Please cite as: F. Ponchio and K. Hormann. Interactive rendering of dynamic geometry. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):914–925, July/August 2008.

# Interactive Rendering of Dynamic Geometry

Federico Ponchio Kai Hormann

Department of Informatics
Clausthal University of Technology

**Abstract**

Fluid simulations typically produce complex three-dimensional iso-surfaces whose geometry and topology change over time. The standard way of representing such "dynamic geometry" is by a set of iso-surfaces that are extracted individually at certain time steps. An alternative strategy is to represent the whole sequence as a four-dimensional tetrahedral mesh. The iso-surface at a specific time step can then be computed by intersecting the tetrahedral mesh with a three-dimensional hyperplane. This not only allows to animate the surface continuously over time without having to worry about the topological changes, but also enables simplification algorithms to exploit temporal coherence. We show how to interactively render such four-dimensional tetrahedral meshes by improving previous GPU-accelerated techniques and building an out-of-core multi-resolution structure based on quadric error simplification. As a second application we apply our framework to time-varying surfaces that result from morphing one triangle mesh into another.

## 1 Introduction

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 character animation, physically-based animation of deformable or rigid objects, or mesh morphing, 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, however, are much easier dealt with 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 then is an iso-surface and in order to get an explicit representation of the sequence, these iso-surfaces are usually extracted one by one with any of the many available algorithms. The whole animation is then given as a set of *individual* triangle meshes with *varying* mesh connectivity which complicates further processing of the sequence, in particular the interpolation between successive frames.

Another approach is 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. Such 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 (see Section 3).

The advantages of this approach to handling dynamic geometry are twofold: firstly, it treats 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 5). 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.

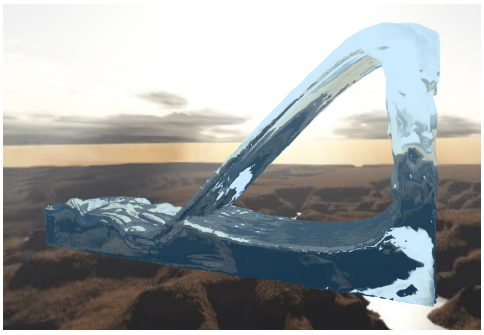

Figure 1: Snapshot from the interactive rendering of the "wave" data set.

### Contributions

In this paper we show how to render such dynamic geometry at interactive frame rates. In particular, we explain how to build a multi-resolution structure for 4D tetrahedral meshes and how to preprocess the data such that the hyperplane-intersection for a certain time value can be computed efficiently on the GPU. The proposed method

- can render large data sets of time-varying surfaces (with many million tetrahedra) in an interactive way,
- allows to continuously interpolate between successive time frames, regardless of any topological and geometric changes that may occur,
- is scalable in the sense that doubling the size of the input data also doubles the pre-processing time and the storage space on disk, but does not affect the frame rate and memory usage for the rendering itself,
- requires only little CPU resources during the rendering session, so that the latter is still available for other computations at the same time.

### Overview

Figure 2 illustrates our framework. In a first step, we build a hypermesh that represents the given sequence of time-varying surfaces (Section 4). We then simplify this hypermesh by successive edge collapses using the quadric error metric (Section 5) and build a patch-based multi-resolution hierarchy (Section 6). The hypermeshes from this hierarchy are then transformed into *dynamic triangles*, a special "GPU-friendly" data structure that allows to render the whole animation efficiently (Section 7).

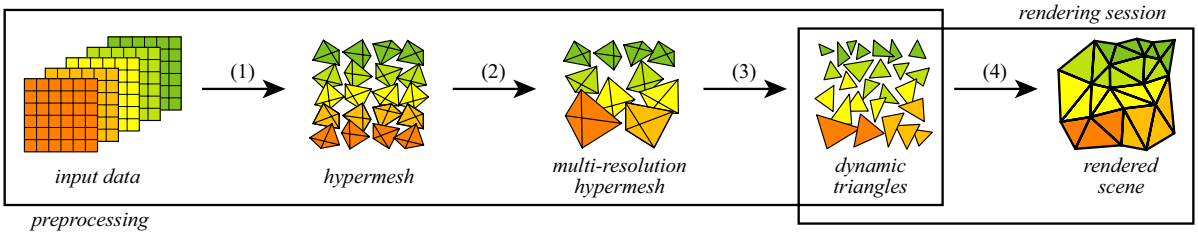


Figure 2: Overview of the proposed framework. The preprocessing phase converts the input data into hypermeshes (1), builds a multiresolution model by utilizing quadric error simplification (2), and converts the hypermesh into a set of dynamic triangles (3). This data structure is optimized for interactive rendering of the scene using the GPU (4).

## 2 Related Work

**Generating Hypermeshes:** Extracting iso-surfaces from volumetric data has a long history, starting with the famous marching cubes (MC) algorithm [22]. Besides the various improvements of this idea in the 3D setting, it has also been extended to extract hypersurfaces from four-dimensional volumes with both tetrahedral [39, 40] and hexahedral elements [31, 4]. 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*.

We implemented a 4D MC algorithm using the table given by Bhaniramka et al. [4] to extract hypermeshes for dynamic geometry that is given as a sequence of volumetric grids (Section 4.1). Such data is produced, e.g. when liquid simulations are computed with the level set method [27, 26] and the latest techniques [25, 16] can compute such detailed simulations that the raw data size easily reaches several GB. Moreover, we developed a simple algorithm to also build hypermeshes from compatibly meshed sequences of triangle meshes (Section 4.2), which can result either from mesh morphing [1] or be generated from general mesh sequences, e.g. by remeshing [2].

**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 that is based on the quadric error metric [13] with multiple-choice randomized collapses (Section 5). Although it has already been discussed by Garland and Zhou [14] how to use quadric error in any dimension, the application to hypermeshes seems new to the best of our knowledge. Note that this is different from working with volumetric tetrahedral meshes [36, 37, 6] in the same way that simplifying triangle meshes in 3D is different from the simplification of planar triangulations.

**Multi-resolution:** Real-time visualization of large hypermeshes requires to use a multi-resolution structure. Solutions for the special case of deforming meshes with constant connectivity can be found in Kircher and Garland [18] and in Shamir et al. [32]. The second technique is based on an adaptation of the Multi-Triangulation technique [30, 12] 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.

Multi-Tessellations [23, 11] are a dimension-independent generalization of Multi-Triangulations and can be used for multi-resolution visualization of hypermeshes. Danovoro et al. [11] describe two ulti-resolution data structures for tetrahedral meshes, one based on edge collapses, the other on vertex decimation, and analyze the *storage cost* and the *representation accuracy*, i.e. the number of tetrahedra needed for a fixed approximation error. However, when considering *rendering efficiency*, i.e. the framerate for a fixed approximation error, Multi-Triangulations and Multi-Tessellations are constrained by the high CPU load that is required to traverse huge DAGs (cf. Section 6.1) and by the fact that it is not possible to fully exploit the GPU processing power. For a detailed discussion on various aspects of simplification, multi-resolution, compression, and visualization of tetrahedral meshes, that are also relevant in the case of hypermeshes, we refer to a recent tutorial by Cignoni et al. [7].

We decided to rather adapt the ideas of Cignoni et al. [8, 10] and to construct a patch-based multi-resolution structure for hypermeshes (Section 6). 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 and Straßer [34].

For hypermeshes, special attention has to be paid to the scaling of the temporal dimension in order to get uniformly sized patches even if the simplified hypermesh contains tetrahedra that are long and thin in time. We therefore adapt the distance metric locally to the shape of the tetrahedra.

**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 (Section 7). Very similar techniques have been proposed for

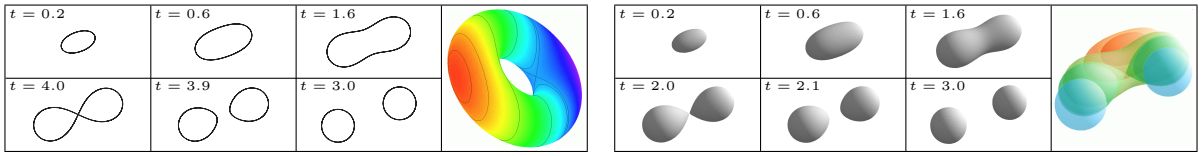


Figure 3: In the same way that an animated curve in 2D can be represented as a surface in 3D (left), an animated surface in 3D can be represented as a hypersurface in 4D (right).

GPU-accelerated iso-surface extraction from tetrahedral 3D meshes [28, 19, 17, 5]. 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 and Westermann [17], 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 7.1) 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.

**Volume Visualization:** A completely different approach to rendering iso-surfaces from four-dimensional volumes is by direct volume visualization. In order to handle large data sets, the latest of these techniques preprocess the data by a clever use of TSP tree data structures [33] and wavelet transforms [15, 38], but for the data size that we consider, they cannot achieve interactive frame rates on a single PC, at least not yet [20, 35].

## 3 Hypermeshes in 4D

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.

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,

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

More precisely, $\boldsymbol{C}$ 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 $t$,

$$\boldsymbol{C} \cap P(t) = C(t).$$

Figure 3 (left) illustrates this concept.

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

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

Again, intersecting $\boldsymbol{S}$ with a $t$-orthogonal hyperplane gives back the surface $S(t)$; see Figure 3 (right) 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 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 4). Without loss of generality we assume the vertices to be ordered according to their $t$-values, i.e., $t_1 \leq t_2 \leq t_3 \leq t_4$.

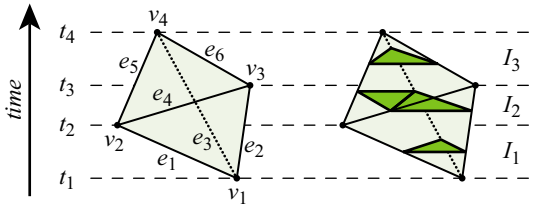


Figure 4: Notation for a tetrahedron (left) and possible cases that occur when it is intersected with a plane (right).

When a tetrahedron $T$ is intersected with a $t$-orthogonal hyperplane $P(t)$, we need to distinguish three cases (see Figure 4). 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 [28, 17].

We shall notice, however, that scalar-valued 3D tetrahedral meshes 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.

## 4 Generating Hypermeshes

Hypermeshes offer a convenient way for representing dynamic geometry and are useful in several applications. For testing and evaluating our multi-resolution rendering framework, we considered time-varying surfaces from fluid simulations as well as animation sequences that morph one triangle mesh into another. For both kind of input data, hypermeshes can be constructed as follows.

### 4.1 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 many cases, $f$ is unknown and only the function values at the vertices of a grid are given.

For example, when the level set method is used for liquid simulations, then $f$ is *the* level set function $\phi(\vec{x}, t)$ [27] 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 MC algorithm and its variants are common tools to compute a triangle mesh that approximates the iso-surface $I(c)$. Adapting the idea of MC, it is possible to extract the iso-surface from a 4D grid as a hypermesh in a similar way, although the number of local configurations that can occur in each cell is much higher; see [31] and [3] for details.

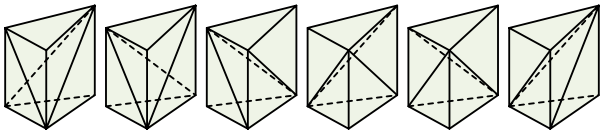

Figure 5: Six different ways to split a prism into tetrahedra.

### 4.2 Compatibly Meshed Sequences

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 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 and Garland [18].

However, 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 5 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. Or, seen the other way round, we need to choose one diagonal for all faces between neighbouring 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. [29] 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: if $v_1, \ldots, v_n$ are the vertices of the mesh, then by always taking the diagonals that have the vertex with the smaller index at the bottom, it is easy to see that the forbidden "cyclic" splits are avoided for all prisms [24].

## 5 Simplification

All of the methods above produce highly over-sampled hypermeshes and it is very desirable to reduce redundancy by simplifying them. We found that the extension [14] of the original QSlim algorithm [13] works very well in our situation. 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.

In the implementation that we used for this article, we followed the approach described by Vo et al. [37] which suggests to use a multiple choice randomized edge-collapse: i.e., the best collapse from a pool of candidates is chosen instead of having all candidates sorted in a priority queue. Wu and Kobbelt [41] showed that this gives a mesh quality that is comparable to that of the standard greedy approach.

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. It can be computed by taking the 4D cross product of three edge vectors of $T$.

Note that the quadric error associated to the edge collapses 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 Multi-resolution

## 6.1 Background on Multi-Tessellations

The concept of Multi-Tessellations (MT) is a very general framework for multi-resolution structures for meshes in arbitrary dimensions and we shall give a brief summary in the following. A detailed discussion of the structure and efficient implementations can be found in [12, 23, 11].

A MT is composed of a coarse mesh and a set of local updates which refine the base mesh by replacing a set of cells (triangles, tetrahedra, etc.) with a set of new cells. In particular, applying any iterative simplification algorithm to a reference mesh defines a MT, where the final result of the simplification is the base mesh and the set of updates consists of the inverted sequence of simplification operations. An update $u_2$ is said to depend on another update $u_1$ if and only if $u_2$ removes some cell that was introduced by $u_1$. The update $u_1$ is then called a parent of $u_2$. This dependency relations induces a partial order among the set of updates and the key observation is that the updates can be applied to the base mesh in any order that preserves this partial order.

A MT is usually represented by a Direct Acyclic Graph (DAG) [30] with one node for each update and an arc from $u_1$ to $u_2$ whenever $u_1$ is a parent of $u_2$. Associated with this arc is the set of cells that are both created by $u_1$ and removed by $u_2$. A subset $S$ of nodes in the DAG is called consistent if for each node in $S$ all its parents are also in $S$. The updates in a consistent subset $S$ can be applied to the base mesh in any total order that extends the partial order so as to create a mesh $M_S$ at an intermediate resolution.

The collection of arcs $C_S$ from nodes in $S$ to nodes not in $S$ is called a *cut* of the DAG, and $M_S$ is exactly the collection of cells associated with all the arcs in $C_S$. Moreover, we can associate with every node $u$ in the DAG an approximation error. The rendering algorithm then performs a traversal of the DAG and selects the cut that minimizes the error given a maximum number of resulting cells. Different metrics can be implemented to calculate the error and additional constraints can be imposed on the traversal.

## 6.2 Patch-based Multi-Tessellation

Usually each update operation in a MT corresponds to an atomic edge-collapse or vertex-removal operation. This ensures a maximum granularity and a smooth transition between different resolutions, but results in very large DAGs that require CPU-intensive computations in the rendering phase.

Due to the fact that GPU speed is increasing at a much faster rate than CPU speed, an established trend in multi-resolution algorithms for meshes [9] is to increase the number of triangles affected by an update operation, moving the refine-coarsen decisions from the level of a single triangle to blocks of triangles. This has the advantage of removing the CPU bottleneck and maximally utilizing the processing power of the GPU. This approach can further be supported by preprocessing the data such that it is handled most efficiently by the GPU. The loss in granularity is compensated for by a much higher triangle per second rendering rate.

Cignoni et al. [10] presented a multi-resolution framework that combines this concept with the ideas of multi-triangulations [12] and we extended it so that it can be used for hypermeshes. In the following we give only a brief description of the approach and in particular the requirements for adapting it to hypermeshes. For more details on this subject we refer to [10] and the references cited therein.

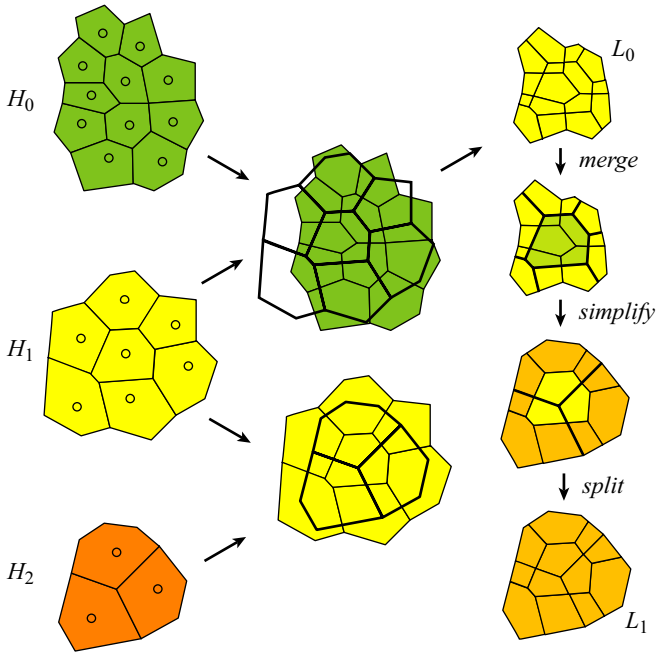


Figure 6: Hierarchy of the $V$-partition (left) and the three steps of the coarsening algorithm (right).

### 6.3 Building the Multi-resolution Model

We start by building the $V$-partition, i.e. a sequence of coarser and coarser partitions $H_0, H_1, \ldots, H_n$ of $\mathbb{R}^4$ using Voronoi clustering. In our implementation we randomly distribute seeds over the hypermesh and apply a few steps of Lloyd's Voronoi relaxation [21]. This technique works as nicely in 4D as it does in 3D.

In the next step we create the partition $L_0 = H_0 \cap H_1$ by intersecting the two finest partitions and split the hypermesh into patches, one for each cell of $L_0$ (see Figure 6). Then we collect for each cell of $H_1$ all patches in $L_0$ that it contains, merge them into one hypermesh, simplify it preserving the boundary, and split it 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 continue this coarsening algorithm for all levels of the $V$-partition. Note that this phase of the construction, which is the most CPU-intensive due to the simplification step, can easily be parallelized because the cells of $H_1$ can be processed independently of each other.

The history of the coarsening algorithm is then encoded as a DAG. We create a node in the graph for each cell of the space partitions $H_i$, $i > 0$, and also add a special node, the sink, that is associated with all the cells of $H_0$ (see Figure 7). Whenever two cells that belong to neighbouring levels $H_i$, $H_{i+1}$ intersect, we create an arc in the graph, directed towards the node in $H_i$. Each arc corresponds to one of the patches of the multi-resolution model that we created in the previous step. It is important to note that the collection of patches associated with the arcs in a cut join together seamlessly and form a complete multi-resolution representation of the hypermesh.

Each patch is stored as an independent hypermesh, using local indexing and replication of the patch boundary vertices. We take care that each patch contains less than 64 K vertices so that local vertex indices with 2 Bytes can be used instead of the 4 Bytes that are usually needed for global indices. 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.

On the other hand, this replication strategy requires a careful boundary management: the attributes of the vertices must have the same values on all replicated boundary vertices to avoid visible artifacts

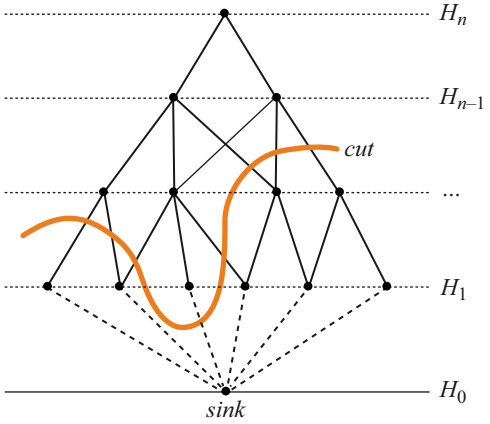


Figure 7: A cut in the DAG.

in the rendering. We keep track of these replicated boundary vertices in the following way. For each patch $p$ we maintain a list of triplets $(v_p, q, v_q)$, where $v_p$ is one of the boundary vertices in $p$ that is replicated as the vertex $v_q$ in one of the neighbouring patches $q$.

We exploit these lists to efficiently unify indices, to mark read-only vertices in the simplification step, and to ensure the consistency of vertex attributes. For example, to compute the 4D normals, we start at the finest level and propagate the ones at the boundary vertices to the next coarser level and then repeat the process up to the coarsest level. This data is not needed for rendering and can be discarded at the end of the preprocessing.

We further manage the set of patches in an out-of-core fashion by storing each patch, along with its boundary information, sequentially on disk and memory mapping it individually when needed. This approach is independent of the particular format in which the patches are stored and allows to use the same structure for rendering. In this case each patch is further preprocessed for GPU-assisted rendering as described in Section 7.1.

Finally, we store the DAG and an array of entries that contains for each patch in the data set the number of faces and vertices, the offset and size on the disk, the bounding sphere, and the estimated geometric error. There are several different structures for actually encoding the DAG [30, 12] with various tradeoffs between space and speed. But given the limited size of the DAG and due to the cluster structure, this is not a critical part of the rendering algorithm as this information is small enough to be kept in RAM even for very large models.

The overall storage cost of this structure is around $120n$ Bytes where $n$ is the number of vertices in the reference mesh, and the overhead of storing the DAG and replicating vertices is negligible. This is considerably smaller than the explicit data structure for tetrahedral MT described in [11] which requires between $450n$ and $880n$ Bytes and is due to the fact that our DAG has a reduced granularity. On the other hand, the edge-based MT that is also described in [11] requires only $36n$ Bytes. Recent compression methods [42] are able to store the topological information of a tetrahedral mesh with approximately 3 Bytes per tetrahedron (or 0.6 Bytes per vertex) for small meshes. If we were to compress the patches this way, the overall cost would be around $34n$ Bytes and even smaller than the edge-based MT, with the advantage that our method allows for an easy parallelization of the compression/decompression threads. However, we decided not to use this highly compressed format because it leads to substantially higher computational costs for extracting the mesh from a cut in the DAG and thus reduces the rendering performance.

We would finally like to note that the structure of our DAG is independent of the particular choice of simplification algorithm and mesh representation. As long as the boundaries of the patches are preserved, any simplification strategy (e.g. edge-collapses, vertex removal, remeshing, clustering) could be applied.

### 6.4 Differences between 3D and 4D

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 5).

For data from the 4D MC algorithm, the obvious choice is to set the time between two frames equal to the size of the marching cube. However, during the simplification process, 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.

### 6.5 Rendering the Multi-resolution Model

The goal of the rendering algorithm is to select a cut in the DAG to adapt the resolution in different parts of the model such that the error in screen-space is as uniform and low as possible and to maintain the visualization interactive, while constrained by the resources that are available: disk-speed, RAM, video-RAM, CPU budget, and GPU budget.

The screen-space error for a patch is computed as follows: if the bounding sphere is outside of the 4D viewing frustum, the error is set to zero, if the viewpoint is inside the bounding sphere, the error is infinite, otherwise we divide the geometric error by the distance from the viewpoint to the bounding sphere. As a result, regions that are further away from the viewer require a lower resolution than those that are close to the camera. Note that the word "distance" refers to the 4D distance and includes the temporal dimension. The error associated with a cut in the DAG is the maximum error of the collection of patches selected by the cut.

For each frame, the algorithm updates the cut in the DAG by performing refining or coarsening operations that correspond to moving the cut up or down in the tree. We associate with each refining or coarsening operation an error: the maximum error among the new selected patches. Each refinement operation consumes resources such as disk, RAM, video-RAM, and GPU budget, while coarsening operations release them.

The algorithm maintains a list of possible refinement and coarsening operations sorted by the error associated with them. Refinement operations, starting from the greatest error, are carried out as long as free resources exists and coarsening operations, starting from the lowest error are performed to free resources. The procedure terminates when the budget is used up and the smallest error in the coarsening list is bigger than the biggest error in the refining list, i.e. we would need to increase the overall error to free resources.

A detailed description of the algorithm can be found in [8, Section 4.3] where the structure of the DAG is identical to the one used here. Moreover, the implementation of the algorithm is described in detail in [10, Section 5], in particular how disk, RAM, and video-RAM can be treated like a multi-level cache that is accessed by a pre-fetching routine and executed in parallel with the rendering routine.

## 7 GPU-assisted Rendering

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 3). But

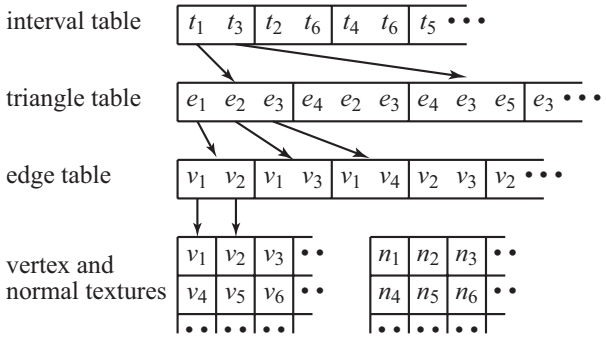


Figure 8: Data structures used for GPU rendering.

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 tradeoffs between storage cost and speed.

The technique of Pascucci [28] 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. [5] 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 and Westermann [17] 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.

We developed yet another strategy which is tailored to our specific needs: it is faster than [5], but requires more space. In Section 8 we compare the performance of both techniques.

### 7.1 Dynamic Triangles

As described in Section 3, 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 4, 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 the other three triangles, so that we store

$$\triangle_1 := \{I_1, (e_1, e_2, e_3)\}, \qquad \triangle_2 := \{I_2, (e_2, e_3, e_4)\},$$
$$\triangle_3 := \{I_2, (e_3, e_4, e_5)\}, \qquad \triangle_4 := \{I_3, (e_3, e_5, e_6)\}$$

for each tetrahedron of the hypermesh $H$.

Like [17] 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 $T \cap P(t) \in M(t)$.

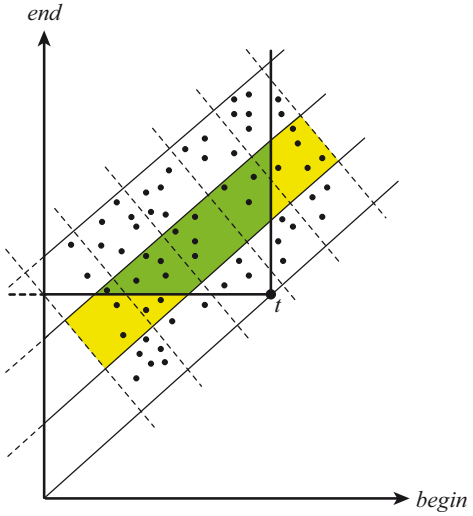


Figure 9: Span space for the lifespans of the dynamic triangles: Each point represents an interval with the abscissa and ordinate referring to the start and end time. The three diagonal bands contain intervals with approximately the same length, with the short-living intervals close to the diagonal. The intervals in the second band that intersect with the current time $t$ are shown in green.

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 8). 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 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 contain the current $t$: the parameter $\lambda$ for the linear interpolation is negative or bigger than 1 for one of its corners. We can then easily cull the triangle by moving the vertex position to the viewpoint so that the whole triangle becomes invisible.

Usually a large fraction of the triangles have a lifespan $I$ that does not contain 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 9). This results in more calls of "glDrawRangeElements" (one per band) but greatly improves the culling efficiency. The most appropriate number of bands depends on the relative cost of calling "glDrawRangeElements" versus that of rendering a tetrahedron.

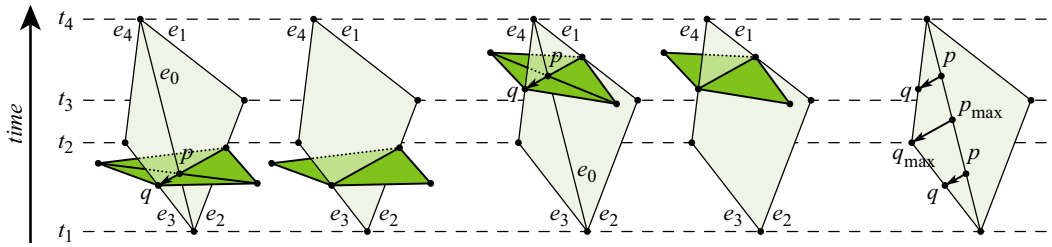


Figure 10: An edge collapse in the triangle mesh $M(t)$ (green) is equivalent to joining two faces of the hypermesh (light green).

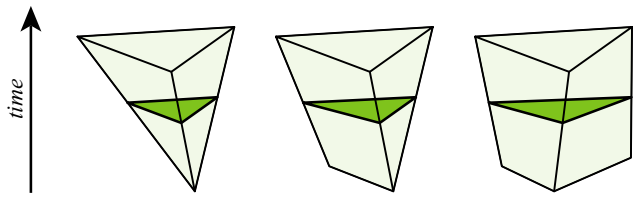


Figure 11: Joining faces of the hypermesh as in Figure 10 generates non-tetrahedral elements.

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 is computationally more expensive: $O(\log(n+m))$, where $n$ is the number of all and $m$ 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.

### 7.2 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 5) 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 10, 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 11).

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.

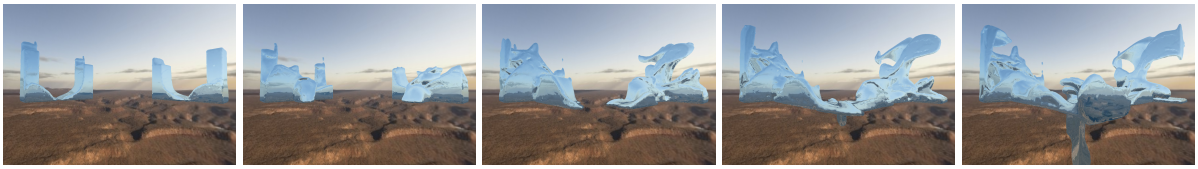

Figure 12: Snapshots from the "dams" sequence.

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 $t = t_2$ 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, as they rely on the fact that the slicing hyperplane is always at constant time.

## 8 Examples

We have applied the methods explained in the previous sections to five data sets: three liquid simulations (*dams*, *drops*, and *wave*) and two morphing sequences (*horse-to-man* and *cloth*); see Figures 12–16, respectively. The timings for the different steps of our processing pipeline and the sizes of the different data structures are summarized in Table 1. Because the full *wave* data set (200 frames) gives almost the same numbers as the *drops* sequence, we show the timings and sizes 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 MC algorithm as described in Section 4.1 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). Note that this step requires almost 10 days of runtime for the large *dams* model, but we did not investigate whether it is possible or not to implement the 4D MC algorithm more efficiently. On the other hand, the hypermeshes of the morphing sequences were constructed from sets of compatibly meshed surfaces as explained in Section 4.2.

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

| data set | | *dams* | *dams (partial)* | *drops* | *wave (partial)* | *horse-to-man* | *cloth* |
|---|---|---|---|---|---|---|---|
| resolution of input data | | 600 frames at $374\times374\times374$ | 200 frames at $374\times374\times374$ | 380 frames at $142\times60\times86$ | 50 frames at $50\times50\times90$ | 200 frames with 36 K triangles | 400 frames with 39.4 K triangles |
| hypermesh | construction | 14163 min. | 4721 min. | 177 min. | 26 min. | 1 min. | 3 min. |
| | tetrahedra | 1172 M | 375 M | 27 M | 6 M | 22 M | 48 M |
| | size on disk | 22.95 GB | 7.67 GB | 551 MB | 131 MB | 448 MB | 980 MB |
| multi-resolution model | construction | 5203 min. | 1649 min. | 157 min. | 32 min. | 112 min. | 216 min. |
| | tetrahedra | 1121 M | 378 M | 19 M | 5.6 M | 0.8 M | 3.1 M |
| | size on disk | 15.2 GB | 5.3 GB | 350 MB | 92 MB | 16 MB | 63 MB |
| dynamic triangles | triangles | 2268 M | 762 M | 55 M | 11.3 M | 1.8 M | 7.1 M |
| | size on disk | 23.5 GB | 8.1 GB | 631 MB | 137 MB | 26 MB | 101 MB |
| | render performance | 20 M triangles/sec. | | | | | |

Table 1: Size, timings, and memory requirements for the tested data sets. The full "wave" data set consists of 200 frames.

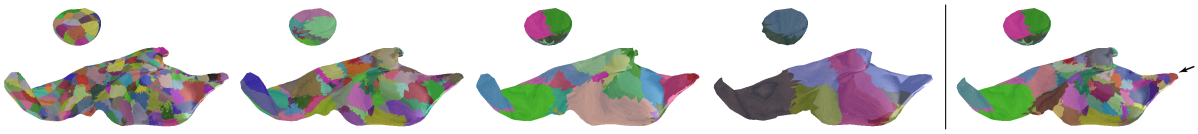

Figure 13: Snapshots from the "drop" sequence with increasing error tolerance (left) and the camera located at the black arrow (right), showing how the size of the patches selected from the multi-resolution hierarchy depends on the error and the viewpoint.

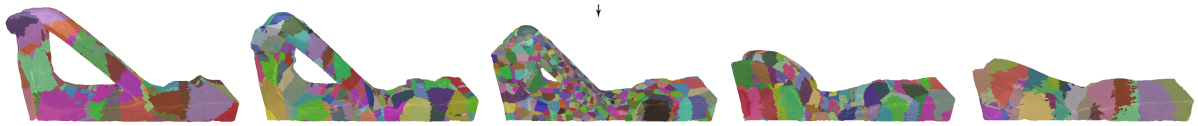

Figure 14: Snapshots from the "wave" sequence for increasing time values with the camera frozen at the time frame in the middle, showing that the size of the patches from the multi-resolution hierarchy depends on temporal distance to the viewpoint.

distributing the simplification steps over multiple computers [10]. After pruning the zero-error leaves from the full multi-resolution model, it becomes even smaller than the initial hypermesh. The multi-resolution models of the morphing sequences are much smaller because they contain many parts that move linearly over time and can thus be simplified much better. And the model of the *horse-to-man* sequence is yet smaller because the fixed connectivity that contains the details of both the horse and the man yields an oversampling in the spatial directions.

Finally, converting the data into dynamic triangles triples the size, but the optimization described in Section 7.2 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 amounts to a total of 5 million dynamic triangles or 190 MB 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, however, can become the bottleneck of the system, and while the algorithm is sufficient for normal playback and interactive visualization, it cannot quickly adapt the resolution of the model to abrupt changes in the viewing position or in time. The implementation of a compression algorithm would greatly improve this limitation, but we have not yet investigated in an algorithm for compressing dynamic triangles.

For a comparison, we also implemented the technique of Buatois et al. [5] 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 changes in the viewpoint.

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 favours small patches, and the rendering performance that increases with bigger patches.

The first four images in Figure 13 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.

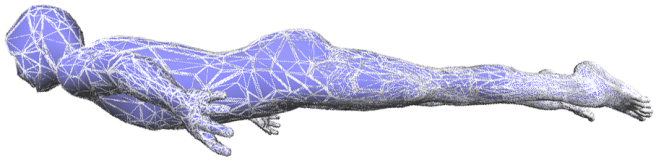

Figure 15: The model resolution is decreasing smoothly with increasing distance from the camera (placed near the feet).

Figure 16: Snapshots from the "cloth" sequence.

The rightmost image of Figure 13 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 15 shows a snapshot from the *horse-to-man* multi-resolution hypermesh where the resolution decreases with increasing distance to the camera (placed at the feet).

Figure 14 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 centre and the resolution decrease as we go back or forth in time without adapting the cut of the DAG.

## 9 Conclusions

In this paper we have shown how to render large data sets of dynamic geometry at interactive frame rates. Most of our processing pipeline builds on the representation of dynamic geometry as a hypermesh and we show how to convert two different kinds of input data to this structure. One of the advantages of hypermeshes is that they allow for a rather straightforward adaptation of an quadric error simplification algorithm, which in turn is the key ingredient to the construction of our multi-resolution model. The latter enables us to adapt the detail of the rendered scene not only to the distance from the viewpoint but also to the resource limits (GPU speed, RAM, disk speed). We further improve the performance of our rendering system by converting the multi-resolution model into a set of dynamic triangles, a data structure that is optimized for GPU rendering and preprocessed such that it maximimally exploits the GPU vertex cache. While the preprocessing of the data scales linearly with the size, the rendering frame rate is approximately constant.

### 9.1 Future Work

We believe that apart from simplification and building multi-resolution structures, many other standard geometry processing tools can be carried over to the 4D setting and be adapted to work with hypermeshes. In particular, we will try to further improve the compression rates for hypermesh representations of dynamic geometry.

For liquid simulations, it would be interesting to combine the simulation itself with our preprocessing pipeline so as to generate the dynamic triangle structure directly from the simulation, preferably with a streaming algorithm. Moreover, the dynamic triangle structure itself may be improved, depending on the new features that the next generation of graphics cards may offer.

We would finally like to apply our method to data sets from other kinds of scientific simulations. For example, stream surfaces in time-varying vector fields could be inspected interactively and texture mapping be used to further highlight important features like velocity or curl.

### Acknowledgements

This joint research project was financially supported by the State of Lower-Saxony and the VolkswagenFoundation, Hannover, Germany. We would like to thank Mark Carlson from Georgia Institute of Technology for kindly providing us with the simulation data of the *drops* sequence, Anders Söderström and Ken Museth from Linköping University for the *dams* and *wave* data sets and Scott Kircher from the University of Illinois for the *horse-to-man* and *cloth* morphing sequences.

## References

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

[2] 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.

[3] P. Bhaniramka, R. Wenger, and R. Crawfis. Isosurfacing in higher dimensions. In *Proceedings of IEEE Visualization 2000*, pages 267–273, Salt Lake City, UT, Oct. 2000. IEEE Computer Society Press.

[4] 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.

[5] 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.

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

[7] P. Cignoni, L. De Floriani, P. Lindstrom, V. Pascucci, J. Rossignac, and C. Silva. Multi-resolution modeling, visualization and streaming of volume meshes. In *Eurographics 2004 Tutorials*, volume 2, Grenoble, France, Aug. 2004. Eurographics Association.

[8] P. Cignoni, L. De Floriani, P. Magillo, E. Puppo, and R. Scopigno. Selective refinement queries for volume visualization of unstructured tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):29–45, Jan./Feb. 2004.

[9] 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.

[10] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Batched multi triangulation. In *Proceedings of IEEE Visualization 2005*, pages 207–214, Minneapolis, MN, Oct. 2005. IEEE Computer Society Press.

[11] E. Danovaro, L. De Floriani, P. Magillo, and E. Puppo. Data structures for 3D multi-tessellations: an overview. In *Data Visualization: The State of the Art*, Kluwer International Series in Engineering & Computer Science, pages 239–256. Kluwer, 2003.

[12] L. De Floriani, P. Magillo, 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.

[13] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH '97*, pages 209–216, Los Angeles, CA, Aug. 1997. ACM Press.

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

[15] S. Guthe, M. Wand, J. Gonser, and W. Straßer. Interactive rendering of large volume data sets. In *Proceedings of IEEE Visualization 2002*, pages 53–60, Boston, MA, Oct. 2002. IEEE Computer Society Press.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] S.-K. Liao, J. Z. C. La, and Y.-C. Chung. Time-critical rendering for time-varying volume data. *Computers & Graphics*, 28(2):279–288, Apr. 2004.

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

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

[23] P. Magillo. *Spatial Operations on Multiresolution Cell Complexes*. PhD thesis, Department of Computer and Information Sciences, University of Genova, 1999.

[24] 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(1):53–61, 2000.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] H.-W. Shen and L.-J. C. K.-L. Ma. A fast volume rendering algorithm for time-varying fields using atime-space partitioning (TSP) tree. In *Proceedings of IEEE Visualization 1999*, pages 371–545, San Francisco, CA, Oct. 1999. IEEE Computer Society Press.

[34] R. Sondershaus and W. Straßer. Segment-based tetrahedral meshing and rendering. In *Proceedings of GRAPHITE '06*, pages 469–477, Kuala Lumpur, Malaysia, Nov. 2006. ACM Press.

[35] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl. Large volume visualization of compressed time-dependent datasets on GPU clusters. *Parallel Computing*, 31(52):205–219, Feb. 2005.

[36] 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.

[37] 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.

[38] C. Wang, J. Gao, L. Li, and H.-W. Shen. A multiresolution volume rendering framework for large-scale time-varying data visualization. In *Proceedings of Volume Graphics 2005*, pages 11–19, Stony Brook, NY, June 2005. IEEE Computer Society Press.

[39] 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.

[40] 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.

[41] J. Wu and L. Kobbelt. A stream algorithm for the decimation of massive meshes. In *Proceedings of Graphics Interface '03*, pages 185–192, Halifax, Canada, June 2003. AK Peters.

[42] C.-K. Yang, T. Mitra, and T. Chiueh. On-the-fly rendering of losslessly compressed irregular volume data. In *Proceedings of IEEE Visualization 2000*, pages 101–108, Salt Lake City, UT, Oct. 2000. IEEE Computer Society Press.