitive, instead it now holds a pointer to a patch holding a submesh of the model found somewhere within the region. Under this approach, the nodes of the tree-like hierarchy could be regarded as

[5]These costs dramatically hinder performance when the number of triangles to be rendered are big.

containers, i.e., each one of them contains a single triangle patch. See figure 8 (the reader may contrast it with figure 1).

(b) To keep the implicit multiresolution encoded scheme found in the recursive structure itself, a given patch must reflect the resolution of the node referencing it. Hence, patches need to be simplified according to their level in the hierarchy. This process is performed off-line and bottom-up. Patches are preoptimized off-line for rendering.

(c) Each triangle patch is processed to obtained a set of triangle strips whose optimal size is determined according to the underlying GPU. In out-of-core techniques patches are efficiently stored in secondary memory.

(d) During execution time only the patches needed to be rendered are loaded into main memory by the operating system through system memory mapping functions.

5. We have provided a common theoretical background for what we have called *batch-based* visualization techniques for large multiresolution polygonal models; and we have introduced the main algorithms for each of their comprising stages (preprocessing, simplification, and rendering), independently of the underlying subdivision scheme adopted.

References

[1] Charalambos, J. P., Forero M. G. y Zuluaga D. Estación médica para procesamiento de imágenes y visualización de la actividad eléctrica cerebral en una reconstrucción tridimensional de la cabeza de un paciente. Accepted for publication in *Ingeniería e Investigación*. Universidad Nacional de Colombia.

[2] Chiang, Y., El-Sana, J., Lindstrom, P., and Silva, C. 2002. Out-of-Core Algorithms for Scientific Visualization and Computer Graphics. Course Notes for IEEE Visualization.

[3] Cignoni, P., Ganovelli, F., Gobetti, E., Marton, F., Ponchio, F., and Scopigno, R. 2003. BDAM - batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum* 22, 3 (Sept.) 505-514.

[4] Cignoni, P., Ganovelli, F., Gobetti, E., Marton, F., Ponchio, F., and Scopigno, R. 2004. Adaptive TetraPuzzles: Efficient Out-of-Core Construction and Visualization of Gigantic Multiresolution Polygonal Models. ACM Transactions on Graphics, 23(3), August 2004. Proc. SIGGRAPH 2004.

[5] Duchaineau, M. A., Wolinsky, M., Sigeti, D. E., Aldrich, C., Mineev-Weinstein, M. B. 1997. ROAMing Terrain: Real-time Optimally Adapting Meshes. *IEEE Visualization '97*, 81-88.

[6] El-Sana, J., and Chiang, Y. 2000. External memory view-dependent simplification. EUROGRAPHICS.

[7] Evans, W., Kirkpatrick, d., Twownsend, G. 2001. Right-Triangulated Irregular Networks. *Algorithmica*, 30(2):264-286.

[8] Gregorski, B., Duchaineau, M., Lindstrom, P., Pascucci, V., and Joy, K. I. 2002. Interactive view-dependent rendering of large IsoSurfaces. In *Proc. IEEE Visualizations*, 475-484.

[9] Hoppe, H. 1997. View-dependent refinement of progressive meshes. In *SIGGRAPH 97 Conference Proceedings*. Addison Wesley, T. Whitted, Ed., Annual Conference Series. ACM SIGGRAPH, 189-198. ISBN 0-89791-896-7.

[10] Hoppe, H. 1998. Smooth view-dependent level-of-detail control and its applications to terrain rendering. In *IEEE Visualization '98 Conf.,* 35-42.

[11] Hoppe, H. 1999. New quadric metric for simplifying meshes with appearence attributes. In *Proceedings of the 10th Annual IEEE Conference on Visualization (VIS-99)*, ACM Press, New York, pages 59-66.

[12] Isenburg, M. 2001. Triangle strip compression. *Computer Graphics Forum 20*, 2, 91-101.

[13] Levoy, M., Pulli, K., Curless, B., Rusinkiewicz, S., Koller, D., Pereira, L., Ginzton, M., Anderson, S., Davis, J., Ginsberg, J., Shade, J., and Fulk, D. 2000. The digital michelangelo project: 3D scanning of large statues. In *Proc. SIGGRAPH*, K. Akeley, Ed., Annual Conference Series, 131-144.

[14] Lindstrom, P. 2003. Out-of-Core construction and visualization of multiresolution surfaces. In *ACM 2003 Symposium on Interactive 3D Graphics*, 93-102,239.

[15] Lindstrom, P., Pascucci, V. 2001. Visualization of large terrains made easy. In *Proc. IEEE Visualization 2001*, 363-370, 574. IEEE Press, Octuber 2001.

[16] Mirin, A. A., Cohen, R. H., Curtis, B. C., Dannevik, W. P., Dimits, A. M., Duchaineau, M. A., Eliason, D. E., Schikore, D. R., Anderson, S. E., Porter, D. H., Woodward, P. R., Shieh, L. J., and White, S. W. 1999. Very high resolution simulation of compressible turbulance on the IBM-SP system. In *Supercomputing '99*, ACM Press and IEEE Computer Society Press.

[17] Ohlberger, M. and Rumpf, M. 1998. Adaptive projection operators in multiresolution scientific visualization. *IEEE Transactions on Visualization and Computer Graphics 4*, 4 (Oct./Dec.), 344-364.

[18] Pajarola, R. 1998. Large scale terrain visualization using the restricted quadtree triangulation. In H. Rushmeier D. Elbert, H. Hagen, editor, *Proceedings of Visualization '98*, 19-26.

[19] Pascucci, V. 2002. Slow growing subdivision (SGS) in any dimension: towards removing the curse of dimensionality. *Computer Graphics Forum 21*, 3 (Sept.), 451-460.

[20] Bibliography POV-Ray online documentation, `http://www.povray.org/documentation/view/3.6.1/279/`

[21] Prince, C. 2000. *Progressive Meshes for Large Models of Arbitrary Topology.* Master's thesis, Department of Computer Science and Engineering. University of Washington, Seattle.

[22] Puppo, E., and Scopigno, R. *Simplification, LOD and Multiresolution, Principles and Applications.* Eurographics Tutorial. Blackwell Publishers, 1997.

[23] Shaffer, E., and Garland, M. 2001. Efficient adaptive simplification of massive meshes. In *Proc. IEEE Visualization 2001*, IEEE Press, 127-134.

[24] Varadhan, G., and Manocha, D. 2002. Out-of-Core rendering of massive geometric datasets. In Proc. IEEE Visualization, 69-76.

[25] Velho, L., Gomes, J. 2000. Variable Resolution 4-k Meshes: Concepts and Applications. *Computer Graphics Forum*, 19(4):195-212.

[26] The Visible Human Project. 2004. `http://www.nlm.nih.gov/research/visible/visible_human.html`.

[27] Direct Rendering Infrastructure. `http://dri.freedesktop.org/wiki`

[28] K Desktop Environment. `http://www.kde.org`

[29] OpenGL. `http://www.opengl.org`

[30] Qt API. `http://www.trolltech.com`

[31] Mesa 3D. `http://www.mesa3d.org`