Multi-Resolution Dynamic Meshes with Arbitrary Deformations

Ariel Shamir Valerio Pascucci Chandrajit Bajaj
The Center for Computational Visualization
TICAM, University of Texas at Austin

January 14, 2000

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 objects pose similar challenges in terms of time and space and need the development of similar solutions. We present the T-DAG, an adaptive multi-resolution representation for dynamic meshes with arbitrary deformations including attribute, position, connectivity and topology changes. We also provide an on-line algorithm for constructing the T-DAG, enabling the traversal and use of the multi-resolution model for partial playback while still constructing it.

1 Introduction

Dynamic scenes in computer graphics are defined by associating some of the scene parameters with a function of time. Global parameters like the position of the viewer (walk-through) or the position of objects can be encoded easily using rigid body transformation or interpolators and behaviors [15]. Local deformations of objects are generally much more complex and more time and space consuming. Multi-resolution techniques have been shown to be an effective tool for handling the complexity of geometric objects. However, most of the work done in this field concentrates on static objects. Moreover, the application of previous proposed solutions for dynamic models is restricted to objects in which the connectivity and topology are fixed over time. We introduce a new multi-resolution representation scheme for dynamic geometric objects where no restriction is imposed on the modification through time in terms of their connectivity or topology. We also present an algorithm for on-line construction of the multi-resolution model over time. This allows traversing the the multi-resolution model of previous time-steps while still augmenting the model, in each time-step, with the newly modified object. Our scheme extends the possibility of using adaptive multi-resolution techniques for display and transmission of general deformable time-dependent meshes.

1.1 Previous Work

There are many approaches for creating multi-resolution representations of geometric data for graphics and visualization [8, 11, 14, 16]. They vary in both the simplification scheme like vertex removal [1], edge contraction [9, 10], triangle contraction [7], vertex clustering [17], wavelete analysis [2, 20], and in the

structure used to organize the levels of detail (either linear order [9, 11] or in a DAG [1, 5, 7, 14]). However, these works assume the finest resolution mesh is static and build a static multi-resolution representation. Our scheme draws from many of them and defines a model for applying them through time, when the underlying geometric mesh is dynamic.

A time dependent data structure for use in the extraction of isosurfaces from large time dependent data is presented in [21]. A temporal branch on need tree (a type of an octree) 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 *(timestep,isoval)* by traversing top down and branching only those parts that hold the correct isovalues for that 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 extractions from time varying data sets, 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 preprocessing. For volume rendering of time dependent data [18] use 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 pixels 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 *(timestep,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 in the same manner. The rendering of each octree-node volume is done separately and the sub-images are composited 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 change in connectivity or topology of the mesh.

An opposite approach was presented earlier in [3] for multiresolution 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, the fact that it is not clear how to define the average of several time-dependent surfaces, especially when topology and connectivity could change over time, means such an approach is difficult to apply on 3D meshes.

We generally use the more conservative approach of viewing time sequentially and supporting multi-resolution in spatial dimension. However, since we separate the temporal information of each vertex, there is no restriction to store this information consecutively. In fact, and any type of multi-resolution can be defined for this data similar to the binary trees of [18]. Another option is to use compression in temporal space. In [12] 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, [12] concentrates on compressing the V matrix, and we concentrate on encoding the G matrix using multi-resolution methods. Therefore both works 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 algorithm for on-line building of 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 computer graphics we assume that objects are represented by triangular surface meshes. This assumption also enables us to include other objects such as scientific simulation meshes in our discussion. We define a mesh $\mathcal{M}$ by a tuple $\mathcal{M} = (P, A, I)$. $P = \{p_i\}$ is a collection of points embedded in E^3 called *nodes* or *vertices*. $A : P \times P \rightarrow \{0, 1\}$ is a relation called adjacency, which defines the connectivity of the mesh. This relation also imposes a certain 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 nodes of the mesh, such as 'color' or 'temperature' ($f_i : P \rightarrow \Re$), or "texture coordinates" ($f_i : P \rightarrow \Re^2$). Note that although we restrict our discussion to objects in $\Re^3$, 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 A along with the decimation dependencies (see section 2.3), the symbolic information.

2.2 Dynamic Meshes

Consider a sequence of time dependent meshes:

$$\mathcal{M}_{t_0}, \mathcal{M}_{t_1}, \ldots, \mathcal{M}_{t_k}$$

where $t_0 < t_1 < \ldots < t_k$. The time dependency means that all mesh components, i.e. attributes, positions and adjacency, become a function of time $\mathcal{M}_{t_i} = (P_{t_i}, A_{t_i}, I_{t_i})$. For our purposes the actual time is irrelevant, and so we can map the time-steps to discrete intervals $\{t_i\} \rightarrow i$. Therefore, we examine the changes between two consecutive meshes $\mathcal{M}_i$ and $\mathcal{M}_{i+1}$, and distinguish between different possible levels of change:

1. Attribute changes: $P_i = P_{i+1}$ and $A_i \equiv A_{i+1}$, but $I_i \neq I_{i+1}$.
2. Position changes: $A_i \equiv A_{i+1}$ but $P_i \neq P_{i+1}$ and $I_i \neq I_{i+1}$.

3. Connectivity changes: $A_i \neq A_{i+1}$, $P_i \neq P_{i+1}$ and $I_i \neq I_{i+1}$, but the topology of the object does not change.

4. Topological changes: $A_i \neq A_{i+1}$, $P_i \neq P_{i+1}$ and $I_i \neq I_{i+1}$, and no restriction on the topology is given.

This broad notation covers a large class of possible dynamic meshes defined in animation, graphics and scientific simulation. This includes any key-framing animations and finite-element simulations of dynamic systems 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 creation and closing of holes, splitting and merging.

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 ability to optimize the representation for models with lower levels of dynamic change (for example by using quantification 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 bottom up. In very general terms this involves three primary decisions: 1. Selecting a decimation primitive, 2. Defining an error (estimate) for priority of decimation and traversal, and 3. Choosing the multi-resolution data-structure. Once these parameters are established, a multi-resolution model can be built by inserting all decimation elements into a priority queue, repeatedly choosing the first element, applying the decimation to the mesh, and recording it 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 ([1, 5, 13]) simplification is performed in multiple levels where each level includes only independent decimation operations (see Figure 1). Further dependencies between decimation operations of different levels are stored as a DAG (Figure 2). A cut in this graph is a collection of edges, which intersect all paths from the root to the leaves once and only once. Any such cut represents a valid multi-resolution approximation of the model [4] (Figure 3). The DAG model allows greater flexibility in generating adaptive approximations, which were not explicitly constructed during the simplification. An outline of the algorithm for creating a DAG based multi-resolution model is presented in Figure 4.

Once the DAG is constructed, a mesh representation can be generated by traversing the DAG from the roots towards the leaves. 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 nodes, 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 (and 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.

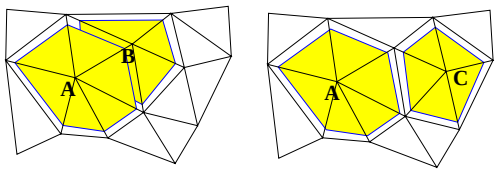


Figure 1: Vertices A and B can not be independently decimated using vertex removal operation (left). Vertices A and C can (right).

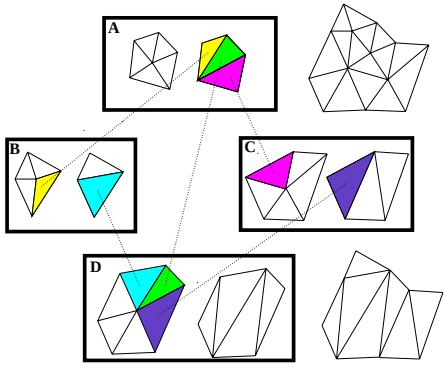


Figure 2: Building the DAG as a model for multi-resolution of a mesh. Each node includes information regarding a specific decimation (vertex removal in this case) and dependencies between the regions in the nodes are encoded as arcs in the DAG.

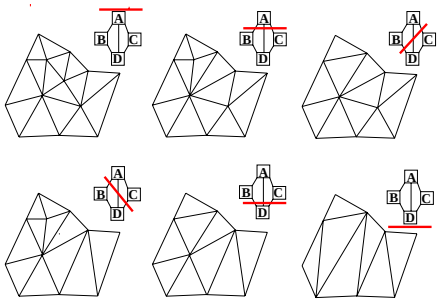


Figure 3: All possible approximations of the mesh of Figure 2 and the respective cuts in the multi-resolution DAG.

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 are also time-dependent.

3 Dynamic Multi-Resolution Model

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 each vertex can be identified precisely through time. Let $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_s$ be the dynamic sequence of meshes. This mapping means that if $p_i \in \mathcal{M}_k$ and $p_j \in \mathcal{M}_n$ then $i = j$ iff p_i and p_j are considered the same vertex in different time-steps (t_k and t_n respectively). In other words, new vertices can appear through time and old vertices can be removed, but each new vertex must have a distinct identifier,

```
DecimateLevels(M, G, tol)
M is the initial fine resolution mesh
G is the DAG of decimation operation
tol is some external predefined tolerance
Q is a priority queue of decimation elements

loop until M is coarse enough {
   clear dependencies for this level
   fill Q with decimation elements from M
   while Q is not empty {
      e = Q → first()
      if e is marked as dependent
         continue
      if e → cost() > tol
         break
      ApplyDecimation(e, M, G)
   }
   raise tol
}
ApplyDecimation(e, M, G)
{
   mark all elements dependent on e
   decimate M using e
   store decimation in G
   store e dependencies in G
   update Q if needed
}
```

Figure 4: An outline of the algorithm for creating a DAG based multi- resolution model. The tol parameters governs the error and stops decimation at each level if the error is too large. However, it is gradually increased as the levels increase in order to continue the hierarchy.

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 in an on-line manner. The underlying assumption is that although dynamic changes are involved, most of the meshes have similar sets of vertices (the worst case means 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, half-edge contraction, or general edge contraction when one of the vertices is mapped to the new vertex (along with a positional change), all comply with this restriction. Let $\mathcal{M}^i$ be a mesh, and let $\mathcal{M}^{i+1}$ be a mesh created by applying such decimation operator to $\mathcal{M}^i$, then we get $P^{i+1} \subset P^i$. This implies that the total number of nodes in the DAG is exactly the number of vertices in the original finest resolution mesh.

Once these restrictions are met, we can identify each node in the DAG with some specific vertex in the dynamic changing mesh. This node signifies 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 this node:

- Vertex attributes.
- Vertex positions.
- Vertex decimation error.
- Parent links in the DAG.
- Child links in the DAG.

We then treat all the fields in the same manner, converting them to a function of time. This is done by attaching a time-tag in the form of a range (i, j) to each value in the field. This range defines the continuous time range (t_i, t_j) where this value is *alive* in this field. Fields such as error estimation, node coordinates or color attributes have only one value alive at each time-step. However, the parent and child links of the DAG may have many different values alive at the same time. The collection of these values for a specific time t_k define the structure of the multi-resolution DAG at time t_k (see Figure 5). We call the collection of all the nodes in all time-steps the **T-DAG**. It is important to note that the DAG constraints are only true for discrete time-steps encoded in the TDAG. Node n_0 in the TDAG might be a child of node n_1 at one time-step, n_0 can be a parent of n_1 in another time-step, and there can be no path between them in yet another time-step.

In addition to the nodal values, an array of root indices is stored in the TDAG in the same manner as parent and child links are stored in the nodes (each value holds a time range tag). The TDAG then supports queries of the form *(time-step,error-tol)*, and returns an approximated mesh for that time satisfying that error tolerance. This is done using basically the same algorithm for top down construction of a mesh from a static multi-resolution representation. Starting from all live roots for this specific time, the graph is traversed top down and at each node the error tolerance is checked. The only difference is that the algorithm must use only live links in the nodes to children or parents for traversal. Other attributes and position information used for example, for rendering the object, are accessed as a function of time, allowing simple modification of the shape and appearance of the mesh through time.

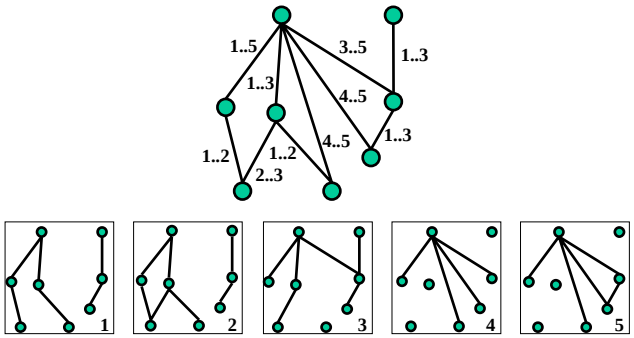


Figure 5: A TDAG with five time-steps, and the DAGs encoded in it.

Since for each time-step an approximating mesh is constructed according to some tolerance, there is no need to explicitly store any mesh connectivity or topological changes in the TDAG. These changes will be encoded implicitly in the graph dependencies while constructing the TDAG.

4 On-line T-DAG construction

The first observation concerning the definition of the TDAG is that any sequence of multi-resolution DAGs, which preserve the previously mentioned mapping restrictions of vertices, can be encoded in a TDAG. Let $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_s$ be a dynamic sequence of meshes, and $\mathcal{MR}_0, \mathcal{MR}_1, \ldots, \mathcal{MR}_s$ be multi-resolution DAGs created for them. If these meshes preserve the mapping restriction, then we can define $\mathcal{P} = \{v | v \in \mathcal{M}_i \text{for some } i\}$, $\mathcal{P}$ is the union of all vertices in all time-steps. We define a node in our TDAG for each such vertex, and encode all $\mathcal{MR}_i$ meshes using these nodes. Obviously, this scheme is practically equivalent to the worst case of building and storing each $\mathcal{MR}_i$ separately. This scheme favors optimizing each specific time-step in terms of traversal time, if a certain error is requested, or the quality of the mesh, if time is the restriction.

The opposite extreme is to use just a single DAG for all time-steps by encoding only the different decimation error in different time-steps. This scheme can actually be beneficial in some cases (See for example Figure 8). However, because a single DAG will not have an optimal structure for all time-steps, this will generally force the traversal to reach down to lower levels of the graph in order to satisfy a given error tolerance. This means sacrificing traversal time or quality to gain lower storage space (only a single DAG).

The key to constructing a good TDAG both in terms of storage-space and traversal, would be to try and merge more efficiently the distinct DAGs of each time-step into a single TDAG ($\mathcal{TD}$):

$$\mathcal{TD} = \mathcal{MR}_0 + \mathcal{MR}_1 + \cdots + \mathcal{MR}_s$$

It would be even more beneficiary, and possibly more feasible if it could be done incrementally:

$$\mathcal{TD}_{i+1} = \mathcal{TD}_i + \mathcal{MR}_{i+1} \text{ , where } \mathcal{TD}_0 = \mathcal{MR}_0$$

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

The algorithm uses the history of the previous time-step decimation order. A decimation operation from the history is extracted using some policy (**getElement**()), and its cost is compared to the cost of applying the same decimation on the current mesh. If the difference is too large (**largeDiff**()= true), this decimation is skipped, otherwise it is performed and recorded for the next time-step. Due to changes in topology or connectivity, some decimations from the previous time-step cannot be found in the current queue, and so they are skipped. Other decimations are possible only on the current mesh, and so the algorithm performs them at the end of each level. An outline of the algorithm is given in Figure 6.

The function (**getElement()**) defines the policy in which the previous time-step decimation is examined. According to our tests the best policy was to examine the decimations of all levels up-to and including the current level of decimation. (**largeDiff**($previouCost, currentCost$)) was usually set to be:

$$previousCost - currentCost > factor * currentCost$$

with factor = 1/10.

Figure 7 illustrates a comparison between decimations of consecutive time-steps of the balls in Figure 10. As can be seen, most of the decimations are carried across the timesteps, and therefore, without sacrificing traversal optimization too much, we can get large overlaps between DAGs of different time-steps and gain in storage.

The construction algorithm augments the TDAG incrementally. This means that the algorithm can be used online, and the TDAG of previous time-steps can be used for display and interaction, while the next time-step is being introduced. This fact is important specifically when the time span of the dynamic model is long or being generated on a remote server. The user does not need to wait until the whole processing is done (or even the whole process of creating the dynamic model is done), but rather he can view intermediate results as soon as they are included.

5 results

In this section we examine several examples for the use of a TDAG. We show the flexibility of the model and the ability of the construction algorithm to encode time-dependent information with different restrictions.

In the first example (see Figure 8), we look at a simulation of two sub-atom particles colliding over a 2D mesh. This simulation tracks several attributes changing over time (e.g. 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 vertex removal, we preserve at each step a Delaunay triangulation (this scheme was presented in [1] for static terrain meshes). On average, the Delaunay triangulation gave good results for all different attributes. The real advantage of using such a scheme is the fact that a single DAG is used for all time-steps and all

```
DecimateConform(M, TD, tol, time, Hin, Hout)
{
M is the initial fine resolution mesh
TD is the TDAG of decimation operation
tol is some external predefined tolerance
time is the current time-step
Hin is the previous order of decimation
Hout stores the current order of decimation
Q is a priority queue of decimation elements

swap Hin and 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
         largeDiff(e'→cost(), e→cost())
         continue
      remove e' from Q
      ApplyDecimation(e, M, TG, 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)
   }
   raise 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 6: An outline of the algorithm for creating a multi-resolution TDAG model. Two external functions govern the degree of conformity between consecutive time-steps: **getElement** from the history storage and **largeDiff** to check the difference in the cost of the decimation. 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 swapping the inHistory and outHistory

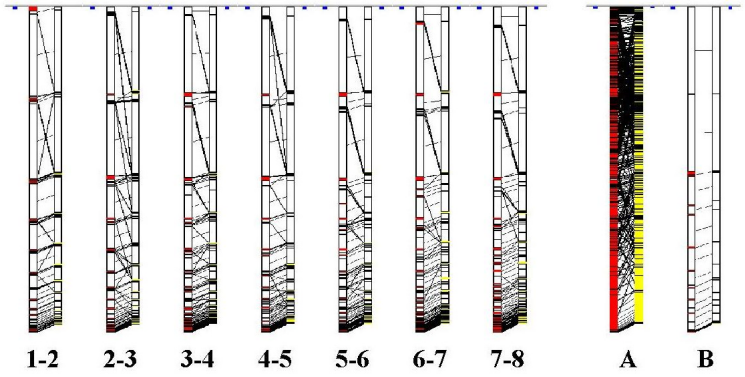


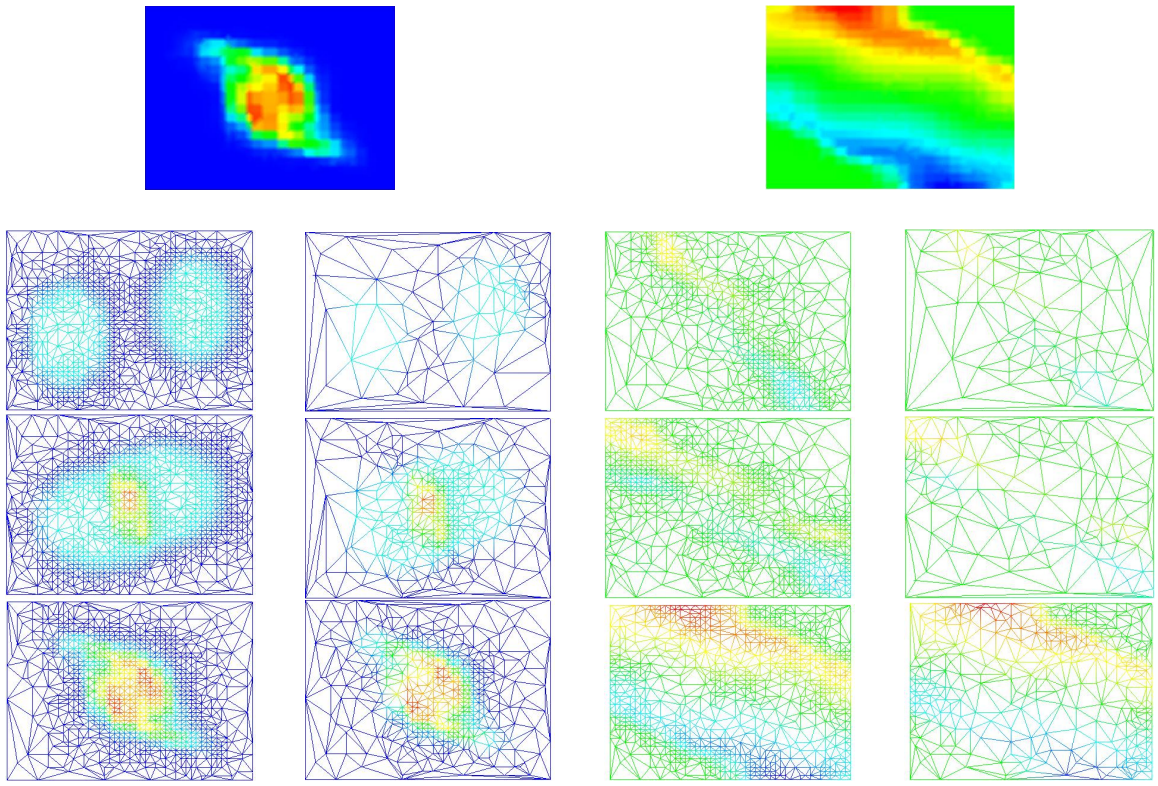
Figure 7: Comparison between decimation order of consecutive time-steps using the TDAG construction algorithm. The decimations are laid out from top to bottom, white spaces signify the same decimation, but links across levels can show that the decimation order is not always preserved. Also, due to connectivity changes there are some decimations that occur only at the left (red) or right (yellow). For comparison, the leftmost orders were created by decimating each time-step independently (A) or by fully conforming to the previous time-step order (B).

attributes. The different triangulations extracted over time are a result of the differences in the numerical information stored in the TDAG (the errors imposed by the decimation).

The price for the efficient storage in the previous example is paid at traversal time. Nodes at lower levels in the DAG need to be visited in order to satisfy a certain error tolerance. In fact, when we move to 3D surfaces, additional tests are necessary beside checking the approximation error when traversing 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. [19]. These types of tests could involve heavy computation, being too time consuming to be practical at traversal time. However, if these tests are carried out during the construction stage of the TDAG, verifying that all cuts in all time-steps satisfy the tests, the traversal time is greatly reduced.

In the second example we encode a sequence of dynamic changing meshes created by two waves colliding (using 7200 faces). We use half-edge contraction as the decimation primitive and the quadratic error metric for tracking decimation cost [6]. During decimation, we penalize the cost of contractions that cause a change in triangle orientation, and omit such checks during traversal. The TDAG constructed then supports faster adaptive multi-resolution viewing for all time-steps (Figure 9).

In cases where the connectivity or topology of the dynamic mesh changes, the differences of the symbolic information through time must be encoded. The last example (Figure 10) shows a ball splitting (or two balls merging) 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 TDAG model these changes were encoded implicitly and seamlessly by the construction algorithm.



(a) (b) (c) (d)

Figure 8: A simulation of collision between two sub-atomic particles encoded in a TDAG. The simulation tracks the interaction of several different variables over 50 time-steps (shown at the enclosed video). The TDAG was built using random vertex removal preserving Delaunay triangulations. This means the symbolic information for all time-steps is the same, but the numerical information is different. The TDAG extracts different triangulations due to the difference in errors over time (top to bottom), difference in tolerance (1% at (a) and (c) and 10% at (b) and (d) sub-columns) and for different variables ((a),(b) show density and (c),(d) show electrostatic scalar field potential).

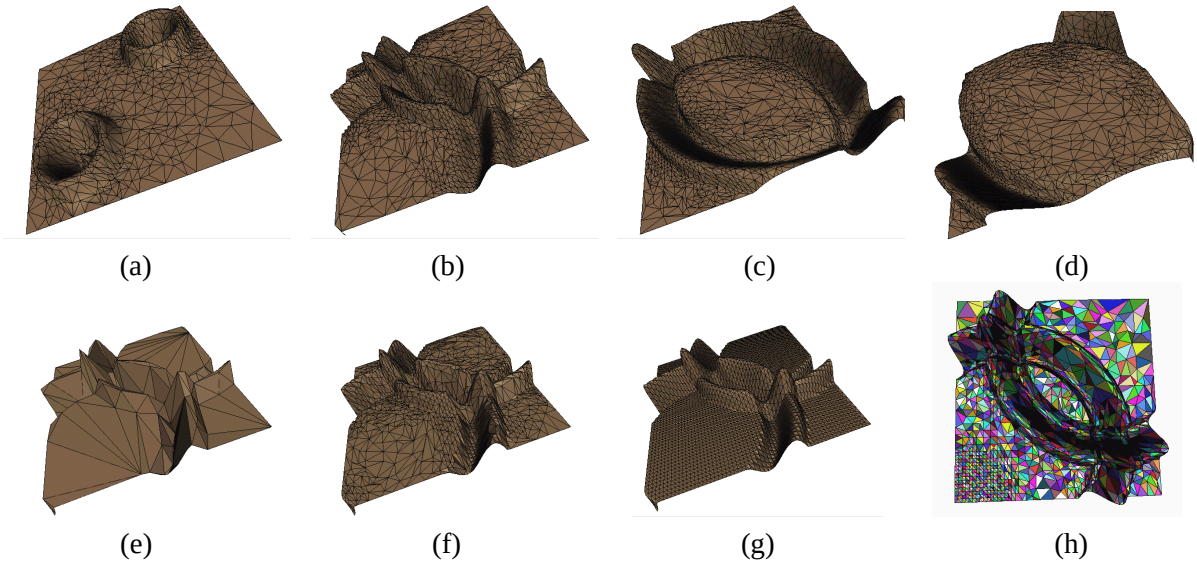


Figure 9: Mesh created by two moving wave fronts. The top row (a) to (d) show several time-steps (from the sequence shown at the enclosed video), the bottom row (e) to (g) show several resolutions of one time-step, and (h) an adaptive triangulation extracted with a viewing focus point at the lower left corner (hence, higher resolution at the corner).

6 Conclusions and Future Directions

Dynamic environment including deformable models are becoming more common as the 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 TDAG, 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. The TDAG 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 TDAG could be large, if one concentrates on a certain 'window' of time-steps, the live information is much smaller. This fact could be used, for instance, 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 TDAG, temporal coherency could be exploited for compression.

Another possibility is to use the TDAG to apply 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.

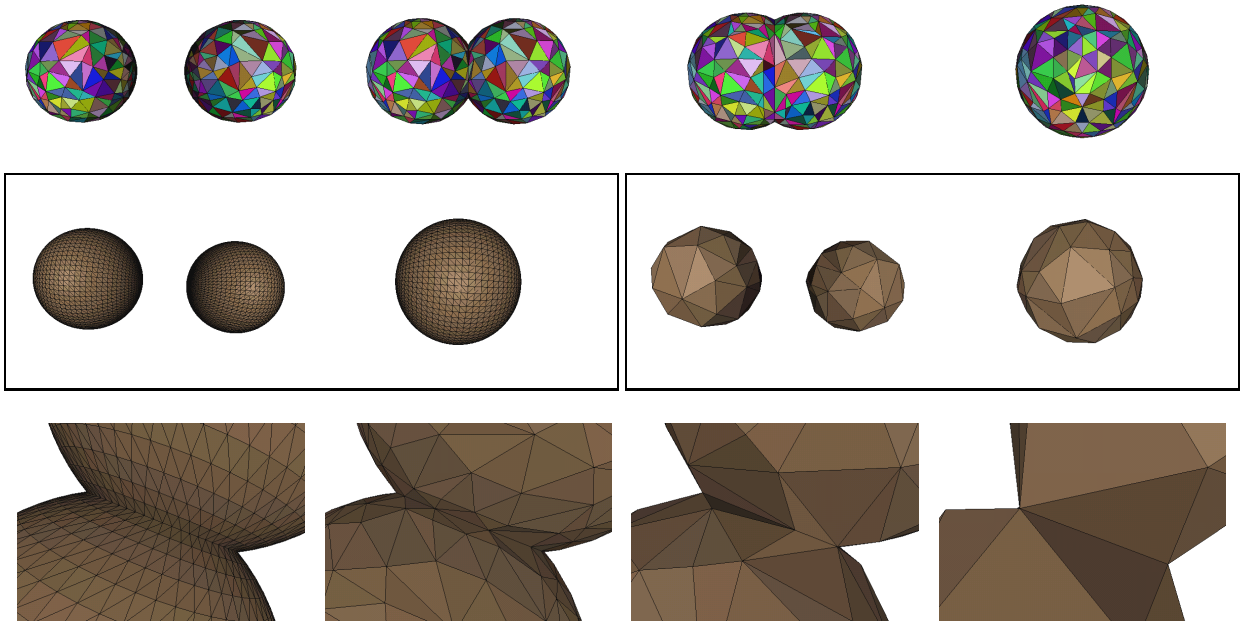


Figure 10: A merging or splitting ball mesh sequence. The top row shows several time-steps (shown at the enclosed video). The middle row shows the starting and ending meshes which have different topology and connectivity at both high and low resolution. The bottom row shows detail of the balls intersection curve at various resolutions. Note that this curve is not a boundary curve but rather a set of internal edges in the mesh.

References

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

[2] 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*, pages 173–180, 1995.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.