Multi-Resolution Dynamic Meshes with Arbitrary Deformations*

Ariel Shamir† Valerio Pascucci‡ Chandrajit Bajaj†

†Center for Computational Visualization
TICAM, University of Texas at Austin

‡Center for Applied Scientific Computing
Laurence Livermore National Laboratory

Abstract

Multi-resolution techniques and models have been shown to be effective for the display and transmission of large static geometric object. Dynamic environments with internally deforming models and scientific simulations using dynamic meshes pose greater challenges in terms of time and space, and need the development of similar solutions. In this paper we introduce the T-DAG, an adaptive multi-resolution representation for dynamic meshes with arbitrary deformations including attribute, position, connectivity and topology changes. T-DAG stands for Time-dependent Directed Acyclic Graph which defines the structure supporting this representation. We also provide an incremental algorithm (in time) for constructing the T-DAG representation of a given input mesh. This enables the traversal and use of the multi-resolution dynamic model for partial playback while still constructing new time-steps.

1 INTRODUCTION

Dynamic scenes in computer graphics are represented by defining some of the scene parameters as functions of time. Global parameters like the position of the viewer (walk-through) or the position of objects can be encoded easily using rigid body transformations or interpolators and behaviors [18]. The representation of deforming objects is generally much more complex and more time/space consuming. Multi-resolution techniques have been shown to be an effective tool for handling complex geometric objects. However, most of the work done in this field concentrates on static objects. Moreover, the application of previously proposed solutions for dynamic models is restricted to objects in which the connectivity and topology are fixed over time. The T-DAG representation introduced here removes such limitations allowing for a unified representation of dynamic geometries where no restriction is imposed on the modification through time in terms of local connectivity or global topology.

The T-DAG data-structure is constructed incrementally over time. The information relative to the model evolution for each new time-step is integrated in the T-DAG as it becomes available. This allows traversing adaptively the multi-resolution model up to time-step t while still augmenting the model with the newly modified object for time-step $t+1$. The scheme extends the possibility of using adaptive multi-resolution techniques for display and transmission of time-dependent meshes with general deformations.

*This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. Research supported in part by NSF grants CCR-9732306, DMS-9873326, ACI-9982297 and Sandia/LLNL DOE ASCI-BD4485

1.1 Previous Work

Many approaches have been developed in the past for creating multi-resolution representations of geometric data for graphics and visualization [9, 14, 17, 19]. They vary in both the simplification scheme like vertex removal [2], edge contraction [10, 13], triangle contraction [8], vertex clustering [20], wavelet analysis [3, 23], and in the structure used to organize the levels of detail (either linear order [10, 14] or in a DAG [2, 6, 8, 17]). However, these techniques are based on the assumption that the finest resolution mesh is static. Our scheme evolves from such approaches removing this basic assumption of static input and allowing to define a multi-resolution representation for dynamic meshes.

A time dependent data structure for the extraction of isosurfaces from dynamic volumetric data is presented in [24]. A temporal Branch On Need Octree (BONO) is created to index the data spatially. Extreme isovalues in each node of the hierarchy are computed for each time step and stored separately. This structure is then used to support queries of the form *(timevalue,isovalue)* by traversing top down and visiting only in the parts that hold the correct isovalues for the current time-step. When a leaf node is reached, the block containing its data is stored in a list. Once the traversal is done, only blocks in the list are read and the isosurface computed. While this structure seems to be very efficient for isosurface extraction, it supports only this specific visualization primitive. It does not apply to meshes where connectivity can change, and it must be built off-line by global preprocessing. The volume rendering approach of time dependent data in [21] uses a spatial octree to partition the data, but each node holds a binary time-tree. Each node in the binary tree holds the average value of the containing octree node sub-volume, along with measurements of the spatial and temporal errors of these voxels in the corresponding time range. These measures serve as an indication of the spatial and temporal coherency of the sub-volume. The structure supports queries of the form *(timevalue,spatial-err,temporal-err)*. The octree is then traversed from the root expanding only the nodes that do not pass the tolerance test. In each node the binary time-tree is traversed likewise. The rendering of each octree-node voxel is done separately and the sub-images are composed using their colors and opacities to create the full image. This structure allows very efficient time-dependent volume rendering, but is tailored for this specific type of visualization and does not support changes in connectivity or topology of the mesh.

An opposite approach was presented earlier in [4] for multi-resolution video. A binary time tree is built by subdividing the time span. Each node corresponds to some averaging of all the images of its time span. The node holds a spatial quadtree built from

this average image. This structure supports multi-resolution in the temporal dimension by accessing the average images, and seems very appropriate for video sequences. However, it is not clear how to define the average of several time-dependent surfaces, especially when topology and connectivity could change over time, and consequently is difficult to apply such an approach to 3D meshes.

In our approach we exploit the difference in nature between time and space. We order time information sequentially while supporting multi-resolution representation in the spatial dimensions. However, since we separate the temporal information of each vertex, there is no restriction in storing this information consecutively. In fact, any type of multi-resolution can be defined for this data similar to the binary trees of [21]. Another option is to use compression in temporal space. In [15] a method is described for compression of time dependent geometry. The vertex positions matrix is decomposed into $P \cdot V \cdot G$, where P is the time interpolation, V is the vertex positions at key time-steps and G is the Geometry interpolation or spatial interpolation. Using those terms, [15] concentrates on compressing the V matrix, while we concentrate on encoding the G matrix using multi-resolution methods. Therefore the two methods are somewhat complementary, and might be combined.

1.2 Contribution

We define a multi-resolution data structure using time-tags for time dependent traversal. In particular we treat the symbolic information (mesh connectivity and decimation dependencies) in a similar manner as we treat the numeric information (attributes and positions of nodes). We show how this extended data structure enables the representation of a larger class of dynamic models including also connectivity and topology changes.

We present an incremental algorithm for building this data-structure. The incremental construction enables the use of the data-structure (of previous time-steps) even while the input is being processed. Moreover, this algorithm enables adjusting the resulting structure according to the tradeoff between optimized storage space and traversal time in each time-step for the creation of meshes.

2 PRELIMINARIES

2.1 Meshes

As customary in many application fields and for visualization, we assume that objects are represented by triangular surface meshes. We define a mesh $\mathcal{M}$ by a tuple $\mathcal{M} = (P, F, I)$. $P = \{p_i\}$ is a collection of points in E^3 called *vertices* (we distinguish between vertices and nodes using the latter in conjunction with graphs). $F \in P \times P \times P$ is a set of tuples of three vertices in P defining the faces (or triangles) of the mesh. Note that the set of faces induces a certain global topology on the object defined by the mesh (number of connected components, genus of each component). $I = \{f_i\}$ is a collection of functions called attributes defined over the vertices of the mesh, such as 'color' or 'temperature' ($f_i : P \rightarrow \mathbb{R}$), or "texture coordinates" ($f_i : P \rightarrow \mathbb{R}^2$). Note that although we restrict our discussion to objects in $\mathbb{R}^3$, the multi-resolution model and construction algorithm can support objects in any dimension. We call I and P the numerical information of the mesh, and F along with the decimation dependencies (see section 2.3), the symbolic information.

2.2 Dynamic Meshes

Consider a time-sequence of meshes: $\mathcal{M}_{t_0}, \mathcal{M}_{t_1}, \ldots, \mathcal{M}_{t_k}$, where $t_0 < t_1 < \ldots < t_k$. All mesh components, i.e. attributes, positions and adjacency, become a function of the time t_i: $\mathcal{M}_{t_i} = (P_{t_i}, F_{t_i}, I_{t_i})$. For our purposes the actual time values are irrelevant, and so we can normalize the time-steps to unitary intervals $\{t_i\} \rightarrow i$. Therefore, we examine the modifications between two consecutive meshes $\mathcal{M}_i$ and $\mathcal{M}_{i+1}$, and distinguish between different possible classes of changes:

1. Attribute changes: $P_i = P_{i+1}, F_i = F_{i+1}, I_i \neq I_{i+1}$.
2. Position changes: $F_i = F_{i+1}, P_i \neq P_{i+1}, I_i \neq I_{i+1}$.
3. Connectivity changes: $F_i \neq F_{i+1}, P_i \neq P_{i+1}, I_i \neq I_{i+1}$, but the topology of the object does not change.
4. Topological changes: $F_i \neq F_{i+1}, P_i \neq P_{i+1}, I_i \neq I_{i+1}$, with no restriction.

This broad notation covers a large class of possible dynamic meshes defined in scientific simulations, graphics and animation. This includes any finite-element simulations of dynamic systems or key-framing animations featuring attribute and positional changes. It includes continuous affine transformations and free form deformations, by sampling the modification function over time, and creating a sequence of changing meshes. But it also includes dynamic meshes with more drastic changes such as the creation and removal of holes, splitting and merging of components etc.

The higher modification level a representation scheme can support the larger the class of meshes it can represent, and the greater its expressive power is. An important factor in the definition of our scheme was the ability to support all different levels of dynamic change without sacrificing too much the possibility to optimize the representation for models with lower levels of dynamic change (for example by using quantization of attributes or compression of geometry positions).

2.3 Multi-Resolution Model

A multi-resolution mesh representation for a geometric object $\mathcal{M}$, is a representation that embodies a set of meshes $\{\mathcal{M}^1, \mathcal{M}^2, \ldots\}$ each of which is in turn a representation for $\mathcal{M}$. These representations can be seen as different approximations of the original object according to some tolerance.

One popular way of creating a multi-resolution model is by decimating an initial mesh $\mathcal{M}$ from the bottom up to a coarse mesh. In very general terms this process involves three primary decisions:

1. Selecting the primitive decimation operation and the basic decimation element e.g. a vertex in vertex removal, an edge in edge contraction. These operation and elements are used in a priority queue which governs the order of decimation.
2. Defining an error function (estimate) introduced by applying the primitive decimation to the mesh. This function is highly dependent on the mesh type and decimation scheme: e.g. terrains [2], scientific data [1], or graphical objects with color and texture attributes [7, 12]. This type of function is used (i) during the construction of the multi-resolution model for priority in the decimation queue, and (ii) for traversal of the resulting model to extract a given mesh.
3. Choosing the type of multi-resolution data-structure used for storage.

Once these are set, the algorithm for building a multi-resolution model consists of inserting all decimation elements into a priority queue, repeatedly choosing the first element, applying the decimation to the mesh, and encoding the decimation and its error in the

data-structure. In a sequential model, the sequence of discrete modifications (decimation operations) is recorded in a linear data structure. According to the direction of traversal and the current approximation, a series of operations from this sequence is performed on the existing mesh either from coarse to fine or from fine to coarse. In the graph model [2, 6, 11, 16, 25] simplification is performed in multiple levels where each level includes only independent decimation operations. The operations are recorded in a Directed Acyclic Graph (DAG). The nodes of the DAG represent decimation operations while the edges represent dependencies between such operations. If we assume that all roots in this DAG are connected to a single virtual super-root, then a cut in this graph is a collection of edges, which intersect all paths from the super-root to the leaves once and only once. Any such cut corresponds to a valid adaptive-resolution approximation of the model [5]. In particular the approximation corresponding to a specific cut can be obtained performing all and only the decimation primitives corresponding to nodes in the DAG below the cut (nodes in a path from the cut a leaf). The DAG model allows greater flexibility since one can generate adaptive approximations that were not explicitly constructed during the simplification process.

The decimation elements e can be vertices in a vertex removal scheme or edges in an edge contraction scheme. A cost function $cost(e)$ associates each element e with the cost (or degradation) due to its removal from the mesh. The priority queue Q keeps always at the top (maximum priority) the element with minimum cost. The *tol* (tolerance) parameter governs the stopping criteria for the decimation in each level. It is initialized to some minimum and gradually increased with the levels in order to continue building the hierarchy G. This helps control the maximum error allowed at each level, but should be used cautiously as it may lead to an unbalanced hierarchy. To guarantee a balanced hierarchy we keep performing the decimation process if the tolerance criterion is satisfied or if a minimum percentage of the mesh vertices have not been decimated yet. Once the DAG is constructed, a mesh representation can be generated by traversing the DAG from the roots towards the leaves, and creating a cut in the DAG. At each step, the error stored at the current node is compared with a given error tolerance. If this tolerance is not met, the cut is advanced from this node to its children, reversing their decimation operations and checking them in the same manner. In order to create coherent triangulations, a node can be included in the cut only if all its parents are already in the cut (not only the parent from which this node was reached). This is corrected by recursively checking and adding all the parents of a node before visiting it.

Our multi-resolution time-dependent model extends the graph-based approach. This means that in addition to the attributes, positions and connectivity information, the dependencies recorded in the graph may also be time-dependent.

3 DYNAMIC MULTI-RESOLUTION MODEL

3.1 Mesh Mapping and Global Vertex Indexing

In order to define the dynamic multi-resolution model we need to have some correspondence mapping between the vertices of the mesh in the consecutive time-steps. One simple approach to accomplish this is to assume that each vertex can be identified precisely through time. Let $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_k$ be the dynamic sequence of meshes. This mapping means that if $p_i \in \mathcal{M}_n$ and $p_j \in \mathcal{M}_m$ then $i = j$ iff p_i and p_j are considered the same vertex in different time-steps (t_n and t_m respectively). In other words the mapping between consecutive time steps is implemented using a global indexing scheme for the vertices in the mesh through time so that new vertices can appear and old vertices can be removed. Each new vertex must have a distinct identifier and the identifiers of removed vertices cannot be reused. This restriction can simplify both the structure and implementation of the algorithms. Additional conditions under which an index can be reused are also possible but tend to become complex for incremental construction of the T-DAG. The underlying assumption is that although dynamic changes are involved, most of the meshes have similar sets of vertices (the worst case will mean each time-step will have a separate vertex set).

Furthermore, we need a similar mapping between vertices of different levels of approximation. Therefore, we restrict our choice of decimation operation to those which preserve a mapping between the vertices before and after applying the operation. For example, vertex removal or Half-edge contraction do not introduce new vertices and therefore the identity of vertices is carried through trivially to any level. By applying some positional change in general edge contraction, one can map the new vertex introduced after the contraction to one of the two old vertices. Let $\mathcal{M}^i$ be a mesh at some resolution, and let $\mathcal{M}^{i+1}$ be a mesh created by applying such decimation operator to $\mathcal{M}^i$, then we have $P^{i+1} \subset P^i$. This implies that the total number of nodes in the DAG is exactly the number of vertices in the original finest resolution mesh. Another possibility in edge contraction is to use a total of $2n$ vertices for all levels when n is the number of vertices in the fine resolution mesh [11].

3.2 Definitions

Once the two mapping restrictions are met, we can identify a node in the DAG with some specific vertex in the dynamic mesh. Such node represents the decimation operation connected with this vertex (e.g. the removal of this vertex or the contraction of an edge adjacent to it). We then attach all the numeric vertex information and symbolic graph information as fields to the nodes. All nodes will have the same set of five fields: (i) vertex attributes, (ii) vertex positions, (iii) node decimation error, (iv) parent node links in the DAG, and (v) child node links in the DAG.

These fields have two basic types: single-valued fields and multiple-valued fields. Error estimation, vertex coordinates, or color components have only a single possible value, and therefore are defined as single value fields. Parent and child links in the DAG have several values associated with them all valid at the same time, and therefore are defined as multiple-valued fields. For a static multi-resolution DAG the fields in all nodes would be static. In a dynamic setting they are time dependent and need to be represented as functions of time. Both the numerical and the symbolic information are treated in the same manner by attaching different time-tags to different values in the fields. These tags are in the form of sequences of ranges (i, j), with $i \leq j$, each defining a continuous time interval (t_i, t_j) where the value is *alive* in its field.

Consider a dynamic mesh with k time-steps. We define a *single-valued* field of a node n as a function $F_s : \mathbb{R} \rightarrow \mathbb{R} \cup \{\perp\}$, and a *multiple-valued* field as a function $F_m : \mathbb{R} \rightarrow 2^{\mathbb{R}} \cup \{\perp\}$. For every $t, 0 \leq t \leq k-1$, $F_s(t)$ is a real number, and $F_m(t)$ is a group of numbers (for simplicity of notation even pointers are assumed to be represented as real numbers). A value x is defined as not *alive* at time step t iff its field returns $\perp$. If x has a time tag (i, j), then we call i *birth time* and j *death-time*. Note that x can have be multiple birth/death times.

> *The* **T-DAG** *is the collection of all values for all time steps of all the fields in all the nodes.*

Note that the overall **T-DAG** can be a general graph but for any specific time-step t, the collection of all alive parent or all alive child links form an actual DAG at time t (see Figure 1). In general, a node n_0 in the T-DAG might be a descendent of node n_1 at time t_i, and an ascendant of node n_1 at time t_j, as long as $(i \neq j)$.

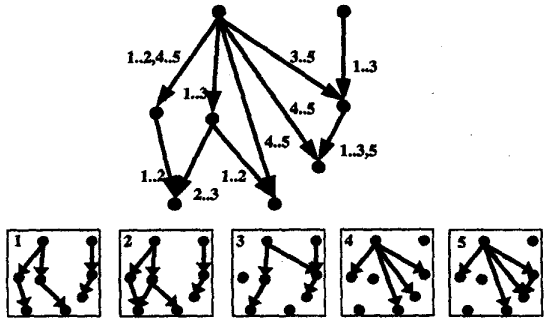

Figure 1: The child links in a T-DAG structure with five time-steps (top), and the five DAGs it represents (bottom). Each child link edge in the top T-DAG carries a tag depicting the range of time-steps in which it is alive.

3.3 Queries

As a model for the dynamic meshes $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_k$, the T-DAG is parametric in two dimensions: resolution and time (see Figure 7). Therefore, every valid T-DAG query needs to instantiate these two parameters. The fundamental query we are interested in is a random-time and random-resolution mesh retrieval in the form *(time = t,tol = ϵ)*. The result of this query is an approximation $\mathcal{M}_t^\epsilon$ of the mesh $\mathcal{M}_t$ which does not differ from $\mathcal{M}_t$ by more than ϵ under some given error function. Note that we use only ϵ in our discussion for simplicity, but the error functions supported can involve more complex computations and additional parameters like in view-dependent error estimate.

The second type of queries we consider is incremental in nature: given $\mathcal{M}_t^\epsilon$ find $\mathcal{M}_{t+1}^\epsilon$ or $\mathcal{M}_{t-1}^\epsilon$, or given $\mathcal{M}_t^{\epsilon_1}$ find $\mathcal{M}_t^{\epsilon_2}$. The incremental queries can be supported trivially using the fundamental query. However, we are interested in finding progressive solutions, where the mesh of previous time step (previous resolution) is progressively updated to arrive at the next time (resolution). The third type of query is the most difficult to support progressively and it involves increments in both time and resolution.

3.4 Time-Dependent Storage and Retrieval

Considering that almost all actual information of the T-DAG is stored at the nodes, the above queries would be translated to basic time-dependent retrieval operations on the fields of the nodes. These operations are of the form:

```
node.getAliveValue(timeStep t)
```

which returns the alive value for a single-valued field (e.g. `getError`, `getPosition`) or:

```
node.getAliveValues(timeStep t, array
&values)
```

which returns the alive values for a multiple-valued field (e.g. `getChildren`, `getParents`).

Let k be the number of time-steps in the T-DAG, and d the maximum single-time size of a multiple-valued field of a node (for example, the maximum number of children for all time-steps). If n is the number of different values for a specific field in all time-steps, then $n \leq kd$. For fast random and incremental access of the time-dependent information ($O(1)$ for single-valued, $O(d)$ for multiple-valued fields), values can be stored in an $O(kd)$ array of all time-steps. However, often this storage is too costly. A better option would be to store the single-valued fields in a list sorted by birth time, and the multiple-valued fields as an interval tree. This will bring down the storage to $O(n)$ in the worst case, but the retrieval time would be worse. For the single-valued fields, incremental access would be constant, but random time access would take $O(\log(n))$ (binary search for the active value). For multiple-valued fields, both random and incremental access would take $O(\log(n) + d)$ for collecting the active values from the tree.

A slightly better approach for multiple-valued fields allows $O(d)$ incremental and sometimes random access, and $O(\log(n) + d)$ for general random access. We store the values in two lists, one sorted by increasing birth times and the other sorted by decreasing death times ($O(n)$ storage). We define a time-window of size $w << k$, where we store all active values for this window in time (i.e. from t_i to t_{i+w}) in an array of $O(wd)$ size. Denote this set of values as $S(t_i, w)$. This allows $O(d)$ random and incremental access for all t with $t_i \leq t \leq t_{i+w}$. Let $S_d(t)$ be the set of values dying in time t and $S_b(t)$ be the set of values born in time t. It is simple to check that $S(t_{i+1}, w) = S(t_i, w) - S_d(t_i) + S_b(t_{i+w+1})$. Similarly, $S(t_{i-1}, w) = S(t_i, w) + S_d(t_{i-1}) - S_b(t_{i+w})$. We keep pointers to the value with smallest birth time greater than t_{i+w} in the birth time array, and to the value with the largest death time smaller than t_i to the death time array. Using those pointers, the incremental (both forward or backward in time) update takes $O(d)$ time. For random access, the binary search to reposition the pointers and the collection of values inside $S(t, w)$ will take $O(\log(n) + d)$.

Overall, incremental time queries can be implemented in $O(d)$ time-complexity with $O(n)$ storage. Also, inside a given window in time, we can preserve $O(d)$ for random time queries.

3.5 Resolution Change

For traversal in resolution space, an array of root indices is stored in the T-DAG in addition to all nodal values. This array is stored in the same manner as any other multiple-valued field. For a give time-step t, the T-DAG can be seen as a static multi-resolution DAG for $\mathcal{M}_t$. Therefore, the type of error functions supported by the T-DAG are exactly the same as those for the static case, including adaptive and view-dependent refinements (see Color Plate 1). Moreover, traversal is done using essentially the same algorithm for top down construction of a mesh from a static multi-resolution representation. For a given time-step t, we start from the set of alive roots at time t, and traverse the graph in a top down manner using only child and parent links which are alive in time t. At each node the error stored for time t is checked using the given error function. Other attributes and position information (used for example, for rendering the mesh), are all accessed as a function of time t, allowing simple modification of the shape and appearance of the mesh through time. Moreover, if the time t remains static, incremental resolution queries on a T-DAG can be supported dynamically by updating the DAG-cut similar to static multi-resolution DAGs.

Although our storage and retrieval scheme favors incremental time queries inside nodes, we adopt a lazy-evaluation scheme for the values of the nodal fields. For a given time t only the fields of nodes visited during the traversal are retrieved and updated to the current time t. As a consequence a node which was above the cut at time t for a specific tolerance, and below the cut at times $t+1, \ldots, t+m-1$, would not be evaluated and updated in those times. If it returns to be above the cut at time $t+m$, the retrieval of values in its fields would become random-time queries instead of incremental. However, since the T-DAG is a multi-resolution model, only a subset of all nodes (and hence, a subset of the vertices) is needed to create an approximating mesh for a given tolerance. Therefore, the cost of random time retrievals of values in some node fields is a minor penalty with respect to the large gain of updating only a sub-set of the nodes at each time-step. Figure 6 displays for different T-DAGs the average traversal time needed to create meshes of various resolutions. These plots are similar in behavior to that of static multi-resolution DAGs [2, 9].

4 T-DAG CONSTRUCTION

Consider a time sequence of meshes $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_k$, which satisfies the global vertex indexing scheme defined in Section 3.1. In this section we introduce and analyze an on-line algorithm for constructing a T-DAG providing a multi-resolution model for the given time sequence of meshes.

4.1 Space-Time Tradeoffs

The trivial scheme that one may consider is to treat each mesh $\mathcal{M}_i$ independently, and construct an independent multi-resolution DAG $\mathcal{MR}_i$ which conforms to the resolution-levels mapping (Section 3.1). The result of this scheme is a sequence of multi-resolution DAGs $\mathcal{MR}_0, \mathcal{MR}_1, \ldots, \mathcal{MR}_k$. To encode such sequence of DAGs in a single T-DAG we can define $\mathcal{P} = \{v | v \in \mathcal{M}_i \text{for some} i 0 \leq i \leq k\}$, as the union of all vertices in all time-steps. Then we construct a node in the T-DAG for each such vertex, and encode all $\mathcal{MR}_i$ meshes using these nodes in the following manner:

```
loop on all timesteps t
   loop on all nodes n in P
      if n ∈ MR_t
         set the fields of n at time t with
         the values of the fields of n in MR_t
      else
         set the fields of n at time t
         to empty fields
```

Let x be a new value for time $t+1$ for some field in node n. If this field is a single-valued field F_s then if $F_s(t) = x$, we postpone the death time of x to be $t+1$. If $F_s(t) \neq x$, we create a new value x for this field and set its birth and death time to be $t+1$. If this field is a multiple-valued field F_m then if $x \in F_m(t)$, we extend the death time of x to be $t+1$. If $x \notin F_m(t)$, we create a new value x for this field and set its birth and death time to be $t+1$.

Using time range tags to encode the lifespan of field values is beneficial when the ranges of the values are long. This gain is expressed both in terms of space (changing the death-time instead of creating a new value) and in terms of traversal time (less time-dependent updates). The longer these chains of similar values are, the greater the gain. Therefore, this T-DAG construction scheme is almost equivalent to the worst case of storing each $\mathcal{MR}_i$ separately. The fact that the DAGs were created independently results in rare occurrences of fields having the same value for consecutive time steps (Figure 2(a)).

The other extreme for a T-DAG creation scheme would be to use just a single DAG for all time-steps. For example, create $\mathcal{MR}_0$, and then instead of looping on the nodes of $\mathcal{MR}_i$, we loop on all $\mathcal{M}_i$, and apply the same decimations as in $\mathcal{MR}_0$ to the mesh $\mathcal{M}_i$. This would mean most fields (e.g. child or parent links) would have values alive for the whole time range, but some (such as coordinate positions or decimation error) would still have changing values over time (Figure 2(b)). In such a scheme the storage space and time-updates are kept to a minimum. However, a single DAG will not have an optimal structure for all different meshes in all time-steps. This will generally force traversals to reach down to lower levels of the graph in order to satisfy a given error tolerance, and results in larger meshes for a given tolerance (see Color Plate 2). Furthermore, it is not always possible to use a single DAG to encode a whole sequence of dynamic meshes. As an example, consider the case of dynamic meshes where the connectivity or the topology changes over time.

The first construction scheme favors optimizing each specific time-step locally in terms of traversal time for a certain error, or the quality of the mesh for time restrictions. The second (which is not always possible) means sacrificing traversal time or quality at the gain of lower storage space, and less time-dependent updates in nodes. Defining a general construction scheme which is optimal both in terms of storage-space and traversal time might be difficult. Instead, we aim at defining a framework where a valid T-DAG can always be constructed, and there is a possibility of control over the different tradeoffs, by using a predefined construction parameter.

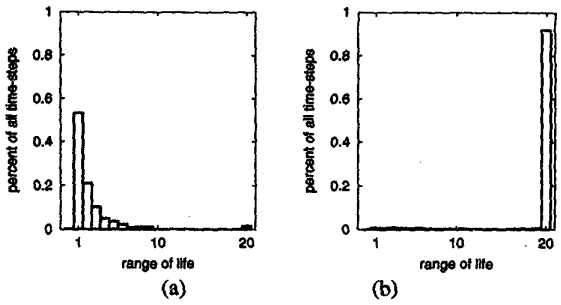

Figure 2: Percent of parent link values of all nodes in a T-DAG covered by parent link values of a given range for two different T-DAGs. The size of T-DAG (b) is around 30% the size T-DAG (a). The two T-DAGs were constructed for the snake example of Color Plate 1. The original dynamic meshes hold 12000 vertices in 20 time-steps. (a) was constructed by simply merging independent DAGs created for each time-step, hence, most of the values have very small time ranges, (b) was constructed by using only a single DAG of the first time-step and applying, whenever possible, the exact same decimations for all other time-steps meshes. Similar distributions occur in most fields of T-DAGs constructed using these two opposite schemes.

4.2 Incremental Construction

In addition to the space-time tradeoffs for a given T-DAG construction, it is often beneficiary to construct the T-DAG incrementally over time. For example, in scientific simulations heavy computations are involved in producing the mesh for each time-step. Instead of waiting until the whole process is complete to construct the model, we would like to be able to visualize in a multi-resolution manner even partial results. Let $\mathcal{TD}_t$ be the T-DAG of times $0, \ldots, t$, and assume $\mathcal{TD}_0 = \mathcal{MR}_0$. An incremental algorithm creates $\mathcal{TD}_i$ by merging successively $\mathcal{TD}_{i-1}$ with $\mathcal{MR}_i$ for all i.

However, creating the DAG $\mathcal{MR}_{i+1}$ and merging it with the previous T-DAG $\mathcal{TD}_{i-1}$ might involve complicated graph matching problems. Instead, the key idea behind the incremental T-DAG construction algorithm is to create at each time-step a multi-resolution DAG using decimations which will conform to the existing T-DAG. This is done by using an enhanced priority in the decimation process, introducing some history considerations which augment the regular priorities of decimation cost. These considerations aim to preserve the structure of the previous time-steps T-DAG.

We define a history of decimations as a sequence of decimation operations that were applied to a mesh, sorted by increasing level and in each level by the order in which they were executed. Our algorithm uses two such history sequences: Hin holds the previous time-step decimations and $Hout$ gathers the current time-step decimations. At the beginning of each time-step (> 0) $Hout$ is assigned to Hin and cleared.

The algorithm (Figure 3) loops on the decimation operations in Hin using **getElement**(). This function returns the next element (edge, vertex etc.) representing the next decimation operation in Hin from levels 0 and up to the current level. This is done since

```
DecimateConform(M, TD, time, Hin, Hout)
{
M is the initial fine resolution mesh
TD is the T-DAG of decimation operation
time is the current time-step
Hin will hold the previous order of decimation
Hout will store the current order of decimation
Q is a priority queue of decimation elements

tol = minimumTol
Hin = Hout, clear Hout
loop until M is coarse enough {
   clear dependencies for this level
   fill Q with decimation elements from M
   while e = Hin→getElement()
      find e' matching e in Q
      if e' is not found or
         e' is marked as dependent or
         e' → cost() > tol or
         largeDiff(e' → cost(), e → cost())
         continue
      remove e' from Q
      ApplyDecimation(e, M, TD, Hout)
   }
   while Q is not empty {
      e = Q → first()
      if e is marked as dependent
         continue
      if e → cost() > tol
         break
      ApplyDecimation(e, M, TD, Hout)
   }
   increase tol
}

ApplyDecimation(e, M, TD, Hout)
{
   mark all elements dependent on e
   decimate M using e
   store decimation in TD(time)
   store e dependencies in TD(time)
   store e in Hout
   update Q if needed
}
```

Figure 3: An outline of the algorithm for one time-step of creating a multi-resolution T-DAG model. The degree of conformity between the current and previous time-steps is governed by the function **largeDiff**. This function checks the difference in cost of the decimation in the current and the previous time-steps. The algorithm first tries to decimate conforming to the previous time-step, and only then reverts to its own priority queue. At each time-step the decimation order is recorded and used in the next time-step by using the decimation histories Hin and $Hout$.

some decimations of previous levels may have been skipped due e.g. to large cost, but they might be applied in the current level. The element extracted is matched to an element in the current priority queue Q. The matching of elements is used in a loose sense, e.g. two edges can match in edge contraction even if the surrounding triangles do not match perfectly. A stricter matching strategy would fail more often, and our goal is to use such matching only to increase the chance of long time ranges for values in the node fields. The decimation is skipped in one of the following four cases: (i) if no matching element is found in the current queue, (ii) if the element found is dependent in this level, (iii) if the cost of applying the decimation is too large ($cost() > tol$), or (iv) if the difference between the previous decimation costs and current is too large (see Section 4.3 for the definition of **largeDiff**). Otherwise, it is performed and recorded in $Hout$ for the next time-step. Some decimations from the previous time-step cannot be found due to topology or connectivity changes. Other decimations are valid only on the current mesh, and so the algorithm performs them at the end of each level.

Figure 4 illustrates a comparisons between the decimation histories of consecutive time-steps of the balls in (Color Plate 4). As can be seen, most of the decimations are carried across the time-steps in the same order, the likelihood of long time ranges for nodal field values is increased, and there is gain in storage and time-values updates.

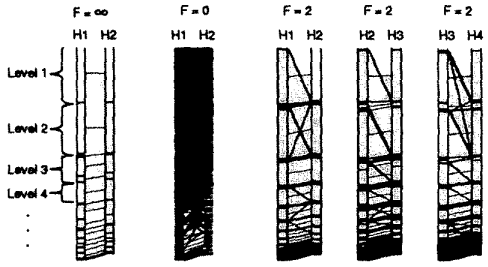

Figure 4: Comparisons between decimation histories of consecutive time-steps for different strategies. The decimation operations are laid out from top to bottom according to their execution order. White regions represent operations that match in both histories. Gray regions represent operations that were performed only at the left or right histories. The links across histories connect regions where the same operations were performed in different order. When $F = \infty$, the histories match exactly apart from places where topology or connectivity changes. When $F = 0$ each history is almost totally different since no similarity constraint is imposed. When $F = 2$ over several time-steps one can see that most of the histories are using the same decimation operations in the same order.

4.3 Space-Time Control Factor

When the conformity between consecutive time-step decimations is large, more attribute values are unchanged across several time steps and the time-range tags of such attributes in the node fields are larger. As discussed earlier, this results in reduced storage size, and faster time-dependent changes. If c_t is the cost of the current decimation in the current time-step, and c_{t-1} is the cost of the same decimation in the previous time-step, then we use **largeDiff**(c_t, c_{t-1}) test function in order to control the level of conformity between the two time-steps. Whenever this function returns true the decimation is skipped. Therefore, for greater conformity more such tests should fail, and for lesser conformity, more tests should succeed. We use as specific **largeDiff**:

$$F \times |c_t - c_{t-1}| > \max(|c_t|, |c_{t-1}|)$$

Therefore, as the value of the factor F increases, the conformity decreases. For example, consider three T-DAGS of the meshes sequence in Figure 7 built with conformity factors $F = 1.1, 2, 10$. These meshes have 7200 faces and 50 time-steps. Figure 5 displays the percent of child links covered by ranges of child link values in the three different T-DAGs. Other nodal field values (e.g. parent links) display very similar patterns, and in fact, the ratio of the total sizes of the three different T-DAGs is $7 : 9 : 20$ for $F = 1.1, 2, 10$ respectively. Figure 6 displays the average traversal time from the roots to a cut satisfying different tolerances for the three different conforming factors. Larger F values means the T-DAG shape will be optimized for each time-step separately, creating higher cuts in the DAG for a given tolerance, and presumably traversing faster.

However, during the traversal, the time-dependent fields need to be updated. Whenever the conformity is larger, this update is smaller and faster and this results in the total traversal being faster.

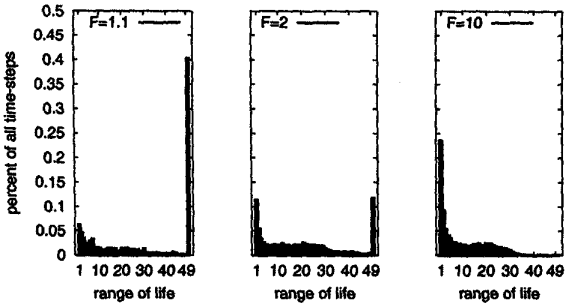

Figure 5: Percent of child link values of all nodes in a T-DAG covered by child link values of a given range for T-DAGs created with different values of the conforming factor $F = 1.1, 2, 10$.

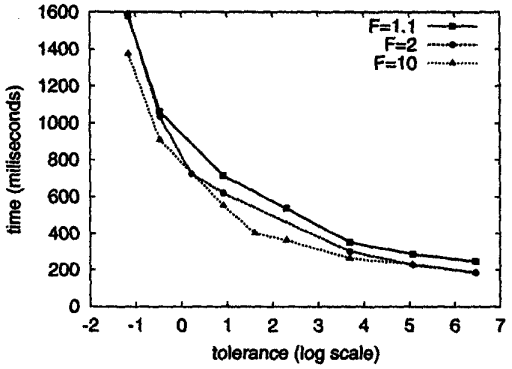

Figure 6: Average traversal time (of 50 timesteps) for a given tolerance for T-DAGS of different conforming factor F.

5 RESULTS

In this section we examine several examples for the use of a T-DAG. We show the flexibility of the model and the ability of the construction algorithm to encode time-dependent information with different restrictions. All examples were created on an Intel Pentium-2 450 mhz machine with 128mb of memory.

In the first example (see Color Plate 3), is the output of a simulation of two sub-atom particles colliding over a 2D domain. This simulation tracks several attributes changing over 50 time-steps (density, electrostatic field, ...). Under these conditions, defining an optimal decimation for all variables could be difficult. Moreover, if new attributes were introduced, this would mean recalculation of the whole structure from the first time-step. Instead, we chose to use a purely geometric condition to govern the decimation process. Using random maximal vertex removal, we preserve at each step a Delaunay triangulation (this scheme was presented in [2] for static terrain meshes). On average, the Delaunay triangulation gives good adaptive triangulation results for all attributes. The real advantage of using such a scheme is in the fact that a single average DAG was used for all time-steps and all attributes. This means that the child and parent links all have the maximum time-range and therefore there are no time-dependent updates. The different triangulations extracted over time are a result of the differences in the error field for different attributes over time.

Considering a 3D surfaces, additional tests are necessary besides checking the approximation error during the traversal of the DAG. For example, in order to preserve correct embedding of the surface, triangles in the neighborhood of the decimation need to be checked for orientation changes, global intersections should be tracked, etc. [22]. These types of tests could involve heavy computation, making it impractical at traversal time. However, if these tests are carried out during the construction stage of the T-DAG, verifying that all cuts in all time-steps satisfy the tests, the traversal time can be reduced. A simpler choice, although it does not guarantee correct embedding, would be to penalize during decimation the cost of contractions that cause a change in triangle orientation, and omit such checks during traversal.

In the second example we encode a dynamic sequence of meshes created by two waves colliding (using 7200 faces). We use half-edge contraction as decimation primitive and quadratic error metric for tracking decimation cost [7]. The T-DAG constructed then supports the extraction of meshes at any point in the two dimensional space of time and adaptive resolution (Figure 7).

Figure 7: Two dimensional space of time (horizontal) and resolution (vertical) defined by the T-DAG model of a mesh of two colliding waves.

In cases where the connectivity or topology of the dynamic mesh changes, the differences of the symbolic information through time must be encoded as changes in the child and parent field values over time. The last example (Color Plate 4) shows a two balls merging (or a ball splitting) with 12,800 faces. The multi-resolution hierarchy was built with quadratic error metric using half-edge contraction. In this case the finest resolution triangulation has different connectivity in most time-steps and there is also a discrete change in topology at one time-step. In fact, using the T-DAG model these changes were encoded seamlessly by the construction algorithm.

6 CONCLUSIONS AND FUTURE DIRECTIONS

Numerical simulations and dynamic environment processing deformable large scale models are becoming more common as the computation power and ability to display high-end graphics evolves. These models are larger and more complex than static geometric models, and therefore necessitate further use of multi-resolution techniques. In this paper we presented the T-DAG, a multi-resolution representation for dynamic meshes with arbitrary change. This model is flexible enough to encode models ranging from the use of a single DAG for all time-steps to more complex graphs with connectivity and topology change. The construction

algorithm is simple enough to be used with many types of decimation operations, yet it is powerful enough to seamlessly encode topology and connectivity changes in the dynamic meshes.

There are several possible extensions for this work. One of the most important remaining challenges for the T-DAG is the update of any adaptive cut dynamically over time instead of creating it by traversing from the roots. Another involves out-of-core computations. The T-DAG structure evolves through time in coordination with the meshes around the current time-step. Although the amount of time-dependent information in each node of the T-DAG could be large, if one concentrates on the window of time-steps, the live information is much smaller. This fact could be used to support out-of-core multi-resolution dynamic models, enabling efficient decomposition of the data into viewing windows of time-steps which can fit into memory. Also, since the temporal information is gathered in the T-DAG nodes, temporal coherency could be exploited for compression.

Lastly, it is possible to use the T-DAG for applying time-dependent constraints for multi-resolution in the same manner as view- and space-dependent constraints are used. For example, an object which moves or deforms rapidly could be displayed in lower resolution than an object which deforms slowly.

References

[1] C. Bajaj and D. Schikore. Topology preserving data simplification with error bounds. *Computers and Graphics*, 22(1):3–12, 1998.

[2] M. de Berg and K. T. G. Dobrindt. On levels of detail in terrains. *Graphical Models and Image Processing*, 60:1–12, 1998.

[3] M. Eck, T. DeRose, T. Duchamp, T. Hoppe, H. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *ACM Computer Graphics Proceedings, SIGGRAPH'95*, Annual Conference Series, pages 173–180, 1995.

[4] A. Finkelstein, C. E. Jacobs, and D. H. Salesin. Multiresolution video. In *ACM Computer Graphics Proceedings, SIGGRAPH'96*, Annual Conference Series, pages 281–290, 1996.

[5] L. De Floriani, P. Magillo, and E. Puppo. Building and traversing a surface at variable resolution. In *Proceedings of the IEEE Visualization Conference VIS'97*, pages 103–110, 1997.

[6] L. De Floriani, P. Magillo, and E. Puppo. Data structures for simplicial multi-complexes. In *Proceedings Symposium on Spatial Databases*, Hong Kong, China, July 1999.

[7] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In Turner Whitted, editor, *ACM Computer Graphics Proceedings, SIGGRAPH'97*, Annual Conference Series, pages 209–216. ACM SIGGRAPH, Addison Wesley, August 1997.

[8] Tran S. Gieng, Bernd Hamann, Kenneth I. Joy, Gregory L. Schussman, and Issac J. Trotts. Constructing hierarchies for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 4(2):145–161, April 1998.

[9] P. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. In *ACM Computer Graphics Proceedings, Annual Conference Series, SIGGRAPH'97, Multiresolution Surface Modelling, Course Notes No. 25*, 1997.

[10] H. Hoppe. Progressive meshes. In *ACM Computer Graphics Proceedings, SIGGRAPH'96*, Annual Conference Series, pages 99–108, 1996.

[11] H. Hoppe. View-dependent refinement of progressive meshes. In *ACM Computer Graphics Proceedings, SIGGRAPH'97*, Annual Conference Series, pages 189–198, 1997.

[12] H. Hoppe. New quadratic metric for simplifying meshes with appearance attributes. In *Proceedings IEEE Visualization'99*, pages 56–66. IEEE Comp. Soc. Press, 1998.

[13] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings IEEE Visualization'98*, pages 35–42. IEEE Comp. Soc. Press, 1998.

[14] R. Klein and J. Kramer. Multiresolution representations for surface meshes. In *Proceedings of the SCCG*, 1997.

[15] J. E. Lengyel. Compression of time-dependent geometry. In *Proceedings of the 1999 ACM Symposium on Interactive 3D Graphics*, Atlanta, Georgia, April 1999.

[16] Paola Magillo. Spatial operations on multiresolution cell complexes (phd thesis). Technical Report DISI-TH-1999-03, Dipartimento di Informatica e Scienze dell'Informazione, University of Genova, Italy, 1993.

[17] A. Maheshwari, P. Morin, and J. R. Sack. Progressive tins: Algorithms and applications. In *Proceedings 5th ACM workshop on Advances in geographic information systems*, Las Vegas, 1997.

[18] INTERNATIONAL ORGANISATION FOR STANDARDISATION CODING OF MOVING PICTURES and AUDIO ISO/IEC JTC1/SC29/WG11 N2995. *MPEG4 standard specifications*, http://drogo.cselt.it/mpeg/standards/mpeg-4/mpeg-4.htm edition.

[19] J. Rossignac and P. Borrel. Multi-resolution 3d approximation for rendering complex scenes. In B. Falcidieno and T. Kunii, editors, *Geometric Modeling in Computer Graphics*, pages 455–465. Springer Verlag, 1993.

[20] William J. Schroeder. A topology modifying progressive decimation algorithm. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization 97*, pages 205–212. IEEE, November 1997.

[21] H. Shen, L. Chiang, and K. Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree. In *Proceedings of the IEEE Visualization Conference VIS'99*, pages 371–378, 1999.

[22] O. G. Staadt and M. H. Gross. Progressive tetrahedralizations. In *Proceedings of the IEEE Visualization Conference Vis98*, pages 397–402, 1998.

[23] E. J. Stollnitz, T. D. DeRose, and D. H. Salesin. *Wavelets for Computer Graphics*. Morgan Kaufmann Publishers, 1996.

[24] P. M. Sutton and C. D. Hansen. Isosurface extraction in time-varying fields using a temporal branch-on-need tree (t-bon). In *Proceedings of the IEEE Visualization Conference VIS'99*, pages 147–154, 1999.

[25] J. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In *Proceedings of the IEEE Visualization Conference Vis96*, pages 327–334, 1998.