Multiresolution structures for interactive visualization of very large 3D datasets

Doctoral Thesis

(Dissertation)

to be awarded the degree of

Doctor rerum naturalium (Dr.rer.nat.)

submitted by

Federico Ponchio

from Vicenza

approved by the

Faculty of Mathematics/Computer Science and Mechanical Engineering,

Clausthal University of Technology

.

Date of oral examination: 16.12.2008

Chairperson of the Board of Examiners: Prof. Dr. Jörg P. Muller

Chief Reviewer: Prof. Dr. Kai Hormann

Reviewer: Prof. Dr. Holger Theisel

.

Contents

Chapter 1

Introduction

The need for interactive visualization of very large surface meshes, consisting of hundreds of millions of triangles, arises naturally in many application domains, including GIS, 3D scanning, geometric modeling, and numerical simulation. However, despite the rapid improvement in hardware performance, these meshes largely overload the computational power and memory capacity of state-of-the-art graphics and computational platforms.

A wide variety of dynamic multi-resolution structures and algorithms have been proposed to face this problem and to work around performance limitations for real-time interaction on general purpose computers. Until a few years ago, the bottleneck of visualization system was the processing power of the GPU, so that these algorithms were designed to generate the most efficient triangulation for the current view-point, even at the cost of complex computations on the CPU.

However, the processing power of the GPU, due in particular to its parallel nature, has been increasing much faster than the CPU. Traditional multi-resolution models were heavily CPU bound, unable to generate model updates at full GPU speed and to efficiently communicate them to the graphics hardware.

So, an emerging trend is to depart from point- or triangle-based multi-resolution models and to adopt a patch-based data structure, from which view-dependent conforming mesh representations can be efficiently extracted by combining precomputed patches. Since each patch is itself a mesh composed of a few thousand primitives, the multi-resolution extraction cost is amortized over many graphics operations, and CPU/GPU communication can be optimized to fully exploit the processing power of the GPU and the complex memory hierarchy of modern graphics platforms.

One of the main challenges in the construction of patch-based multi-resolution structures is enforcing continuity of the surface along the boundary of the patches at different resolutions. The few existing patch-based multi-resolution techniques fall short of a satisfactory solution because they require complicated and expensive boundary management, do not allow for triangle strips or other mesh optimizations, perform expensive per-triangle operations, or do not scale to very large models.

1.1 Contribution

In this thesis we present a series of patch-based multi-resolution structures for terrains, general meshes and animations, which combine either triangle bin-tree hierarchy or a Multi-Triangulation (MT) approach with efficient, GPU friendly data structures and out-of-core techniques, where boundary continuity is automatically enforced.

In particular we describe: a novel out-of-core, patch-based multi-resolution framework, Batched Multi-Triangulation (BMT), that extends the MT and encompasses a wide class of multi-resolution structures; a robust and elegant solution to the boundary continuity problem and the creation of a well conditioned BMT by introducing the concept of *V-partition*; an efficient multi-threaded rendering engine; a general parallel subsystem for the external memory processing and simplification of huge models.

Among the clustered multi-resolution algorithms BMT stands out for its flexibility and simplicity: the local representation and processing of the model is decoupled from the high level multi-resolution structure and thanks to a trivial locking mechanism, no special boundary management is needed at rendering time, to avoid cracks in the model. We will present a number of applications of this framework to specific fields, such as terrains, generic meshes and animations which appeared in various international journals and conferences:

1. the work about terrain visualization described in Chapter 4 was presented in [17] and later adapted to planet visualization in [18];
2. a regular space subdivision based technique for interactive rendering of massive meshes [19] was presented at Siggraph, and later a more general approach based on irregular space subdivision is described in [20]. Both techniques are described in Chapter 5.
3. the work about dynamic meshes described in Chapter 6 appeared in IEEE Transaction on Visualization and Computer Graphics [90].

1.2 Organization of the Work

Chapter 2 will introduce a few basic concepts about triangle meshes, simplification, multi-resolution and a quick overview of the Multi-Triangulation (MT), a dimension-independent framework for continuous multi-resolution modeling based on meshes. A number of multi-resolution structures will be shown to be special instances of the MT framework.

The issues regarding interactive visualization of massive models will be explored in Chapter 3, focusing on the advantages and difficulties of a patch-based structure. We will introduce the concept of V-partition as a solution to the problem of generating efficient patch-based multi-resolution structures.

Applications of this theoretical framework to the case of terrains, generic meshes and dynamic geometry will be presented in Chapters 4, 5 and 6 along with an in-depth discussion of design choices, implementation details and experimental results.

Chapter 2

Background and Terminology

This chapter introduces some basic terminology and background about meshes and presents an overview of the main simplification algorithms and multi-resolution concepts. Particular care is devoted to the Multi-Triangulation framework, introduced in Section 2.3.2, which is the basis for the comprehension of the following chapter.

2.1 Meshes

A *mesh* is defined as a collection of points (vertices) connected by polygonal faces which usually represent a piecewise linear approximation of a surface. Two very common subclasses are *triangle meshes*, which require all faces to be triangles and *tetrahedral meshes*, where every face is a tetrahedron.

An abbreviation is often prepended to the term *mesh* to indicate the geometrical space where the vertices are defined: in a 4D mesh the coordinates of its vertices live in $\mathbb{R}^4$. A 2.5D mesh informally indicates a mesh that can be projected on a plane without self-intersections, as in the case of terrains.

3D triangular meshes are a well established data structure in computer graphics and are generated by many acquisition devices and CAD applications. Dedicated hardware (GPU) is especially optimized for rendering triangular meshes: the operations of vertex projection, scan conversion, bilinear interpolation are highly optimized and can be performed in parallel.

A mesh is said to be *conforming* if all its faces join properly: at most two faces meet along an edge; if additionally the neighbor of every vertex is homeomorph to a closed ball the mesh is said to be *manifold*. Figure 2.1 illustrates a few examples of manifold, conforming and non conforming meshes.

The *accuracy* of a mesh M to represent a surface S can be defined using the local Hausdorff distance or any other suitable metric. In general, to obtain higher accuracy it is necessary to increase the *resolution* of the mesh, loosely defined as the inverse of the square root of the area of its triangles.

A *regular mesh* is defined by the uniform subdivision of a face into scaled copies of itself, while a *semi-regular mesh* is generated by uniformly refining the faces of an irregular mesh. In a

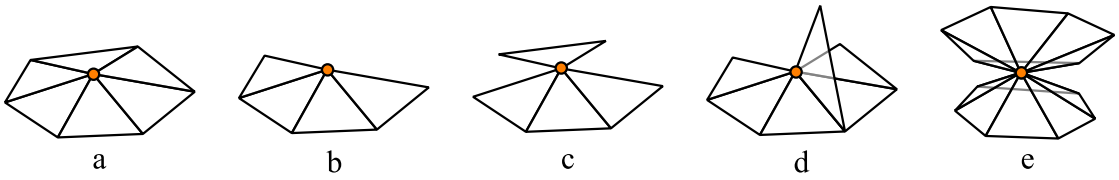


Figure 2.1: *Conforming and manifold condition in the neighborhood of a vertex: a) is manifold, b) is a manifold with boundary, c) and e) are conforming but not manifold, d) is not conforming.*

regular triangle mesh every vertex is *regular* (has valence six), in a semi-regular mesh only a few vertices are *extraordinary*. Encoding mesh connectivity and editing meshes is much simpler for regular or semi-regular meshes, though, in general, regular meshes require much higher resolution to reach the same accuracy.

2.1.1 Representation

There are essentially two classes of representation of triangle meshes based on the encoding of edges or triangles as primary elements.

Edge-based representation structures can represent generic meshes (i.e. meshes where the faces are not limited to triangles) and encode the connectivity of the mesh through local adjacency information stored on the edges. Examples of edge-based structures are the *winged-edge structure* [6], the *double-connected edge list* [83] and the widely used *half-edge structure* [80]. These structures allow for efficient navigation of the mesh elements, by traversing the neighborhood of either a face or a vertex, and provide simple mechanisms for local mesh modifications such as subdivision or edge-collapse. Only manifold meshes can be represented using edge-based structures.

A simple triangle-based representation, the *triangle soup*, is an array of vertices, where each triple of consecutive vertices defines a triangle. This representation is highly redundant: each vertex is replicated multiple times, one for each triangle containing it. A more compact and common way to represent a triangle mesh is the *indexed* structure: the vertex coordinates are stored in an array and each face is defined by the position (index) of its three vertices in the array. Graphic hardware is more efficient in the rendering of indexed structures where data transfer is minimized and it is possible to easily cache vertex transformations. Tetrahedral and k-dimensional meshes can be represented with analogous structures.

2.1.2 Appearance error metric

Given two meshes M and M' we would like to define an error metric $E(M, M')$ measuring their *similarity*. A convenient definition of similarity is application dependent, but in general, for rendering application, similarity of appearance is the natural choice. The appearance of a model M can be loosely defined as the set of raster images produced by a renderer over all possible views; we can measure the distance in appearance between two models by measuring the distance between the two sets.

Different metrics to measure the distance between two images have been proposed; the most commonly used [5] defines the distance between two images I_1 and I_2 (n pixels each) as the

average sum of the squared differences between corresponding pixels:

$$||I_1 - I_2|| = \frac{1}{n} \sum_n |I_1[i] - I_2[i]|^2$$

where the distance between pixels is measured using, for example, the difference of the two RGB vectors. The appearance metric would then integrate these differences over all viewpoints, but since it is unrealistic to evaluate *all* points of view, a suitable subsample is chosen.

The main advantage of this kind of metric is its obvious optimality, and an interesting feature is the removal of hidden details, due to occlusion culling. However, there are a few technical and conceptual difficulties with this kind of metric:

1. it depends on the rendering parameters, light settings and other details which are difficult to define mathematically;

2. it is difficult to properly sample the view point space;

3. rendering is a computationally expensive procedure: the large set of view points makes the evaluation impractical.

2.1.3 Geometric error metric

Another approach to evaluate the difference between two meshes is to measure the distance between corresponding points. In most cases an explicit correspondence is not available and we resort to measure the distance between closest points. The distance between a point v and a model M is defined as its distance to the closest point w on M:

$$d(v, M) = \min_{w \in M} ||v - w||$$

Definition 1 *The Hausdorff distance between two sets M_1 and M_2 is defined as:*

$$H_\infty(M_1, M_2) = \max \big(\max_{v \in M_1} d(v, M_2), \max_{v \in M_2} d(v, M_1) \big)$$

This metric measures the maximum deviation between two models: if $H_\infty(M_1, M_2)$ is small, every point of M_1 is close to a point in M_2 and vice versa. A useful variant of the Hausdorff distance measures the average squared distance between the two models:

$$H_2(M_1, M_2) = \frac{1}{A_1} \int_{v \in M_1} d^2(v, M_2) + \frac{1}{A_2} \int_{v \in M_2} d^2(v, M_1)$$

where A_1 and A_2 are the respective areas of M_1 and M_2. In practice both these metrics are approximated using a discrete set of points on the two meshes since the cost of an exact computation is prohibitively expensive [23].

Most of the published results which evaluate the quality of approximation algorithms use these metrics, or slight variations of them [71, 22, 43].

2.2 Simplification

Given a triangle mesh M at high resolution, the problem of producing a triangulation M' that represents the same surface with a smaller number of triangles and lower accuracy is known as *mesh simplification*. Simplification is used whenever an application does not require the maximum accuracy to represent a mesh, or it is not able to manage the full resolution because of storage issues or computational complexity. A typical example is interactive rendering where there is a bound on the maximum number of primitives that can be processed in each frame: this limit can be met by reducing the resolution of the mesh, at the price of an increased approximation error.

An important topic in mesh simplification is dealing with the topology of the object: depending on the application, it might be necessary to preserve the topology of the model in the simplified mesh, or it might be beneficial to reduce the topological complexity along with the geometric resolution.

A broad classification can divide simplification algorithms into *iterative*, which modify the mesh locally in a sequence of steps and *non-iterative*, which modify the mesh globally in a single step. In general iterative methods have a finer control over the resolution of the mesh while non-iterative methods allows for easier management of topological modifications.

2.2.1 Non-iterative methods

We can identify three classes of non-iterative simplification methods. A first class of algorithms uses *volumetric* techniques: a signed distance from the surface is stored on a 3D grid and the surface is extracted using a standard marching cube algorithm [74]. The extraction method ensures a manifold output mesh and manages automatically the topological simplification, while the resolution of the mesh is indirectly controlled by the resolution of the grid. Variations of this technique involve using an adaptive grid [3], which might help to prevent the loss of high frequency details, resulting from the uniform sampling.

A second class is based on *clustering*: merging groups of nearby triangles or vertices of the mesh. Rossignac and Borrel [99] replace clusters of nearby vertices with a single vertex removing the resulting degenerate triangles. They use a regular grid to create clusters of vertices, so that the error is bounded by the diagonal of the cells. The method is very simple and fast, it has no requirements on the input mesh, but the quality of the simplification is very poor and there is no control over the topology of the mesh. Different clustering methods [76] can improve substantially the quality of the simplification, but at the cost of a higher computational cost.

The *coplanar face merging* by Hinker and Hansen [50] partitions the model into coplanar or nearly coplanar clusters of triangles and merges them into a single polygon which is then triangulated; the algorithm preserves sharp features and topology. A similar algorithm, *superfaces* by Kalvin and Taylor [58], introduces an additional step, by simplifying the boundary of the cluster before re-triangulating, and allows bounding of the error.

A last class of *re-tiling* methods, initially proposed by Turk [117] generates a new set of vertices distributed over the original mesh, optimizes their position by analyzing the local curvature, and creates a new triangulation. These algorithms preserve the topology of the mesh and are especially suited for smooth surfaces, however, by using particular techniques for the placement of the vertices, also sharp features can be preserved.

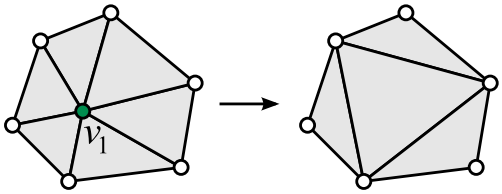


Figure 2.2: *Vertex decimation: a vertex is removed and the resulting hole re-triangulated.*

2.2.2 Iterative Methods

Iterative methods can be classified accordingly to the kind of local modification employed, which falls into two main classes: *vertex decimation* and *pair contraction.*

Vertex Decimation

In the method proposed by Schroeder et al. [107], at each step the algorithm selects a vertex, removes all the faces adjacent to that vertex and re-triangulates the resulting hole. The original algorithm was limited to manifold surfaces because the re-triangulation involved a projection of the local surface on a plane. The same projection ensured that the topology of the model was preserved.

Subsequent papers [106] removed the limitation about topology and used more accurate (albeit still approximate) error metrics [113, 61, 14, 64]. *Simplification envelopes* from Cohen et al. [24] manage to bound the global error and to preserve topology building two polynomial approximations of the surface at a distance $+\epsilon$ and $-\epsilon$ from the surface, and constraining the simplification not to intersect the envelopes.

Pair Contraction

Most currently used simplification algorithms are based on vertex pairs collapse: they replace all occurrences of v_i with v_j, remove the resulting degenerate triangles and optionally move v_i to a new position (see Figure 2.3). This operation modifies the geometry and the connectivity and might also change the topology. The algorithms in literature differ on a few details:

1. the preservation of topological constraints (if present)
2. the optimization of the final position of the collapsed vertex
3. the error metric used to determine which pair to collapse

The basic operation of a pair collapse does not require the starting mesh to be manifold, but it does not guarantee the end result to be manifold as well. A common strategy is to consider pairs which belong to the same edge as candidates for contraction, which is required (but not sufficient) to preserve topology: this strategy ensures topological simplification (closing holes) but it will not merge non-connected components. If topological preservation is a priority, an additional test discards all the candidates that would create a non-manifold mesh. General pair contraction, where the vertices v_i, v_j are not necessarily connected by an edge, but sufficiently close, has been

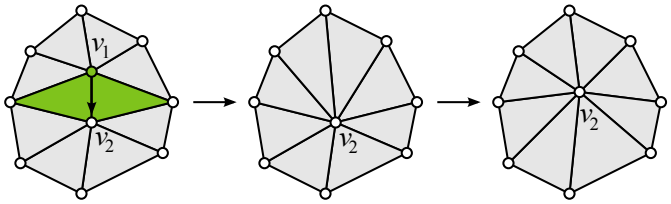


Figure 2.3: *Pair contraction: v_1 collapses over v_2, the resulting degenerate triangles are removed, the position of v_2 is optimized.*

proposed in [44, 91] allowing for merging of non-connected components. The contraction of two non adjacent vertices, however, always results in the creation of a non manifold region.

Most pair contraction algorithms follow a greedy approach to select the sequence of collapses: an error value is assigned to each collapse, associated with the degree of approximation resulting, and the lowest error pair is selected. The main difference between the various methods in literature consists in the method used to compute the error associated to a collapse.

Historically the first algorithm based on edge collapses was *mesh optimization* by Hoppe [55] which used a complicated energy minimization involving the Hausdorff distance of the original and simplified mesh (H_2 introduced in Section 2.1.3), the number of vertices and the length of the edges (to improve triangle quality). Additional edge splits and edge flip operations were considered. Later in *progressive meshes* [51] Hoppe proposed a variant where only the edge collapse operation was allowed and a different energy function was used to better preserve surface features. This method produces high quality results and allows for a multi-resolution model to be easily built; however, the quality of the simplification, comes at the price of very long computation times.

A different approach from Gueziéc [48, 49] defines a tolerance volume around the approximated surface and ensures that the original mesh remains in that volume. The volume information is also used to choose the position of the collapsed vertices so as to preserve the volume of the model. Additional constraints prevent changes in topology, badly shaped triangles or inverted triangles.

Ronfard and Rossignac [98] developed a local metric based on two terms: *local tessellation error* (LTE) which measures the rotation of the normals of the triangles involved in the collapse and *local geometric error* (LGE) which measures the maximum of the squared distance of the new vertex position from the plane defined by the involved triangles. The motivation of it is to avoid flipping triangles and moving vertices vertically with respect to the surface. Ronfard and Rossignac derive localized Hausdorff error bounds from their metric, and their method is fairly efficient compared to more exact algorithms.

The *quadric error metric* developed by Garland and Heckbert [44] is similar to LGE, also defining errors in terms of distances to sets of planes while using a much more efficient implicit representation of these sets. Instead of explicitly storing the planes, each vertex is assigned a single symmetric 4×4 matrix which can measure the sum of squared distances of a point to all the planes in the set. Under suitable conditions, the eigenvectors and eigenvalues of a quadric, accumulated over a smooth surface region, are determined by the principal directions and principal curvature of the surface. While the quadric metric is less precise in assessing the approximation error, the resulting algorithm can produce high quality approximations very rapidly.

The simplest way to choose the final position of the collapsed vertex is just to pick one of the two initial positions [98, 51]. Better approximation of the surface can be achieved by optimizing the final position, for example based on the conservation of the volume [49, 71], on the minimization of the distance from some set of planes [44] or on the shape of the generated triangles.

2.3 Multi-Resolution

In general not all applications accessing a mesh require the maximum accuracy in every part of the model. The resources required can be reduced by locally adapting the resolution of the mesh to the needs of the application, reducing the overall number of elements that must be handled.

A multi-resolution model is a data structure where alternative representations of an object at different resolutions are precomputed and stored. A mesh is then constructed on-the-fly by searching, among the various representations, those that best reflect the given requirements in accuracy and resolution. We will see in the following sections how iterative simplification (and coarsening) algorithms naturally define multi-resolution structures and determine their properties.

2.3.1 Characteristics of a Multi-Resolution Model

To evaluate a multi-resolution structure a number of properties play an important role:

Properties of the Input Mesh

Many multi-resolution structures impose constraints on the class of valid input data: they might require manifold meshes, or might perform badly if the samples are not uniformly distributed in space, or have a high topological complexity. In particular some structures works only on terrains (2.5D) or require data sampled on predefined regular patterns.

Expressive Power

The *expressive power* of a multi-resolution structure measures how many different representation of the model are available. A high number of different approximations means that the structure can better adapt the resolution of the representation to the required approximation.

Adaptation Speed

Strictly related to the expressive power, the *adaptation speed* of a multi-resolution structure measures the fastest possible change of resolution over space. For example, in the case of visualization algorithms it is important that the size of the triangles projected on the screen remains approximately constant, independently from the actual distance from the viewpoint. In this case the resolution of the mesh decreases in inverse proportion with the distance d from the view point and the speed is $o(d)$. Other applications such as collision detection, benefit from higher adaptation speed.

Compression Factor

The resolution needed to achieve a target accuracy is defined as *compression factor*. Data structures based on regular grids usually require a much higher resolution than those based on irregular triangulations. This property is important in rendering application where the frame rate is limited by the number of processed primitives.

Storage Space

A multi-resolution structure imposes some (often non negligible) storage overhead over the original model. It is important to keep this overhead to a minimum since disk access is usually the bottleneck for large models processing and visualization.

Performances

To obtain good performances on an actual computer, a multi-resolution structure must take into account restrictions and optimizations of the platform used. The algorithmic complexity of the extraction, the support for out-of-core data, the load balance between CPU and GPU, and the data optimization for the graphic card, are the most important factors in visualization applications.

Smooth Transitions

The possibility of gradually deforming the extracted mesh between different representations avoids abrupt changes (*popping* effect) during rendering is referred as *geomorphing*. A few structures allow easy implementation of this technique.

2.3.2 Multi-Triangulation

Iterative simplification algorithms allow to progressively decrease the resolution of a model through a sequence of *local* modifications of the mesh. Some modification are mutually *independent*, meaning that we can change the order in which they are applied, while other modifications will modify elements generated by some previous modifications and thus *depend* on those modifications. It is possible to pick any order of the modifications in the sequence, provided that we preserve the partial order introduced by these dependencies; in this way a variety of intermediate representation can be created, at different resolutions.

Multi-Triangulation [39, 94, 95] (MT) is a very general multi-resolution framework which encodes all partially ordered set of modifications over a base mesh. Most multi-resolution models based on triangle meshes can be interpreted as special cases of Multi-Triangulation, where restrictions on the rules for the local modifications, spatial structure, data encoding allow for different trade-offs in performances, space, expressive power, models supported etc. An in-depth presentation of the Multi-Triangulation can be found in [78, 37]; we will summarize the fundamental results and algorithms in the following.

Definition 2 *A* modification pattern C *is an operation that replaces a subset of a mesh* $old(C)$ *with a different set of faces* $new(C)$.

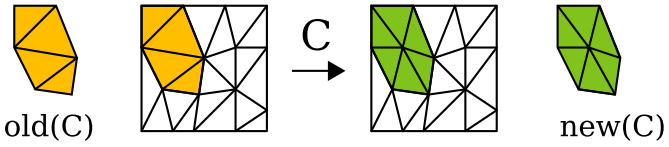


Figure 2.4: *A modification pattern: the orange triangles are replaced by the green ones*

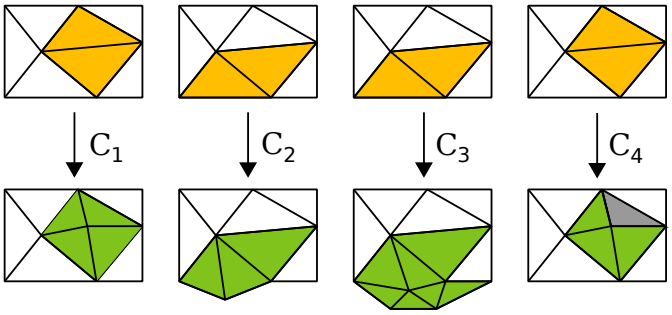


Figure 2.5: *Four modifications: only C_1 and C_2 are valid modification since C_3 violates condition 1 and C_4 condition 2 (a hole, in gray, is created).*

In Figure 2.4 the modification pattern C replaces the orange triangles ($old(C)$) with the green ones ($new(C)$). Obviously, a modification can be applied to the model at a certain stage of simplification only if the faces to be removed exist, and, by adding some conditions on the preservation of the boundary, we can assure that conforming meshes are preserved.

Definition 3 *Let Γ be a k-dimensional mesh in $\mathbb{R}^n$ and C a modification pattern. C is* valid *for Γ if $old(C) \subseteq \Gamma$. In addition C is* conforming *if:*

1. *the mesh obtained by deleting $old(C)$ from Γ (Γ_C) intersects $new(C)$ only in the common boundary of $old(C)$ and $new(C)$*
2. *the common boundary of Γ_C and $old(C)$ must be included in the boundary of $new(C)$ ($\delta\Gamma_C \cap \delta old(C) \subseteq \delta new(C)$).*

The first condition ensures that $new(C)$ does not affect Γ outside of the region occupied by $old(C)$, apart from the boundary. It is a geometric condition, related to self-intersection of a mesh. The second condition states that the boundary of $new(C)$ must match with the boundary of the hole created in Γ by the removal of $old(C)$ (see Figure 2.5 for examples of valid and invalid modifications). It can be proved that applying conforming modifications preserves the manifold properties of a mesh.

A sequence of modifications $C_1 \dots C_n$ applied to a mesh Γ_0 is valid (resp. conforming) if at each step C_i is a valid (resp. conforming) modification for Γ_i, result of applying C_{i-1} to Γ_{i-1}. A valid sequence of modifications is shown in Figure 2.6. The first and last modifications (C_0, C_6) transform the empty set into a mesh and viceversa.

Definition 4 *Given a valid modification sequence $C_0 \dots C_n$ we say that C_i* directly depends *on C_j if $new(C_j)$ and $old(C_i)$ have some faces in common.*

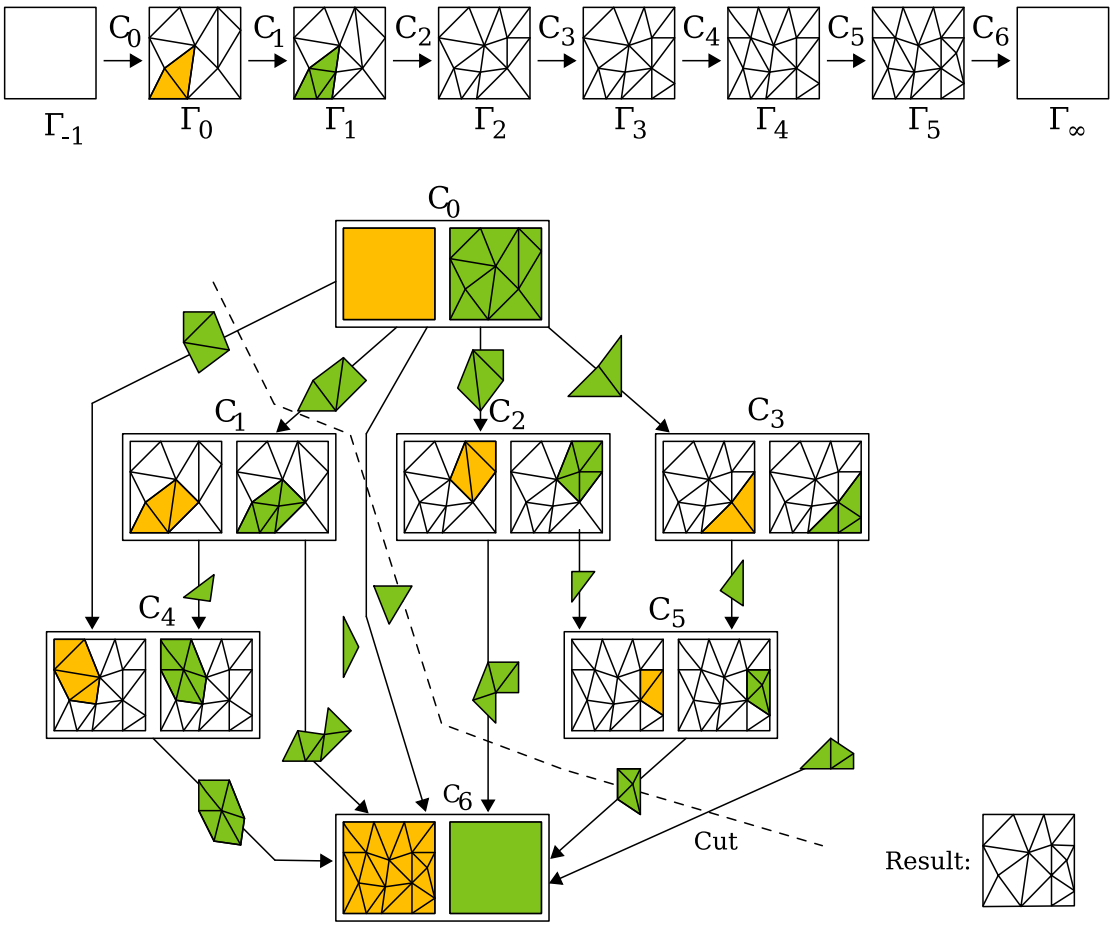


Figure 2.6: *Enriched MT.*

The transitive closure of the direct dependency relationship creates a dependence relationship between modification patterns. In order to avoid cyclical dependencies we introduce the concept of *redundancy*:

Definition 5 *A sequence is* non-redundant *if it does not recreate faces eliminated by some precedent modification:* $new(C_j) \cap old(C_i) = \emptyset, i < j$.

This means that a face is either present in Γ_0 or created once in some $new(C_i)$ and either present in Γ_{n+1} or deleted in some $old(C_i)$.

The dependency relation induces a partial order on a non-redundant modification sequence: $C_i < C_j$ if C_j depends on C_i. It can be shown that every permutation of the C_i that preserves the partial order is a valid sequence: in practice independent modifications can be applied in arbitrary order. We encode the direct dependency relation of a modification sequence into a *direct acyclic graph* (DAG), where nodes represent modifications and arcs represent direct dependencies between them: an arc (C_i, C_j) is created if C_j directly depends on C_i, see Figure 2.6. The non-

redundancy property ensures that there are no cyclic relations. We add two special nodes to the graph: C_0 (the root) which represents the modification going from the empty set to Γ_0, and C_∞ (the sink) representing the modification from Γ_{n+1} to the empty set.

The multi-triangulation structure in the DAG can be more easily pictured if we attach to each arc (C_i, C_j) the faces in the intersection of $new(C_i)$ and $old(C_j)$: it is easy to prove that the collection of arcs entering C_i is $old(C_i)$ and the collection of arcs exiting C_i is $new(C_i)$.

Definition 6 *A* cut *in the DAG is a subset of the tree such that for every node C_i all ancestors of C_i are in the cut as well. The* front *of the cut is the set of arcs that connect a node in the cut to a node outside.*

Equivalently a *front* can be defined as a collection of arcs so that each path from the root to the sink contains exactly one of such arcs. And the corresponding cut as the set of nodes encountered in the paths from the root to any arc of the front. The set of faces associated to the arcs in the front (the "result" on the lower left in Figure 2.6), creates a representation of the model, namely the representation obtained by performing all the modifications in the cut, in some dependency respecting order.

2.3.3 Query Operations on a Multi-Triangulation

Each arc in the DAG represents a portion of the higher resolution mesh with a certain accuracy corresponding to a certain geometric error λ. This value can be computed locally by the Hausdorff distance, or more conveniently be approximated from data computed by the simplification algorithm. For example, in the case of quadrics, the error can be derived by the quadric metric itself [66]. For dense meshes another simple approach just uses the average edge length of the triangles as error value.

A basic query operation on a multi-tessellation computes a cut in the DAG such that each arc in the front satisfies a certain accuracy requirement. In general the approximation required for a portion of the mesh does not depend only on the grade of approximation of the original surface, but also from factors such as the distance from the viewpoint, the intersection with the view frustum or a region of interest, the size of the triangles in the cell. It is then convenient to define extraction strategies based on a general function which associates an error value ϵ to each arc, by combining the above factors. The error associated to each node in the DAG is then the maximum of the error of the arcs exiting from that node.

We are interested either in the smallest mesh which satisfies the accuracy requirements, expressed as a minimum error threshold τ and eventually combined with a region of interest, (*minimum-size* queries) or the mesh which best approximates the error function and its size is within a given upper limit (*best-approximation* queries).

Minimum-Size and Best-Accuracy Queries

A solution to a *minimum-size* query can be computed by a greedy top-down traversal of the DAG, recursively advancing the front to include the node with the highest error until all the errors of the arcs in the cut are below the threshold τ. A cut is valid only if it contains all the ancestors of every modification so we must recursively check all the ancestors of every node to be added to the cut, and add them first. This procedure is computationally expensive and

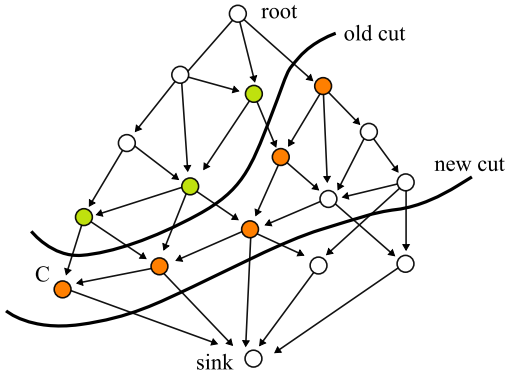


Figure 2.7: *Updating a cut: in green the candidates for exclusion, in orange candidates for inclusion. Notice how the inclusion of the modification C in the cut triggers a cascade of dependencies forcing the new cut to include many other modifications.*

not bounded since the inclusion of a node might trigger a cascade of ancestor inclusions (see Figure 2.7).

Error saturation is the procedure of modifying the error function associated to the nodes of a DAG such that the error of a node is always lower than the error of its parents. This guarantees, in the previous algorithm, that a node is never included in the cut before its parents and the expensive recursive check can be avoided. Applying error saturation on view-dependent error functions results in a conservative error estimation which will possibly increase the size of the extracted mesh. In Section 3.6.1 we show how error saturation can be performed in a visualization framework.

Additional constraints might be added to the extraction algorithm such as an upper limit to the size of the resulting mesh, the computation time or other resources. The previous algorithm already guarantees a uniform accuracy of the extraction according to the error metric by always including the greater error node first, and when combined with *error saturation* generates a valid cut at each step. It can then be interrupted as soon as one of the resource limits is passed thus answering to a best-accuracy query.

Updating a Cut

In many applications the error function changes gradually between one query and the next one, so that it is more efficient to modify the previous extraction by adding and removing nodes from a cut to meet the requested accuracy and resource limits: including a node in the cut will improve the accuracy, while removing a node will decrease the size of the extracted mesh and free resources.

All the candidate nodes for inclusion and exclusion (see Figures 2.8 and 2.9) are organized in two queues sorted accordingly to their error: the maximum error of $old(C)$ for inclusions and the maximum error of $new(C)$ for exclusion, which are updated as we modify the cut. The order of inclusion and exclusion operations might affect the result for non error-saturated DAGs.

For minimum-size queries, we perform all the allowed eliminations first and then all the necessary inclusions: the result is the smallest possible extraction thet meets the accuracy re-

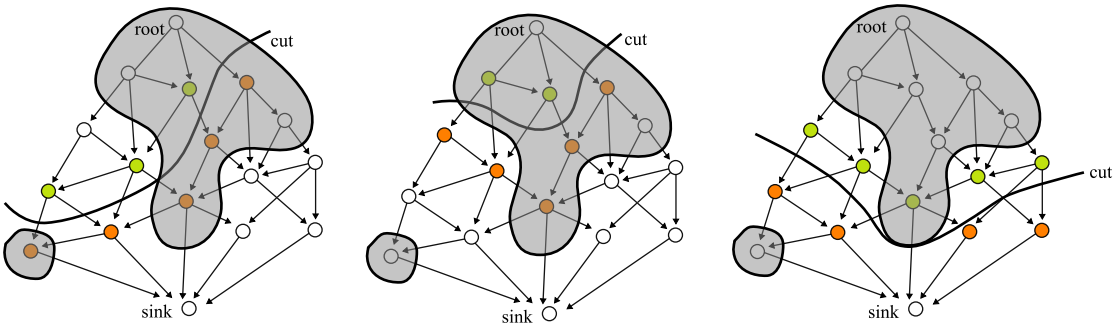


Figure 2.8: *Updating a cut with a non-saturated error function, in the gray area the nodes have an error higher than the threshold. Here we perform elimination first (in green) and then inclusion (in orange). Notice how a few nodes are first removed and then added back.*

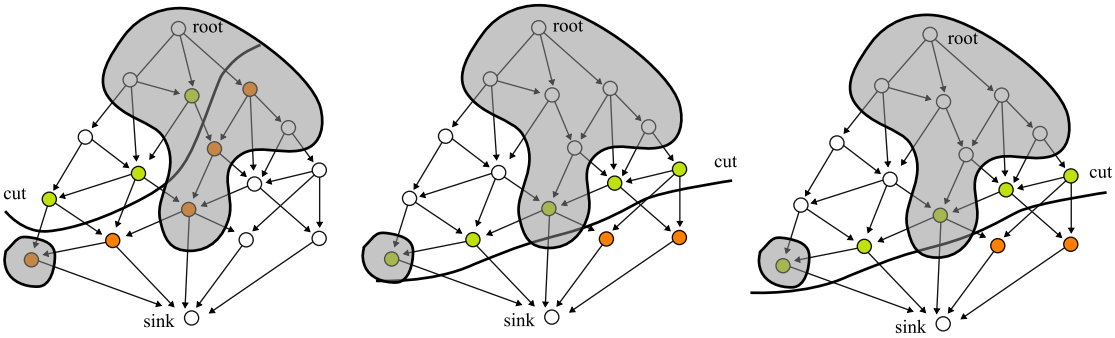


Figure 2.9: *Updating a cut with a non-saturated error function: here we perform first the inclusions (in orange). No operation is undone but the cut includes a few unnecessary nodes.*

quirement. However, it might happen that some eliminated nodes are re-included as ancestors of some necessary inclusion as shown in Figure 2.8. Reversing the order of operations will generate a sub-optimal cut, but minimizes computation time since it ensures that no operation is undone (see Figure 2.9).

For best-accuracy queries the preferred strategy is to balance the cut within budget constraints by alternating two phases:

1. perform operations from the inclusion queue (add nodes with the higher error into the cuts) until the budget limit is met.

2. perform operations from the exclusion queue (eliminate nodes with the smallest error from the cut) until removing a node would increase the maximum error.

The algorithm stops when no operation (inclusion or exclusion) can be performed.

2.3.4 Properties of the Multi-Triangulation

The following concepts allow us to define rigorously many of the properties introduced in Section 2.3.1 and thus to make sound comparisons between different multi-resolution structures.

Height

The *height* of a MT is the maximum length of a path in the DAG. A class of multi-resolution meshes has a logarithmic height if and only if, for any multi-resolution mesh M in the class, the ratio between the height of M and the logarithm of the total number of nodes in the DAG describing M is bounded by a constant.

Width

The *width* of a node C is is the number of faces in $new(C)$ and the width of a MT is the maximum width of its nodes. A class of multi-resolution meshes has a *bounded width* if and only if there exists a constant bounding the width of any multi-resolution mesh in the class.

Growth

The *redundancy* of a MT is the ratio between the total number of faces in a MT and the number of faces in the highest resolution representable model. Since a cut is also a MT the definition of redundancy applies also to a cut and we can define the *Growth* of a MT as the maximum redundancy of all the possible cuts. A class of multi-resolution meshes (e.g., the class of all multi-resolution meshes, built on the basis of a given type of modifications) has a *linear growth* if and only if there exists a constant b such that the growth of any multi-resolution mesh in the class is less then b. It is not difficult to prove that a bounded width implies a linear growth.

A linear growth is fundamental for the extraction of meshes with a time complexity which is proportional to the size of the output, while a bounded width and a logarithmic height are important for efficiently answering spatial selection queries.

Expressive Power

The *expressive power* of a multi-resolution model is defined as the ratio between the total number of possible cuts and the total number of faces in the multi-resolution mesh. In general, the expressive power is inversely proportional to the number of faces involved in a single modification. Smaller modifications also imply fewer dependency links, and these two properties imply more possibilities of combining the cells of the model into meshes. However, this also implies a bigger DAG and more computational power is required to extract a surface.

2.4 Historical Multi-Resolution Structures

Existing multi-resolution models can be described in the framework of Multi-Triangulation by finding the specific set of allowed modifications and how data structures can be mapped onto the DAG. It is then possible to evaluate and compare the properties defined in the previous section.

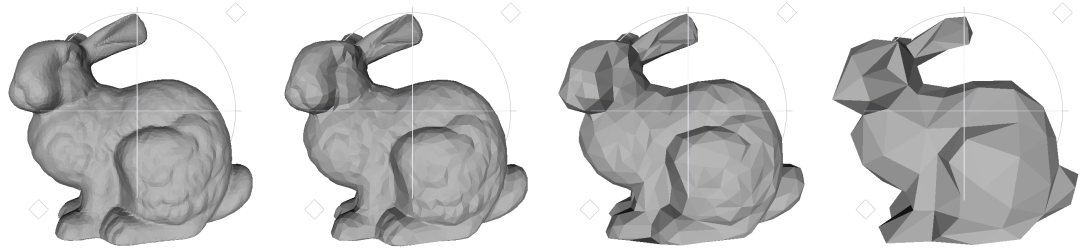


Figure 2.10: *An example of a layered model.*

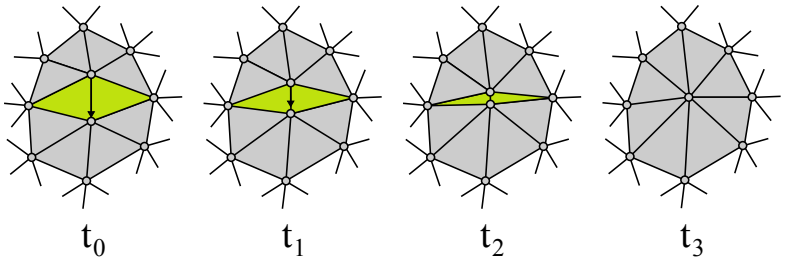


Figure 2.11: *The position of the vertices in an edge collapse move linearly in time resulting in a smooth transition.*

2.4.1 Discrete Level of Detail

The simplest method to create a multi-resolution structure is to generate a discrete set of decreasing resolution models at a predefined accuracy (Figure 2.10), and let the application select which level is more appropriate [40, 10]. The set of modifications corresponding to a discrete set of models (layered model) is simply a sequence of global modifications replacing all the cells of the mesh at once. The DAG is thus a simple list and this class of models has logarithmic height, linear growth (due to the fixed ratio in resolution), but not bounded width.

Advantages of this structure are its simplicity, the possibility to choose any simplification algorithm and to optimize and preprocess each level; however, the expressive power of this structure is very low and it provides no support for adapting accuracy in different parts of the model (the adaptation speed is zero), so that the model will be undersampled or oversampled, depending on the viewpoint. This is especially bad for terrains where the distance to the viewpoint varies greatly. Overall this method is simple and effective for models where the distance from the viewpoint is roughly the same for all the elements of the structure [10].

Smooth Transitions

A second problem in the naive implementation of layered models is the 'popping' effect when changing from one resolution to the other. A first method to obtain a smoother transition is to perform a smooth dissolve effect between the images rendered using two different resolutions and cross-alpha blending [41]. This requires of course to render the two levels.

Another alternative is to interpolate between the geometry of two consecutive levels (Fig-

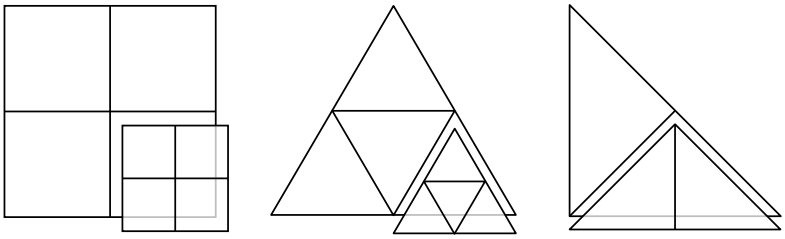


Figure 2.12: *Some 2D regular hierarchical structures: quad-trees, quaternary triangulation, right triangles splitting.*

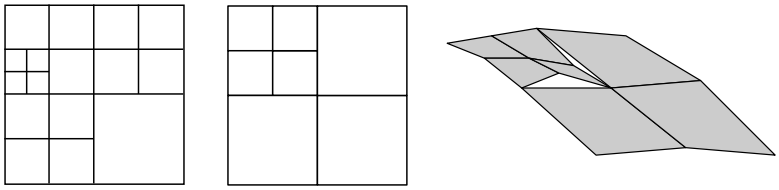


Figure 2.13: *A variable resolution extraction of a quad-tree; notice the cracks in the perspective view.*

ure 2.11). This technique is called *geomorphing* and was used originally in terrain visualization systems [25, 34, 75], and later on also for general meshes [51]. It requires a correspondence between the vertices of M_{i+1} (higher resolution) and M_i (lower resolution) while the interpolation is performed using the connectivity of M_{i+1} and interpolating the positions of the vertices.

2.4.2 Nested Models

Nested model structures are generated by the recursive subdivision of an initial domain (see Figure 2.12) to produce a variable resolution model. In the MT framework they are represented by trees where each node C_i represents the splitting of a cell into a complex. The atomic splitting operations C_i are not valid modifications in the MT setting: they modify the boundary of $old(C_i)$, usually splitting some edge. This results in a complex set of rules where a modification can be applied only together with some other modification in the neighboring cells. A nested model tree can be converted into a multi-triangulation by joining the operations so that the new modifications are in fact valid, e.g. in Figure 2.17.

The DAG corresponding to nested models based on fixed patterns in the recursive subdivision (see Figure 2.12) reduces to a list (all modifications in a level must be joined together to avoid cracks), turning them into layered models in practice. To allow for more expressive power, the strategy is to introduce a new set of modifications that preserve at least part of the boundary.

An exemplar case are the *quad-tree* based multi-resolution models. A *quad-tree* is the quaternary tree resulting from the recursive subdivision of a square domain into four squares. The tree can be pruned at different levels to obtain a variable resolution, but combining cells from different levels produces a non conforming mesh (see Figure 2.13).

Originally the problem was solved by adding the constraint that two adjacent leaves must differ at most for one level (*restricted quad-tree*) and then triangulating the cells accordingly to

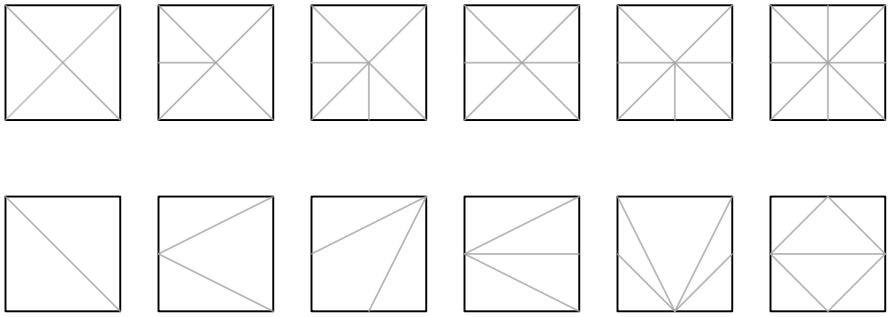


Figure 2.14: *Two possible sets of predefined triangulations for the cells of a quadtree.*

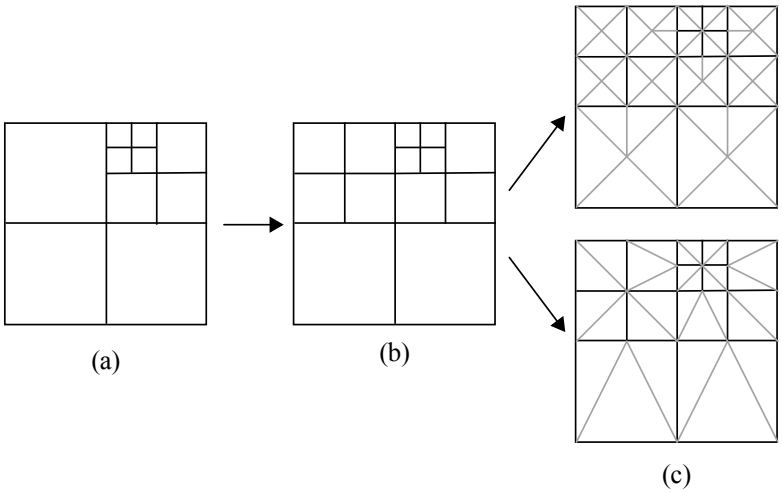


Figure 2.15: *The global procedure: unrestricted quad-tree (a), restricted quadtree (b), different possible triangulations (c) of the latter.*

predefined patterns to eliminate the cracks: two possible set of patterns are shown in Figure 2.14, respectively used in [101] and [129]. The choice of the patterns depends on the levels of the nearby cells.

This can easily be interpreted in the MT context by grouping the patterns into modifications rules. Figure 2.16 shows how this can be done for the second set of triangulation patterns. Similar structures based on quaternary triangulations have been proposed in the red/green triangle refinement algorithm by [47, 129].

A *right triangle hierarchy* is another important example of nested models: they are a binary tree based on the recursive split of a right triangle into two (see Figure 2.12(c)). In Figure 2.17 a hierarchy of right triangles is turned into a MT: notice how the "square" transformations implicitly encode the fact that the levels of the cell which are in a conforming composition can only differ by one.

All the meshes produced by triangulation of a quad-tree according to the patterns shown in Figure 2.14(a) can also be extracted from a hierarchy of right triangles, but the opposite is not true. Quite a few papers exploit this structure for the purpose of terrain visualization [67, 30, 31, 87, 70]. The main differences among them consist in the choice of which data is encoded implicitly

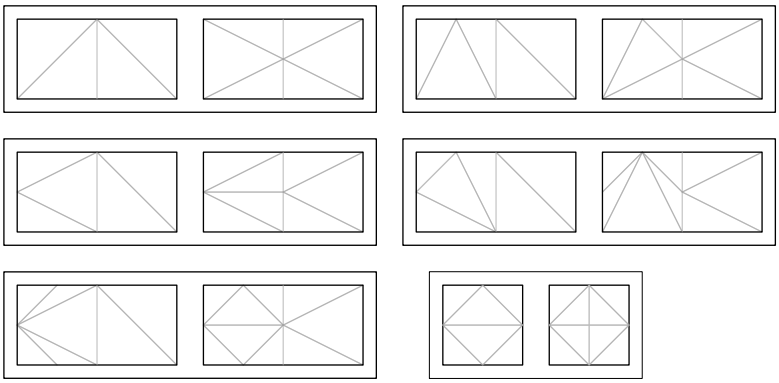


Figure 2.16: *The set of modifications that can be used to generate the same representation as the restricted quad-tree triangulate accordingly to [129]. Using only the bottom-right modification results in the right-triangle hierarchy.*

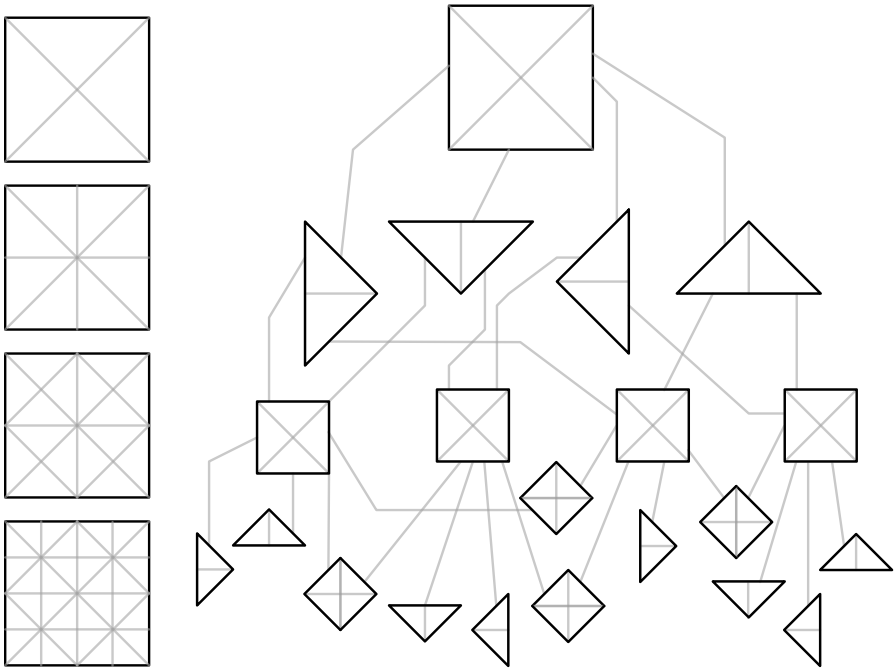


Figure 2.17: *A hierarchy of right triangles and its MT representation.*

and which explicitly and the support for optimizations such as storage strategies, geomorphing, triangle strips, etc.

The uniformly sampled data used by the regular nested models affects their compression factor (see Section 2.3.1), since more triangles are needed to adapt to the high frequencies in the terrain, compared to an irregular mesh. At the same time the regularity allows for very compact data structures, efficient compression and fast preprocessing times.

Figure 2.18: *Triangle refinement rules for hierarchical triangulations.*

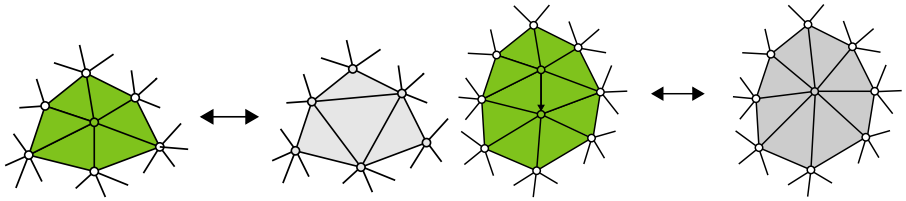


Figure 2.19: *Vertex decimation/insertion operation on the left, vertex collapse/split operation on the right, the part of the mesh affected is in green.*

Hierarchical Triangulations

Irregular nested models were originally developed to allow representation of irregularly triangulated height fields and achieve a better compression factor. A new set of refinement rules (in Figure 2.18) was proposed by [104], where up to four new vertices can be inserted in each triangle.

A more general setting for generating hierarchical triangulations has been proposed in [38] (*Hierarchical Delaunay Triangulation*) where a triangle is refined by inserting an arbitrary number of points and building a Delaunay triangulation. In order to obtain conforming representations from the multi-resolution mesh, it is necessary that the same points are inserted on the common edge of two adjacent triangles. This can be done by implicitly inserting new vertices based on an approximation metric.

2.4.3 Models Based on Iterative Simplification

An iterative algorithm naturally defines a multi-triangulation: each step in the iterative algorithm, usually either *vertex insertion/removal* or *pair contraction/split* (Figure 2.19) is in fact a *modification* (see Definition 2). The multi-triangulation encodes the most general multi-resolution structure based on an iterative algorithm and allows us to compare a number of different multi-resolution models, describing them under the same framework.

Sequential Models

Sequential models encode the sequence of operations of a refining algorithm (or the inverse sequence of a simplification algorithm), allowing for reconstruction of every intermediate step. This approach has a few important advantages: the reconstruction of the model is progressive, the number of operations required is proportional to the final resolution and the sequence of

operations can be easily compressed. The DAG of the corresponding MT is a simple list where each modification directly depends on the previous one. This approach naturally requires little space to store the DAG, but has very low expressive power since it is not possible to change the order of the modifications, and in particular this does not guarantee a logarithmic height.

The *progressive meshes* structure, introduced originally by Hoppe [51], encodes the sequence of operations of a pair contraction algorithm. A pair contraction atomic operation is the removal of a vertex, the computation of a new position and the deletion of degenerate triangles. The reverse operation describes which vertex is split, the new position of the two vertices and which triangles are to be inserted. A similar structure, proposed by Klein and Straßer in [62] records the output of a vertex removal algorithm.

Vertex Hierarchies

Vertex hierarchies are a natural result of pair collapse simplification algorithms: each modification (edge collapse) is associated with a vertex, so that dependencies in the DAG between nodes become dependencies between vertices. The two new vertices created in a vertex-split obviously depend on their parent and these dependencies are naturally organized in a binary tree. There is another set of "horizontal" dependencies due to triangles modified by nearby collapses, which are harder to correctly define outside of a MT framework.

Early works which introduce vertex hierarchies failed to encode them properly: Xia and Varshney [124, 123] adds direct dependencies between all vertices which lie on an edge with the vertex being split. This is redundant since those are not always *direct* dependencies. The same set of dependencies are explicitly stored by Gueziéc in a DAG. Redundant links produce overhead in the model structure and decrease the expressive power.

Hoppe [52] and Maheshwari et al. [79] both introduce unnecessary edges in the DAG and miss some dependencies, while Luebke and Erikson [77] just ignore the second set of dependencies. Even with an inconsistent DAG these algorithms ensure topological correctness of the extracted surface but some extraction may generate triangles never created by the original simplification algorithm, and thus with unknown error.

Almost all of the algorithms which depend on a refining or simplification algorithm cannot theoretically guarantee logarithmic height: this is due to the fact that they have little or no control over the simplification. For example, if a pair contraction algorithm collapses always an edge containing the last collapsed pair, it can create a long chain of direct dependencies and the DAG turns into a chain. In practice this never happens and these models can be considered to have logarithmic height.

Delaunay Pyramids

Delaunay pyramids are based on the iterative insertion of a set of vertices in a Delaunay triangulation: each inserted vertex and resulting local re-triangulation define a modification. In the model proposed by [8], the triangles created in each level are stored explicitly, together with all the 'conflicts' (intersections) between triangles in consecutive levels.

In a similar model presented by De Berg and Dobrindt [28] each level is obtained from the previous by eliminating a maximal set of independent vertices, so that each vertex creates an independent modification. The triangles appearing in each level are explicitly stored, together

with the modification: each new vertex is linked to the old and the new triangles.

Klein et al. [61] discard the explicit storing of the triangles and resort instead to dynamic re-triangulation of the mesh, greatly reducing the redundancy of the structure. Each vertex is assigned a geometrical error in a view-independent decimation pass and at rendering time the resolution of the model is updated by‘ computing the view-dependent error for each vertex. In this setting the DAG is completely implicit, and while the space requirement is minimized, the computational cost of an extraction is substantially increased by the on-the-fly re-triangulation.

Chapter 3

Visualization of Massive Models

Large datasets of complex geometric models are the output of a number of technologies, such as model acquisition, physically-based simulations and computer aided design (CAD). The size and complexity of these datasets have been exponentially increasing along with decreasing cost of data storage and processing power. Interactive visualization of these massive datasets requires a number of techniques to cope with hardware limitations, in particular computational power of CPU and GPU, data transfer bandwidth and latency among the various components of the system.

We divide these techniques into three classes:

1. *filtering*: techniques to discard all the information which is not relevant for the current scene
2. *data management*: techniques to retrieve the selected data and move it to the graphic hardware in the most efficient manner
3. *rendering*: techniques to fully exploit the computational power of the graphic hardware

We focus primarily on the visualization of large static polygonal models using rasterization techniques. Dynamic scenes present an additional layer of problems related to the necessary update of the spatial hierarchies and are usually segmented into a large number of objects instead of a single large surface, thus requiring a different approach.

3.1 View-Dependent Filtering

The size of massive models such as CAD environments, scanned urban data, cultural heritage models, or digital elevation models (DEMS) can be formed by more than 10^9 samples and their size keeps increasing with advances in sensor and information technology. In contrast, the resolution of current displays (currently at a resolution of two million pixels at $10 - 100Hz$) is somewhat limited by the human visual system and it is increasing at a much lower rate.

Interactive visualization can be seen as a filtering approach, where the relevant information needed for the current scene is efficiently extracted from the model and fed to the rendering

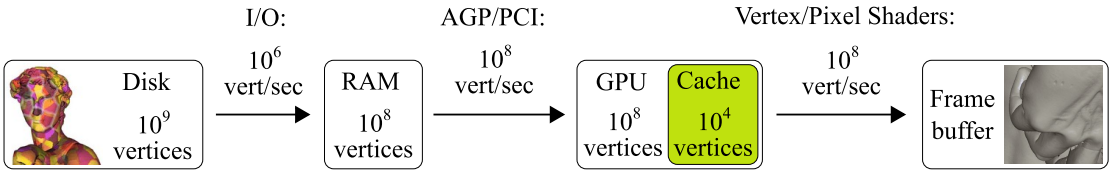


Figure 3.1: *Approximate bandwidth and storage limits for the rasterization pipeline.*

pipeline (see Figure 3.1). A number of techniques are needed to keep the application output-sensitive and not input-sensitive:

- *visibility culling*: the parts of the model that do not contribute to the image must be removed from the rendering pipeline as early as possible.
- *detail reduction*: the level of detail of the representation must be proportional to the resolution of its projection on the screen: in other words the size of the projected triangles must be approximately constant. This requires the resolution of the rendered mesh to decrease with the distance from the viewpoint.

The relative importance of these three topics and the efficiency of the specific adopted solution depends heavily on these characteristics of the dataset:

- *dimension*: terrains can be considered 2.5D, standard objects or scenes are 3D and animations 4D. The multi-resolution structure is usually simpler for low dimensional datasets.
- *depth complexity*: this value measures the ratio between hidden and visible surfaces from an average point of view and it is strictly related to the topology of the object. Occlusion culling is particularly important for high depth complexity models.
- *sampling*: the distribution of the vertices can be regular (as in a grid), dense (irregular but with uniform sampling density) or uneven. Data management and organization structures are usually specialized for one of the cases.

According to this classification we can distinguish a small number of typical datasets:

- *Local terrain model* (2.5D): flat, uniform regular sampling
- *Planetary terrain models* (2.5D): spherical, uniform regular sampling
- *Laser scanned models* (3D): moderately simple topology, low depth complexity, uniform irregular sampling
- *3D CAD models and urban environments* (3D): complex topology, high depth complexity, uneven sampling
- *Simulation results* 3D, 4D, complex topology, high depth complexity, high frequency details, irregular, sparse sampling

Rendering techniques based on rasterization have many points in common with structures developed to speed up ray tracing: hierarchical structures, approximate representations and memory management issues.

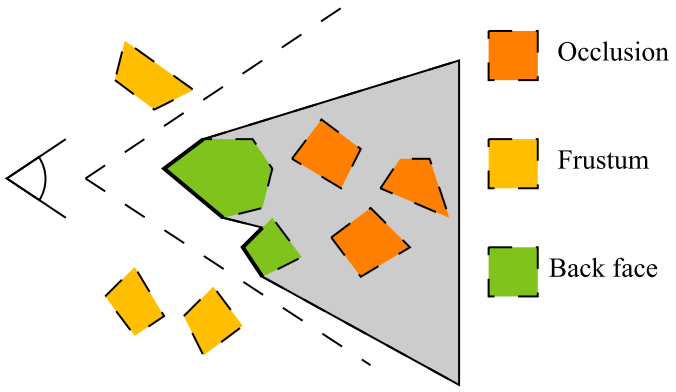


Figure 3.2: *Three kinds of visibility culling.*

3.1.1 Visibility Culling

In general only a fraction of the model geometry is actually visible while the rest is invisible either because it is occluded or outside the view-frustum. A straightforward strategy to reduce the rendering complexity of the scene is to compute the set of objects that contribute to the current image and thus reject large parts of the scene if possible. This process of computing a visible subset of a scene is called visibility culling and is an essential ingredient for high depth complexity models to make applications output sensitive. In general an exact computation of the set of visible primitives is too expensive, and algorithms aim for a *conservative* visibility estimation: hidden objects could be mistakingly classified as visible, but not vice versa, or an *approximate* estimation where visible objects might be culled if their contribution to the final image is small.

Back-face culling (see Figure 3.2) rejects triangles whose normal points away from the viewer and as such are occluded if the object is closed. View-frustum culling simply determines whether the triangle is fully outside of the view-frustum. Both techniques are trivially implemented by the GPU, as they are local, per-primitive operations. Occlusion culling removes primitives blocked by groups of other objects and is potentially a far more effective technique for high depth complexity models. It is however, a global approach since occlusion can occur between distant parts of the models, and its implementation is much more difficult. Occlusion culling approaches fall into two broad classifications: *from-point* and *from-region* visibility algorithms.

From-region algorithms spatially subdivide the scene into disjoint cells and for each compute a *potentially visible set* (PVS) of primitives: ideally the set of primitives that are visible from some view-point in the cell. This expensive computation is performed in a preprocessing phase; during rendering, the algorithms render only the primitives in the PVS of the cell where the view-point is currently located. This technique is mainly used for specialized applications, such as urban walking or building interiors, where the possible positions of the view-point are severely constrained and the nature of the scene naturally defines the spatial partition.

In from-point algorithms the visibility is computed from the current view-point, with little or no precomputed data, usually using occlusion queries provided by the graphic hardware, where the visibility of a group of primitives is tested against the current state of the z-buffer. The most common approaches require a front-to-back scene traversal where the visibility of a group of primitives is tested by checking the visibility of its bounding box. The main disadvantage

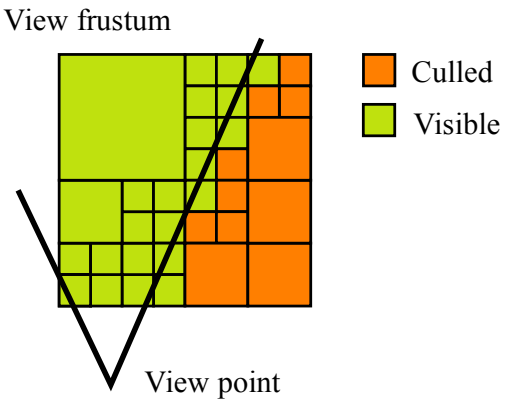


Figure 3.3: *Hierarchical view-frustum culling.*

of hardware occlusion queries is the latency between issuing a query and the availability of the result.

Visibility culling methods are optimized with the help of a spatial index that allows to quickly reject entire subtrees of the spatial hierarchy and a coherent spatial access to the model. A per-primitive implementation is not very efficient, since the geometry must still be loaded from disk, sent to the graphic card and then transformed so that only the shading computation is saved. It is possible to discard this geometry much earlier in the graphic pipeline by adopting a spatial hierarchy.

There are two major approaches to spatial indexing: spatial partitioning and bounding volume hierarchies. Bounding volume hierarchies conceptually organize geometric primitives in a bottom-up manner by hierarchically grouping bounding volumes of objects, while spatial partitioning schemes subdivide the scene in a top-down manner into a hierarchy of disjoint cells that contain the entire scene. Quite a number of spatial partitioning schemes have been proposed in the past. Hierarchical grids, octrees and kd-trees are the most popular methods, and kd-trees are usually considered the option of choice for massive models. More details can be found in [102] e.g.

3.1.2 Detail Reduction

In highly complex scenes a lot of detail might be too small to contribute significantly to the final image, for example whenever many primitives project to a single pixel. This is particularly true for graphics systems that are mainly geometry limited and not pixel fill-rate limited, which is the case for most current systems.

Multi-resolution techniques, described in Section 2.3, must adapt dynamically the resolution of the model, based on the distance from the view-point, and provide the bulk of the complexity reduction where the model has not a very high depth complexity. The main objective of the general multi-resolution structures based on irregular triangulations was to compute on the CPU the minimum number of triangles to render each frame, so as to minimize the amount of computation needed by the graphic card.

In recent times the vast majority of view-dependent level-of-detail methods were all based on multi-resolution structures where refine-coarse decisions were taken at the triangle/vertex

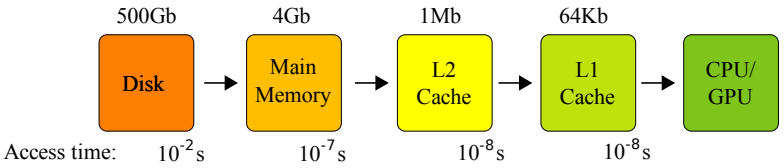


Figure 3.4: *Memory hierarchy: lower memory levels are larger in size and slower in data access speed.*

primitive level. These approaches result in very large trees that need to be traversed with two major consequences: they require a lot of CPU power, since for each primitive some error function needs to be evaluated, and random access to sparse memory to collect the selected primitives and to assemble the simplified model. Detail selection then becomes the bottleneck of the entire rendering process.

Nowadays, consumer graphics hardware is able to sustain rendering rates of tens of millions of triangles per second; this has two important implications: the CPU workload of the adaptive extraction must be reduced to a few cycles per triangle, and the size of the scene increases to millions of triangles. Managing and storing very large dependence graphs at run-time becomes inefficient due to random-access and cache coherence problems. Another important change in GPU architecture is the dramatic increase in speed when data is transferred, stored and rendered in blocks on the video memory, and the primitives are stripified and optimized to exploit the vertex cache on the graphic card.

Both the computational cost involved in selecting the best set of triangles to be rendered and the complexity of assembling geometry in a GPU friendly format, prevent these methods from exploiting the full power of the GPU. Moreover, due to the parallel nature of the GPU, the processing rate of GPUs is increasing much faster than that of CPUs, thus widening the gap. Solutions to overcome these bottlenecks are shown in Section 3.4.

3.2 Data Management

Data management techniques optimize the data transfer between various parts of the system: main storage, RAM and GPU memory. Efficient transfer of data to the processing units has become increasingly important given the fact that memory bandwidth and data access speed grow at significant lower rates than processing power.

System architectures employ memory hierarchies, where each level of memory serves as a cache for the next level (see Figure 3.4), to reduce data access time and bridge the increasing gap between computation performance and memory bandwidth. The access times of different levels of the hierarchy (L1/L2 caches, main memory, disk and network) vary by orders of magnitude, so that the number of cache misses becomes one of the most important performance issues. Moreover, whenever a cache miss occurs, a large block of data, containing the accessed data, is moved between memory levels. Since data is moved in a large blocks, it is critical to store data that are likely to be accessed sequentially close to one another.

Three data management techniques are commonly employed to optimally use the available memory bandwidth: out-of-core caching techniques, data layout methods and compression methods.

3.2.1 Out-of-Core

External memory (or out-of-core) techniques minimize the amount of virtual memory needed by organizing the data such that it is possible to keep in main memory only the fraction of the data currently processed, and are often enhanced with prefetching methods to further reduce data latency. There are two main classes of out-of-core processes: batched processes, which stream data from disk through memory in a fixed order, and online processes, which are basically a specialized version of the general purpose virtual memory system managed by the OS and impose no restriction on the data access order.

In the first strategy, also known as *streaming meshes* [57], the portion of the mesh kept in memory resides in a sliding window over the data stream: vertices are added and removed from a partial, but seamless reconstruction of the mesh while the window spans the entire model. This technique has the advantage of extreme memory coherence, sequential data storage access and a very simple data format at the price of fixed access order to the data, and it is best suited for offline mesh processing (which can even be pipelined for increased throughput) and long term storage.

The second strategy can effectively reduce the number of disk reads by organizing data in a disk-efficient block structure and by prioritizing the cache based on the specific application needs: efficient queries require to read large blocks of contiguous data due to the high latency of disk access. An example of a block-based out-of-core algorithm was proposed by Cignoni et al. [21] in the context of mesh simplification. The biggest challenge resulted in the complexity of indexing and referencing adjacent data in nearby blocks.

In general, out-of core techniques are *cache-aware* since they need to know the fundamental parameters of the cache: the size of the main memory and the size of the most efficient block size. For a more extensive introduction to out-of-core processing relevant to visualization algorithms, see the IEEE Visualization 2002 course notes [109].

3.2.2 Layout Techniques

Coherent access to the data has significant impact on the performance, since the system architecture performs a block transfer whenever a cache miss occurs between two adjacent levels of the memory hierarchy. This block-fetching mechanism assumes that runtime applications access data coherently. In fact efficient rendering methods (either rasterization or ray tracing) usually access vertices and triangles coherently in the geometric space, in wich the triangle meshes are embedded.

Triangular meshes have a natural 2D manifold structure while storage requires a 1D linear data representation. A mapping of 2D data into 1D linear layout cannot preserve contiguity: regions that are close in the 2D geometric space can be stored far away in the 1D representation. The number of cache misses can be reduced by using a mapping that stores nearby vertices in the mesh as close as possible in the 1D layout.

An example of a data layout method is the reordering of the sequence of vertices processed during rasterization in a triangle strip to reduce the number of GPU vertex cache misses [54], where a good rendering sequence can improve the rendering performance up to a factor of six. This is considered a cache-aware technique since the sequence is optimized with respect to the GPU vertex cache.

Cache-oblivious layouts [127] do not require any specific cache parameters such as block sizes, and minimize the number of cache misses for different sizes of the cache, while providing the same performance improvements. Since the optimal cache block size varies with the various level of the cache hierarchy and the type of system, using cache-oblivious layouts allows for significant performance improvements without the need to change code or reprocess data.

3.2.3 Compression

Compression techniques allow to trade computational power for storage space and bandwidth, and are usually employed to reduce the bandwidth bottleneck while accessing external storage and especially network storage. Mesh compression techniques can be classified in two main classes: techniques which preserve the connectivity of the mesh and techniques which re-mesh the model into a semi-regular mesh before the compression stage. The second class takes advantage of the degrees of freedom in approximating a surface with a mesh to generate meshes with high regularity and the fact that regular connectivity can be compressed very efficiently. These methods, however, can only be applied when the application allows for lossy compression or the topological complexity of the surface is not too high.

The *geometry data*, the coordinates of the vertices, can be compressed through *quantization* and/or *prediction* techniques. The positions of the vertices are usually stored in floating point representation, however, if some reduction in precision can be tolerated the positions of the vertices can be snapped on a grid, thus reducing the number of bits necessary to encode it.

Prediction techniques strive to guess the position of a vertex from the already decoded neighbors while assuming some regularity of the mesh; the predicted value is used for entropy coding. The main drawback of predictive methods is that the mesh traversal is dictated by the connectivity scheme, which is in general independent of the geometry. The performance of these compression algorithms is further penalized from the fact that the geometry information dominates the connectivity information.

A new class of recent *geometry-driven* compression techniques [42] achieve better overall results encoding the positions of the vertices through recursive subdivision of space and encoding the updates in the connectivity required by each subdivision. A comprehensive survey of the many algorithms for generic mesh compression can be found in [2].

In the contest of large model visualization an important property is random access to compressed meshes. Recent techniques [126] split the mesh into a set of clusters and compress each cluster independently.

3.2.4 Prefetching

Prefetching techniques aim at reducing data access latency by predicting future data requests and loading it into memory before it is actually requested by the application. Usually prefetching is implemented with the help of multithreading where one thread is in charge or performing all the disk I/O without actually blocking the rendering thread.

The actual prediction is highly application dependent, but in general relies on temporal coherence resulting from a smooth change of view-point and direction change. For example, in typical terrain exploration tasks the expected future position of the viewer can be extrapolated with good accuracy based on the last few positions. A common strategy employs a prefetching

thread which executes the same refinement algorithm as the rendering code, taking as input the predicted camera position instead of the current one. When the refinement terminates, instead of rendering the patches or binding the textures, it simply checks whether the required graphics objects are in the cache. If not, it advises the operating system kernel that the pages containing their representation will likely be accessed in the near future.

3.3 Hardware Accelerated Rendering

The increase in computational power of the GPU allows for interactive rendering of scenes with millions of triangles. To do so, however, it is necessary to transfer data efficiently to the GPU, optimize the rendering sequence to take advantage of the various caches and encode the geometric data in a specific format. We will give an overview of OpenGL methods to use graphic hardware to the maximum of its possibilities. DirectX provides similar capabilities.

3.3.1 Data Transfer

The first method employed for transferring geometric information to the GPU is called *immediate mode*, where a call to glVertex() is done for each vertex:

```
glBegin(GL_TRIANGLES);
for (i = 0; i < numtriangles; i++) {
  glNormal3fv(normal[i]);
  // other per-vertex attributes
  glVertex3fv(vertex[i]);
}
glEnd();
```

This method, though flexible, is very CPU intensive (since it requires a function call for each vertex) and inefficient in data transfer because the AGP or PCI-express bus is optimized for large block transfers.

When the geometry does not change too frequently between frames, *display lists* can be used: this technique stores the data and OpenGL commands on the GPU memory and amortizes the transfer cost between several frames. OpenGL drivers allow to transfer groups of primitives using a single API function, *glDrawArrays*:

```
glVertexPointer(3, GL_FLOAT, 0, &vertices[0]);
glEnableClientState(GL_VERTEX_ARRAY);
glDrawArrays(GL_TRIANGLES, 0, num_vertices);
```

This method has both the advantage of reducing the CPU load and dramatically increasing the data transfer rate, but requires an expensive 'triangle soup' data structure.

OpenGL supports also block transfer for indexed mesh data structure, where three consecutive indices in the *indices* array create a triangle:

```
glDrawElements(GL_TRIANGLES, indices.size(),
               GL_UNSIGNED_INT, &indices[0]);
```

This has the obvious advantage of reducing the amount of transferred data, yet all the geometry needs to be transferred to the graphic card for each frame.

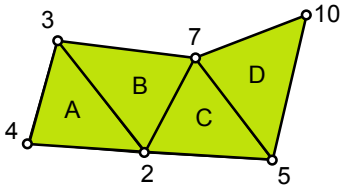


Figure 3.5: *Diagram of four triangles, A, B, C, and D, created by the sequence of vertex indices 4, 3, 2, 7, 5, 10.*

The *vertex buffer object* (VBO) extension allow to store the vertex coordinates and the triangle indices in the GPU memory:

```
/* allocate a new vertex buffer object */
glGenBuffersARB(1, &vbo_id);
glBindBufferARB(GL_ARRAY_BUFFER_ARB, vbo_id);

/* copy vertex data (vbo_data) into the vertex buffer
   the data is moved to the graphic card */
glBufferSubDataARB(GL_ARRAY_BUFFER_ARB, offset,
                   vbo_size, vbo_data, GL_STATIC_DRAW_ARB);

/* allocate a index buffer */
glGenBuffersARB(1, &ebo_id);
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, ebo_id);

/* copy the indices into the buffer
   the data is moved to the graphic card */
glBufferDataARB(GL_ELEMENT_ARRAY_BUFFER_ARB,
                ebo_size, ebo_data, GL_STATIC_DRAW_ARB);

/* finally draw the primitives, no transfer needed */
glDrawElements(GL_TRIANGLE_STRIP, indices_size, GL_UNSIGNED_INT, NULL);
```

3.3.2 Cache Optimization

The computation needed to process a vertex can be shared among triangles by using *triangle strips*. A triangle strip is a sequence of vertices where the first three vertex indices (0, 1, 2) in the sequence construct a triangle, and any subsequent vertex v_n creates a triangle with the vertices v_{n-1} and v_{n-2} (see Figure 3.5); each vertex processing is thus shared among three triangles and the index size is reduced to about one third. Generating a good decomposition of a mesh into long triangle strips is computationally intensive and must be done in a preprocessing step. Speckmann and Snoeyink [114], for example, compute the triangle strips for triangulated irregular networks by creating a spanning tree of the dual graph, and then extract the strips from a depth-first traversal of the tree.

The hardware implementation of this rendering sequence requires only a small First-In-First-Out vertex cache for previous vertices. Enlarging the size of the cache improves the performance of the system since a higher percentage of the already transformed vertices can be reused in subsequent triangles. Recent algorithms such as [7] and [118] optimize not only the length of the triangle strips but also the cache coherence.

OpenGL supports vertex arrays combined with triangles strips:

```
glDrawElements(GL_TRIANGLE_STRIP, indices.size(),
               GL_UNSIGNED_INT, &indices[0]);
```

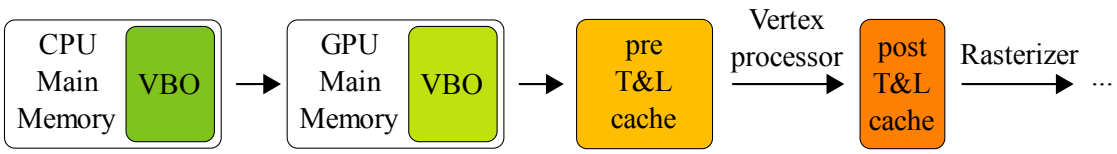


Figure 3.6: *Memory stages and vertex caches.*

To reduce the number of function calls it is advisable to join the triangle strips using degenerate triangles, which are automatically discarded by the driver.

3.3.3 Data Format

Encoding the vertex coordinates and triangle indices in the *correct* format improves the frame rate by avoiding unnecessary conversions and making optimal use of the specialized hardware. Triangle indices should be encoded using unsigned short integers (16 bits) to reduce memory requirements:

```
glDrawElements(GL_TRIANGLES, indices.size(),
               GL_UNSIGNED_SHORT, &indices[0]);
```

or the driver allowed to do itself the conversion by drawing the triangles in batches of less then 2^{16} vertices:

```
glDrawRangeElements(GL_TRIANGLES,
                    indices.min(), indices.max(),
                    indices.size(),
                    GL_UNSIGNED_INT, &indices[0]);
```

Since the coordinates of the vertices are the bulk of the data it can be advantageous to compress them into 16 bits: this reduces storage space and bandwidth requirements, virtually enlarges the (pre-transform) cache and does not require too much processing power. Coordinates can be stored as short integers instead of floats, and quantized relatively to the bounding box of the object. A different approach which uses a simple vertex shader program was presented in [13]. It should be noted that lossy compression can be problematic for replicated vertices: the algorithm must ensure that the two quantized vertices result in the same exact coordinates, otherwise crack artifacts will appear in the mesh.

In conclusion, under the assumption of a vertex transform limited application, the most efficient data structure consists of small (order of a few thousands) batches of cache coherent triangle strips, stored in VBOs. The difference in performance with respect to immediate mode is very large: a GeForce 8600 in immediate mode is capable of rendering about 8 million triangles, while the figure rises to 160 millions when using VBO. Adding cache coherence, triangle strips and data format optimization can increase the performance by more than two times.

3.4 Patches as Primitives

Many techniques have been proposed to reduce the workload of the CPU and generate GPU optimized geometry by adapting previous algorithms: a first step was to amortize the cost of the

extraction over several frames [67, 30, 52]. RUSTIC [89] and in CABTT [65] are extensions of the ROAM algorithm and improve rendering performance by caching geometry on the GPU as vertex arrays and rendering them as triangle strips. The main difference between them is that RUSTI clusters are static subtrees of the bin-tree hierarchy which are preprocessed and saved, while CABTT dynamically generates and caches subtrees. The total number of triangles per frane at a given error treshold is increased due to the use of precomputed clusters, however the overall rendering performance is vastly increased by the better usage of the GPU.

3.4.1 Patch-Based Structures

A different class of methods employs a patch-based approach: the granularity of the primitive is moved from points or triangles to small contiguous portions of a mesh. The increased granularity has the obvious advantage of reducing the number of per-triangle CPU operations. An increased granularity results also in a reduced expressive power, since the extracted meshes will be suboptimal with respect to a generic MT. Given the constant *adaptation speed* (Section 2.3.1) required by visualization algorithms, the overhead will be only a constant factor which depends on the size of the patches.

On the other hand, the batched structure allows for aggressive GPU optimization of the triangle patches: local coherence can be exploited and triangle strips can be easily used; this results in a huge boost in terms of performance of the graphic card: the increase in rendering rate (one order of magnitude of triangles per second at least) largely compensates for the suboptimal extraction of the mesh. Larger updates might also result in more evident *popping effects*, however, the very small average dimension of the triangles, due to the much higher resolution attainable, greatly reduces the effect. Geomorphing techniques might still be applied to remove popping.

3.4.2 Out-of-Core and Vertex Replication

Patch based structures are naturally suited for out-of-core management: the system loads in memory only the actually needed blocks. An example of a patch-based structure for out-of-core editing and simplification of huge meshes based on an octree is presented in [21]. The main issue in this kind of structures is represented by the vertices shared among patches which are stored in only one patch and need to be referenced in the others. At run time an incomplete representation of the mesh is built where vertices not loaded are marked as unreadable. Replication of the vertices would require to enforce consistency of the replicated vertex attributes through all the editing phase.

However, in a rendering application patches must be stored in a GPU ready fashion as a small indexed mesh, to avoid loading nearby patches and the process of merging vertices, thus requiring vertex replication of the boundary vertices. The storage cost of vertex replication is in the order of the square root of the number of vertices, thus affordable for reasonably sized patches; however, care must be taken to ensure consistency of the replicated vertex attributes. Most of the complexity of the implementation of patch-based structures lies in the management of the referenced or replicated vertices.

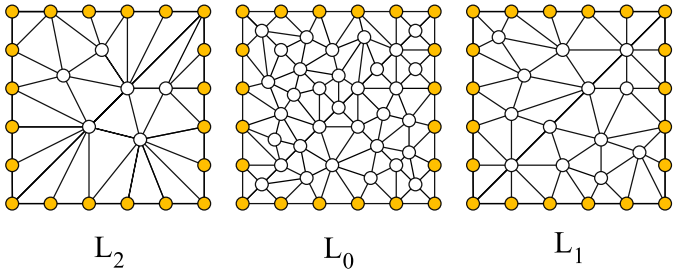


Figure 3.7: *By marking and preserving the boundary during simplification patches of different levels always join seamlessly.*

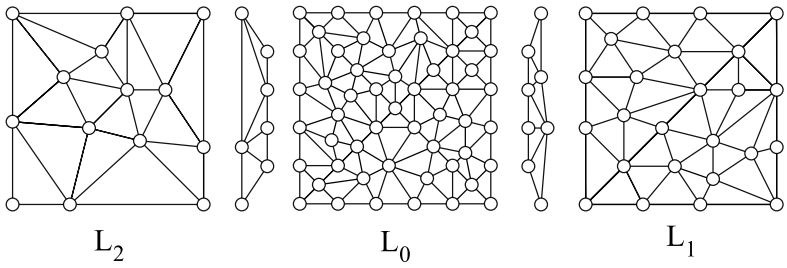


Figure 3.8: *Stitches: unconstrained simplification produces non matching boundaries, a set of small triangle strips (one for each couple of neighboring patches of different levels) fills the cracks.*

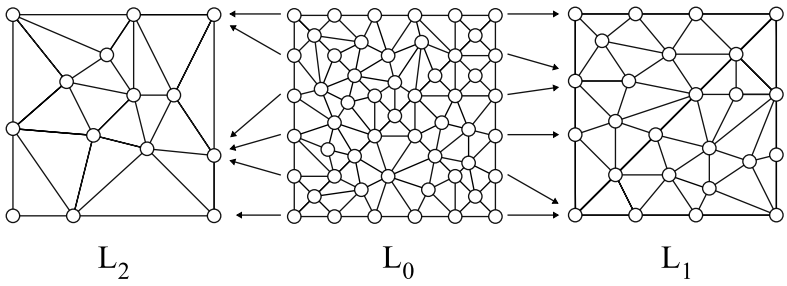


Figure 3.9: *Geomorphing: the position of the boundary vertices (and possibly the interior) interpolates smoothly between different levels.*

3.4.3 Boundary Management Problem

The main challenge for multi-resolution patch-based triangulation algorithms is to avoid cracks while assembling blocks from different levels in the hierarchy. Three different solutions have been proposed to solve this problem:

- *marking*: during simplification, boundary vertices are marked as read-only and preserved: in this way two neighboring blocks always connect seamlessly (see Figure 3.7). Naive implementations of this method allow a boundary to remain unchanged along many levels of the hierarchy, affecting the scalability of the model.
- *stitches*: the boundaries from neighboring blocks of different resolutions are connected

through a strip of thin triangles, either pre-computed or generated on the fly (see Figure 3.8). This requires additional drawing calls, selecting the correct strip for each boundary and assembling the vertices of the strip (which belongs to different patches). This makes this approach viable only on regular structures.

- *geomorphing*: vertex attributes are interpolated between different levels, based on the distance from the boundary or the distance from the view-point (see Figure 3.9). Methods based on the distance from the boundary require a regular structure or some additional data and maintain much of the stitches complexity. Methods based on the distance of the vertex from the view point require an error driven extraction algorithm: the resolution of the model depends from some global function, which must be computed per vertex. This can be computationally expensive and makes it difficult to implement error metrics based on the actual simplification error.

3.5 Batched Multi-Triangulation

An elegant and simple solution to the boundary management problem is the patch-based approach to the Multi-Triangulation framework, introduced in Section 2.3.2. The MT was designed as a general way to formalize and implement multi-resolution models based on simplicial complexes and atomic update operations. In theory there is no restriction to increase the scope of the update operation to a small patch of simplices. In practice it is not trivial to enforce bounded width and limited height, as defined in Section 2.3.4, and at the same time to ensure that the boundary of the patches will be eventually simplified in another level in the hierarchy. A general solution to this problem is the Batched Multi-Triangulation (BMT) which is based on the concept of *V-partitions*.

3.5.1 V-partition

A *V-Partition* is a sequence of coarser and coarser partitions over a mesh; they can be simplified and merged together to form a well-behaving BMT. Let H be a partition of a model S into n disjoint regions $h_1, \ldots, h_n$ and $G = g_1, \ldots, g_t$ be a partition of the same model into t disjoint regions. We denote with $H \cap G$ the *intersection* of the two partitions defined as:

$$H \cap G = \bigcup_{i=0..h, j=0..t} \{h_i \cap g_j\}.$$

The construction of a multi-resolution structure over a model S starts with the definition of a sequence of coarser partitions $H_0, \ldots, H_n$. In the subsequent step we create the partition $L_0 = H_0 \cap H_1$ by intersecting the two finest partitions and splitting the model into patches, one for each cell of L_0 (see Figure 3.10).

Then we collect, for each cell of H_1, all the patches in L_0 resulting from its intersection with H_0, merge them into one large patch, simplify its representation by preserving the boundary, and split it into a set of new patches by intersecting it with H_2. Overall, this creates a set of coarser patches that correspond to the cells of $L_1 = H_1 \cap H_2$. We apply this coarsening algorithm to all levels of the V-partition. The elements of the partition H_{k+1} can be assembled either using the elements of L_k or the elements of L_{k+1}.

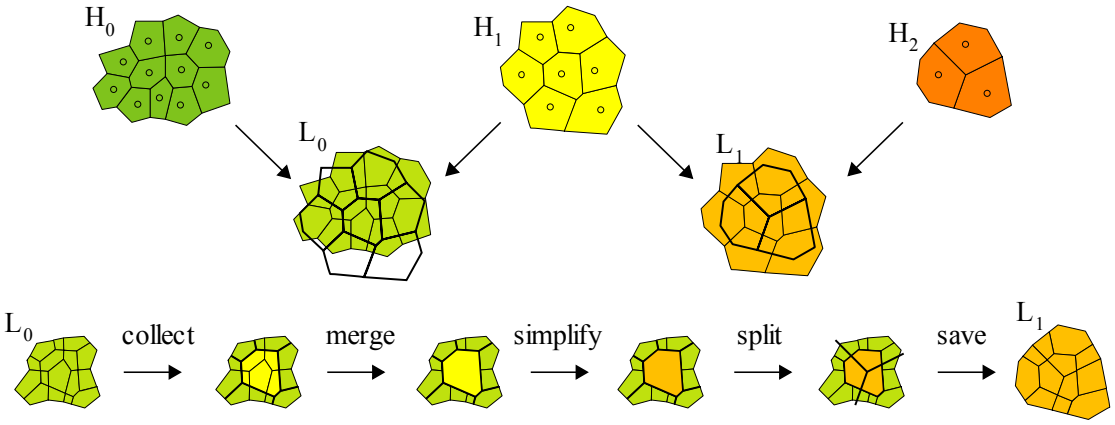


Figure 3.10: *Hierarchy of the V-partition (left) and the three steps of the coarsening algorithm (right).*

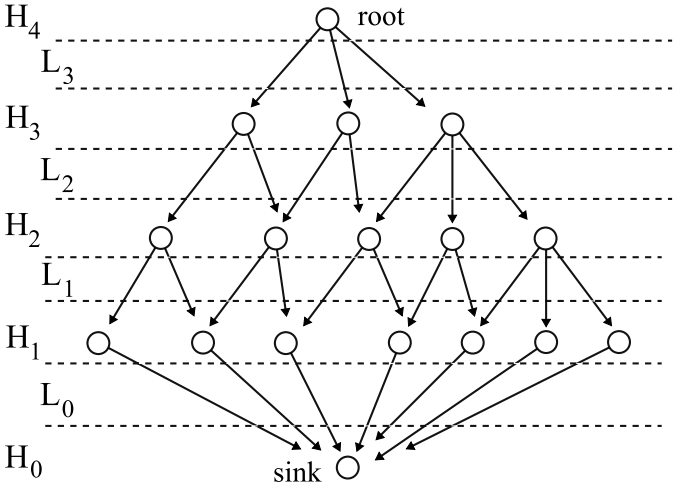


Figure 3.11: *DAG resulting from a V-partition.*

The simplification process described above can be interpreted in terms of local modifications in the MT framework, and thus encoded in a DAG (see Figure 3.11). Each element a of H_k corresponds to a node in the DAG and an edge goes from a in H_k to a node b in H_{k+1} (if a and b intersect). The patch in L_k which results from the intersection of a and b is associated to that arc.

Notice how the structure of the DAG is effectively decoupled from the choice of the simplification algorithm and the mesh representation. As long as the boundary of a patch is preserved, any algorithm (edge-collapse, vertex removal, remeshing, clustering, etc.) can be applied.

Finally, it is important to stress that the concept of *V-Partitions* is independent from the geometric dimension of the model or the choice of the primitive used to represent the surface, as long as the boundary of a patch can be preserved during simplification.

3.5.2 Well Conditioned BMT

While the MT framework guarantees a correct extraction and seamless patch assembly, still certain conditions on the V-partition need to be met to guarantee a good performance of the multi-resolution algorithm.

- As a general rule all the patches in the MT should have the same size (the same number of triangles) for an efficient memory management and adaptive multi-resolution. This translates in all the elements of H_k being about the same size in each level of the hierarchy.
- To ensure a *limited height*, the number of elements should decrease geometrically in each partition H_k. Combined with the above requirement, this implies that the simplification step should reduce the number of triangles by a constant factor (usually 0.5 to reduce the storage overhead to a reasonable amount) so that the elements of H_{k+1} should cover about twice the area of the elements of H_k.
- The number of dependencies (arcs in the DAG) should be minimized to maximize the expressive power of the model. This requires to minimize the number of intersections in H_{k+1} for any element a in H_k. Hierarchical structures such as quadtrees and right triangle hierarchies (Section 4.1) fulfill this requirement, but the condition that the element of a partition must be compact in shape (ideally spherical) is enough. A uniform distribution of the partition elements ensures a linear growth, since an element of H_i should intersect a roughly constant number of elements of H_{i+1}. Compact patches are also beneficial for a number of reasons: the number of boundary vertices is minimized, per patch error strategies remain accurate, and visibility culling is more effective.
- Boundary vertices locked during a simplification step should not remain blocked in the subsequent levels, as this would create patches with badly shaped triangles and, ultimately, prevent further simplification. This requires that each partition H_k should not share borders with H_{k-1} and H_{k+1} (it is *independent*). Hierarchical structures do not fulfill this requirement.

3.5.3 V-partitions Construction

We propose to build the partitioning using a Voronoi like approach. Given a set of 3D points $Q = v_0, \ldots, v_k$, called *seed set*, we define the V-partition of a mesh T into patches $V_Q = \{Q_0^T, \ldots, Q_k^T\}$

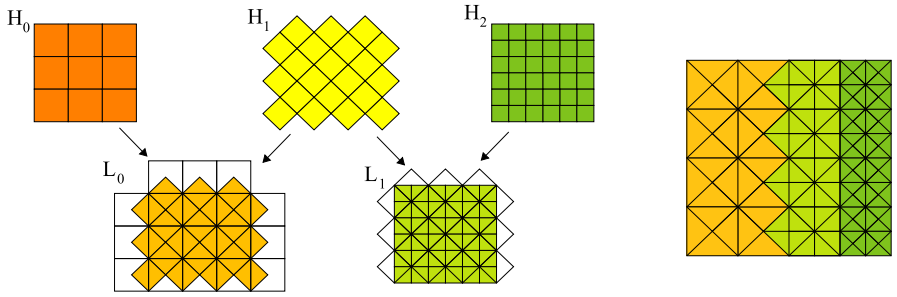


Figure 3.12: *The sequence of partitions obtained by placing seeds on the centers and corners of a square grid generate the well known bin-tree hierarchy used for 2D terrain multi-resolution models.*

by defining Q_i^T as the patch composed of the faces that are nearest to the seed point v_i. Note that it is not required for these patches to be composed of a single connected component. To build the multi-resolution model, we need a sequence of seed sets $Q_0, \ldots Q_k$, and the corresponding V-partitions $H_0, \ldots, H_k$, of decreasing granularity.

It should be noted that the sequence of partitions does not need to adapt to the geometric characteristics of the mesh (like curvature or density). In this approach the adaptivity is handled during the MT traversal. If a portion of the mesh presents more features, its simplification will yield a bigger error and therefore during the MT traversal that portion will be maintained at a finer resolution.

We propose two approaches to build the sequence of seed sets, the first one generates a regular partitioning of space, while the second one generates a sequence of irregular partitions.

Regular V-partitions

A simple and effective method to build V-partitions is to use a regular recursive seed distribution scheme. Considering the two dimensional case, illustrated in Figure 3.12: we start by placing vertices on a regular grid, obtaining a partition into squares, then we continue placing seeds on the mid-points of the edges of these squares, obtaining another finer partition into squares (rotated by 45 degree), and so on.

With this approach, the L_i partitions form the well known triangle bin-tree hierarchy used in the BDAM approach [17], presented in Section 4.2. This approach can be extended to the three-dimensional case by considering a regular grid and placing seeds at cube centers, face centers and edge centers. The sequence of partitions H_i that we obtain (where Voronoi regions are cubes and octahedra) forms the same patterns of diamonds as the Slow Growing Subdivision scheme used also in the recent Tetrapuzzles [19] approach. This subdivision is explained in detail in Section 5.2. Many other recursive 2D subdivision schemes can be obtained by regular seed placement, for example hexagonal subdivisions shown in Figure 3.13 (also described as dual $\sqrt{3}$ subdivisions).

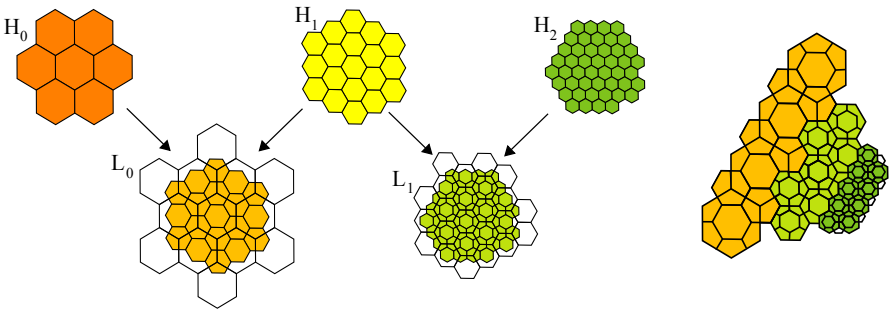


Figure 3.13: *The sequence of partitions obtained by recursively placing seeds on the vertices of a hexagonal grid.*

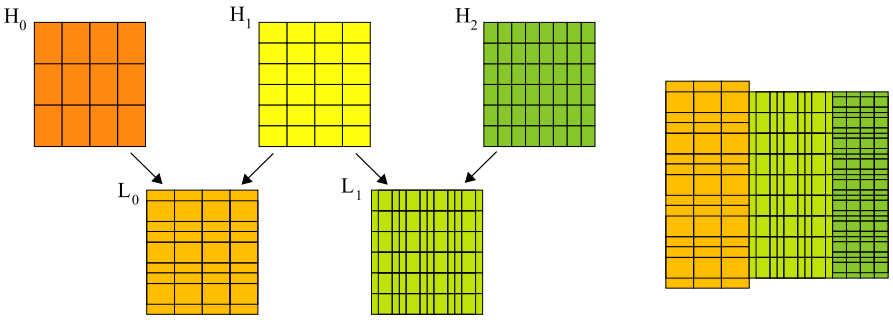


Figure 3.14: *The sequence of partitions obtained by recursively placing seeds on the vertices of a rectangular grid.*

Irregular V-partitions

Irregular V-partitions can be built by generating an irregularly distributed set of seeds, using the Voronoi approach described above. If the seed points for each level are fairly distributed and in general position with respect to the seeds of the other levels all the needed properties will be respected. To ensure a uniform size of the patches a simple and effective strategy is to generate the seeds randomly and apply a few iterations of Lloyd's Voronoi relaxation algorithm [72, 29].

Lloyd's algorithm starts with an initial distribution of samples and consists of repeatedly executing one relaxation step:

1. the Voronoi diagram of all samples is computed, and the faces of the mesh are assigned to their cells,

2. the samples are moved to the barycenter of each cell.

Each time a relaxation step is performed, the distribution of the points becomes more even: closely spaced points move further apart, and widely spaced points move closer together. This approach is used in Batched Multi-Triangulations [20] (Section 5.3) and in Dynamic Meshes (Section 6.4.1).

3.6 Architecture

The architecture of a rendering application based on BMT can be divided into four components: an out-of-core system which manages the data on disk; a DAG with a related set of algorithms to traverse it (eventually with various degrees of visibility culling); a rendering component which manages the VBO buffers and textures; a prefetching thread to load (or preload) data from disk without stalling the entire application.

In the next three chapters we will discuss in detail the application of this framework and the specific design and implementation to different classes of models: 2.5D terrains with regular V-partitions in Chapter 4, 3D meshes with a regular and irregular V-Partitions in Chapter 5, and finally, in Chapter 6, 4D meshes (dynamic geometry) with irregular V-partitions. We follow a common approach in all these applications, in particular regarding DAG traversal driven by screen space error and out-of-core management.

3.6.1 Screen Space Error Saturation

During the construction phase a geometric error *lambda* is assigned to each node in the DAG, that corresponds to its degree of approximation of the underlying surface and results from the simplification process. Ideally the geometric error should be computed using the Hausdorff metrics H_∞ or H_2 defined in Section 2.1.3, but due to the large run times required we usually adopt some approximation.

The error function that controls the traversal of the DAG should capture the appearance error (Section 2.1.2) from the current view-point. We use the *screen projection error* ϵ which is simply the projection of the model space error λ on screen. In order to have a conservative estimate of the error, the closest point in the bounding volume of the patch should be used. Bounding spheres are particularly suited for this error metric, as the computation of the error requires only a few operations (Figure 3.15). This metric provides an upper bound for the number of pixels by which a geometric picture will be displaced in the simplified mesh with respect to the original mesh. A user specified *error threshold* τ can then be simply expressed in pixels.

The *error saturation* technique, introduced in Section 2.3.3, modifies the error function such that, for every point of view, the error associated to a parent node is always higher than the error of any child. This property can be enforced in a simple bottom-up approach where the geometric error of the leaf nodes is set to zero, and recursively the geometric error λ of every node

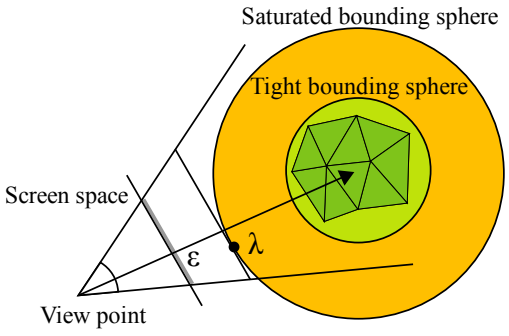


Figure 3.15: *Screen based projection error.*

is increased until it is bigger than the geometric error of all its children. Similarly, the bounding sphere must be enlarged to enclose the bounding spheres of all its children in the DAG, in orderto ensure a monotonic view-dependent error function.

The main advantage of error saturation in the context of BMT is not the simplification of the DAG traversal for minimum-size and best-resolution queries, but the fact that non saturated error functions might trigger a cascade of update operations due to the recursive checks, each one updating a block of primitives: this usually generates spikes in data access and transfer and increases latency in the application. The saturated bounding sphere used for view-dependent error calculation is inefficient for frustum culling purposes, therefore we store also the radius of the original bounding sphere (*tight bounding sphere*).

3.6.2 Out-of-Core

The out-of-core implementation for BMT is straightforward: each patch is stored independently on disk as a single block, in the format chosen for the specific application: indexed mesh, triangles strips or regular grids. eventually compressed. An index is used to keep track of the patches along with the DAG, and since it needs only a few bytes per node, even for very large models it can easily fit into memory. For each patch we need to store in the index:

- location of the patch in the disk (4 bytes)
- number of vertices and triangles in the patch (2 + 2 bytes)
- size of the patch on disk and once extracted into ram (2 + 2 bytes)
- geometric error (4 bytes)
- saturated and tight bounding sphere (5 floats or 20 bytes)
- if needed, visibility culling additional data such as the cone of normals (24 bytes)

As a result between 36 and 60 bytes are needed per patch, and assuming an average of one thousand triangles per patch and a very large model consisting of a billion triangles the space occupied by the index ranges from 36 to 60MB. The DAG space requirement, either implicit or explicit, is comparable: the number of nodes is 3 or 4 times less and the number of links is the same as the number of patches. Since a well conditioned V-partition exhibits linear growth (Section 3.5.2) the number of links per node is also roughly constant and small.

Data access is simple: whenever a patch is needed, its offset in the data file is looked up in the index and the corresponding region is memory mapped. The patch is then eventually uncompressed and ready to be read, while unused patches are deallocated (saved back on disk and memory un-mapped) using a *Least Recently Used* algorithm. During the construction of the multiresolution model, when new patches are created, we can simply append them at the end of the file and update the index correspondingly.

An associative database such as *BerkleyDB* which associates a key to a chunk of data could be used to manage the relative complexity of patch management. However, given the simplicity of this data structure and the ability to fine tune many aspects of the data access we prefer to implement our own solution. For example, in order to further enhance the cache coherence of the data on disk we can arrange the order of the patches so as to maximize locality: we

order the patches first level by level first and then by geometric proximity, following a space-filling curve. The dimension of the models (2.5D, 3D or 4D) and the kind of V-partition used (regular, irregular) of course determine the optimal curve to be used. The exact details of the data organization will be presented in the following chapters.

3.6.3 Parallellization

The preprocessing stage for huge models can be quite computationally expensive, in particular when using high quality simplification algorithms: simplification and mesh optimization (e.g. triangle strip construction) are actually the most time consuming steps in the construction of a BMT. The whole simplification process, however, is inherently parallel (see Figure 4.12), because of the operations involving each element of H_i (collect the corresponding patches in L_{i-1}, join them, simplify and optimize as shown in Figure 3.10) can be performed independently.

In all the various implementations of the BMT framework each cluster of patches H_i is dispatched to a computer in a small network, processed and the results assembled back into the BMT structure, resulting in a much faster construction stage.

Chapter 4

Applications to Terrains

Interactive visualization of very large digital elevation models (DEMs) is a recurring task in a number of application domains such as Geographic information systems (GIS), virtual reality, games, flight simulation and scientific visualization. While general data structures and algorithms for multi-resolution methods are in principle also applicable to digital terrain models, dramatically higher efficiencies can be achieved by tailoring the algorithm specifically for 2.5-dimensional surfaces.

In this chapter we shall review the existing literature on terrain visualization and present our contribution: BDAM and P-BDAM, two effective specializations of the BMT framework.

4.1 Regular and Semi-Regular Models

The most general multi-resolution models are based on irregular triangulations with unrestricted connectivity, see Section 2.3.2: thanks to their flexibility, fully irregular structures are capable of producing the minimum complexity mesh representation for a given error measure, but require explicit encoding of mesh connectivity, hierarchy and dependencies.

Regular and semi-regular models impose strong constraints on mesh connectivity, sampling patterns and update operations which require a higher resolution (up to an order of magnitude for terrains) to reach the same accuracy as irregular structures. On the other hand in these models most of the information (connectivity, DAG, vertex coordinates, texture coordinates, bounding volumes) can be maintained in implicit form resulting in huge storage space savings and faster extraction, rendering and editing. Input data most often comes sampled on a regular grid so that meshes with regular connectivity are easily adapted to terrains.

Multi-resolution schemes based on quad-trees or triangle bin-tree triangulations, introduced in Section 2.4.2, allow for continuous variable resolution surfaces, easy construction and management and fast extraction of semi-regular triangulations.

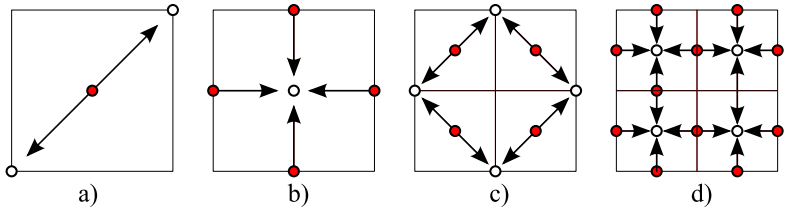


Figure 4.1: *Dependence relations of restricted quad-tree triangulations: a) and c) same level dependencies, b) and d) dependencies on next level.*

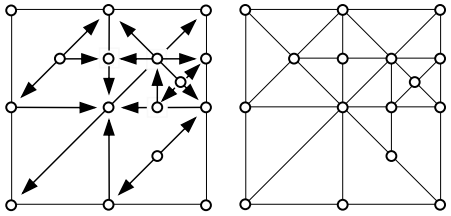


Figure 4.2: *Example of restricted quad-tree triangulation and corresponding vertex dependencies.*

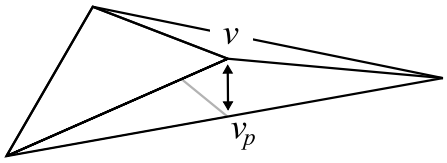


Figure 4.3: *The removal of a a vertex displaces the surface at most of the distance between v and v_p.*

4.1.1 Restricted Quad-Trees

The restricted quad-tree technique, introduced in Section 2.4.2, can be applied to terrains represented as 2.5-dimensional surface data consisting of points with associated height on a regular 2D-grid. Early work by [110, 111] was later improved by [67] which introduced a set of vertex dependencies that can be used to prevent cracks and create conforming triangulations at variable resolution.

The restricted quadtree triangulation imposes a hierarchical dependence relation between vertices as shown in Figure 4.1. Each vertex to be included in a triangulation requires some other vertex to be included first and when no dependence relation is violated the triangulation is conforming (Figure 4.2).

The metric used for the extraction is based on the displacement of the surface generated by the removal of a vertex. The approximation error associated to each vertex v is the distance d from its projection v_p on the edge of the two triangles it depends on (Figure 4.3). These approximation errors are transformed into a screen space-error metric (Section 3.6.1) by projection onto the screen space. The extraction algorithm refines and coarsens the model by inserting and removing the vertices based on some threshold, while the dependencies are recursively enforced by inserting the required vertices.

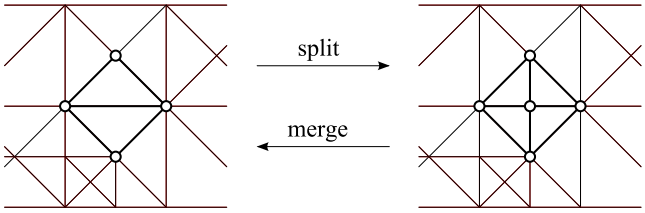


Figure 4.4: *Split and merge operation on a bin tree triangulation.*

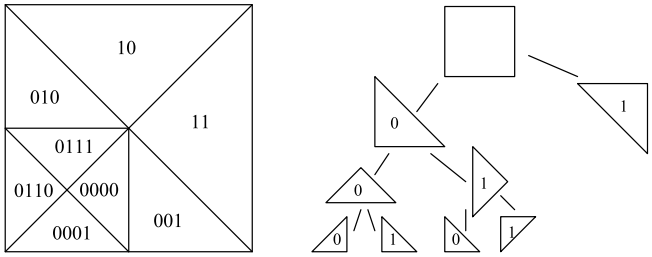


Figure 4.5: *RTIN binary labeling.*

The approach presented in [87] takes full advantage of the fact that the quad-tree hierarchy and the dependency relations can be defined entirely implicitly, by appropriate point indexing and recursive functions, reducing the storage cost down to the elevation data and approximation errors values per vertex, and improving the speed of building and extraction algorithms.

4.1.2 Triangle Bin-Trees

Triangle bin-tree hierarchies (also known as hierarchy of right triangles, HRT) are strictly related to quad-trees hierarchies, and were first presented in the context of terrain rendering by [30] as ROAM (*Real-time Optimally Adapting Meshes*). The extraction algorithm of ROAM is based on the split-merge operation described in Figure 4.4 and performs these operations to adaptively refine or simplify the current mesh. This is essentially the algorithm described in Section 2.3.3: two priority queues, based on an error metric defined on the triangles, store the triangles that can possibly be split or merged accordingly to a given error threshold. The dependence relations of [67] are here expressed as *forced splits*: a triangle cannot be split unless its neighbor across its longest edge is from the same level in the hierarchy.

In the ROAM technique the bounding error hierarchy is the *wedge*: a triangular prism whose height is the maximum height of all its children. The estimate of the screen error projection is then performed based on the corner closest to the viewpoint and frustum culling based on the visibility of the corners. No error saturation technique is adopted: a split operation might trigger a cascade of forced splits to maintain a conforming triangulation.

Right-triangulated Irregular Networks (RTIN) as presented in [32] focus on the efficient representation and fast traversal of bin-trees hierarchies, and its approach is almost identical to the ROAM method and only differs in notation and representation of the hierarchy. RTIN introduces

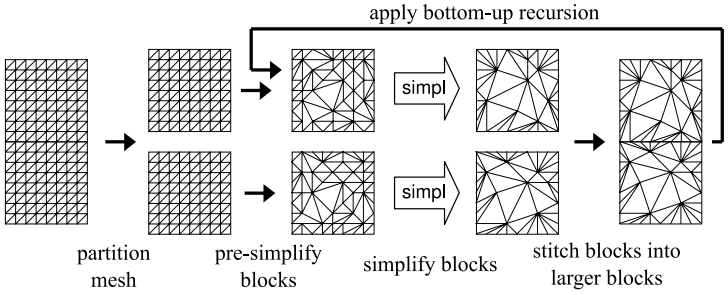


Figure 4.6: *Steps in the hierarchical block based simplification.*

a binary labeling scheme, shown in Figure 4.5, where child triangles are labeled as *left* and *right* (0 and 1) with respect to the split vertex of their parent.

This binary tree allows for the connectivity to be completely implicit and a RTIN triangulation can be represented by the binary tree encoding the splits and the elevation values of the triangle vertices, while the x and y coordinates can be computed on the fly based on the labeling of the triangle. A second contribution of RTIN is an efficient neighbor finding scheme also based on the labeling. This function is needed during splitting operations to ensure the consistency of the triangulation by forced splits.

4.1.3 Clustered Triangulations

We saw in Section 3.4 how a recent trend in multiresolution algorithms is to move the granularity of the primitive to small portions of a mesh and how the solution to boundary continuity problem, introduced in Section 3.4.3, becomes a central ingredient of clustered triangulation algorithms.

Marking

In [115] Toledo et al. extends a hierarchical approach introduced by Hoppe [53], which relies on marking and preserving boundary vertices during simplification to ensure consistency between adjacent blocks belonging to different levels of the hierarchy. The terrain is subdivided in square patches and hierarchically merged and simplified to build a quadtree. During the simplification phase, the external boundaries are marked and preserved (see Figure 4.6). While simple and effective for small models, this approach does not scale over a few levels because a large number of vertices can never be simplified: the number of boundary vertices in each patch double at each level of the hierarchy so that after a few levels they prevent significant complexity reduction.

Stitches

Another common approach subdivides the terrain into square patches, and generates several levels of detail for each patch. In [105], for example, the LODs are computed with a ROAM multi-resolution structure. Each level of each patch is then optimized for rendering: triangle strips or triangle fans are generated, vertex cache coherence is exploited as much as possible, the vertex data is organized so as to optimize CPU/GPU communication efficiency.

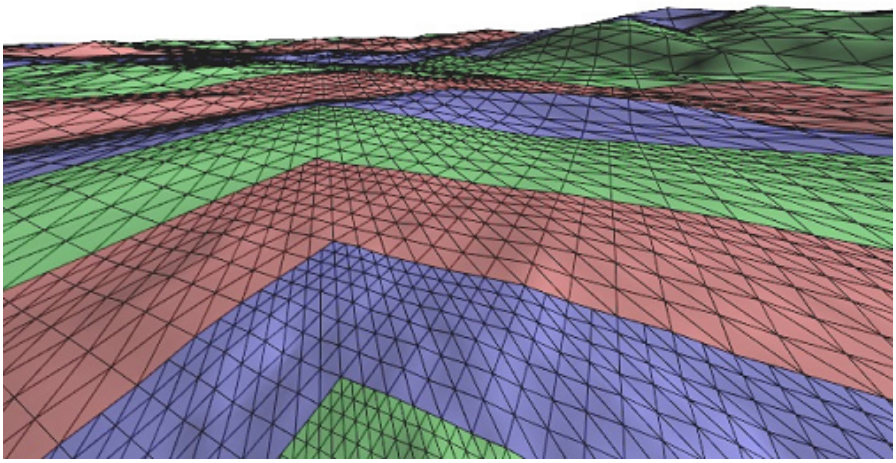


Figure 4.7: *Concentric rings of decreasing resolution in the terrain around the viewpoint generated by the geoclipmaps. Picture taken from [75].*

Since each tile is simplified independently, different LODs will not have the same boundary connectivity so, at run time, the boundaries of neighboring tiles are stitched with an appropriate set of zero-area triangles. These additional triangles are generally of very low quality and need to be stored or computed for each pair of adjacent blocks.

In the approach proposed by Losasso and Hoppe [75], the geometry clipmap, the terrain is rendered as a set of nested regular grids with decreasing resolution centered about the viewer (see Figure 4.7). These grids represent filtered versions of the terrain at power-of-two resolutions, and are stored as vertex buffers in video memory. As the viewpoint moves, the clipmap levels shift and are incrementally refilled with data. To prevent spatial aliasing, an image is pre-filtered into a mipmap pyramid of power-of-two grids.

A sequence of gaps form between regions at different levels due to the power of two sampling of the meshes. To eliminate those gaps and provide temporal continuity, the geometry near the outer boundary of each region L_n is morphed progressively into the geometry of the coarser level L_{n-1}. The amount of morphing is a function of the distance of the vertex to the boundary. The morphing eliminates the large gaps, but the T-junctions along the boundary still result in missing pixels during rasterization. To solve this problem a strip of zero-area triangles is drawn along the boundary connecting different regions, and given the regular connectivity of the structure the strip can be easily assembled on-line without need for additional storage or fetching data from nearby regions.

Since the connectivity is fixed and the geometrical data is stored as simple grids of heights, normals and colors, this technique easily supports compression and out-of core management. While extremely efficient in storage and rendering, this structure might require orders of magnitude more triangles to achieve the same accuracy as an irregular triangulation.

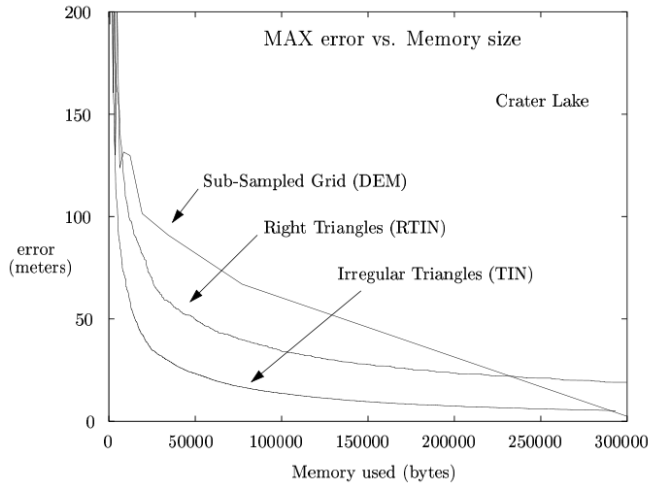


Figure 4.8: *Max error vs. memory size for TIN, RTIN and sub-sampled grids. This graph is taken from [33].*

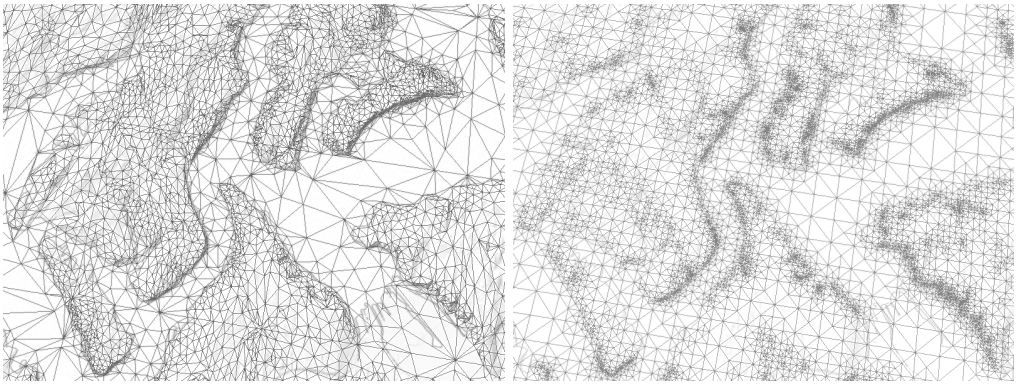


Figure 4.9: *TINs can easily adapt to high frequency variations of the terrain such as cliffs, while many subdivision levels are needed for regular subdivision meshes, that spend a large fraction of the triangle budget for following edges.*

4.2 BDAM

Batched Dynamic Adaptive Meshes (BDAM) [17], with its derivatives Planet-sized-BDAM [18] (P-BDAM) and Compressed-BDAM [46] (C-BDAM), is the first technique to fully employ a patch-based triangulation structure: the coarse topology of the multi-resolution structure is based on a *hierarchy of right triangles* and decoupled from the actual structure of the patch. In this way these structures combine the advantage of regular multi-resolution structures in terms of effective adaptive extraction with the high adaptivity of *triangulated irregular networks* (TINs).

A TIN is much more efficient than a hierarchy of right triangles (HRT) in terms of number of triangles needed to represent a surface at a certain level of accuracy: Figure 4.8 shows a comparison of the max error versus memory occupation between TIN meshes simplified using

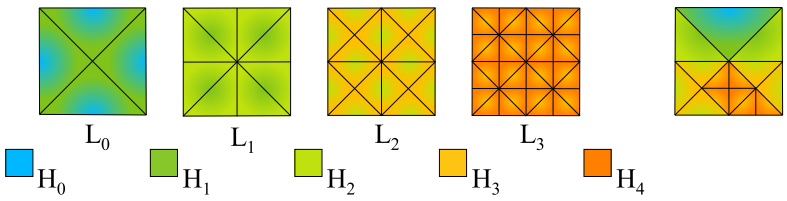


Figure 4.10: *An example of BDAM extraction: each triangle represents a terrain patch composed by many triangles. Each error value is marked by a different color: the blending of the color inside each triangle corresponds to the smooth error variation inside each patch.*

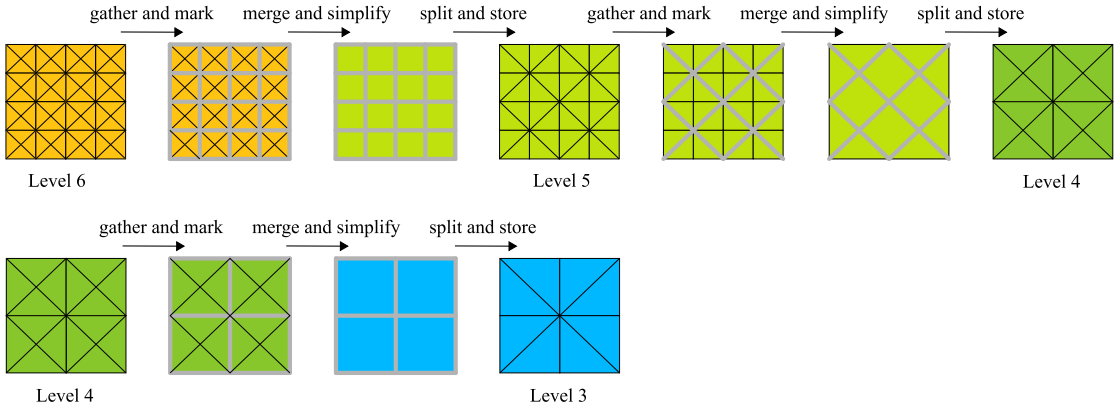


Figure 4.11: *Construction of a BDAM through a sequence of simplification and marking steps. Each triangle represents a terrain patch composed of many triangles. Colors correspond to different errors.*

quadrics assuming 30 bytes per vertex, RTIN extracted meshes which use 40 bytes per vertex and subsampled grids which require only 2 bytes per vertex. The better performance of TINs is mainly due to the fact that they adapt much better to the high frequency variations of the terrain [33] (see Figure 4.9).

The input domain is recursively subdivided to a hierarchy of right triangle clusters, until each bin-tree node consists of a small TIN, containing a few thousands of triangles (Figure 4.10). In practice also regular triangulations or any other convenient surface description supporting the basic operations of merging, simplification and splitting described in the next section can be used.

4.2.1 Construction

The upper levels in the hierarchy are built by joining every four adjacent triangular partitions that form a square, called *diamond*, see Figure 4.12, and split it into two regions. This is the same split-merge operation described in the ROAM method (Figure 4.4) with an important difference: the algorithm works with clusters of triangles.

To ensure the correct matching between triangular patches, we exploit the HRT property that each triangle can correctly connect either to: triangles of its same level, triangles of the

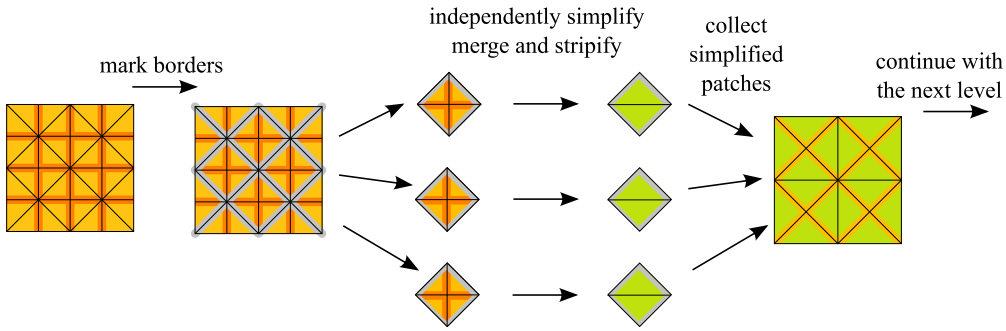


Figure 4.12: *Construction of a BDAM through a sequence of simplification and marking steps. Each triangle represents a terrain patch composed of many triangles.*

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.

To enforce the correct connectivity, all the vertices lying onto the longest edges are marked as non-modifiable, so that they will still exist in the next level of the hierarchy. This marking splits the mesh into a set of square shaped sub-meshes, each one made of four triangular patches joined along their shortest edges. These submeshes correspond to the modifications in the DAG of a HRT.

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 each of the two triangular patches corresponds to a triangle at the next level of the bin-tree hierarchy. This sequence of merge, simplify and split operations can easily be described in the context of V-partitions, specifically as depicted in Figure 3.12 on page 46.

The simplification always halves the number of vertices of the sub-mesh, thus the size of patches is approximately constant everywhere in the hierarchy. This means that the width of the MC is bounded, while the structure of the HRT ensures logarithmic height and linear growth.

An important advantage of upgrading the primitive is the decoupling of the simplification algorithm from the actual structure of the DAG. In fact any simplification algorithm which preserves the outer boundary, is suitable for the BDAM. To maximize the advantage of using TINs as primitives we choose an edge collapse simplification driven by the efficient quadric error metric [44] which combines speed and precision.

Simplification and triangle strip construction are actually the most time consuming steps in the construction of a BDAM and in certain applications it might be beneficial to increase the preprocessing speed adopting faster simplification algorithm, for example vertex clustering. As shown in Section 3.6.3, the whole construction phase can be parallelized. It is particularly easy to see in the case of BDAM, because each diamond is simplified and stripified independently (see Figure 4.12): thus each diamond can be dispatched to a computer in a small network, processed (merge, simplify, split and triangle stripping steps) and the result assembled back into the BDAM structure.

Each patch is then treated as an independent small mesh. We chose to replicate the vertices

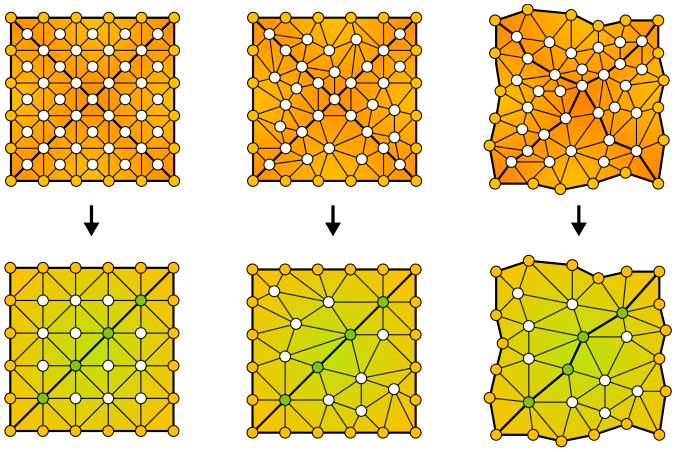


Figure 4.13: *Three different variants of the BDAM local structure: regular (C-BDAM), irregular with constrained borders (P-BDAM), irregular (BDAM).*

shared between adjacent patches because the overhead is small, in the order of the square root of the triangles in the cell, and it is more than compensated for by a number of optimizations:

- each cluster can be stored independently on disk, thus allowing for a straightforward out-of-core implementation.
- we can switch to 16 bit indexing (provided that each cell is smaller than 64k triangles)
- we can aggressively optimize the triangle stripping and vertex cache coherence for maximum GPU performance
- each patch can be compressed using a lossless or lossy algorithm with an important caveat: the shared vertices must be consistent with the adjacent clusters.

The structure of the HRT is then encoded implicitly as a binary tree.

4.2.2 Textures

BDAM supports usage of textures, but in this case placement of the vertices on the border of the quad-tree cells needs extra care to ensure compatibility with texture boundaries: their projection must lie exactly on the boundary of the patch triangular shape resulting from the regular quad-tree subdivision of the terrain (see Figure 4.13b). This results in a constraint on the simplification of a region: no triangle must cross the splitting diagonal if this diagonal is vertical or horizontal in texture space. If no texture is used the space alignment of the vertices along the boundary might be noticeable and it is better to drop this constraint: the patches resulting from the splitting will have unconstrained zigzag borders instead of straight ones (Figure 4.13c).

Due to the large dimension of the textures associated with the elevation data, we need to partition them into chunks before rendering. The total size also generally exceeds typical core memory size, hence the texture management unit has to deal with out-of-core memory handling

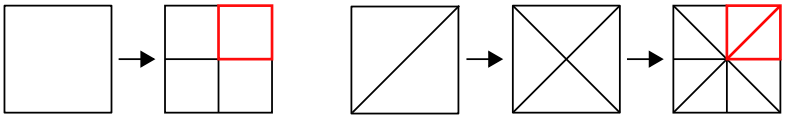


Figure 4.14: *A texture quadtree element is associated to a pair of adjacent geometry bintree elements. View-dependent refinement is performed using a combined top-down traversal of the texture and geometry trees.*

techniques. These considerations lead to a multi-resolution 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 quad-tree instead of the geometry bin-tree. If the original image covers the same region of the elevation data, each texture quad-tree element corresponds to a pair of adjacent geometry bin-tree elements (Figure 4.14), and descending one level in the texture quad-tree corresponds to descending two levels in the associated pair of geometry bin-trees. This correspondence can be exploited in the preprocessing step to associate object-space representation errors to the quad-tree levels, and in the rendering step to implement view-dependent multi-resolution textures and geometry extraction in a single top-down refinement strategy.

4.2.3 Bounding Volumes and Screen Space Error

The concept of nested/saturated errors (Section 2.3.3 and 3.6.1) was introduced in the terrain visualization context by Pajarola [87] and allows to extract a coherent triangulation with a simple stateless visit of the bin-tree [87, 69], thus greatly simplifying the extraction algorithm with respect to a generic DAG.

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. The error for each patch is evaluated as the maximum vertical difference to the original one. To perform this computation quickly by exploiting graphics hardware, we render the original and simplified meshes under orthographic projection and evaluate the difference among the corresponding depth buffers. The Hausdorff distance would be too costly to evaluate for such big models.

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. 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 bounding spheres (Figure 4.15) is also built during the preprocessing in order to enforce the following rules:

1. the bounding sphere of a patch includes all children bounding spheres;
2. two patches adjacent to a hypotenuse must share the same bounding sphere which encloses both.

These conditions can easily be derived in the MT framework, where the bounding volumes used for saturated view-dependent error enclose an entire node of the DAG (corresponding to the support of a modification). In the case of HRT this corresponds to a diamond and explains

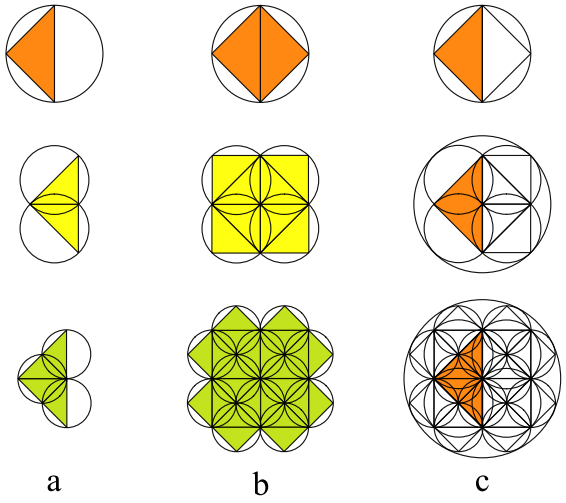


Figure 4.15: *In column a) the descendant of a patch in the hierarchy of triangles and the strict bounding spheres. In column b) the descendant of a node in the DAG and their strict bounding sphere. In column c) how the bounding sphere of a patch must be enlarged to encompass the bounding sphere of all its descendants in the DAG.*

the second condition. These bounding spheres are used to compute screen space errors as in Section 3.6.1 and also for view frustum culling, using the *tight* radius.

4.2.4 Extraction Algorithm

View-dependent refinement of the HRT is driven by screen space error computed using the bounding sphere of a patch and its object-space geometry and texture errors: 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 λ between the texture and the geometry object space errors (see Figure 4.15). The refinement condition, once the closest point from the viewpoint is computed, requires only one multiplication to check if $\lambda > \tau * distance$ where τ is an error threshold expressed in pixels.

The refinement procedure described in pseudo code at the end of the section in the procedure *refine*, 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 in the procedure *geo_parent_refine* refining the two geometry bin-trees 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 one level refinement step in the texture quadtree corresponds to two refinement steps

into the geometry bin-tree, 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. 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 as 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.

View frustum culling is easily done as part of the recursive refinement, exploiting the tight bounding volumes. Since a patch bounding volume contains all the geometry of a given subtree (in the HRT bin-tree, not in the implicit DAG), recursion can stop without rendering whenever the bounding volume is detected as invisible.

```
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
```

A prefetching thread (Section 3.2.4) is run in parallel performing the same refinement algorithm on a predicted camera position and advises the operating system when a required object is

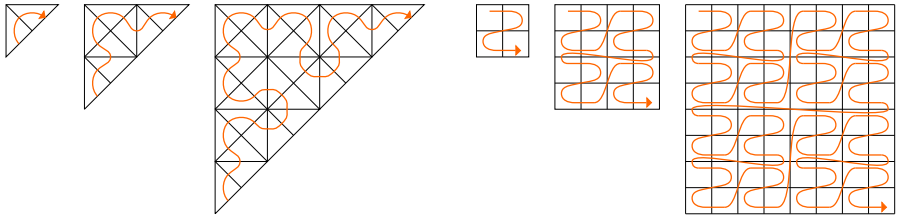


Figure 4.16: *Filling curves used for coherent storage of geometry (left) and textures (right).*

not in cache. On Linux, this is done by executing the *madvise* system call to instruct the kernel that it would be advantageous to asynchronously read the indicated pages ahead if they are not already in core. This technique blends well with the virtual memory based external memory management subsystem. In particular, the main rendering code does not need to be aware of the prefetching component, and we exploit the extensive performance optimizations of the operating systems virtual memory manager, such as reordering of requests to reduce seek access time and overlapping of computation and disk access.

4.2.5 Out-of-Core Data Layout

Following the approach described in Section 3.6.2 each patch is stored independently on disk as a small indexed mesh while an index containing the information necessary for the traversal, visibility culling and error computation is maintained in core.

Similarly to Lindstrom and Pascucci [69], the data layout is optimized to improve memory coherence 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, therefore all data is sorted first 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 4.16 illustrates the indexing schemes used for data layout.

4.2.6 P-BDAM

The main problem with very large datasets is the fact that the highest precision data-type on the graphic hardware is the IEEE floating point: a mantissa limited to 23 bits leads to noticeable vertex quantization problems and camera jitter. In P-BDAM [18], we exploit BDAM's structure for planetary sized rendering applications: the surface of the planet is partitioned into a number of square tiles, therefore managing a forest of BDAM hierarchies instead of a single tree.

The tiles have an associated (u, v) parametrization, which is used for texture coordinates and to construct the geometry subdivision hierarchy (see Figure 4.17). The number and size of the tiles is arbitrary, however, the local parametrization requires that a single precision floating point representation is accurate enough for representing local coordinates (i.e. there are less than 2^{23} texels/positions along each coordinate axis). Errors and bounding volumes need to be propagated to neighboring tiles through the common edges in order to ensure continuity for the entire dataset.

To overcome the floating point accuracy issues we use a small patch of triangles as a primitive expressed in barycentric coordinates with respect to a triple of points, *base corners*, whose

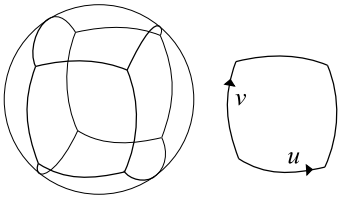


Figure 4.17: *A large dataset such as a planet can be decomposed into tiles with a local parametrization.*

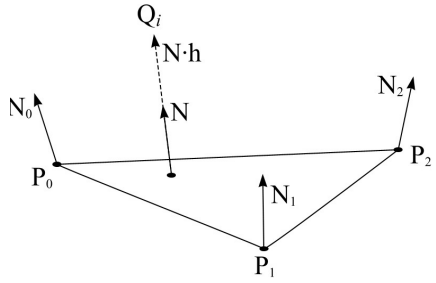


Figure 4.18: *P-BDAM patches are represented as arbitrary triangulations of points over a displaced triangle.*

coordinates are encoded in double precision (Figure 4.18). Each base corner vertex contains a pair of texture coordinates T_i, that correspond to the position of the vertex in (u, v) coordinates, as well as a planetocentric position P_i and a normal vector N_i , that are computed from T_i at patch construction time as a function of the particular projection used. The vertices Q_j of the internal triangulation are stored by specifying a barycentric coordinate and an offset along the interpolated normal direction, and all the information required at rendering time is linearly interpolated from the base corner vertex data (see Figure 4.18). As for BDAM, the interior of the patch is an arbitrary triangulation of the vertices, that is represented by a cache-coherent generalized triangle strip stored as a single ordered list of vertex indices.

The only aspect that requires particular care is the computation of the planetocentric positions, since all other information is local to the patch. We therefore store P_i in double precision. At each frame, we render all patches in camera coordinates, simply subtracting the camera position from P_i on the host before converting them to single precision for communicating it to the graphics hardware. This way a single reference frame is used for each frame, and positional accuracy decreases with the distance from the camera, which is exactly what we want. In contrast to common linear transformation approaches [68, 96], neighboring patches remain unconditionally connected because displaced vertex values only depend on the common base corner vertices (along the edges, the weight for the opposite vertex is null). The conversion cost (9 subtractions and 9 floating point conversions) is negligible, since it is amortized over all the internal triangles. Moreover, the transformation from barycentric to Cartesian/texture coordinates can be efficiently computed from corner data on the GPU, using a simple vertex program, shown below.

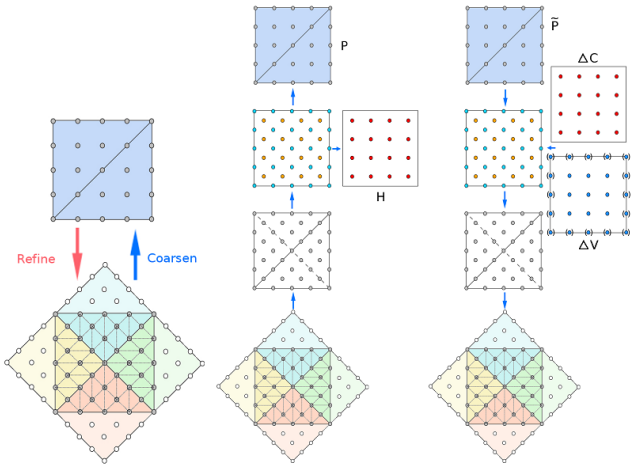
```
struct a2v {
  float4 uvh: POSITION;  //baricentric position and displacement
};

struct v2f {
  float4 hpos: HPOS;     // view normalized position
  float4 tex0: TEX0;     // texture coord. for mapping
  float4 tex1: TEX1;     // texture coord. for bottom/left clipping
  float4 tex2: TEX2;     // texture coord. for top/right clipping
};

v2f vpl_displaced_tripatch(
    a2v vertex,
    uniform float4x4 pvm,
    uniform float4 texture_border_width,
    uniform float4 one_minus_two_texture_border_width,
    uniform float4 P0,    uniform float4 T0,   uniform float4 N0,
    uniform float4 P0P1, uniform float4 T0T1, uniform float4 N0N1,
    uniform float4 P0P2, uniform float4 T0T2, uniform float4 N0N2) {

  v2f result;
  // Uncompress displacement (two shorts to a float)
  float h = vertex.uvh[2]*327.67 + vertex.uvh[3]*0.01;

  // Interpolate using barycentric coordinates
  float4 N = N0 + vertex.uvh[0]*N0N1 + vertex.uvh[1]*N0N2;
  float4 P = P0 + vertex.uvh[0]*P0P1 + vertex.uvh[1]*P0P2 + h*N;
  float4 T = T0 + vertex.uvh[0]*T0T1 + vertex.uvh[1]*T0T2;

  // Compute output position and texture coordinates
  result.hpos = mul(pvm,P);
  result.tex0 = texture_border_width + T*one_minus_two_texture_border_width;
  result.tex1 = T;
  result.tex2 = float4(1,1,1,1)-T;
  return result;
}
```

This has the important advantage that, since the vertices of the internal triangulation are invariant in barycentric coordinates, they can be stored in a static vertex array directly in graphics memory, and the rendering routine can fully benefit from the post-transform-and-lighting cache of current graphics architectures, which is fully exploited when drawing from the indexed representation. In particular, since the vertex program is executed only at cache misses, its cost is amortized over multiple vertices.

4.2.7 Compression

To reduce data transfer bandwidth and memory footprints, many multi-resolution techniques integrate compression methods, which becomes a fundamental ingredient of the approach for very large terrains. Geometry clipmaps [75] and grid based structures make straightforward use of standard 2D compressors or multilevel wavelets and reconstruct the normal information at run-time providing a very compact representation.

The Compressed Batched Dynamic Adaptive Meshes (C-BDAM) technique [46], combines the generality and adaptivity of bin-tree multi-resolution structures with high compression rates of regular grid techniques, demonstrating the flexibility of the BDAM approach. In BDAM each triangular region is a pre-computed irregular mesh, while in C-BDAM two triangular regions compose a *diamond* which is a regular triangulation with vertices on a grid. The split-merge operation, shown in Figure 4.19, consists of adding or removing vertices at the center of the square cells of the grid.



(a) Coarsening and refinement (b) Wavelet view (c) Near lossless scheme

Figure 4.19: *C-BDAM wavelet compression scheme.*

The vertex attributes are incrementally encoded following a two-step wavelet based scheme, in which lossy wavelet predictions are corrected to keep approximated values within user defined bounds.

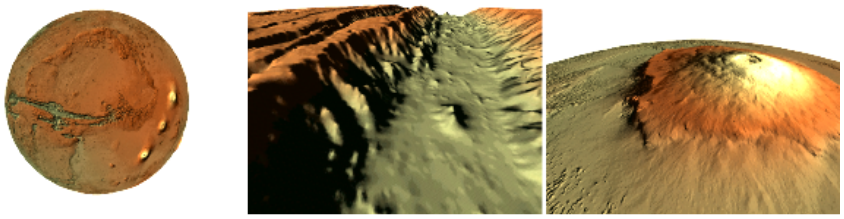
In P-BDAM we have chosen a simple solution that favors mesh decompression speed over compression ratio. We quantize the barycentric coordinates to 12 bits (sufficient to encode over 16M positions in a triangular patch), reorder vertices in strip occurrence order, delta encode them, and compress the result using the LZO lossless compression method (available on http://www.oberhumer.com/), an open source data compression library based on a Lempel-Ziv variant which is suitable for decompression in real time. This typically compresses data to less than 50% of the original size, while supporting a decompression rate of over 15M triangles/second.

4.3 Results

We evaluated the rendering performance of the BDAM technique on a number of fly-through sequences over the Puget Sound area and that of the P-BDAM on a low-altitude fly-over of the Mars model (in Figure 4.20). For the Puget models we used a window of 800x600, a screen tolerance of 1 pixel and a resident set size of the application of 160MB, for Mars a window of 640x480 pixels, a tolerance of 3 pixels, and a set size of 98MB, demonstrating the effectiveness of the out-of-core data management.

In Table 4.1 are reported sizes and preprocessing times for the Puget and Mars dataset. The average vertex size for each patch in BDAM is 200 vertices, and is 512 in P-BDAM. 12 bin-trees are used for Mars, processed in parallel on 5 workstations. The results were collected on a Linux 2.4 PC with two AMD Athlon MP 1600MHz processors, 2GB RAM, a NVIDIA GeForce4 Ti4600 graphics board, and a MAXTOR 6L060J3 60GB IDE disk.

Figures 4.21 and 4.22 illustrate the rendering performance of the application: in the prefetch-



(a) Frame 100: Approaching Mars (b) Frame 1240: Inside Mariner Valley (c) Frame 2780: Arriving at the Olympus Mons

Figure 4.20: *Frames from the Mars dataset fly-over.*

Model	Size (vertices)	Tris	Time(h:m:s)	Output size
Puget	1K x 1K	2M	6:35	2x14MB
Puget	4K x 4K	32M	1:42:33	2x196MB
Puget	8K x 8K	128M	6:39:23	2x765MB
Mars	6 x 13.3K x 13.3K	2G	0:36:00 + 6.30:00	12x737MB, 4.5Gb compressed
	6 x 16.3k x 16.3K tex	1.5G	1:00:00 + 1:00:00	1.2GB

Table 4.1: *Sizes and processing times for the construction of the Puget and Mars datasets.*

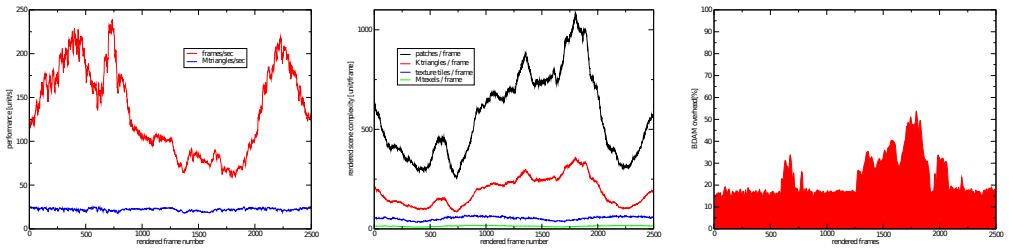


Figure 4.21: *Performance Evaluation for the Puget dataset fly-over in BDAM.*

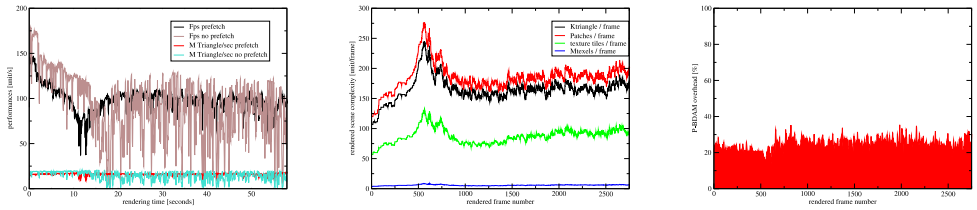


Figure 4.22: *Performance Evaluation for the Mars dataset fly-over in PBDAM.*

ing version, we were able to sustain an average rendering rate of roughly 22 million textured triangles for BDAM and 16 million for P-BDAM, where the reduced performance is mainly due to the increased complexity of the vertex shader.

By comparison, on the same machine, the peak performance of SOAR [70] for a 4Kx4K subset of the dataset was measured at roughly 3.3 millions of triangles per second, even though SOAR used a smaller single resolution texture of 2Kx2K texels. The geoclipmap [75] algorithm,

published in 2004, achieves 59 million triangles per second, using a more recent graphic card; on the same hardware BDAM reaches 65 million triangles per second. The bottleneck for both algorithms lies in the geometry transformation phase of the pipeline, and the geomorphing of the vertices in geoclipmaps requires a small overhead. For a fair comparison also the error in screen space should be considered: the regular sampling employed in geoclipmaps results in a higher error at a given triangle budget. The space and preprocessing times for Geoclipmaps are far lower due to lossy compression (0.1 bytes per sample against 1.8) and a simplification algorithm based on subsampling. Applying similar techniques to BDAM (as in C-BDAM) produces comparable results.

The time overhead of BDAM and P-BDAM structure traversal, measured by repeating the test without executing OpenGL calls, is only about 22% of the total frame time, demonstrating that we are GPU bound even for handling extremely large out-of-core data sets. Rendered scene granularity is illustrated in Figure 4.21 (middle) and 4.22 (middle): even though the peak complexity of the rendered scenes exceeds 350K (240K for P-BDAM) triangles and 20M texels (8.5M for P-BDAM) per frame, the number of rendered graphics primitives per frame remains relatively small, never exceeding 1000 patches and 130 texture blocks per frame. Since we are able to render such complex scenes at high frame rates, it is possible to use very small pixel thresholds, virtually eliminating popping artifacts, without the need to resort to costly geomorphing features.

Chapter 5

Applications to Generic Meshes

In Chapter 3 we discussed the need for view-dependent interactive rendering algorithms in order to deal with very complex 3D models. The case of general surface meshes presents a significantly harder challenge than terrains because the arbitrary topological complexity makes it impossible to use regular structures and implicit DAGs. For this reason the most used algorithms are based on vertex hierarchies, introduced in Section 2.2.2, or on the multi-triangulation (MT) described in Section 2.3.2.

The overall strategy is conceptually similar to the one described in Section 3.4 and adopted for terrain visualization algorithms in the previous chapter: the primitives have been upgraded to patches of triangles in order to reduce the CPU and fully utilize the increasing GPU computational power and to easily manage out-of-core large models. Again, the main challenge for these algorithms is how to stitch together patches at different resolutions while keeping a seamless boundary.

In this chapter we review the existing literature on patch-based multiresolution algorithms and present our solutions: Tetrapuzzles and Batched Multi-Triangulation.

5.1 Extending Progressive Meshes

In literature, the first line of research approaching these problems have been developed working on extensions of the Hoppe's *progressive meshes* [51] (PM). The domain of the model is subdivided hierarchically into blocks and each block processed to generate a PM.

In his PhD thesis, Prince [93] implements such a hierarchical subdivision to generate and view progressive mesh representations of large models, which would not fit in memory. The construction process consists of:

1. spatially subdivide the input mesh into smaller blocks, using a simple regular 3D grid;
2. simplify each block of data until some user-specified threshold is reached. The result is a progressive mesh;
3. "stitch" together (i.e. merge) neighboring blocks of the mesh, forming larger blocks;

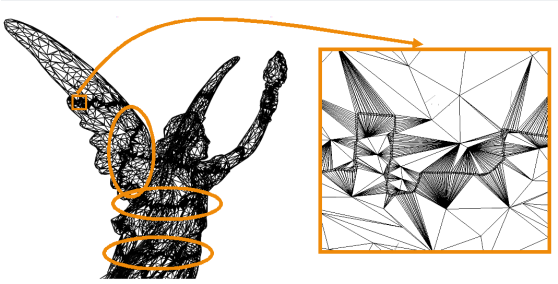


Figure 5.1: *Boundary under-simplified vertices. Figure taken from [128].*

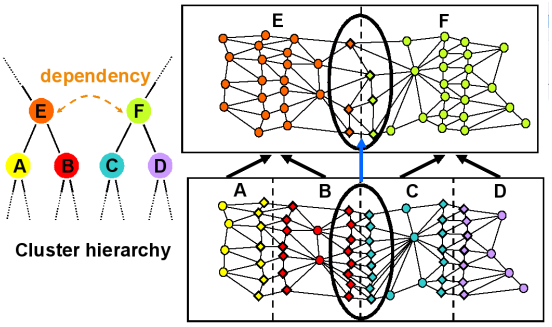


Figure 5.2: *Dependencies between clusters are introduced when simplifying the boundary. Figure taken from [128].*

4. repeat steps 2 and 3 in a hierarchical fashion, until all the pieces of the original mesh have been stitched back together.

Here, the boundary problem is resolved by simply disallowing simplification of the boundary vertices for each block and ensuring that neighboring blocks join seamlessly at any level of the hierarchy. These constrained vertices on the boundary however propagate to the upper levels of the hierarchy, resulting in a dramatic increase of the number of triangles and decreasing the quality of the triangulation; specifically, the boundary of each blocks is heavily under-simplified compared to the interior (see Figure 5.1).

Later, Yoon et al. [128] proposed in *Quick View Dependent Rendering* (QVDR) three practical improvements over this basic structure: exploiting temporal coherence, the visibility of each cluster is dynamically computed with real-time occlusion queries, also high GPU throughput is obtained by rendering PM's directly from GPU memory, finally the boundary problem is alleviated by introducing cluster dependencies.

The introduction of a dependency between two clusters remove the constraints on the shared vertices thereby allowing the simplification of the merged boundary (see Figure 5.2). However, at runtime, splitting a cluster forces all its dependent clusters to split, otherwise a visible crack would be formed at the boundary; a parent cluster cannot be collapsed unless all of its dependent clusters have also been collapsed.

Cluster dependencies reduce the expressive power of the resulting multi-resolution. This effect is particularly noticeable where a large group of clusters is connected by dependence links.

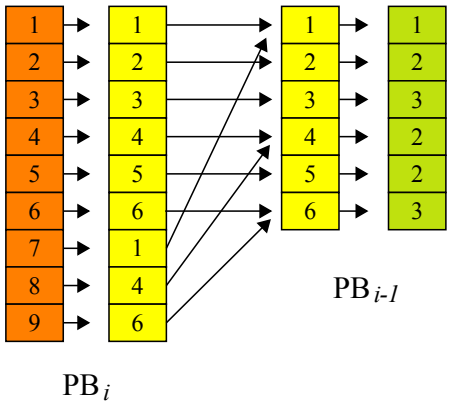


Figure 5.3: *Transition between LODs in the progressive buffer.*

Unfortunately this problem is unavoidable with this approach: either the expressive power of the multi-resolution is reduced by introducing more dependencies or the quality of the triangulation suffers in order to preserve a crack-free surface. The heuristic used in QVDR to achieve an acceptable trade-off is to allow chain dependencies only when all clusters share some vertices and to disallow dependencies between clusters that will be merged within two levels above in the hierarchy.

It is interesting to note that these conditions on generating dependencies can be seen as an approximation of the rules to the construction of a well conditioned V-partition (see Section 3.5.1) where each chain (group of mutually dependent patches) is an element of a partition, and corresponds to the two conditions in Section 3.5.2: each element of a partition must be compact and a boundary must not survive many levels in the hierarchy.

Recently, Sanders and Mitchell presented *progressive buffers* [103], also based on a partition of the mesh into clusters and progressive meshes multiresolution. A progressive buffer (PB) is a series of vertex and index buffers that represent the mesh at different levels of detail. Figure 5.3 shows two levels of detail of a progressive buffer: PB_i and PB_{i-1}. This structure allows for smooth geomorphing between the two levels of detail.

During the construction of the multiresolution model the mesh is split into multiple clusters and for each cluster a progressive buffer is built. The simplification of boundary vertices is allowed but consistency is enforced between adjacent clusters.

In the rendering stage, the geomorphing weights are computed per-vertex, based on the distance from the camera as shown in Figure 5.4, where r is the radius of the clusters, and the parameters s and k determine how the distance from the camera is translated into a LOD. The geomorphing interval e is chosen such that all vertices have finished geomorphing when the cluster switches LOD from PB_i to PB_{i-1}.

Since the geomorphing weights are computed globally, they match exactly on boundary vertices across clusters, avoiding cracks, and due to the smooth change in position no "popping" effects are visible while changing resolution. This removes the boundary simplification constraints and allow for seamless reconstruction while still using "GPU-friendly" static vertex and index buffers. The parameters s and k are adjusted in real-time to maintain a given framerate.

The geomorphing approach has some drawbacks: it increases the complexity of the vertex

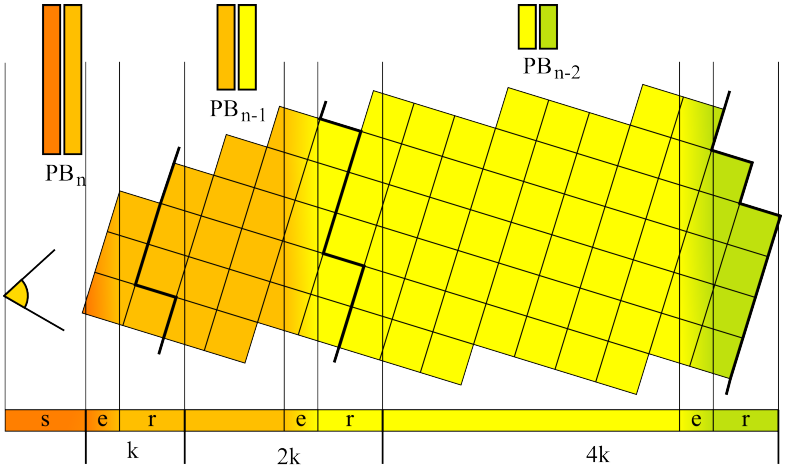


Figure 5.4: *Vertex LOD as a function of distance from the camera.*

shader and does not allow for mesh cache optimization and triangle strips. The drop in performance is significant especially since the bottleneck of most applications lies in the geometry transformation stage of the pipeline.

5.2 Tetrapuzzles

Tetrapuzzles is a technique for out-of-core construction and view-dependent visualization of very large surface models based on a regular conformal hierarchy of tetrahedra to spatially partition the model. The partitioning consists of a binary forest of tetrahedra, whose roots correspond to a decomposition of a cube into six tetrahedra around a major diagonal and whose other nodes are generated by tetrahedron bisection. This operation consists of replacing a tetrahedron s with the two tetrahedra obtained by splitting s at the midpoint of its longest edge by the plane passing through this point and the opposite edge in s. Splitting a tetrahedron in the original cube subdivision results in a 1/2 square pyramid (Figure 5.5a), the next subdivision produces a 1/4 pyramid (Figure 5.5b) and the third splitting a 1/8 pyramid (Figure 5.5c). The shapes produced by the recursion are cyclic: every three levels of decomposition the resulting shapes are similar to the original and half the size.

To guarantee that a conforming tetrahedral mesh is always generated after a bisection, all the tetrahedra sharing their longest edge with s are split at the same time. Such a cluster of tetrahedra is called a diamond (see Figure 5.5). Three kinds of clusters are generated and correspond to the smallest conforming updates that a tetrahedron bisection allows:

1. six 1/8 pyramids split into twelve 1/2 pyramids forming a cube
2. four 1/2 pyramids split into eight 1/4 pyramids forming an octahedron
3. eight 1/4 pyramids split into sixteen 1/8 pyramids forming an octahedron

The hierarchy of this tetrahedral structure has the important property that, by selectively refining or coarsening it on a diamond by diamond basis, it is possible to extract conforming

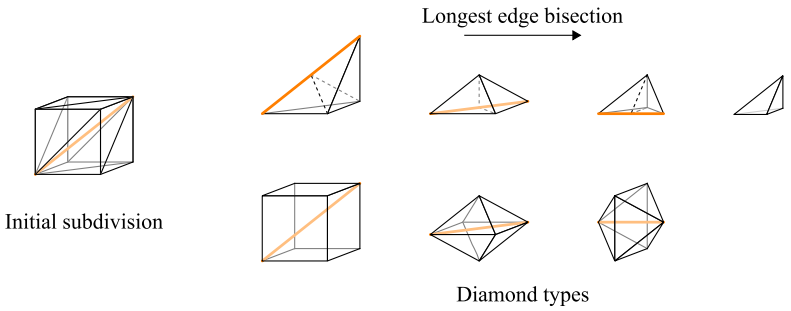


Figure 5.5: *Hierarchy of tetrahedra for space partitioning. The longest edge is colored in red, while next level edges are light-dashed.*

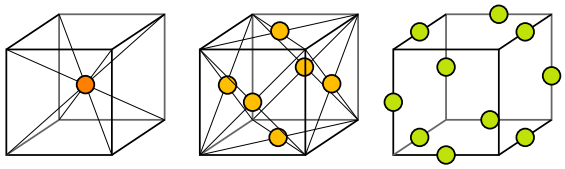


Figure 5.6: *Hierarchy of tetrahedra for space partitioning can be described as a Voronoi based V-partition placing the seeds on the center of the cubes, the faces and the edges.*

variable resolution volumetric mesh representations. We exploit this property to construct a level-of-detail structure for the surface of the input model. This structure is obviously the 3D analogue of the hierarchy of right triangles exploited in BDAM [17] (see Section 4.2).

This hierarchical structure can also be described as a V-Partition based on Voronoi space partitioning: the H_{3i} space partitioning is created by a set of seeds placed in the centers of a cubic subdivision of space, the H_{3i+1} has the seeds placed on the center of the faces of the same cubic subdivision, and the H_{3i+2} is generated placing the seeds on the center of the edges (see Figure 5.6). Every three levels the edge length in the cubic subdivision of the space is halved. The tetrahedra are generated from the intersection of two consecutive diamond partitions H_i. The structure of the patches generated by the intersection of the hierarchical partition with the David mesh is shown in Figure 5.7.

5.2.1 Differences from BDAM

The different characteristics of a full three-dimensional model with respect to a terrain, in particular the irregular sampling and the lack of a natural projection over a plane, require a more general approach to the splitting of the model in the hierarchy, error computation, visibility culling and spatial indexing.

Construction

Since the model is not regularly sampled and distributed in space, the six binary trees encoding the hierarchy of tetrahedra will not be balanced as in the case of terrains in BDAM. The forest is instead built in a top-down fashion, through recursive insertion of mesh triangles by starting

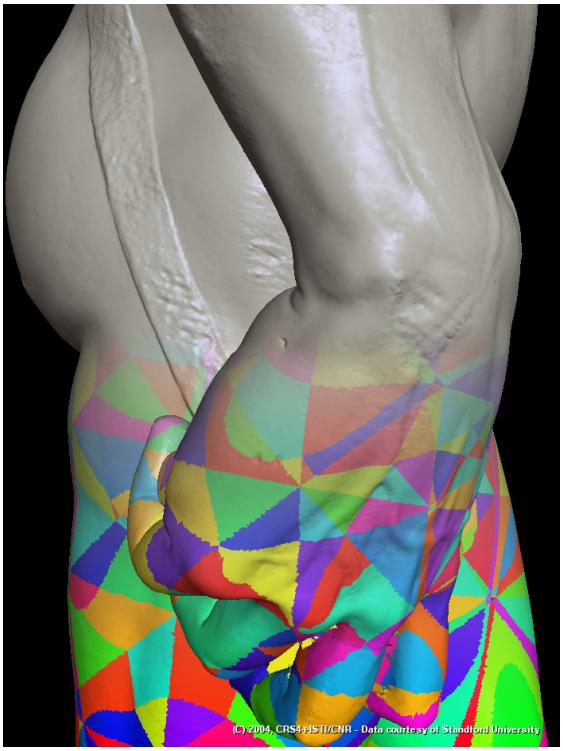


Figure 5.7: *Tetrapuzzles rendering of the David dataset at 1mm error.*

from an initial subdivision of the mesh bounding box into six tetrahedra around a major box diagonal.

When a new triangle is inserted, we locate the leaf that contains its center point and, if the number of triangles already contained in it does not exceed the maximum, we insert the new one into the associated triangle bucket. Otherwise, we refine the hierarchy by tetrahedron bisection and recursively continue the insertion procedure. Each time a tetrahedron is split, all the triangles in the associated bucket are reassigned to its children by a recursive application of the insertion procedure. The end result is a tetrahedron graph that describes the subdivision structure using a DAG of diamonds and a set of triangle buckets associated with leaf tetrahedra that cover the mesh.

The graph, rather small since each node typically contains a few thousand triangles, is maintained in main memory for efficiency reasons, while triangle buckets are stored on disk. This simple partitioning scheme, that clusters triangles solely based on the location of their center point, does not adapt to surface features, and, as for all spatial clustering methods, could lead to patches with exceedingly complex boundaries, and produces clusters with an uneven number of triangles.

Spatial Organization and Data Indexing

To maximize memory locality, all data in the tree and in the corresponding patch repository is therefore sorted by level, then by geometric proximity, by ordering the nodes in a given level by

	000	001	010	011	100	101	110	111
000	000000	000001	000100	000101	010000	010001	010100	010101
001	000010	000011	000110	000111	010010	010011	010110	010111
010	001000	001001	001100	001101	011000	011001	011100	011101
011	001010	001011	001110	001111	011010	011011	011110	011111
100	100000	100001	100100	100101	110000	110001	110100	110101
101	100010	100011	100110	100111	110010	110011	110110	110111
110	101000	101001	101100	101101	111000	111001	111100	111101
111	101010	101011	101110	101111	111010	111011	111110	111111

Figure 5.8: *The Morton code in 2D.*

increasing z-value [100] of their center point.

The z-order, or Morton-order, is a space-filling curve with high locality: the distance in the curve of two points close in space is on average small. This optimizes the probability that two patches adjacent in space have consecutive storage position on disk and eventually on the various caches in the system, thus reducing overall access time.

The z-value of a point in an n-dimensional grid can be computed by interleaving the bits of the binary representation of the $x_0 \dots x_n$ coordinates (each represented using a fixed number of bits) and in two dimensions corresponds to the z-curve; in Figure 5.8 we show a two-dimensional example.

Each tree is stored as a memory mapped linear array, and each of its nodes, corresponding to a particular tetrahedron, contains the following information:

- a reference to the associated patch data (vertex attributes and connectivity) in a patch repository;
- the tight bounding sphere and bounding cone of normals for the patch;
- the saturated model space error and bounding sphere of the neighborhood;
- the index of child nodes in the linear arrays, which correspond to the two tetrahedra generated by bisection.

Error Metric

The error associated to the cluster is computed differently: we have taken the common approach of deriving the model space error ϵ directly from the quadric metric ϵ_q (see, e.g., [66]). We employ the simple formula $\epsilon = s\sqrt{\epsilon_q}^3$ where s is an empirical scale factor for converting to world units. The scale factor is determined prior to rendering time by finding the smallest value of s leading to no image difference in a fixed number of random views, when setting the screen space tolerance below 1 pixel.

For simple clustered back-face culling we compute the normal cones which, together with the bounding spheres, are computed using optimal methods from computational geometry for

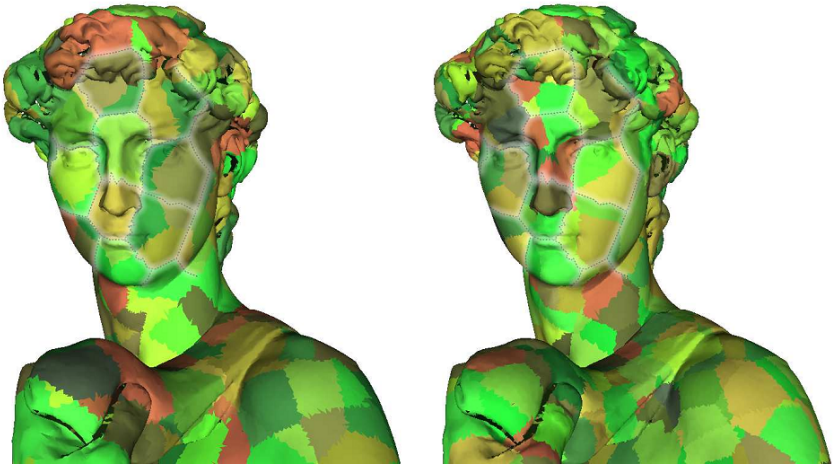


Figure 5.9: *Two consecutive V-partition levels L_i, L_{i+1} of the David mesh. The enhanced lines show some of the borders of the partition H_i that remains unchanged between the two steps.*

finding the minimum enclosing ball of points (for leafs) and minimum enclosing ball of balls (for all other nodes) [35].

Finally we apply the error saturation technique to the bounding spheres and error values. The algorithm is analogous to the case of a right triangle hierarchy: the bounding sphere of a patch must be the same for all the patches in the same *diamond* (Figure 5.5), and must enclose all the bounding spheres of its children. A simple bottom-up procedure is used to enforce this condition.

Extraction Algorithm

The variable resolution rendering algorithm is very similar to BDAM, we just added a hierarchical back-face culling test: we traverse the hierarchy and test if the current node is invisible or fully back-facing by checking the tight bounding sphere and cone of normals of the associated cluster against the current view volume and eventually cull away the entire branch of the tree. If the node is potentially visible, we test whether its patch is an accurate enough representation by measuring its saturated screen space error (Section 3.6.1). If so, we can render the associated patch, otherwise we continue the recursive refinement with the node's children. The consistency of the extracted mesh is guaranteed by the same saturation techniques employed in BDAM: we enforce the two conditions that the bounding sphere of all the tetrahedra in a diamond must be the same and include all the bounding spheres of the children.

5.3 Batched Multi Triangulation

In Batched Multi Triangulation [20] (BMT) we generalize the tetrapuzzles and BDAM approaches by proposing a clustered multi-resolution framework based on the Multi-Triangulation (MT) and introducing the V-partition (presented in Section 3.5.1) as a robust technique to build a well conditioned data structure. We favored an irregular adaptive V-partition which has the

advantage of minimizing boundary and generating more uniform and compact clusters with respect to a fixed space partition. The Voronoi volumes generated by the Lloyd relaxation algorithm, described in Section 3.5.3, follow closely the surface and produce much more compact intersections: compare the patches in Figure 5.7 and 5.9. An irregular V-partition requires an explicit DAG and thus a more complex data structure and extraction algorithm. The additional computational cost is however negligible, given the limited size of the DAG.

The error saturation technique for an irregular V-partition is only apparently more complicated then tetrapuzzles (Section 5.2.1): using the DAG it is simple to find the children of each patch and the patches belonging to the same node, which corresponds to a diamond in tetrapuzzles and an element of H_i in BMT. The same bottom up procedure can be used to enforce the two conditions on bounding spheres and errors.

5.3.1 Boundary Management During Construction

While tetrapuzzles rely on the unification of vertices when merging patches for the simplifications step, in BMT we choose to allow for replicated vertices possibly having different attributes such as normals. Boundary vertices, that are shared between patches, are replicated but identified for easier patch processing. For each patch p we maintain the list L of all the vertices that have external dependencies. The entries of L are triplets (v_p , q, v_q), denoting, for each vertex v_p in p, the patch q that refers to it and its position v_q inside q.

When a set of patches P is requested for being processed and modified in-core, we exploit these lists to efficiently unify vertex indexes and to mark the vertices that are referred by not loaded patches. When the in-core mesh portion has to be written back, the user can change the mesh partitioning scheme, defining a new set of patches that covers the same mesh portion. In this case, we also update the boundary lists of the patches that are not loaded but referring to vertices in the current portion P. Once we have a partition sequence, starting from the finer partition we have to load patches in memory, to simplify them and effectively build the whole MT structure. Moreover it is rather simple, with this scheme, to perform out of core mesh healing processes like smoothing and small hole filling.

Note that this approach is somewhat independent from how the patches are actually stored. This allows to use the same structure also for rendering purposes, just changing the final format of the stored patches.For sake of rendering efficiency we can store patches as optimized triangle strips with precomputed normals in this case, and then optionally compress them with an algorithm designed for a fast decompression stage.

5.4 Results and Comparisons

An experimental software library and a rendering application for both tetrapuzzles and BMT have been implemented on Linux using C++ with OpenGL. We have tested our system with a number of large surface models.

5.4.1 Preprocessing

Table 5.1 lists numerical results for the tetrapuzzles out-of-core preprocessing method for a number of runs on all the test datasets. The tests were executed on a moderately loaded network

Model	Size (triangles)	Time	Size Ply	Size Out size
Bonsai	6M	9 min	520Mb	76Mb
David 1mm	56M	104 min	1154Mb	967Mb
S. Matthew	373M	665 min	7611Mb	5887Mb

Table 5.1: *Numerical results for the construction of the tetrapuzzles model relative to a small cluster of 4 PC's. The output size includes the normals per vertex.*

Model	Size (triangles)	Time	Size Ply	Size BMT size
Lucy	28M	54 min	520Mb	854Mb
David 1mm	56M	97 min	1154Mb	1640Mb
S. Matthew	373M	357 min	7611Mb	11600Mb

Table 5.2: *Numerical results for the construction of the BMT model relative to a small cluster of 4 PC's. The BMT size includes the normals per vertex.*

Model	Size (triangles)	Time	Size Ply	Size BMT size
Power Plant	12.2M	64 min	485Mb	652Mb
Isosurface	100M	350 min	2543Mb	3726Mb
S. Matthew	373M	2369 min	9611Mb	13992Mb

Table 5.3: *Numerical results for the construction of the BMT model relative to a small cluster of 4 PCs. The Ply size here includes the normals per vertex.*

of 4 PCs running Linux 2.4. Each PC has two CPU Athlon 2200+ CPUs, 1GB DDR memory, a 70GB ATA 133 hard disk, and a Ethernet 100 Mb/s network connection. We constructed all multi-resolution structures with a prescribed maximum leaf size of 4000 triangles/tetrahedron for the partitioning phase and an average non-leaf size of 2000 triangles/tetrahedron for the bottom-up construction phase.

Overall processing times range from about 3K-4K triangles/s for 1 CPU to 15K-30K triangles/s for 14 CPUs. Current state-of-the-art high quality out-of-core methods achieve simplification rates up to 70K triangles/s [109]. Our speed is currently slower mainly because we generate a full multi-resolution structure, as opposed to a single small model, and simplification is only a single step of diamond processing that also includes cache-coherent stripification, mesh compression, and optimal bound computation, each costing as much as simplification. As the number of CPUs increases, construction time, initially dominated by diamond processing, starts to be dominated by raw I/O. The almost linear reduction in processing time shows the efficiency of the distributed approach. The large I/O overhead is due to the repeated access to the temporary triangle repository during bottom-up construction, stored on a slow IDE disk.

In Table 5.2 we present the result for the BMT technique also produced on a small cluster of 4 PCs: the timings are similar and the small difference (both in slightly reduced preprocessing time and increased output size) are due to lack of a compression stage. A comparison with QVDR technique is given in Table 5.3, which also produces similar results.

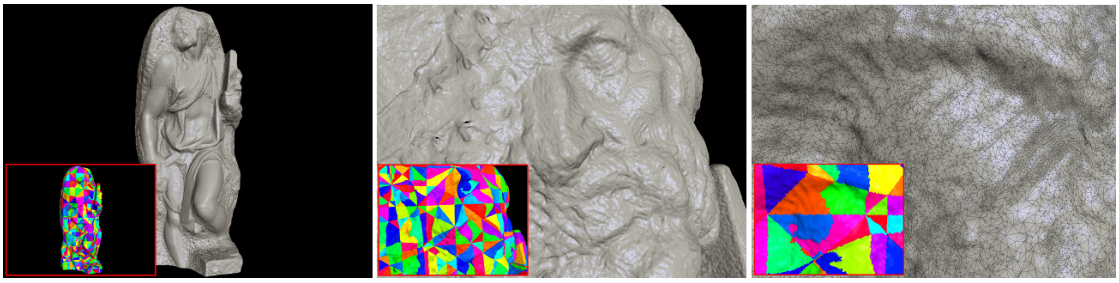


Figure 5.10: *View-dependent rendering of the St. Matthew dataset. The full resolution model contains 373 million triangles and is inspected at over 40 fps on a commodity PC platform. The main images present the mesh rendered with Gouraud shading using 4x Gaussian Multisampling on a 1280x1024 window, while the small inset figures depict the adaptive mesh structure with a different color for each patch. The rightmost image also shows the adaptive triangulation.*

5.4.2 Rendering

All the tests for the BMT algorithm were done with window size 800x600 on a Windows machine equipped with an AMD Athlon 64, 2 GHz, 512 MB Ram, SCSI hard disk, bus AGP 8x and graphics card GeForce 6800 GT. The BMT algorithm is able to render around 4M triangles per frame at 35 fps with a precision of 1 pixel, computed as the average length of the triangles projected onto the screen. The adaptive tetrapuzzles approach sustains an average rendering rate of 70 million triangles per second on a Linux equipped PC with a GeForce FX 5800 Ultra Graphics, and produces comparable results on the same hardware setting. QVDR, instead, sustains 771K triangles per frame at 17fps on a GeForce Ultra FX 5950 GPU.

Geomorphing could be implemented in tetra-puzzles or BMT: the simplification step of each node (element of H_i) would be recorded as a progressive mesh and split into n progressive meshes $L_{j,i_1}, \ldots L_{j,i_n}$, one for each patch. At run time a per-node error would ensure that all the patches perform the same amount of geomorphing to avoid cracks. Patches in different nodes would not generate cracks due to the constrained boundary. The algorithms have similar characteristics for the rest.

On similar machines, the peak performance of Lindstrom's multi-resolution vertex clustering method [66] was measured at roughly 3 million triangle per second, with a representation update latency of up to 1s, even though the model was radically downsampled during preprocessing to only 3M polygons.

The increased performance of both BMT and tetrapuzzles 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-transform-and-lighting cache with optimized indexed triangle strips. This compares favorably also with the latest point rendering approaches, that render in-core models of a few million polygons at 10M-50M points/second depending on filtering quality [26, 11]. The overhead of the prefetching and rendering code, measured by repeating the test without executing OpenGL calls, is only about one third of total frame time, and is mostly due to external data access, mainly I/O wait and patch decompression.

Chapter 6

Applications to Animated Models

Although advances in massive-model rendering techniques for static models have been numerous, relatively few research efforts have focused on dealing with time-varying, dynamic, or animated models. Dynamic geometry occurs in many fields of computer graphics, basically whenever three-dimensional objects are considered over time or some other fourth parameter dimension. When performing mesh morphing, character animation or physically-based animation of deformable or rigid objects, the topology of the objects usually does not change and it is feasible to use explicit surface representations such as triangle meshes. Often, further care is taken to maintain a fixed mesh connectivity throughout the animation so as to allow for a more efficient processing of the sequence, e.g. texture mapping or compression.

Topological changes are much easier dealt by using an implicit representation of the object to be animated. Water and other fluids, for example, are usually simulated by discretizing the appropriate differential equations and carrying out the computations on a volumetric grid. For each time step, the result of the simulation is an iso-surface and in order to get an explicit representation of the sequence, these iso-surfaces are usually extracted one by one. The whole animation is then given as a set of individual triangle meshes with varying mesh connectivity which complicates further processing of the sequence. A critical point is in particular the interpolation between successive frames.

The approach we followed in *Interactive Rendering of Dynamic Geometry* [90] however, was to interpret the whole animation sequence as a data set in a four-dimensional space and represent dynamic geometry as a general four-dimensional mesh. In this chapter we will show how to adapt algorithms and data structures described previously (simplification, batched multi-resolution, and rendering) to this specific case.

6.1 Hypermeshes

A *hypermesh* is a collection of tetrahedra, but in contrast to volumetric tetrahedral meshes, its vertices have four coordinates: three spatial and one temporal. Slicing such a 4D mesh with a three-dimensional hyperplane that is perpendicular to the time axis gives a triangle mesh in 3D that represents the animation at a certain time frame.

The advantages of this approach in handling dynamic geometry are twofold: firstly, it treats

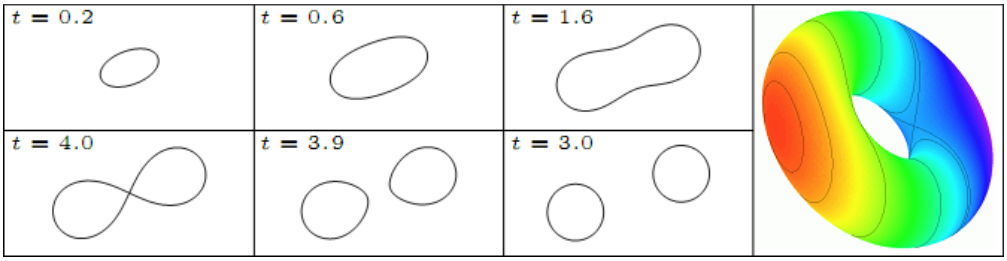


Figure 6.1: *An animated curve in 2D can be represented as a surface in 3D.*

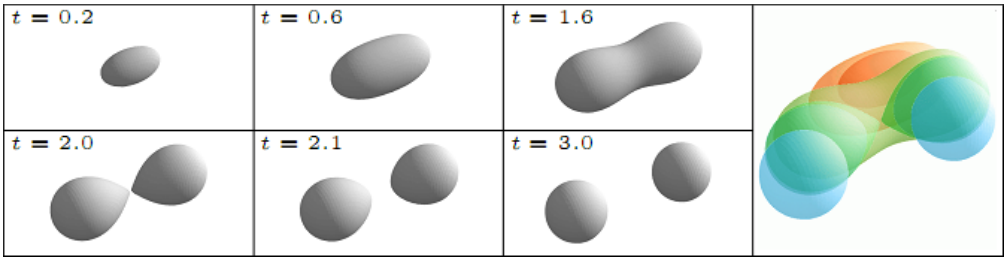


Figure 6.2: *An animated surface in 3D can be represented as a hypersurface in 4D.*

the time coordinate in the same way as the spatial coordinates and thus it is possible to adapt existing 3D algorithms to this setting; in particular, this allows simplification algorithms to exploit temporal coherence (see Section 6.3). Secondly, topological changes that may occur in the geometry as it is animated over time do not need to be treated in a special way, because they are naturally built-in features of hypermeshes.

The concept of embedding an animated sequence of objects in a space with one more dimension is certainly not new, but nevertheless let us quickly review and formalize the basics. Let's drop one coordinated for illustration purposes. Assume that we are given for any parameter $t \in \mathbb{R}$ a curve $C(t) \subset \mathbb{R}^2$ in the plane. Then by adding t as a third coordinate, we can represent the union of all $C(t)$ as a 3D object,

$$C = \{(C(t), t) : t \in \mathbb{R}\} \subset \mathbb{R}^3.$$

More precisely, C is a two-dimensional surface in 3D. Slicing this surface with the plane $P(t) = \{(x, y, t) : (x, y) \in \mathbb{R}^2\}$ that is orthogonal to the t-axis just gives back the curve at parameter value t,

$$C \cap P(t) = C(t).$$

Figure 6.1 illustrates this concept.

Likewise we can embed a sequence of surfaces $S(t) \subset \mathbb{R}^3$ into R^4 to give the *hypersurface*

$$S = \{(S(t), t) : t \in \mathbb{R}\} \subset \mathbb{R}^4.$$

Again, intersecting S with a t-orthogonal hyperplane gives back the surface $S(t)$; see Figure 6.2 for an example.

In the same way that it is common to use triangle meshes to describe surfaces in 3D, it is also natural to represent a hypersurface in 4D as a tetrahedral *hypermesh* H. In contrast to volumetric

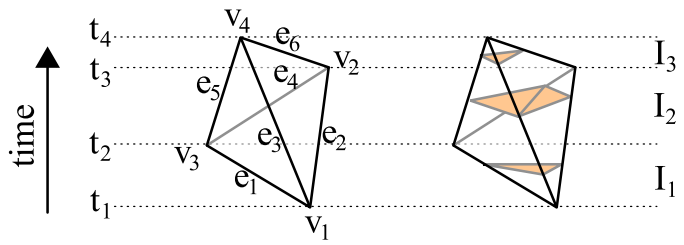


Figure 6.3: *Notation for a tetrahedron (left) and possible cases that occur when it is intersected with a plane (right).*

tetrahedral meshes, the vertices $v_i = (x_i, y_i, z_i, t_i)$ of a hypermesh are four-dimensional, but each tetrahedron still is the convex hull of four vertices, $T = [v_1, v_2, v_3, v_4]$, with six edges $e_1, \ldots, e_6$ (see Figure 6.3). Without loss of generality we assume the vertices to be ordered according to their t-values, i.e., $t_1 \leq t_2 \leq t_3 \leq t_4$.

When a tetrahedron T is intersected with a t-orthogonal hyperplane $P(t)$, we need to distinguish three cases (see Figure 6.3). If $t < t_1$ or $t > t_4$, then the intersection is empty. For $t \in [t_1, t_2]$ and $t \in [t_3, t_4]$, it is a triangle whose corners are the intersections of P with the edges e_1, e_2, e_3 or e_3, e_5, e_6, respectively, and if $t \in [t_2, t_3]$, then we get a quadrilateral that we split into the two triangles whose corners are the intersections of $P(t)$with e_2, e_3, e_4 and e_3, e_4, e_5. The union of the triangles that we get by intersecting all tetrahedra of a hypermesh H in this way is a triangle mesh $M(t) = H \cap P(t)$ with vertices in $\mathbb{R}^3$ in the same way that the intersection of a triangle mesh with a plane gives a planar polygon.

Without loss of generality, we assume the tetrahedra to be non-degenerated in the sense that not all four corners have the same t-coordinate. The intersection with $P(t)$ would be a volume in this case, but as long as the sequence $S(t)$ is continuous in t, such kind of degeneracy does not occur.

Computing the intersection $M(t)$ is very similar to the extraction of an iso-surface from a 3D tetrahedral mesh with scalar values assigned to its vertices. In fact, if the scalar values are interpreted as the time coordinate, then both operations are exactly the same and the analysis of the different intersection cases can also be found, e.g. in [88, 59].

Scalar-valued 3D tetrahedral meshes, however, are special cases of hypermeshes and not vice versa: the triangle meshes $M(t)$ can intersect in 3D for different values of t and therefore it is not possible to simply interpret the time coordinate of the hypermesh vertices as a scalar attribute and convert H into a 3D tetrahedral mesh.

6.2 Generating Hypermeshes

Both explicit and implicit representation of dynamic meshes can be easily converted into hypermeshes with straightforward modification of the algorithms to extract 2D surfaces from implicit representation or terrain grids.

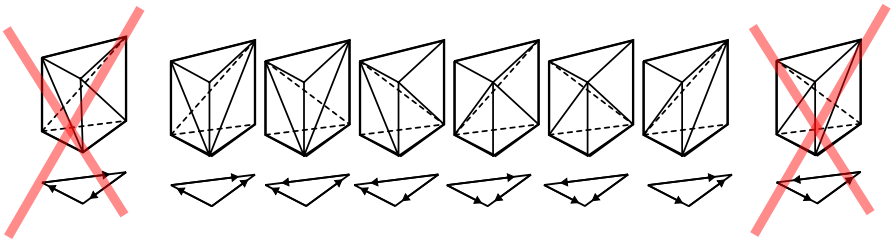


Figure 6.4: *Six different valid ways to split a prism into tetrahedra, and the two "forbidden" diagonal configurations.*

6.2.1 Iso-surfaces from 4D Grids

Given an n-dimensional scalar function $f : \mathbb{R}^n \to \mathbb{R}$, one is often interested in the iso-surface

$$I_f(c) = \{x \in \mathbb{R}^n : f(x) = c\}$$

for some iso-value c. In most cases, f is sampled at the vertices a grid. For example, when the level set method is used for liquid simulations, then f is th e*level set* function $\phi(\vec{x}, t)$ [86] and the surface of the simulated liquid is the iso-surface $I_\phi(0)$. For practical reasons, the simulation is usually computed on a regular 3D grid and by collecting these grids for all time steps, the whole simulation sequence becomes a regular scalar-valued 4D grid.

For $n = 3$, the marching cubes (MC) algorithm [74] and its variants are common tools to compute a triangle mesh that approximates the iso-surface $I(c)$. The MC algorithm has also been extended to extract hypersurfaces from four-dimensional volumes with both tetrahedral [120, 121] and hexahedral elements [97, 9]. Like the original MC algorithm, these 4D variants compute a piecewise linear approximation of the hypersurface, i.e. a tetrahedral mesh with four-dimensional vertices, or simply a *hypermesh*, although the number of local configurations that can occur in each cell is much higher.

Such data is produced, e.g. when liquid simulations are computed with the level set method [86, 85] and the latest techniques [84, 56] can compute such detailed simulations that the raw data size easily reaches several GB.

6.2.2 Compatibly Meshed Sequences

We developed a simple algorithm to also build hypermeshes from compatibly meshed sequences of triangle meshes (Section 6.2.2), which can result either from mesh morphing [1] or be generated from general mesh sequences, e.g. by remeshing [4].

In many cases, dynamic geometry is represented as a sequence of meshes where the connectivity is fixed while the positions of the vertices change over time. Morphing [1] algorithms, cloth simulations and other non-skeletal animations produce surfaces that undergo big non-rigid deformations. In order to accurately approximate the surface in all frames, the resulting meshes are usually very dense because once a detail requires a fine resolution in some frame, this structure is also present at all other time steps. A clever approach to thin such meshes and build an adaptive hierarchy has been proposed by Kircher [60].

We can also use our machinery to handle this kind of data because a compatibly meshed

sequence can easily be converted into a hypermesh. As a triangle moves from one time frame to the next, it creates a prism in $\mathbb{R}^4$, so that the whole sequence can be seen as a collection of such prisms. Splitting each prism into three tetrahedra as shown in Figure 6.4 finally yields a hypermesh. We must only take care that the splitting of the prisms is compatible in the sense that the diagonal splits of the quadrilateral faces must match and avoid configuration where no diagonals meet in a vertex, which would require additional vertices to split the prism into tetrahedra. Seen the other way round, we need to choose one diagonal for all faces between neighboring prisms such that all prisms end up with one of the six possible splits. Note that this needs to be done only for one layer of prisms as the connectivity is the same in all other layers.

The same splitting problem has been discussed by Porumbescu et al. [92] and to find a solution, we could use their algorithm, which works well in practice but is not proven to converge. However, there is a much simpler strategy: each face between neighboring prism correspond to an edge in the base mesh (Figure 6.4), and the problem reduces to pick an orientation for each edge of the mesh such that in no triangle three edges form a cycle. Any spanning tree would be a solution, or even simpler as shown in [81], by indexing the vertices of the mesh and by always taking the orientation from the vertex with the smaller index to the higher one no cycle can be created.

6.3 Simplification

Hypermeshes are often over-sampled and too large to fit into the memory of the graphics card or even the main memory. As for triangle meshes, both issues can be addressed by simplifying them. For simplifying dynamic geometry we implemented an algorithm [45] that is based on the quadric-error metric [44] introduced in Section 2.2 on page 12.

In the adapted version, this simplification algorithm performs successive collapse operations that each remove one edge and all incident tetrahedra. The selection of which edge to collapse is guided by the quadric error metric which in our case measures the approximation error in space and time simultaneously. Note that this is different from working with volumetric tetrahedral meshes [116, 119, 15] in the same way that simplifying triangle meshes in 3D is different from the simplification of planar triangulations.

Instead of the usual greedy approach we used a *multiple-choice randomized collapse* algorithm [82, 122, 119], which produces almost the same quality but with significantly reduced computation costs. Since no global structure (e.g. a priority queue) needs to be maintained, the algorithmic structure of multiple choice decimation is extremely simple: in each step a random set of candidates is picked and the best among these candidates is chosen. Usually a small number of 10 to 15 random candidates is sufficient to produce meshes with the same quality of those produced by greedy algorithms.

The quadric error of x with respect to the tangent space at a vertex v is given by

$$Q_v(x) = (x - v)^T A (x - v)$$

where A is the sum of outer products of normals,

$$A = \sum_i n(T_i)\, n(T_i)^T$$

and the summation index i ranges over the set of all tetrahedra T_i incident to v. The normal $n(T)$ of a 4D tetrahedron T is well-defined because T is "flat" in the sense that it is contained in

a hyperplane of codimension one. The normal can be computed by taking the 4D cross product of three edge vectors of T.

The quadric error associated to the each edge collapse is linear invariant: if we apply a linear transformation M to the data, run the algorithm and transform the result back with M^{-1}, then we get the same as using the algorithm with the untransformed data. Indeed, if we apply M to the data and accordingly M^{-T} to the normals, we have

$$Q_{Mv}(Mx) = (x - v)^T M^T \left[M^{-T} A M^{-1}\right] M(x - v) = Q_v(x).$$

In particular, this means that Q is scale invariant in the time direction and thus we are free to choose the time scale.

6.4 Multi-Resolution

Multi-resolution structures for the special case of deforming meshes with constant connectivity can be found in Kircher and Garland [60] and in Shamir et al. [108]. The second technique is based on an adaptation of the Multi-Triangulation technique [94, 36] and allows changes in the topology and connectivity of the sequence of meshes, but it is unable to exploit temporal coherence of the surfaces when no unique correspondence between the vertices of the meshes in the sequence is given.

Two multi-resolution data structures for tetrahedral meshes, based on edge collapse and vertex decimation simplification algorithms, are described in [27], extending the MT structure described in Section 2.3.2. A recent tutorial [16] discusses simplification, multi-resolution, compression, visualization of tetrahedral meshes, which are relevant also in the case of hypermeshes.

All these algorithms focus on storage cost and accuracy of the representation in terms of number of tetrahedra for a given approximation error. For these reasons the granularity of the multiresolution is as small as possible, resulting in huge DAGs with all the drawbacks explained in Section 3.4.

In the following we will describe a patch-based multi-resolution structure for hypermeshes based on the BMT framework. In particular, this allows to extract and render consistent meshes with view-dependent resolution at interactive rates in combination with out-of-core techniques to handle large meshes. A similar approach has recently been used for tetrahedral 3D meshes by Sonderhaus et al. [112].

6.4.1 Batched Multi-Triangulation for Hypermeshes

The BMT technique for the construction of the multi-resolution structure can be applied to hypermeshes with some obvious modification due to the increased dimensionality: Lloyd's Voronoi relaxation [73] for the construction of the patches is directly applicable in the 4D case; the sequence of merge, simplify and split described in Section 3.5.1 is exactly the same. The increased dimensionality, however, exacerbates a few problems which were overlooked in the previous settings.

In general, the ratio between the size of the boundary and the interior of an n-dimensional object grows with n. Therefore, the patches of a hypermesh have on average much more boundary vertices than those of a triangle mesh. Thus, it is crucial to keep the patches well-shaped, because

otherwise it may quickly happen during the construction of the hierarchy that the patches consist of mostly boundary vertices and cannot be simplified any further.

For dynamic geometry, one important factor that has a big influence on the shape of the patches is how the time unit is related to the three spatial units, because we measure distances with the standard Euclidean norm in 4D. Decreasing the time scale makes the patches longer and thinner in time and vice versa, and we would like to find the "correct" scale that gives patches as uniformly sized as possible. Note that choosing the time scale does not affect the simplification process (see Section 6.3).

For data from the 4D marching cubes algorithm, the obvious choice is to set the time between two frames equal to the size of the marching cube. During the simplification process, however, tetrahedra in surface regions that move little, become elongated in the time direction. During the Lloyd relaxation we therefore use for every Voronoi cell an individual distance norm that weights the time direction such that the average shape of the tetrahedra in that cell is uniform in all directions with respect to this norm. Of course, the same idea could also be used in the 3D setting, but for surfaces it is much less crucial to have well-shaped patches.

The storage cost for a patch with m tetrahedra and n vertices is then $16n + 8m$ bytes. Even though this requires more vertices to be replicated at the boundary of the patches, it reduces the overall memory requirement by about 40%, because the number of replicated vertices is comparatively small. This approach is somewhat independent from how patches are actually stored and allow to use the same structure for rendering. In this case each patch is further preprocessed for GPU-assisted rendering as described in Section 6.5.1. The overall storage cost of this structure is around $120n$ bytes where n is the number of vertices in the reference mesh, while the overhead of the DAG storage and vertex replication is negligible. By comparison the explicit data structure for tetrahedral MT described in [27] requires from $450n$ to $880n$ bytes. This is due to the reduced granularity and expressive power.

The Edge-Base MT also described in [27] requires $36n$ bytes. It is, however, a compressed format which requires a substantially higher computational cost with respect to the explicit structure. Compression of topological information in tetrahedral meshes [125] achieves approximately 3 bytes per tetrahedra (or 0.6 bytes per vertex) on small meshes. If we were to compress each patch the overall cost would be $34n$ bytes, with the advantage that it would be trivial to parallelize the compression/decompression threads.

6.4.2 Rendering the Multi-resolution Model

The *geometric error* for a patch is derived from the quadrics used for the simplification similarly to Tetrapuzzles and BMT, thus also accounting for the temporal dimension. To compute the *screen space error* we just divide the geometric error from the distance from the viewpoint to the bounding sphere. Note that the word "distance" refers to the 4D distance and includes the temporal dimension. Frustum culling also can be easily extended in the 4D case by additionally checking for intersection in the temporal dimension.

Under this error metrics regions that are further away from the viewer, both in time and space, require a lower resolution than those that are close to the camera. In Figure 6.11 it is apparent how the distance in time affects resolution. The quick decrease of the resolution in time allows for a complete 4D model to be extracted and maintained in memory and allows for rendering of a complete model without latency even for sudden changes in view-direction or time.

6.5 GPU-Assisted Rendering

Intersecting a hypermesh with a three-dimensional hyperplane gives a triangle mesh in 3D and by using standard features of modern graphics cards it is possible to compute this intersection directly on the GPU. Very similar techniques have been proposed for GPU-accelerated iso-surface extraction from tetrahedral 3D meshes [88, 63, 59, 12]. In fact, any such unstructured grid can be turned into a hypermesh by interpreting the scalar values given at the vertices as a fourth time coordinate. Extracting the iso-surface is then equivalent to computing the hyperplane intersection.

Our algorithm is similar to the idea of Kipfer et al. [59], and also uses an edge-based data structure, but instead of using SuperBuffers (which are not a standard extension) to avoid redundant intersection computations, we pre-order the primitives so as to make optimal use of the GPU vertex cache. A novel feature of our approach is that we completely discard the tetrahedral structure of the hypermesh and convert it into the special data structure of *dynamic triangles* (Section 6.5.1) instead. In short, each tetrahedron is replaced by four triangles and for each triangle we precompute the time interval for which it is part of the intersecting hyperplane. In this way, about 40% less faces need to be rendered.

The basic algorithm for rendering a hypermesh at a certain time-coordinate t is to slice all tetrahedra with the hyperplane that intersects the time axis at t and is perpendicular to it (see Section 6.1). But doing this in the CPU and sending the resulting triangles to the graphics card is inefficient because the CPU is much slower in processing geometry than the GPU. There exist a few techniques to perform these computations on the GPU and each one could be used to render the patches of the multi-resolution structure, while offering different space-speed trade-offs.

The technique of Pascucci [88] basically processes a quad for each tetrahedron by storing the coordinates of the four vertices in the vertex attributes of each vertex of the tetrahedron and performing a marching tetrahedra on-the-fly. It is very fast but the multiple replication of the vertex coordinates makes it impractical for big datasets, due to the memory overhead. Buatois et al. [12] avoid this replication by storing the tetrahedral indexed structure in textures and using the texture access capability of the vertex shader. Unfortunately, the access to the vertex texture is quite slow in current graphics cards and this technique requires 20 accesses per tetrahedron. Kipfer et al. [59] developed an edge-based approach, whose main strength is that it avoids redundant computations of edge-surface intersections. This technique, however, takes advantage of a specific feature of a single graphics card vendor: the SuperBuffer extension of ATI cards.

It should be stressed that *any* tetrahedral visualization algorithm could be used in combination with the batched multi-triangulation structure for hypermeshes. We developed yet another strategy which is tailored to our specific needs, in particular it is faster than [12] while requiring more space. In Section 6.6 we compare the performances of the two techniques.

6.5.1 Dynamic Triangles

As described in Section 6.1, three cases need to be distinguished when the hyperplane $P(t)$ intersects a tetrahedron T, and four different kind of triangles can occur. We store all of these four triangles as *dynamic triangles* in the sense that each triangle is associated with a "lifespan" and the three edges on which its vertices lie. In the notation of Figure 6.3, the first triangle "lives" during the time interval $I_1 = [t_1, t_2]$ with vertices on the edges e_1, e_2, e_3 and likewise for

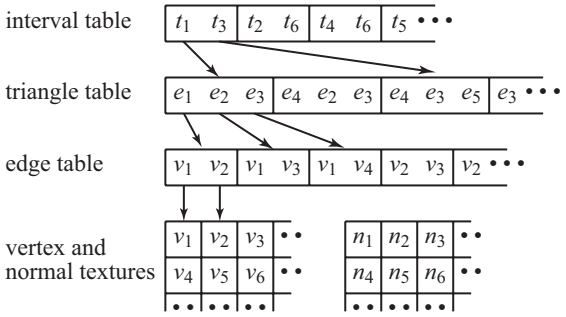


Figure 6.5: *Data structures used for GPU rendering.*

the other three triangles, so that we store

$$\triangle_1 := \{I_1, (e_1, e_2, e_3)\}, \quad \triangle_2 := \{I_2, (e_2, e_3, e_4)\},$$

$$\triangle_3 := \{I_2, (e_3, e_4, e_5)\}, \quad \triangle_4 := \{I_3, (e_3, e_5, e_6)\}$$

for each tetrahedron of the hypermesh H.

Like [59] we further build an edge table for the edges of all tetrahedra. In this table, each edge $e = [v_i, v_j]$ is stored as a pair of indices (i, j) to the vertices that this edge connects. We can now discard the hypermesh data structure because the dynamic triangles, the edge table, and the vertex list contain all the necessary information needed for computing the triangle mesh $M(t) = H \cap P(t)$ for any given time parameter t on the GPU.

Suppose that $t \in I$ for some triangle $\triangle = \{I, (e_1, e_2, e_3)\}$ from the tetrahedron T. For each corner we then use the edge table to look up the indices of the two endpoints v_1 and v_2 and read their coordinates from the vertex list. We finally interpolate the (x, y, z)-coordinates of v_1 and v_2 linearly with the weight $\lambda = (t - t_1)/(t_2 - t_1)$ to get the 3D position of the triangle $T \cap P(t) \in M(t)$.

In order to implement this strategy in OpenGL we store the 4D coordinates of the vertices in an RGBA texture and use vertex buffers to store the triangle table (`ELEMENT_ARRAY_BUFFER_ARB`) and the edge table (`ARRAY_BUFFER_ARB`). The function call "glDrawElements" takes triples of indices from the triangle table and feeds the corresponding values from the edge table to the vertex program, which in turn uses these values to look up the coordinates of the edge endpoints in the texture. It then interpolates them to get the actual 3D coordinates of the corner (see Figure 6.5). Note that any vertex attribute can be treated in the same way as the vertices themselves by storing them in additional textures. In our implementation (shown below) we did this for normals to enable Phong shading and refraction.

We can detect in the vertex program if the lifespan I of a triangle does not contains the current t: the parameter λ for the linear interpolation is negative or bigger than 1 for one of its corners. We can easily cull the triangle by moving the vertex position to the viewpoint so that the whole triangle becomes invisible.

```
!!ARBvp1.0
OPTION NV_vertex_program3;

# Directional light standard transform and lighting modelview
PARAM mv[ 4 ] = { state.matrix.modelview };
```

```
# Modelview*projection matrix
PARAM  mvp[4]       = { state.matrix.mvp };
# Inverse transpose of modelview for normal transform
PARAM  mvinv[4]     = { state.matrix.modelview.invtrans };

PARAM  lightDir     = state.light[0].position;
PARAM  halfDir      = state.light[0].half;

#These values already takes into account material properties
PARAM  ambientCol   = state.lightprod[0].ambient;
PARAM  diffuseCol   = state.lightprod[0].diffuse;
PARAM  specularCol  = state.lightprod[0].specular;
PARAM  time = program.local[0];

TEMP   n1, n2, color, dots, v1, v2, t1, t2, tmp, norm, pos, coef;

#loading first vertex position from texture
MUL pos.y, vertex.position.x, time.x;
FRC pos.x, pos.y;
MUL pos.y, pos.y, time.y;
TXL v1, pos, texture, 2D;

#loading first vertex position from texture
MUL pos.w, vertex.position.y, time.x;
FRC pos.z, pos.w;
MUL pos.w, pos.w, time.y;
TXL v2, pos.zwxy, texture, 2D;

#loading normals
TXL n1, pos, texture[1], 2D;
TXL n2, pos.zwxy, texture[1], 2D;

#interpolating vertex positions
SUB tmp.x, time.w, v1.w;
SUB tmp.y, v2.w, time.w;
ADD tmp.w, tmp.x, tmp.y;

MUL v1.xyz, v1, tmp.y;
MAD v1.xyz, tmp.x, v2, v1;

#remapping normals from texture values [0,1] to [-1, 1]
MAD n1, 2, n1, {-1, -1, -1, 0};
MAD n2, 2, n2, {-1, -1, -1, 0};

MUL n1.xyz, n1, tmp.y;
MAD n1.xyz, tmp.x, n2, n1;

MOV v1.w, tmp.w;

#computing transformed position and storing in texture
DP4 result.texcoord[ 0 ].x, mv[ 0 ], v1;
DP4 result.texcoord[ 0 ].y, mv[ 1 ], v1;
DP4 result.texcoord[ 0 ].z, mv[ 2 ], v1;
DP4 result.texcoord[ 0 ].w, mv[ 3 ], v1;

#trasforming the vertex position in clip coordinates
DP4   pos.x, mvp[0], v1;
DP4   pos.y, mvp[1], v1;
DP4   pos.z, mvp[2], v1;
DP4   pos.w, mvp[3], v1;


#culling out of time-frame vertices (we move them to (0, 0, 0)
SLT tmp.x, 0, tmp.x;
SLT tmp.y, 0, tmp.y;
MUL tmp.x, tmp.x, tmp.y;
SUB tmp.y, 1, tmp.x;

MUL pos, pos, tmp.x;
MAD pos, {0, 0, 1, 0}, tmp.y, pos;
MOV result.position, pos;
```

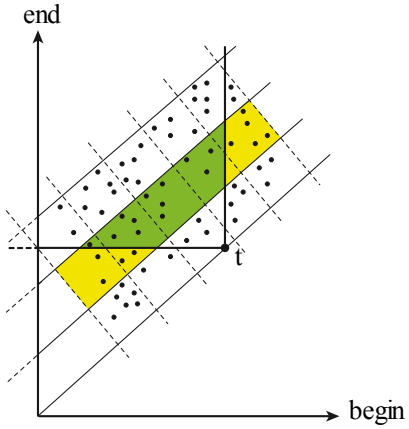


Figure 6.6: *Span space for the lifespans of the dynamic triangles: Each point represents an interval with the abscissa and ordinate referring to the start and end time. The three diagonal bands contain intervals with approximately the same length, with the short-living intervals close to the diagonal. The intervals in the second band that intersect with the current time t are shown in green.*

```
#ransformin the normal in eye coordinates.
DP3   norm.x, mvinv[0], n1;
DP3   norm.y, mvinv[1], n1;
DP3   norm.z, mvinv[2], n1;

# Normalizing  and passing to fragment program as texture coords
DP3  norm.w, norm, norm;
RSQ  norm.w, norm.w;
MUL  result.texcoord[1], norm, norm.w;

## Lighting done in fragment shader
MOV  result.color, diffuseCol;
END
```

Usually a large fraction of the triangles have a lifespan I that does not contains the current t and we would like to process in the GPU only the active ones. Computing the active triangles requires a lot of CPU computations and sending the primitives to the graphics card for each frame requires a lot of bandwidth.

It is more efficient to sort the triangles according to the centre of their lifespan, store them on the graphics card as element array buffers and only process the interval that contains the active triangles. In this way we process a certain number of non-active triangles, but each primitive is transferred to the graphics card only once.

A simple solution is to subdivide the global lifespan of the patch into n regular intervals and store for each interval the indices of the first and last triangle that intersect the interval. The number of intervals compromises between accuracy and space. The problem with this solution is that the dynamic triangles in the patch have lifespans that vary considerably in length. Regardless of the sorting order, the interval that contains the active triangles will usually contain many non-active triangles, which results in a quite inefficient culling.

To prevent this, we group intervals of approximate equal length together in a few "bands" and apply the simple solution above for each band (see Figure 6.6). This results in more calls of "glDrawRangeElements" (one per band) but greatly improves the culling efficiency. The most

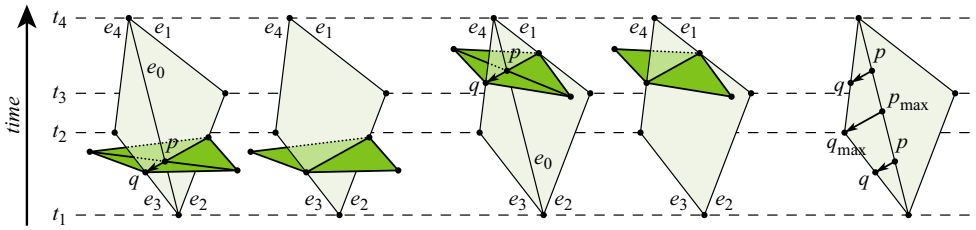


Figure 6.7: *An edge collapse in the triangle mesh $M(t)$ (green) is equivalent to joining two faces of the hypermesh (light green).*

appropriate number of bands depends on the relative cost of calling "glDrawRangeElements" versus that of rendering a tetrahedron.

Another solution is to use an interval tree to compute the active triangles and compact the list into a few intervals such that no interval contains a long sequence of non-active triangles. This is more robust with respect to irregular distributions of lifespans and more accurate, but it also requires more space ($O(n)$) and it is computationally more expensive: $O(\log N + m)$, where m is the number of active triangles.

In our examples, we experienced that about 10% of all triangles are active for each frame on average. The single-band approach typically requires to process approximately 5 times as many triangles as needed, but only 1 out of 5 triangles are actually rendered by the vertex shader. Instead, when using 4 to 6 bands and a number of regular intervals equal to a quarter of the number of tetrahedra, this rate improved to 3 out of 4 triangles on average in all our examples. The space overhead for this banded structure is 4 bytes per tetrahedron.

6.5.2 Optimizing Dynamic Triangles

Overall, the space needed to store the dynamic triangles exceeds that of the indexed hypermesh by about 65%, but we can use two approaches to reduce this number. Whenever two or more vertices of a tetrahedron T have the same time coordinate, some of the intervals I_1, I_2, I_3 are empty and the corresponding dynamic triangles and edges can be removed from the lists. Thus, upon simplification of the mesh (see Section 6.3) we try to align as many vertices as possible in time. This typically removes about 20% of all dynamic triangles.

Another idea to reduce the number of dynamic triangles is based on the observation that the triangle mesh $M(t)$ contains many thin triangles which contribute little to the geometric shape. Thus, if we were to apply a quadric-error simplification algorithm on $M(t)$, we would reduce the number of triangles considerably without significantly increasing the error.

As shown in Figure 6.7, collapsing an edge of $M(t)$ is equivalent to removing an edge from the hypermesh H and combining two faces to form a quadrilateral. In this example, collapsing the vertex p into q removes two faces from $M(t)$, the edge e_0 from H and forms the quadrilateral with edges $e_1, \ldots, e_4$. At the same time, the tetrahedra adjacent to edge e_0 are combined to a volumetric element that is not a tetrahedron (see Figure 6.8).

The structure that results from such an operation is of course not a hypermesh anymore, still its intersection with any time plane is again a valid triangle mesh without gaps, because it is the result of an edge-collapse over a valid triangle mesh.

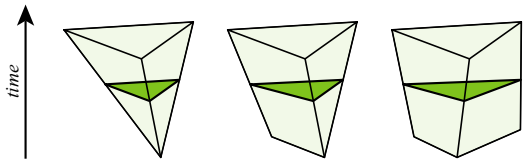


Figure 6.8: *Joining faces of the hypermesh as in Figure 6.7 generates non-tetrahedral elements.*

The quadric error associated with this edge collapse varies with $t \in [t_1, t_4]$ and we can easily determine the maximum: Regarding the surface M, the quadric error associated with the collapse of p into q is $(p-q)^T A(p-q)$, where A is the quadric form associated with the point p. This form is computed using the normals of all the surface triangles incident to it, which in turn correspond to the intersections of the hyperplane $P(t)$ with the tetrahedra adjacent to e_0. The 3D normals of these triangles are the projections of the 4D normals from the corresponding tetrahedra onto $P(t)$. Therefore, they do not change with t as the point p "slides" over the edge e_0 and A is constant. Similarly, the vector $p-q$ only changes in length but not in direction. So the maximum error is taken at $t = t_1$ where the vector is longest.

With this strategy, we can reduce the number of dynamic triangles by another 30% with only a small additional error. It should be noted that these optimizations cannot be applied to general 4D applications, relying on the fact that the slicing hyperplane is always at constant time.

6.6 Examples

We have applied the methods explained in the previous sections to three data sets: three liquid simulations (*dams*, *drops*, and *wave*) and two morphing sequences (*horse-to-man* and *cloth*); see

data set	*dams*	*drops*	*wave (partial)*	*horse-to-man*	*cloth*
input resolution	600 frames at 374×374×374	380 frames at 142×60×86	50 frames at 50×50×90	200 frames with 36 K triangles	400 frames with 39.4 K triangles
hypermesh					
construction	14163 m	177 m	26 m	1 m	3 m
tetrahedra	1172 M	27 M	6 M	22 M	48 M
size on disk	22.95 GB	551 MB	131 MB	448 MB	980 MB
BMT					
construction	5203 m	157 m	32 m	112 m	216 m
tetrahedra	1121 M	19 M	5.6 M	0.8 M	3.1 M
size on disk	15.2 GB	350 MB	92 MB	16 MB	63 MB
dynamic triangles					
(in M tri.)	2268	55	11.3	1.8	7.1
size on disk	23.5 GB	631 MB	137 MB	26 MB	101 MB
render speed	20 M triangles/sec.				

Table 6.1: *Size, timings (in minutes), and memory requirements for the tested data sets at various stages of preprocessing: extraction of the hypermesh, multi-resolution model, dynamic triangles optimization. The full "wave" data set consists of 200 frames.*

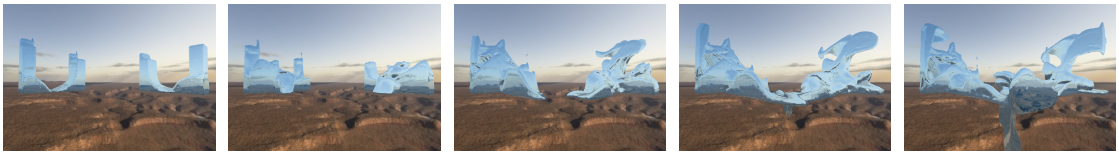


Figure 6.9: *Snapshots from the "dam" sequence.*

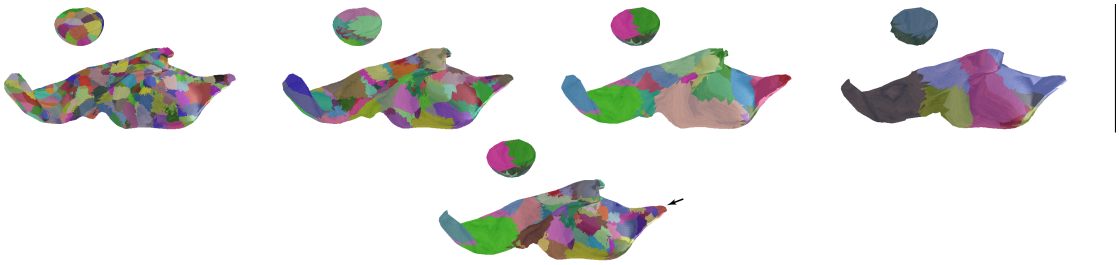


Figure 6.10: *Snapshots from the "drop" sequence with increasing error tolerance (left) and the camera located at the black arrow (right), showing how the size of the patches selected from the multi-resolution hierarchy depends on the error and the viewpoint.*

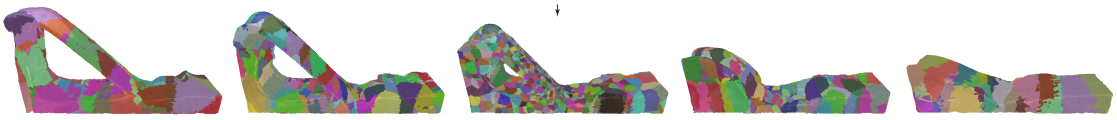


Figure 6.11: *Snapshots from the "wave" sequence for increasing time values with the camera frozen at the time frame in the middle, showing that the size of the patches from the multi-resolution hierarchy depends on temporal distance to the viewpoint.*

Figures 6.9, 6.10, 6.11, and 6.12, respectively. The timings for the different steps of our processing pipeline and the sizes of the different data structures are summarized in Table 6.1. Because the full *wave* data set (200 frames) gives almost the same numbers as the *drops* sequence, we show the timings and size that result from the first 50 frames only to underline that the numbers are basically linear in the size of the data set.

The three liquid simulations were extracted from regular grids with the 4D marching cubes algorithm as described in Section 6.2.1 and most of the time is spent in the big lookup table for the different configurations that can occur in the marching hypercube (64 K cases). This step requires almost 10 days of run time for the large dam model, since we did not investigated more efficient implementations of the 4D marching cubes algorithm. Instead, the hypermesh of the morphing sequence was constructed from a set of compatibly meshed surfaces as explained in Section 6.2.2.

The main cost in the construction of the multi-resolution model is the simplification algorithm, which has not been optimized for speed. Note that it would be possible to speed up this algorithm by distributing the simplification steps over multiple computers as explained in Section 3.6.3.

After pruning the zero-error leaves from the full multi-resolution model, it becomes even smaller than the initial hypermesh. The multi-resolution model of the *horse-to-man* sequence is much smaller than the others as it can be simplified much better. One reason is that the

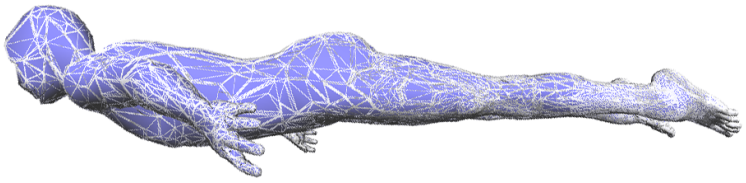


Figure 6.12: *The model resolution is decreasing smoothly with increasing distance from the camera (placed near the feet).*

sequence contains many parts that move linearly over time, another is that it is oversampled in the spatial directions due to the fixed connectivity that contains the details of both the man and the horse.

Finally, converting the data into dynamic triangles triples the size, but the optimization described in Section 6.5.2 reduces the total size by about 40% and we end up with a data set which is comparable to the initial hypermesh in size, but contains all the levels of detail and is optimized for being rendered efficiently with the GPU.

On average, we experienced a render performance of 20 million triangles per second (actual triangles rendered). Thus, at a desired frame rate of 40 frames per second, the rendering algorithm can choose half a million triangles from the multi-resolution model to display the model with the lowest possible error. Since only about 10% of the triangles in a patch are active at a certain time, this amount at a total of 5 millions dynamic triangles, or 190Mb of video RAM. Each patch is used for a few seconds of running animation so that at any given frame only a few patches need to be updated.

Disk access becomes the bottleneck of the system, and while sufficient for normal playback and interactive visualization, the algorithm cannot quickly adapt the resolution of the model to abrupt changes in view-position or time. The implementation of a compression algorithm would greatly improve this limitation, unfortunately, there are still no compression algorithms for dynamic triangles available.

For a comparison, we implemented the technique of Buatois et al. [12] on the multi-resolution tetrahedral structure, and the render performance was only about 10 million triangles per second. This is mainly due to the higher number of vertex texture accesses (20 against an average of 7). However, the lower memory footprint made it faster in adapting the resolution to the view change.

All computations and interactive renderings of the models were performed on a PC with a Xeon 2.8 GHz processor, 1 GB of RAM, and an NVidia GeForce 7900 XT graphics card. The CPU usage peaked at about 30% during the rendering.

The average size of a patch in the multi-resolution model is 3000 tetrahedra. We found this number to be the best compromise between a good granularity of the multi-resolution structure that favors small patches, and the rendering performance that increases with bigger patches.

The first four images in Figure 6.10 show how the size of the patches increases while the camera zooms out of the scene. Since each patch consists of approximately the same number of triangles, the resolution of the triangulation decreases and the model is rendered with an increasing error tolerance.

The rightmost image of Figure 6.10 shows that the size of the patches increases with the distance from the camera (black arrow on the right). As a result, all triangles of the model have approximately the same size on screen. Analogously, Figure 6.12 shows a snapshot from the *horse-to-man* multi-resolution hypermesh where the resolution decreases with increasing distance to the camera.

Figure 6.11 further shows that the resolution of the model also changes with the distance from the camera in time. In this example, the camera was frozen at the time of the frame in the center and the resolution decrease as we go back or forth in time without adapting the cut of the DAG.

Chapter 7

Conclusions and Future Work

In this work we demostrated the effectiveness of *Batched Multi-Triangolation* as a patch-based multi-resolution framework, due to the increasing performance gains obtained by upgrading the graphic primitive from a single triangle to a small patch of triangles.

In this framework it is possible to exploit all the advantages of patch-based structures: the CPU load can be minimized due to small DAGs and no per-triangle operations, it is possible to make optimal use of the memory caches at the various stages of the pipeline, the available bandwith is optimized by using block transfer and in general rendering performances are vastly improved through extensive preprocessing. At the same time the concept of *V-partition* applied to the Multi-Triangulation structure allows for seamless reconstruction at different levels of detail without the need for special management of the boundary of the patches and simple out-of-core data structures coupled with parallel processing.

The most important contribution is the ability to decouple simplification algorithm, mesh storage format and visualization technique from the multiresolution structure. This is demonstrated by the wide range of applications to specific fields, such as terrains, generic meshes and animations: in each different contest it was possible to combine regular and irregular multiresolution data partitioning, regular grids or irregular meshes and different simplification and preprocessing algorithm such as triangle strips, compression, dynamic triangle optimization.

7.1 Future Work

There are currently few techniques for multiresolution editing and probably none for seamless multiresolution texture mapping on generic 3D models. How BMT could be extended to support these important tools will be discussed in the following sections.

A number of other minor improvement still deserve investigation:

- Given the relative improvement of processing power over data transfer bandwith efficient compression algorithms would reduce latency and data storage. This is particularly crucial in the case of dynamic meshes.
- A current problem with BMT is “popping”: abrupt changes in appearance while the resolution of the model is updated, due to the patch-based approach. In most situations this

effect is barely noticeable due to the very small size of the average triangles. Expecially under severly constrained bandwidth a smoother resolution change is required. Geomorphing techniques could be implemented to remove this problem. As usual the challenge is to ensure consistency along the boundaries.

- Lastly it seems worthwhile to investigate better and faster V-partition generation. Voronoi coupled with Lloyd's relaxation algorithm generates uniform clustering and very compact clusters, at the expence of a few pass over the data. A current problem is that a good intersection between different levels of the V-partition is not guaranteed, only very probable. It would be desiderable to apply the relaxation algorithm in parallel on all the levels of the V-partition taking into account also the intersection between consecutive levels.

7.1.1 Textures

Each patch of a BMT can be regarded as a small mesh allowing to use all the available tools for texture processing, a major problem is texture discontinuities along the boundary of the patches. This problem is common to all atlas-based techniques but in our case is particularly difficult due to the fact that patches from different levels share a common border.

A first approach is to extend the textures between adjacent patches and blend between them. This method unfortunately breaks the paradigm that each patch should be independent since it requires to access data from nearby patches depending on the current extraction. This reintroduces quite a lot of complexity in the rendering and construction algorithms.

A more promising approach is to decouple the texture multiresolution structure from the geometry. Each vertex would then know which atlas map it belongs to and blending would be performed on those triangles spanning different maps.

7.1.2 Editing

BMT is a multiresolution technique especially tailored for visualization purpouses, however, it does not seem impossible to support local mesh operations such as smoothing, hole filling and generic editing.

A simble idea is to assemble a small number of high level resolution patches, edit the mesh, resplit into the original patches and propagate the changes up in the hierarchy. For this purpouse a fast simplification algorithm (such as vertex clustering) would be preferable. A major problem arises when local editing substantially changes the number of primitive in a region (e.g. hole filling or mesh cleaning), as that would require changes into the macrostructure of the V-partition.

Bibliography

[1] M. Alexa. Recent advances in mesh morphing. *Computer Graphics Forum*, 21(2):173–196, 2002.

[2] P. Alliez and C. Gotsman. Recent advances in compression of 3D meshes. In *Proceedings of the Symposium on Multiresolution in Geometric Modeling*, pages 3–26. Springer-Verlag, 2003.

[3] C. Andujar, D. Ayala, P. Brunet, R. Joan-Arinyo, and J. Sole. Automatic generation of multiresolution boundary representations. *Computer Graphics Forum (Proceedings of Eurographics 1996)*, 15(3):87–96, 1996.

[4] N. Anuar and I. Guskov. Extracting animated meshes with adaptive motion estimation. In B. Girod, M. A. Magnor, and H.-P. Seidel, editors, *Proceedings of Vision, Modeling, and Visualization 2004*, pages 63–71, Stanford, CA, Nov. 2004. IOS Press.

[5] A. J. Baddeley. An error metric for binary images. In W. Förstner and S. Ruwiedel, editors, *Robust computer vision: quality of vision algorithms*, pages 59–78, Bonn, 1992.

[6] B. G. Baumgart. Winged-edge polyhedron representation for computer vision. In *National Computer Conference*, May 1975.

[7] O. Belmonte, I. Remolar, J. Ribelles, M. Chover, and M. Fernández. Efficiently using connectivity information between triangles in a mesh for real-time rendering. *Future Gener. Comput. Syst.*, 20(8):1263–1273, 2004.

[8] M. Bertolotto, L. De Floriani, and P. Marzano. Pyramidal simplicial complexes. In *Solid Modeling '95, 3rd Symp. on Solid Modeling and Appl.*, pages 153–162, Salt Lake City, Utah, May 1995.

[9] P. Bhaniramka, R. Wenger, and R. Crawfis. Isosurface construction in any dimension using convex hulls. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):130–141, Mar./Apr. 2004.

[10] R. Borgo, P. Cignoni, and R. Scopigno. An easy to use visualization system for huge cultural heritage meshes. In D. Arnold, A. Chalmers, and D. Fellner, editors, *Proceedings of the VAST 2001 Conference*, pages 121–130, Athens, Greece, Nov. 28-30, 2001. ACM Siggraph.

[11] M. Botsch and L. Kobbelt. High-quality point-based rendering on modern GPUs. In *PG 2003: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, pages 335–343, Washington, DC, USA, 2003. IEEE Computer Society.

[12] L. Buatois, G. Caumon, and B. Lévy. GPU accelerated isosurface extraction on tetrahedral grids. In *Advances in Visual Computing (ISVC 2006)*, volume 4291 of *Lecture Notes in Computer Science*, pages 383–392. Springer, 2006.

[13] D. Calver. Vertex decompression using vertex shaders. In *ShaderX*. Wolfgang Engel, 2002.

[14] A. Ciampalini, P. Cignoni, C. Montani, and R. Scopigno. Multiresolution decimation based on global error. *The Visual Computer*, 13(5):228–246, 1997.

[15] P. Cignoni, D. Costanza, C. Montani, C. Rocchini, and R. Scopigno. Simplification of tetrahedral volume data with accurate error evaluation. In *Proceedings IEEE Visualization 2000*, pages 85–92. IEEE Computer Society, 2000.

[16] P. Cignoni, L. De Floriani, P. Lindstrom, V. Pascucci, J. Rossignac, and C. Silva. Multiresolution modeling, visualization and streaming of volume meshes. In *Eurographics 2004 - Tutorial*, 2004.

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

[18] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Planet–sized batched dynamic adaptive meshes (P-BDAM). In *Proceedings IEEE Visualization 2003*, pages 147–155, Conference held in Seattle, WA, USA, October 2003. IEEE Computer Society Press.

[19] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Adaptive tetrapuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Transactions on Graphics*, 23(3):796–803, Aug. 2004. Proceedings of SIGGRAPH 2004.

[20] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Batched multi triangulation. In *Proceedings IEEE Visualization*, pages 207–214, Conference held in Minneapolis, MI, USA, October 2005. IEEE Computer Society Press.

[21] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):525–537, 2003.

[22] P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification algorithms. *Computers And Graphics*, 22(1):37–54, 1998.

[23] P. Cignoni, C. Rocchini, and R. Scopigno. METRO: measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, June 1998.

[24] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, and W. Wright. Simplification envelopes. In *Siggraph 1996, Computer Graphics Proceedings*, pages 119–128, Aug. 6-8 1996.

[25] M. A. Cosman and R. A. Schumacker. System strategies to optimize CIG image content. In *Proceedings of the Image II Conference*, pages 463–480. Image Society, Tempe, AZ, Jun 1981.

[26] C. Dachsbacher, C. Vogelgsang, and M. Stamminger. Sequential point trees. *ACM Transactions on Graphics*, 22(3):657–662, 2003.

[27] E. Danovaro, L. D. Floriani, P.Magillo, and E. Puppo. Data structures for 3d multi-tessellations: an overview, in: Data visualization: The state of the art. *Proceedings Dagstuhl Seminar on Scientific Visualization*, 2003.

[28] M. de Berg and K. Dobrindt. On levels of detail in terrains. In *Proc. 11th Annual ACM Symp. on Computational Geometry*, Vancouver, B.C., June 1995. Also available as Utrecht University tech report UU-CS-1995-12, ftp://ftp.cs.ruu.nl/pub/RUU/CS/techreps/CS-1995/.

[29] Q. Du, V. Faber, and M. Gunzburger. Centroidal Voronoi tessellations: applications and algorithms. *SIAM Review*, 41(4):637–676, 1999.

[30] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Aldrich, and M. Mineed-Weinstein. Roaming terrain: Real-time optimally adapting meshes. In *roceedings IEEE Visualization 1997*, pages 81–88, 1997.

[31] W. Evans, D. Kirkpatrick, and G. Townsend. Right triangular irregular networks. Technical Report 97-09, University of Arizona, 1997.

[32] W. S. Evans, D. G. Kirkpatrick, and G. Townsend. Right-triangulated irregular networks. *Algorithmica*, 30(2):264–286, 2001.

[33] W. S. Evans, D. G. Kirkpatrick, and G. Townsend. Right-triangulated irregular networks. *Algorithmica*, 30(2):264–286, 2001.

[34] R. L. Ferguson, R. Economy, W. A. Kelley, and P. P. Ramos. Continuous terrain level of detail for visual simulation. In *Proceedings of the 1990 Image V Conference*, pages 145–151. Image Society, Tempe, AZ, Jun 1990.

[35] K. Fischer and B. Gärtner. The smallest enclosing ball of balls: combinatorial structure and algorithms. *International Journal of Computational Geometry and Applications*, 14(4-5):341–378, 2004.

[36] L. D. Floriani, P. M., and E. Puppo. Efficient implementation of multi-triangulations. In *Proceedings of IEEE Visualization 1998*, pages 43–50, Research Triangle Park, NC, Oct. 1998. IEEE Computer Society Press.

[37] L. D. Floriani and P. Magillo. Multiresolution mesh representation: Models and data structures. In M.Floater, A.Iske, and E.Qwak, editors, *Tutorials on Multiresolution in Geometric Modelling*, pages 363–418. Springer-Verlag, 2002.

[38] L. D. Floriani and E. Puppo. Hierarchical triangulation for multiresolution surface description. *ACM Transactions on Graphics*, 14(4):363–411, October 1995.

[39] L. D. Floriani, E. Puppo, and P. Magillo. A formal approach to multiresolution modeling. In W. Straßer, R. Klein, and R. Rau, editors, *Theory and Practice of Geometric Modeling*, pages 302–323. SpringerVerlag, 1996.

[40] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Siggraph 1993, Computer Graphics Proceedings*, pages 247–254, 1993.

[41] T. A. Funkhouser, C. H. Séquin, and S. J. Teller. Management of large amounts of data in interactive building walkthroughs. In *1992 Symposium on Interactive 3D Graphics*, pages 11–20, 1992. Special issue of Computer Graphics.

[42] P. Gandoin and O. Devillers. Progressive lossless compression of arbitrary simplicial complexes. In *ACM Transactions on Graphics*, pages 372–379, 2002.

[43] M. Garland. *Quadric-Based Polygonal Surface Simplification*. PhD thesis, Carnegie Mellon University, Computer Science Department, 1999.

[44] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *Siggraph 1997, Computer Graphics Proceedings*, pages 209–216, aug 1997.

[45] M. Garland and Y. Zhou. Quadric-based simplification in any dimension. *ACM Transactions on Graphics*, 24(2):209–239, Apr. 2005.

[46] E. Gobbetti, F. Marton, P. Cignoni, M. D. Benedetto, and F. Ganovelli. C-BDAM - compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum*, 25(3), Sep 2006.

[47] G. Greiner and R. Grosso. Hierarchical tetrahedral-octahedral subdivision for volume visualization. *The Visual Computer*, 16:357–365, 2000.

[48] A. Guéziec. Surface simplification with variable tolerance. In *Second Annual Intl. Symp. on Medical Robotics and Computer Assisted Surgery (MRCAS '95)*, pages 132–139, Nov 1995.

[49] A. Guéziec. Surface simplification inside a tolerance volume. Technical report, Yorktown Heights, NY 10598, Mar 1996. IBM Research Report RC 20440 http://www.watson.ibm.com:8080/search_paper.shtml.

[50] P. Hinker and C. Hansen. Geometric optimization. In *Proceedings of IEEE Visualization 1993*, pages 189–195, October 1993.

[51] H. Hoppe. Progressive meshes. In *Siggraph 1996, Computer Graphics Proceedings*, pages 99–108, aug 1996.

[52] H. Hoppe. View-dependent refinement of progressive meshes. *Computer Graphics*, 31(Annual Conference Series):189–198, 1997.

[53] H. Hoppe. Smooth view-dependent level-of-detail control and its aplications to terrain rendering. In *Proceedings of IEEE Visualization 1998*, pages 35–42, 1998.

[54] H. Hoppe. Optimization of mesh locality for transparent vertex caching. In *Siggraph 1999, Computer Graphics Proceedings*, pages 269–276, 1999.

[55] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Siggraph 1993, Computer Graphics Proceedings*, pages 19–26, Aug. 1993.

[56] B. Houston, M. Nielsen, C. Batty, O. Nilsson, and K. Museth. Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM Transactions on Graphics*, 25(1):151–175, Jan. 2006.

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

[58] A. D. Kalvin and R. H. Taylor. Superfaces: Polygonal mesh simplification with bounded error. *IEEE Computer Graphics and Applications*, 16(3):64–77, May 1996.

[59] P. Kipfer and R. Westermann. GPU construction and transparent rendering of iso-surfaces. In G. Greiner, J. Hornegger, H. Niemann, and M. Stamminger, editors, *Proceedings of Vision, Modeling, and Visualization 2005*, pages 241–248, Erlangen, Germany, Nov. 2005. IOS Press.

[60] S. Kircher and M. Garland. Progressive multiresolution meshes for deforming surfaces. In *Proceedings of Symposium on Computer Animation 2005*, pages 191–200, Los Angeles, CA, July 2005. ACM Press.

[61] R. Klein, G. Liebich, and W. Straßer. Mesh reduction with error control. In *Proceedings of Visualization '96*, pages 311–318, Oct. 1996.

[62] R. Klein and W. Straßer. Generation of multiresolution models from cad-data for real time rendering. In W. Straßer, R. Klein, and R. Rau, editors, *Theory and Practice of Geometric Modeling*. Springer Verlag, Berlin, Heidelberg, New York, 1996.

[63] T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. In *Proceedings of Pacific Graphics 2004*, pages 186–195, Seoul, South-Korea, Oct. 2004. IEEE Computer Society Press.

[64] L. Kobbelt, S. Campagna, and H.-P. Seidel. A general framework for mesh decimation. In *Graphics Interface*, pages 43–50, 1998.

[65] J. Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *Proceedings IEEE Visualization 2002*, pages 259–266. IEEE, Oct 2002.

[66] P. Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *SI3D*, pages 93–102, 2003.

[67] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner. Realtime, continuous level of detail rendering of height fields. In *Siggraph 1996, Computer Graphics Proceedings*, pages 109–118, 1996.

[68] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, A. Op, and B. N. Faust. An integrated global GIS and visual simulation system. Technical Report GIT-GVU-97-07, Georgia Institute of Technology, 1997.

[69] P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *Proceedings of IEEE Visualization 2001*, pages 363–370. IEEE Press, Oct. 2001.

[70] 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.

[71] P. Lindstrom and G. Turk. Fast and memory efficient polygonal simplification. In *Proceedings IEEE Visualization 1998*, pages 279–286. IEEE, 1998.

[72] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[73] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[74] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Siggraph 1987, Computer Graphics Proceedings*, 21(4):163–169, July 1987. Proceedings of SIGGRAPH '87.

[75] F. Losasso and H. Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Transactions on Graphics*, 23(3):769–776, Aug. 2004.

[76] K.-L. Low and T.-S. Tan. Model simplification using vertex-clustering. In *1997 Symposium on Interactive 3D Graphics*, pages 75–82. ACM SIGGRAPH, 1997.

[77] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Siggraph 1997, Computer Graphics Proceedings*, pages 199–208, Aug. 1997.

[78] P. Magillo. *Spatial Operations on Multiresolution Cell Complexes*. PhD thesis, Università degli Studi di Genova, 1999.

[79] A. Maheshwari, P. Morin, and J. Sack. Progressive TINs: Algorithms and applications. In *ACM-GIS*, pages 24–29, 1997.

[80] M. Mäntylä. *An introduction to solid modeling*. Computer Science Press, Inc., New York, NY, USA, 1987.

[81] N. Max, P. Williams, and C. T. Silva. Approximate volume rendering for curvilinear and unstructured grids by hardware-assisted polyhedron projection. *International Journal of Imaging Systems and Technology*, 11:53–61, 2000.

[82] M. Mitzenmacher, A. Richa, and R. Sitaraman. The power of two random choices: A survey of the techniques and results. In *Handbook of Randomized Computing*. Kluwer, 2000.

[83] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7:217–236, 1978.

[84] M. Nielsen and K. Museth. Dynamic tubular grid: An efficient data structure and algorithms for high resolution level sets. *Journal of Scientific Computing*, 26(3):261–299, Mar. 2006.

[85] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*, volume 153 of *Applied Mathematical Sciences*. Springer, 2003.

[86] S. Osher and J. A. Sethian. Fronts propagating with curvature dependent speed: Algorithms based on Hamilton-Jacobi formulation. *Journal of Computational Physics*, 79(1):12–49, Nov. 1988.

[87] R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *Proceedings IEEE Visualization 1998*, pages 19–24, Research Triangle Park, NC, 1998. IEEE Computer Society Press.

[88] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. In O. Deussen, C. Hansen, D. A. Keim, and D. Saupe, editors, *Proceedings of VisSym 2004*, pages 293–300, Konstanz, Germany, May 2004. Eurographics Association.

[89] A. A. Pomeranz. ROAM using surface triangle clusters (RUSTiC). Master's thesis, University of California at Davis, 2000.

[90] F. Ponchio and K. Hormann. Interactive rendering of dynamic geometry. *IEEE Transaction on Visualization and Computer Graphics*, 14(4):914–925, Jul 2008.

[91] J. Popović and H. Hoppe. Progressive simplicial complexes. In *Siggraph 1997, Computer Graphics Proceedings*, pages 217–224, 1997.

[92] S. D. Porumbescu, B. Budge, L. Feng, and K. I. Joy. Shell maps. *ACM Transactions on Graphics*, 24(3):626–633, July 2005. Proceedings of SIGGRAPH 2005.

[93] C. Prince. *Progressive Meshes for Large Models of Arbitrary Topology.* PhD thesis, University of Washington, 2000.

[94] E. Puppo. Variable resolution terrain surfaces. In *Proceedings of the 8th Canadian Conference on Computational Geometry*, pages 202–210, 1996.

[95] E. Puppo. Variable resolution triangulations. *Computational Geometry*, 11(3-4):219–238, 1998.

[96] M. Reddy, Y. Leclerc, L. Iverson, N. Bletter, and K. Vidimce. Modeling the digital earth in vrml. In *In Proceedings of SPIE - The International Society for Optical Engineering. Volume 3905*, pages 390–399, 1999.

[97] J. C. Roberts and S. Hill. Piecewise linear hypersurfaces using the marching cubes algorithm. In *Visual Data Exploration and Analysis VI*, volume 3643 of *Proceedings of SPIE*, pages 170–181, San Jose, CA, Jan. 1999.

[98] R. Ronfard and J. Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, 15(3), aug 1996. Proc. Eurographics '96.

[99] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering complex scenes. In B. Falcidieno and T. Kunii, editors, *Modeling in Computer Graphics: Methods and Applications*, pages 455–465, Berlin, 1993. Springer-Verlag. Proceeding of Conference, Genoa, Italy, June 1993. (Also available as IBM Research Report RC 17697, Feb. 1992, Yorktown Heights, NY 10598).

[100] H. Samet. *Applications of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[101] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[102] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[103] P. V. Sander and J. L. Mitchell. Progressive buffers: view-dependent geometry and texture LOD rendering. In *ACM SIGGRAPH 2006 Courses*, pages 1–18, New York, NY, USA, 2006. ACM.

[104] L. L. Scarlatos and T. Pavlidis. Hierarchical triangulation using cartographic coherence. *CVGIP: Graphical Model and Image Processing*, 54(2):147–161, 1992.

[105] J. Schneider and R. Westermann. GPU-friendly high-quality terrain rendering. *Journal of WSCG*, 14(1-3):49–56, 2006.

[106] W. J. Schroeder. A topology modifying progressive decimation algorithm. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 205–212., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[107] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. *Siggraph 1992, Computer Graphics Proceedings*, 26(2):65–70, July 1992.

[108] A. Shamir, V. Pascucci, and C. Bajaj. Multi-resolution dynamic meshes with arbitrary deformations. In *Proceedings of IEEE Visualization 2000*, pages 423–430, Salt Lake City, UT, Oct. 2000. IEEE Computer Society Press.

[109] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *IEEE Visualization - Tutorial*, 2002.

[110] R. Sivan. *Surface modeling using quadtrees*. PhD thesis, College Park, MD, USA, 1995.

[111] R. Sivan and H. Samet. Algorithms for constructing quadtree surface maps. *Proc. 5th Int. Symposium on Spatial Data Handling*, pages 361–370, aug 1992.

[112] R. Sondershaus and W. Straßer. Segment-based tetrahedral meshing and rendering. In *Proceedings of GRAPHITE 2006*, pages 469–477, Kuala Lumpur, Malaysia, Nov. 2006. ACM Press.

[113] M. Soucy, A. Croteau, and D. Laurendeau. A multi-resolution surface model for compact representation of range images. In *Intl. Conf. on Robotics and Automation*, pages 1701–1706, May 1992.

[114] B. Speckmann and J. Snoeyink. Easy triangle strips for TIN terrain models. *International Journal of Geographical Information Science*, 15(4):379–386, 2001.

[115] R. Toledo, M. Gattass, and L. Velho. QLOD: a data structure for interative terrain visualization. Technical Report TR01-13, VISGRAF Laboratory, 2001.

[116] I. J. Trotts, B. Hamann, K. I. Joy, and D. F. Wiley. Simplification of tetrahedral meshes. In *Proceedings of IEEE Visualization 1998*, pages 287–295, Research Triangle Park, NC, Oct. 1998. IEEE Computer Society Press.

[117] G. Turk. Re-tiling polygonal surfaces. *Siggraph 1992, Computer Graphics Proceedings*, 26(2):55–64, July 1992.

[118] O. M. van Kaick, M. V. G. da Silva, and H. Pedrini. Efficient generation of triangle strips from triangulated meshes. *Journal of WSCG*, 12:475–482, 2004.

[119] H. Vo, S. Callahan, P. Lindstrom, V. Pascucci, and C. Silva. Streaming simplification of tetrahedral meshes. Technical Report UCRL-CONF-208710, Lawrence Livermore National Laboratory, Apr. 2005.

[120] C. Weigle and D. C. Banks. Complex-valued contour meshing. In *Proceedings of IEEE Visualization 1996*, pages 173–180, San Francisco, CA, Oct. 1996. IEEE Computer Society Press.

[121] C. Weigle and D. C. Banks. Extracting iso-valued features in 4-dimensional scalar fields. In *Proceedings of Symposium on Volume Visualization 1998*, pages 103–110, Research Triangle Park, NC, Oct. 1998. IEEE Computer Society Press.

[122] J. Wu and L. Kobbelt. A stream algorithm for the decimation of massive meshes. In *Proceedings of Graphics Interface 2003*, pages 185–192, Halifax, Canada, June 2003. AK Peters.

[123] J. C. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, 1997.

[124] J. C. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In R. Yagel and G. M. Nielson, editors, *Proceedings of IEEE Visualization 1996*, pages 335–344, 1996.

[125] C.-K. Yang, T. Mitra, and T.-C. Chiueh. On-the-fly rendering of losslessly compressed irregular volume data. In *IEEE Visualization*, pages 101–108, 2000.

[126] S. Yoon and P. Lindstrom. Random-accessible compressed triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1536–1543, Nov.-Dec. 2007.

[127] S. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious mesh layouts. *ACM Transactions on Graphics*, 24(3):886–893, 2005.

[128] S. Yoon, B. Salomon, R. Gayle, and D. Manocha. Quick-VDR: interactive view-dependent rendering of massive models. *IEEE Visualization*, pages 131–138, 2004.

[129] D. Zorin, P. Schröder, and W. Sweldens. Interactive multiresolution mesh editing. *Computer Graphics*, 31(Annual Conference Series):259–268, 1997.