2011-07-28

Fast Relabeling of Deformable Delaunay Tetrahedral Meshes Using a Compact Uniform Grid

David C. Frogley
Brigham Young University - Provo

Fast Relabeling of Deformable Delaunay Tetrahedral Meshes
Using a Compact Uniform Grid

David C. Frogley

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements of the degree of

Master of Science

Michael Jones, Chair
Eric Mercer
Jay McCarthy

Department of Computer Science

Brigham Young University

December 2011

ABSTRACT

Fast Relabeling of Deformable Delaunay Tetrahedral Meshes
Using a Compact Uniform Grid

David C. Frogley
Department of Computer Science, BYU
Master of Science

We address the problem of fast relabeling of deformable Delaunay tetrahedral meshes using a compact uniform grid, with CPU parallelization through OpenMP. This problem is important in visualizing the simulation of deformable objects and arises in scientific visualization, games, computer vision, and motion picture production. Many existing software tools and APIs have the ability to manipulate 3D virtual objects. Prior mesh-based representations either allow topology changes or are fast. We aim for both. Specifically, we improve the efficiency of the relabeling step in the Delaunay deformable mesh invented by Pons and Boissonnat and improved by Tychonievich and Jones. The relabeling step assigns material types to deformed meshes and accounts for 70% of the computation time of Tychonievich and Jones' algorithm.

We have designed a deformable mesh algorithm using a Delaunay triangulation and a compact uniform grid with CPU parallelization to obtain greater speed than other methods that support topology changes. On average, over all our experiments and with various 3D objects, the serial implementation of the relabeling step of our work reports a speedup of 2.145 over the previous fastest method, including one outlier whose speedup was 3.934. When running in parallel on 4 cores, on average the relabeling step of our work achieves a speedup of 3.979, with an outlier at 7.63. The average speedup of our parallel relabeling step over our own serial relabeling step is 1.841

Simulation results show that the resulting mesh supports topology changes.

Keywords: object representations, geometric algorithms, modeling packages

ACKNOWLEDGMENTS

I express thanks to all those who made this work possible, through various means of support, help, and encouragement. Specifically, thank you to Dr. Michael Jones, my advisor, for the idea for this project, for always being a positive motivator, and for spending countless hours selflessly giving feedback, critiques, ideas, and counsel.

Thank you to Anthony Selino for being my go-to guy for sanity checks and mathematical correctness. Even while tirelessly working on his own thesis, he still took time to listen to my varied ideas (ridiculous or otherwise) and to double-check my equations and algorithms that spanned two whole whiteboards.

Finally, thank you to Emily Arrington Frogley, Kathryn Frogley, Danielle Frogley, Christine Frogley, Jacob Henry Frogley, and Joseph Edward Frogley, my wife and children, for being willing to sacrifice time with "Daddy" so I could get through school as quickly as possible. Especially, thank you for your prayers, offered multiple times daily, on my behalf. No other hands have pushed and pulled more effectively and lovingly as yours and God's.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1 – INTRODUCTION

1.1 – Background

This thesis addresses fast relabeling of deformable Delaunay tetrahedral meshes using a compact uniform grid. This work falls into the modeling subfield of computer graphics. Modeling refers to representing three-dimensional objects, and possibly changes to those objects, in a virtual environment.

Several methods for representing objects exist in computer graphics, and we focus on meshes. A triangle mesh, the most common mesh, is a collection of points in 3D space grouped into triangles that approximate the closed surface of the object. Tetrahedral meshes, which we use for our work, are similar but use tetrahedrons whose outer-most faces approximate the closed surface of the object. Outer-most faces are faces that touch tetrahedrons labeled with different materials or that border unlabeled—empty—space. The surface normal of an object is parallel to the face normal of an outer-most face.

Meshes offer a good balance between expressiveness and efficiency. Here, expressiveness refers to the ability of a deformable model to represent splitting/merging and creating/filling holes, also known as topology changes. Meshes can also represent concavities such as caves/grottos, arches, and overhangs.

Modeling with meshes that can split and merge is important in film, scientific simulation, and computer games. Currently, many solutions use representations other than meshes, sacrificing

1

expressiveness for speed, or sacrificing efficiency or automation for expressiveness. Our work provides a middle ground that is expressive and uses meshes while requiring less time than the previous fastest implementation of a mesh that splits and merges.

Current object modeling solutions include representations such as height maps, voxel grids, implicit and curved surfaces, and triangle meshes. These representations all have strengths that make them ideal for some scenarios and weaknesses that make them unsuitable for others. Height maps are fast but inexpressive. Voxel grids are intuitive and expressive but scale poorly since they scale with an object's volume instead of its surface area, and surface extraction can be expensive. Implicit surfaces are easily split and merged but require an expensive surface extraction step before rendering. Curved surfaces are difficult to split and merge but are easily rendered. Triangle meshes are expressive and scale well, but the expressiveness is difficult to achieve automatically and is therefore often done manually, which is tedious since 3D objects can be composed of thousands or even millions of triangles.

Two existing solutions attempt to make automation of topology changes in triangle meshes easier and faster. Both are based on a three-step process: 1) the motion step, in which vertices are moved; 2) the remeshing step, in which a new mesh is generated for the moved vertices; and 3) the relabeling step, in which the triangles or tetrahedrons of the new mesh—called "cells"—are relabeled. Relabeling of cells in the new mesh is the computational bottleneck of this process— in Tychonievich and Jones' [2010] work the relabeling step took up 70% of the total computation time.

2

Pons and Boissonnat [2007] use a Delaunay mesh, described in chapter 2, in conjunction with a segtree to obtain a mesh that splits and merges with automatic remeshing and relabeling, but their solution is slow and memory intensive. Tychonievich and Jones [2010] improve on Pons and Boissonnat's work by using a Delaunay mesh with a Delaunay hierarchy [Devillers, 2002], greatly improving performance and decreasing memory requirements. However, their solution is still slow because it depends on the Delaunay hierarchy, though it is faster than the segtree.

Our work improves on Tychonievich and Jones' work by using a compact uniform grid [Lagae and Dutré, 2008] instead of the Delaunay hierarchy. The compact uniform grid can be built quickly and has almost constant-time lookup for individual points. Furthermore, we parallelize portions of our algorithm using OpenMP [Barney], which further enhances performance on multi-core machines.

1.2 – Thesis Statement
Relabeling tetrahedral meshes using a compact uniform grid is more efficient in time and space than existing approaches using a Delaunay hierarchy or segtree; furthermore, the compact uniform grid creates opportunity to exploit concurrency to improve time performance further.

CHAPTER 2 – RELATED WORK

This work is related to prior work both in industry and academia. We give a brief description of software tools related to deformable modeling as well as an in-depth discussion of relevant prior research in deformable modeling.

Prior work in deformable modeling can be grouped into Eulerian and Lagrangian methods. In Eulerian methods, such as voxel grids, properties are assigned to regions of space. As objects move and deform, the labeling of those regions changes to reflect changes in the contents of the regions, but the regions themselves remain constant. In Lagrangian methods, such as triangle meshes, properties are assigned to objects that move through space. As objects move and deform, the objects carry their properties with them. Each approach has advantages and disadvantages as will be discussed below.

2.1 – Height Maps

Some software tools—World Machine's World Machine 2 [2008], Bundysoft's L3DT [2010], Johannes Rosenberg's GeoControl 2 [2008], and Quad Software's Grome 2 [2009]—use height maps for their objects and therefore cannot support concavities such as overhangs, grottos, and arches, or topology changes, and the object must be converted to a triangle mesh for rendering. Height maps are an Eulerian representation. Our implementation uses a tetrahedral mesh that supports concavities and topology changes. The outside faces of the tetrahedrons are the triangle mesh that is rendered, and therefore no surface extraction is necessary.

4

A height map is a 3D object generated from a 2D texture image in which each pixel value represents the height of the object at that point. Height maps are much more compact representations of an object than triangle meshes. However, height maps are incapable of representing concavities such as overhangs because only a single elevation can be stored per location, thus making them more useful for objects at a distance instead of for close-up detail. Height maps are used in fast erosion simulations [Beneš et al., 2006]; however, such erosion simulations also suffer from a lack of concavities as well as a lack of topology changes, making them less realistic.

Height maps can be rendered with ray tracing algorithms, in which the map is converted to columns of voxels, or by converting the map to a triangle mesh. Figure 1 gives an example of rendering a height map: on the left is the 2D texture that stores the object's height field, in the middle is the map converted to a triangle mesh, and on the right is the final rendering using filled triangles.
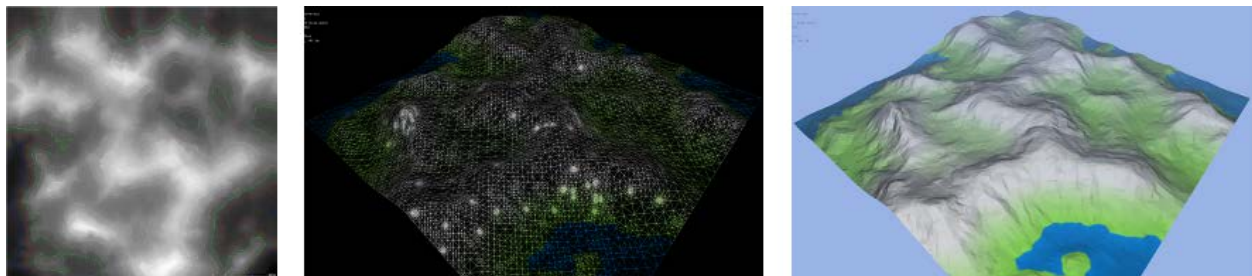
Figure 1 - Left: a height map as a texture, with white being high and black being low; Center: a height map converted to a triangle mesh for rendering; Right: a height map rendered as a surface with filled triangles (created with L3DT).

2.2 – Voxel Grids

A voxel grid, which is also an Eulerian representation, is a 3D generalization of a pixel that represents the internal volume of an object. Voxels are used often in the visualization and

analysis of medical and scientific data, such as 3D models constructed from a series of medical scans.

Voxel grids are used in representations of terrain, such as [Jones et al., 2010], [Ito et al., 2003], and [Beneš et al., 2006], because of their ability to represent concavities such as overhangs, caves, and arches. Figure 2 shows a house constructed of voxels. The interior of the house also contains voxels, even though they aren't visible.
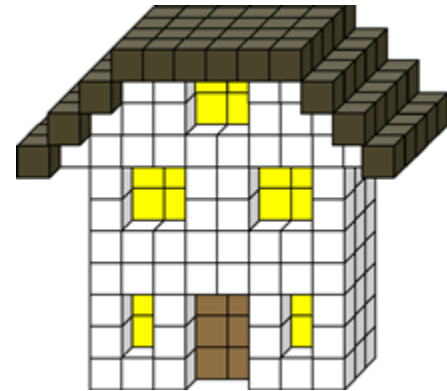
Figure 2 - A house constructed of voxels. Note that the interior of the house also contains voxels, even though they aren't visible.

Voxel grids trivially support topology changes in dynamic scenes because each voxel is independent of its neighbors, but surface extraction for rendering is time-consuming, and the representation scales with the volume rather than with the surface of an object. While some implementations approximate a voxel grid using a data structure such as an octree, storing only enough outer layers of voxels to compute curvature, voxel grids are still less efficient than surface-based representations. Lorensen and Cline [1987] developed an algorithm they call Marching Cubes, which is *the* way to extract a surface for several object representations, including voxel grids. However, the algorithm is slow. Since our work uses a tetrahedral mesh to represent only the surface of an object, the surface extraction is quite simple, and the algorithm computation time is linear in the number of tetrahedrons.

Jones et al. [2010] define weathering algorithms on voxel grids in which objects are deformed based on local mean curvature. To increase efficiency, the implementation by Jones et al. only

6

stores the outer-most two to three voxels for an object, and even then the algorithm is too slow. This is further evidence that an Eulerian representation such as a voxel grid is not ideal for the types of problems we address. Topology changes in voxel grids are trivial because voxels are not explicitly connected in space. However, the grid must also be converted to a triangle mesh for rendering. Our implementation also supports topology changing deformations but is more efficient because it scales with the surface area rather than the volume and does not require a computationally intensive surface extraction step.

2.3 – Implicit and Curved Surfaces

Lagrangian representations of curved surfaces, such as the one seen in Figure 3, using splines support efficient manipulation operations but do not readily support automated topology changes. Sederberg et al. [2003] developed T-splines—an improvement on the more-common B-splines used for NURBS surfaces— which were later refined by Sederberg et al. [2004].
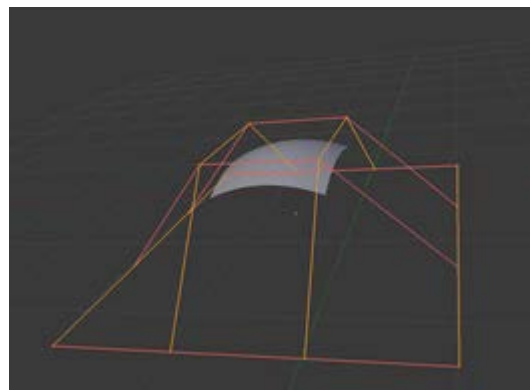
Figure 3 - NURBS surface with its control points (created in Blender 2).

NURBS surfaces are implicit surfaces commonly used by software tools such as Autodesk's Maya [2011], E-on Software's Vue [2010], and Blender Foundation's Blender 2 [2009] for complex shapes and objects. Since the surface is implicit, however—meaning that its shape is governed by control points instead of by actual points on the surface of an object—the surface must be extracted to a triangle mesh. This can be done more efficiently than for voxels, but deforming the object is more difficult and less precise.

7

2.4 – Triangle Meshes

A triangle mesh is a set of triangles connected by their common edges and vertices. Triangle meshes are often used in 3D modeling software tools and in graphics APIs like OpenGL [Open Graphics Library].

A triangle mesh is an approximation of a closed surface with no explicit representation of the interior of an object, and therefore a triangle mesh scales more efficiently than a voxel grid. Triangle meshes are commonly used because hardware devices are optimized to operate efficiently on triangles
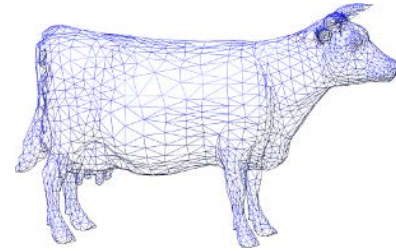
Figure 4 - A triangle mesh approximating the surface of a cow-shaped object.

grouped into meshes. In our work, we focus on triangle meshes because they are easily rendered and are expressive. However, the problem with meshes is that allowing topology changes to the mesh requires extra work to maintain a non-degenerate mesh. Figure 4 shows an object—a cow—whose surface is approximated using a triangle mesh.

Software tools that do deal directly with triangle meshes for their objects—E-on Software's Vue [2010], Autodesk's Mudbox [2011] and Maya [2011], and Blender Foundation's Blender 2 [2009]—do not support automatic remeshing of objects when their vertices are moved, so triangles or tetrahedrons can become inverted, degenerate, or overlapping. For topology changes, the vertices must be remeshed completely by hand. Our implementation supports automatic deformation, remeshing, and relabeling for the entire object.

8

2.5 – Delaunay Deformable Meshes

Following are a few relevant terms from computational geometry related specifically to Delaunay meshes. These terms are used in the discussion of Delaunay meshes that follows the definitions.

Circumsphere: A circumsphere is the sphere that exactly circumscribes a triangle in 2D or a tetrahedron in 3D space. In other words, the circumsphere intersects the tetrahedron at all four vertices and nowhere else. All triangles and tetrahedrons have a circumsphere.

Circumcenter: The center point of a circumsphere is called the circumcenter. The circumcenter is equidistant from all the vertices of a triangle or tetrahedron; therefore, in 2D the circumcenter is located at the intersection of the perpendicular bisectors of the three edges, as seen in Figure 5.
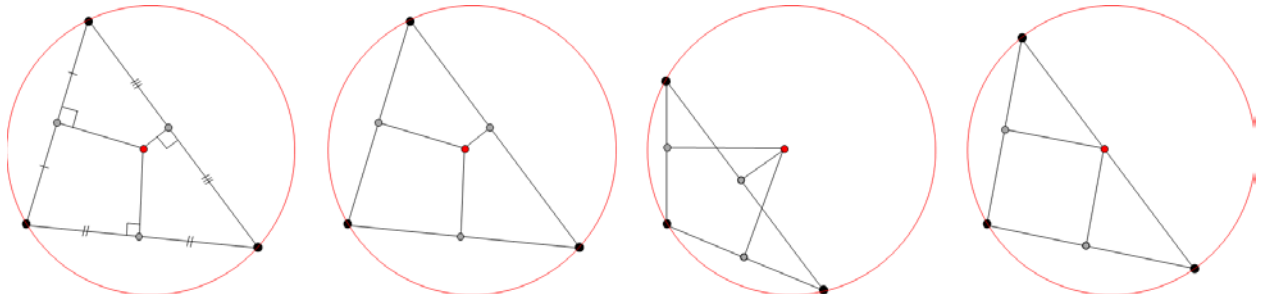
Figure 5 - Left: a circumcenter is computed by finding the intersection of the perpendicular bisectors of each edge; Left-center: circumcenter of an acute triangle; Right-center: circumcenter of an obtuse triangle; Right: circumcenter of a right triangle.

Delaunay Property: In 3D, a Delaunay mesh is a kind of triangle mesh that uses tetrahedrons grouped such that the circumsphere of any tetrahedron—called a "cell"—in the mesh does not contain a vertex of any other cell in the mesh, with the single exception of two cells sharing the

Figure 6 - A Delaunay mesh with the triangles' circumcircles.

9

same circumsphere. Two cells share the same circumsphere if all the vertices of both cells lie on the sphere itself, but none of the vertices of either cell lie inside the sphere. Figure 6 shows a simple Delaunay mesh in 2D with the circumcircles for each "cell." Notice that no vertex in the mesh lies inside any of the circumcircles.

Pons and Boissonnat [2007] introduce the concept of topology-adaptive meshes using restricted Delaunay triangulations to simplify deformations of Lagrangian surfaces. In this algorithm, the mesh representing the object is deformed, the vertices are remeshed, and the material queries are performed in the deformed mesh in order to label the remeshed triangles or tetrahedrons—again, called "cells." In 3D, a Delaunay cell is comprised of a tetrahedron with its associated material label. If an object in simulation frame n deforms to a new arrangement in frame $n + 1$, then the critical operation in Pons and Boissonnat's algorithm is querying a representation of the deformed object in order to relabel cells in frame $n + 1$. To accelerate these queries, Pons and Boissonnat use a segtree when querying the deformed mesh directly, making the lookup inefficient in terms of speed and memory usage.

Figure 7 outlines the steps of Pons and Boissonnat's algorithm. On the left is the original Delaunay mesh. Next, the vertices of the mesh have been moved so the object is deformed. However, since the vertices maintain their connectivity, some of the triangles in the mesh—the triangles outlined in yellow—are inverted and/or overlapping. A new Delaunay mesh is then created using the moved vertices, and each triangle's circumcenter is computed, marked with blue Xs in the figure. Note that the new mesh is unlabeled—each triangle is white. Finally, each new triangle's circumcenter is used as a query point by finding the triangle in the old deformed

10

mesh that contains it, and each new triangle receives the same material label as the old triangle containing its circumcenter.
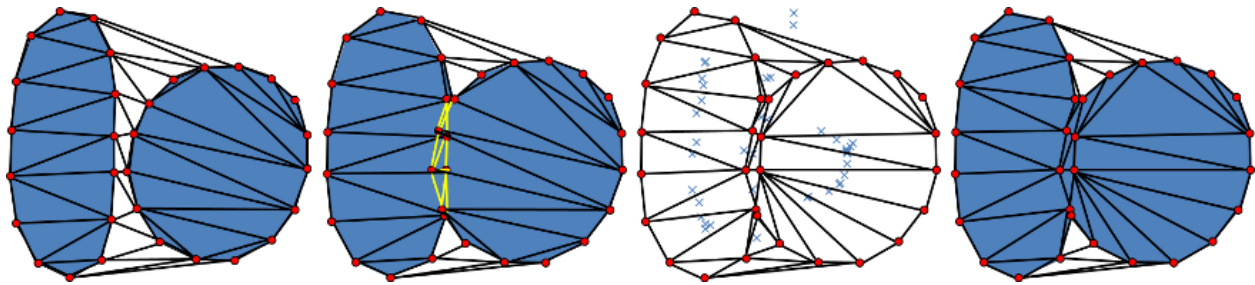

Figure 7 - Pons and Boissonnat's remeshing and relabeling steps - Left: original Delaunay mesh; Left-Center: deformed mesh with inverted and/or overlapping triangles; Right-Center: new Delaunay mesh with its circumcenters; Right: relabeled mesh.

The material lookup step—relabeling—in Pons and Boissonnat's algorithm can be improved using the Delaunay hierarchy introduced by Devillers [2002]. The Delaunay hierarchy is a data structure that stores the complete Delaunay triangulation as its root node. Each interior node in the hierarchy has exactly one child node, and each child node stores a Delaunay triangulation of a random subset of its parent's vertices. The storage complexity of the hierarchy is $n \log(n)$, as is the creation time complexity. While $n \log(n)$ is typically good, since each node in the hierarchy must create a complete Delaunay mesh, which is a computationally intensive and complex operation, the creation is slow compared to our implementation. The time complexity of traversing the grid for point queries is roughly $\log(n)$, which is also typically good. However, the traversal also involves many complex computations, resulting in a lookup that is much slower than our implementation for some values of n.

Tychonievich and Jones [2010] improve on the work of Pons and Boissonnat [2007] by using the Delaunay hierarchy [Devillers, 2002] described above. Instead of querying the deformed mesh for material labels, they compute the average movement of each vertex of a cell from the old

mesh to the new one and then apply the inverse of that average movement to the circumcenter of the cell, finally querying the old Delaunay mesh—not the deformed mesh—for the material associated with the "reverse-advected" circumcenter. Since Tychonievich and Jones use the Delaunay hierarchy, the relabeling portion of their algorithm is significantly faster and requires much less memory than Pons and Boissonnat's work. However, Tychonievich and Jones' relabeling step still consumes 70% of their computation time, making it an ideal candidate for optimization, including parallelization.

Other approaches to remeshing or repartitioning triangle meshes have been investigated. However, these methods are not suitable for this application because they do not easily support querying [Fürnstahl, 2005], rely on static objects [Barreira and Penedo, Alliez] or are defined only for simple shapes [VanderZee et al., 2008].

2.6 – Acceleration Structures

The fundamental step of deformable Delaunay mesh algorithms is called the "relabeling step," which involves finding the unique tetrahedron in the previous mesh that contains a given point. To avoid comparing every lookup point against every tetrahedron, the mesh is subdivided using one of a variety of data structures called acceleration structures. One example, the Delaunay hierarchy, was described in the previous section, and this section discusses a few other possible structures, including kD trees such as binary trees, quadtrees, and octrees, as well as the compact uniform grid. This is not an exhaustive treatment but focuses on work most closely related to or used by this thesis.

A kD tree is a spatial partitioning data structure that groups nearby objects—points, triangles, or tetrahedrons in our case—into subdivisions of k-dimensional space. The subdivisions are typically axis aligned. One specialized kD tree implementation is the binary tree, which is used both in 2D and in 3D. Another implementation is the quadtree, which subdivides space into four equally sized subspaces. The 3D equivalent of the quadtree is the octree. Each branch of a kD tree may go to an arbitrary recursive depth, independent of the other branches, based on the number of objects in the node, the node's depth in the tree, or some other criteria. kD trees are useful for applications such as searches involving multidimensional search keys like a nearest neighbor search. They are also commonly used as acceleration structures in ray/path tracing to minimize the number of ray-object intersection tests.

Figure 8 shows examples of these three types of kD tree. The left and center images show a binary tree and quadtree, respectively, for the same set of points. The right image shows a generic octree.
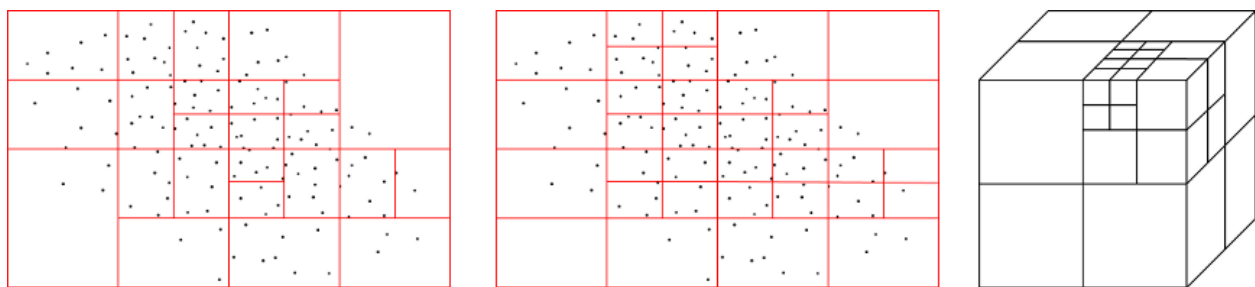
Figure 8 - Left: binary tree in 2D; Center: quadtree over same data in 2D; Right: generic octree in 3D.

Binary trees are typically built by starting with a root node that contains all the points/objects in a set. The space is then split across its largest dimension at the midpoint of that dimension, or the mean or median point in that dimension of the points within the node, forming two subspaces. Each point or object in the node is then assigned to whichever subspace contains or intersects it.

For points, each point is associated with exactly one subspace, but for objects the object may cross the split plane and therefore be associated with both subspaces. Finally, each subspace is recursively split to an arbitrary depth or until a maximum number of associated points/objects is reached. Construction time complexity is $n \log(n)$ in the number of points/objects in the space, as is clearly seen in Figure 9, which depicts the hierarchy representations for the trees in Figure 8. Quadtrees and octrees are treated similarly.
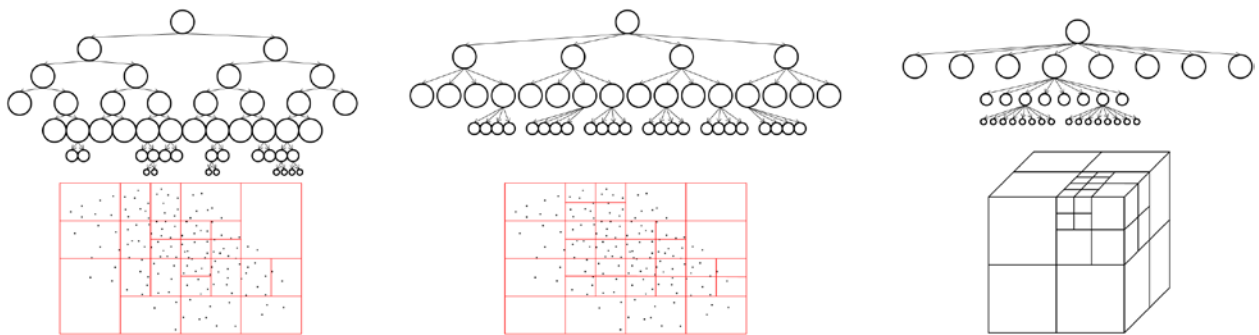
Figure 9 - The node hierarchy representations for the trees in Figure 8, showing clearly that the storage complexity of each tree is $n\log(n)$.

Since traversal time complexity for a kD tree is $n \log(n)$, performance of our algorithm using a kD tree (both for a binary tree as well as for an octree) was actually worse, even in parallel, than the Delaunay hierarchy, which has a traversal time complexity of $\log(n)$. That led us to use the compact uniform grid.

A uniform grid is a spatial partitioning data structure that divides a 3D volume into equally sized subspaces, allowing for fast creation time, since it uses no recursion, and fast lookup time for individual points.

Figure 10 - A uniform grid over a triangle mesh.

14

Uniform grids were used in one of the first systems for interactive ray tracing. Most ray tracing work focuses on fast traversal of the data structure and parallelizing the build process. Uniform grids 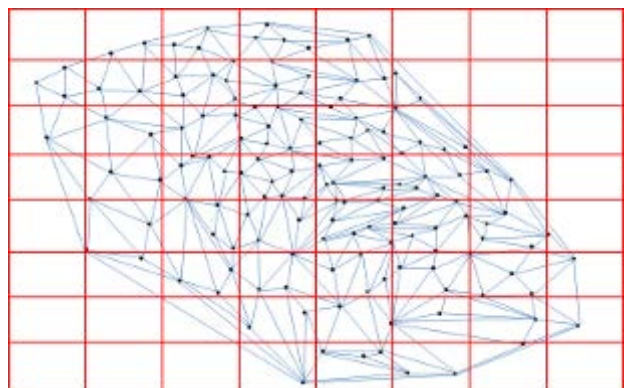are ideal for fast building because the build time is linear in the number of objects—points, triangles, or tetrahedrons—in the space; therefore, uniform grids are commonly used for dynamic and interactive ray tracing, which involves rendering dynamic and interactive scenes instead of static ones. Figure 10 shows a generic uniform grid over a triangle mesh. Note that, on average, the number of triangles overlapping any given grid cell is a small subset of the total mesh. Also, the number of grid cells overlapped by any given triangle is also small.

Lagae and Dutré [2008] developed a uniform grid algorithm that is memory efficient, builds quickly, and provides fast point lookup that is essentially a hash function for points in 3D space. The compact uniform grid was originally intended for dynamic ray tracing, but our work adapts the compact uniform grid for use with tetrahedrons instead of triangles and adds individual point lookups instead of ray intersection algorithms.

While the concept of deforming a mesh is fairly straightforward, it involves some complex steps that require a more in-depth look, such as remeshing the new vertices, circumcenter computation, acceleration structure construction, and relabeling the new Delaunay mesh. For our Delaunay triangulation code, we used a computational geometry library called CGAL [2010], which has several built-in features that make the triangulation possible.

3.1 – Algorithm Overview

CGAL [2010] can create a Delaunay triangulation automatically simply by adding points to a triangulation object. Once the Delaunay triangulation is created, the Delaunay cells—the tetrahedrons—must be labeled as being either internal or external to the object, since the triangulation algorithm creates some tetrahedrons outside the surface of the object in order to create a convex hull. To determine the label for each cell, we compute each cell's circumcenter and look up the position of the circumcenter in a geometric description of the newly created object to determine if the cell is inside the object. If not, the cell is labeled as "air." Otherwise, it is given the label of the object containing it. For a scene containing multiple objects, each object has its own material label, but otherwise the algorithm behaves the same.

After the cells of the mesh have labels, each vertex then receives a label, too, marking it as a border vertex on the boundary between two material types or as a floater that is internal to an object. This vertex labeling is used for rendering the outer-most faces of the cells.

The mesh is deformed by transforming vertices to new positions using a user-defined function. This function takes points in a triangulation and returns a set of points. The cardinality of the set of points might not be preserved by the function. The transformed points are stored in a second Delaunay triangulation object, which is constructed incrementally as points are computed. The final step in the algorithm is to swap the two triangulation objects. After the points are transformed and remeshed, they must be relabeled.

Cells in the new mesh are not necessarily comprised of the same vertices as in the old mesh, so we must approximate the contents of each new cell based on the contents of the prior cells. Like Tychonievich and Jones, we use reverse advection of the circumcenters to compute query points. Reverse advection involves computing the circumcenter of each cell, computing the movement of each vertex of that cell from the old mesh to the new one, and applying the inverse of the average of that movement to the circumcenter. The reverse-advected circumcenters are the query points for the prior mesh.

For each query point, we find the Delaunay cell in the prior mesh that contains it. Next, we get the material label for that cell. If no cell contains the point, the material label is "air." Finally, the Delaunay cell in the new mesh associated with the query point is assigned the material label. Looking up points in the old mesh was the most time-consuming step in the algorithm in the previous fastest implementation by Tychonievich and Jones [2010]. The process of using a compact uniform grid to accelerate looking up query points is discussed next.

3.1.1 – The Compact Uniform Grid

This section closely follows the description given in [Lagae and Dutré, 2008]. We adapted Lagae and Dutré's compact uniform grid representation, which was originally created for ray and path tracing. The compact grid has minimal memory requirements when compared to dynamic array approaches because it stores exactly one index per grid cell and exactly one index per tetrahedron reference. The grid can also be built in linear time.

The number of grid cells M should be linear in the number of tetrahedrons N, or

$$M = \rho N, \tag{1}$$

where ρ is called the *grid density*. The number of grid cells M is the product of the resolution of the grid in each dimension, and according to Lagae and Dutré, cubically shaped grid cells work best. The resolution of the grid is given by

$$M_i = S_i \sqrt[3]{\frac{\rho N}{V}}, (i \in \{x, y, z\}), \tag{2}$$

where S_i is the size of the bounding box of the grid in dimension i, and V is the volume of the bounding box. The literature contains several recommendations for the value of ρ, and after experimentation we chose $\rho = 1$ because it resulted in the best runtime performance for our experiments.

18

As part of grid construction, we must determine which grid cells are overlapped by each Delaunay cell. Like Lagae and Dutré, we use the bounding box for each Delaunay cell in insertion calculations rather than performing more accurate intersection tests. This slightly increases the grid's memory footprint and slightly increases the number of inside/outside tests performed during querying, but for our tests the savings in time far outweighed the costs.

The compact representation of the uniform grid consists mainly of two static arrays L and C. L consists of the concatenation of all tetrahedron *lists*, and C stores for each grid *cell* the offset of the corresponding tetrahedron list in L. A grid cell's tetrahedron list is the list of tetrahedrons that intersect the grid cell. L is a 1D array, and C is a 3D array of size $M_x \times M_y \times M_z$ linearized in row major order into a 1D array of size M. C supports both 3D indexing of the form $C[z][y][x]$ and 1D indexing of the form $C[i]$.

The size of the tetrahedron list of the grid cell with 1D index i is given by $C[i+1] - C[i]$. Note that this is invalid for the last object, since $C[N]$ does not exist. Therefore, the array C is extended with one extra position.

To perform inside/outside checks for a given point against all tetrahedrons in a given grid cell i, we can simply iterate across all tetrahedrons indexed in array L between $L\big[C[i]\big]$ and $L\big[C[i+1]\big]$.

The space complexity of the compact uniform grid is linear in the number of tetrahedrons, which is better than a tree since a tree's space complexity is $n \log(n)$. Note that since the grid uses 32-bit integer indices instead of pointers, there is an additional savings for 64-bit operating systems.

3.1.2 – Building the Compact Uniform Grid

```
 1. for (i = 0; i < N; ++i) {
 2.   for each grid cell j intersected by tetrahedron i {
 3.     ++C[j];
 4.   }
 5. }
 6.
 7. for (i = 1; i <= M; ++i) {
 8.   C[i] += C[i-1];
 9. }
10.
11. for (i = N-1; i >= 0; --i) {
12.   for each grid cell j overlapped by tetrahedron i {
13.     L[--C[j]] = i;
14.   }
15. }
```

Figure 11 - Pseudo code for incrementing tetrahedron list size of overlapped grid cells, accumulating tetrahedron list sizes, and inserting tetrahedron indices into object lists. Since we chose $\rho = 1$, then $M = N$, and we can assume that, on average, each tetrahedron overlaps one cell.

First we compute the bounding box for the scene using the minimum and maximum vertices from the mesh. Next, using equation (2), we determine the grid resolution M_i in each dimension. We then compute the size of each grid cell's tetrahedron list, which size is stored in the cell array. $C[i]$ records the size of the tetrahedron list of the grid cell with 1D index i. This value is used to compute the offsets of each tetrahedron list. The joint size of all object lists is needed for allocating the tetrahedron lists array L.

To compute the tetrahedron list sizes, we first allocate the array C with size $M + 1$ and initialize each entry to 0. Then we iterate over all tetrahedrons in the mesh and increment the object list size of all overlapped grid cells using the algorithm in Figure 11 lines 1 through 5.
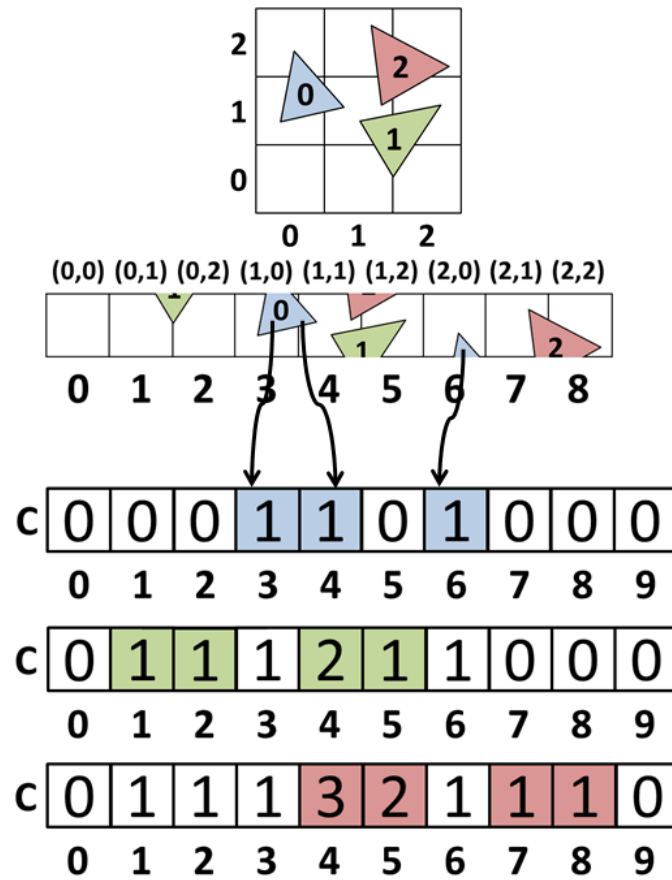
20

Figure 12 - A scene with 3 objects linearized in row-major order.

Figure 12 shows an example of a scene with three triangles, numbered 0, 1, and 2, along with the uniform grid for the scene. On top is the 2D representation of the grid, and just below it is the 1D linearization in row-major order of the scene.

Since the grid has nine cells, $M = 9$. We allocate array C to be size 10 to hold the one extra position at the end as mentioned before. The lower half of Figure 12 shows C as we first increment the indices of the cells overlapped by triangle 0, then by triangle 1, and finally by triangle 2.

To determine which grid cells are overlapped by a tetrahedron, we compute the minimum point P_{min} and maximum point P_{max} of the bounding box for each tetrahedron. We then compute the grid coordinates of the grid cells containing these two points:

$$Min_i = \left\lfloor \left(\frac{P_{min_i} - i_{min}}{M_i} \right) S_i \right\rfloor, i \in \{x, y, z\}, \tag{3}$$

$$Max_i = \left\lfloor \left(\frac{P_{max_i} - i_{max}}{M_i} \right) S_i \right\rfloor, i \in \{x, y, z\}, \tag{4}$$

Where Min is the grid cell containing the minimum point, Max is the grid cell containing the maximum point, S_i is the size of the bounding box of the grid in dimension i, i_{min} is the minimum extent of the bounding box in dimension i, and i_{max} is the maximum extent of the bounding box in dimension i. Once we have the minimum and maximum grid cells, we consider all grid cells in the range $[Min_i, Max_i]$ in each dimension to intersection the tetrahedron.

Once we have computed the size of each tetrahedron list, we compute the offsets of the tetrahedron lists by accumulating the sizes of the tetrahedron lists. Inserting tetrahedron indices into tetrahedron lists is not possible without keeping track of how many tetrahedron indices are already inserted during each iteration over the tetrahedrons. Therefore, instead of computing the offset of each grid cell's tetrahedron list, we compute the offset to the next grid cell's tetrahedron list. So $C[i]$ now stores the offset of the tetrahedron list with 1D index $i + 1$ ($C[i]$ points one past the end of the object list of the grid cell with 1D index i) using the algorithm in Figure 11 lines 7 through 9. At the end of this loop, the joint size of all object lists is stored in $C[N]$. Also note that

22

since these lines perform a very simple integer addition, and since $M = N$, the computation time for this loop is negligible when compared to the other two.
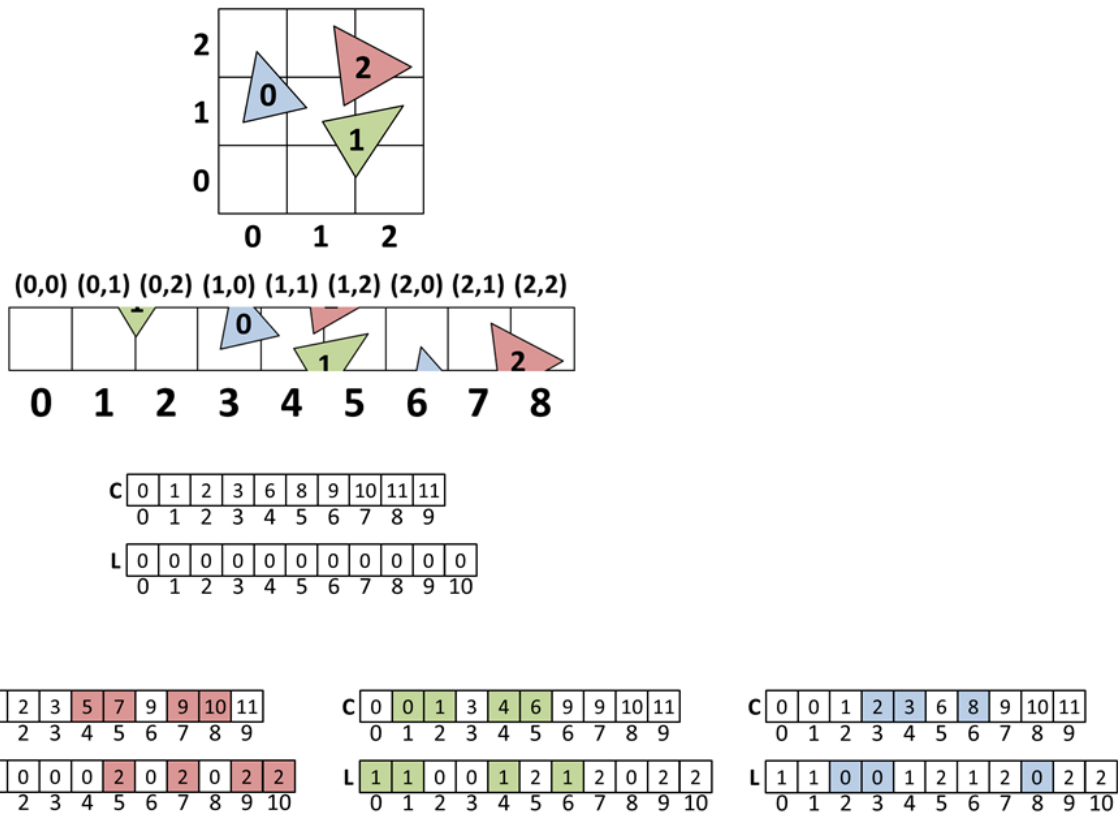

Figure 13 - Array C after accumulating tetrahedron list sizes, array L allocated with size 11, and arrays C and L as we iterate over the tetrahedrons again and decrement overlapped cells and insert indices.

Figure 13 continues the example from Figure 12, showing again the scene with three triangles. In the middle of the figure, we see arrays C and L. Array C holds the result of accumulating tetrahedron list sizes using Figure 11 lines 7 through 9. $C[N]$ holds the combined size of all tetrahedron lists, so as described earlier we allocate array L to be size $C[N]$, which is 11 in our example.

The final step in building the grid is to insert object indices into the object lists in L. We start with the newly allocated array L and insert the object indices by reverse iterating over all tetrahedrons and, for each grid cell overlapped by the tetrahedron, decrementing the offset of the grid cell and storing the tetrahedron index at that offset, as seen in the algorithm in Figure 11 lines 11 through 15.

The bottom portion of Figure 13 shows the result of filling the tetrahedron list array L. For each grid cell overlapped by a tetrahedron, we decrement the cell's value $C[i]$. Then we use the new cell value in $C[i]$ as an index into array L, and set $L[C[i]]$ equal to the tetrahedron's index.

Each grid cell is associated with an element in array C. The value of the associated array element gives the offset into array L for that grid cell's tetrahedron list. Figure 14 shows the final results.
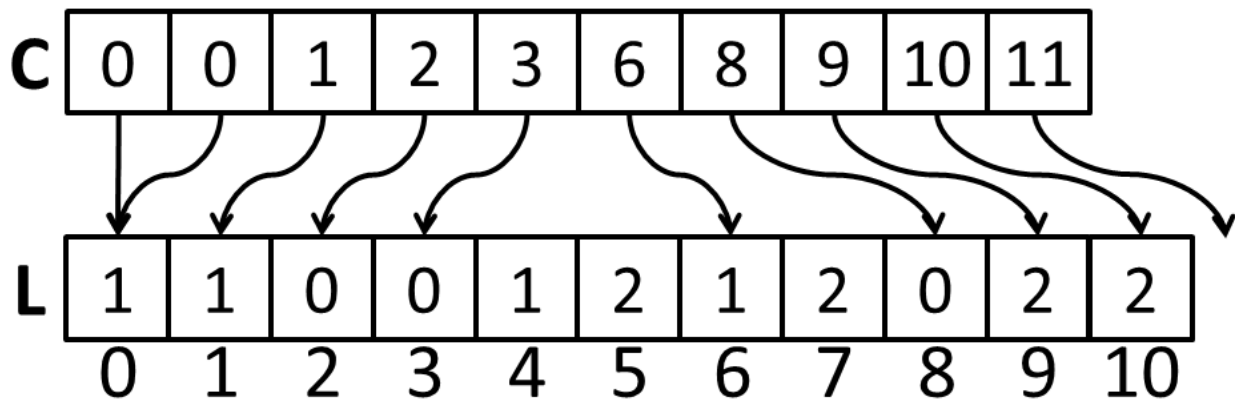

Figure 14 - Array C gives the offset of each grid cell's tetrahedron list in L. Array L stores the concatenation of all the tetrahedron lists, which hold the indices of the tetrahedrons that intersect each grid cell.

In this way, the object lists are filled backwards, and the cell array contains all the correct offsets. The indices in each object list are also sorted. The time complexity of the build algorithm is

linear in the number of tetrahedrons, and no additional memory is required. Since we chose $\rho = 1$, then $M = N$, and we can assume that, on average, each tetrahedron overlaps one grid cell.

The size of each cell's tetrahedron list is given by $C[i + 1] - C[i]$. In Figure 14 we can see that grid cell 0 intersects no triangles, grid cell 1 intersects triangle 2, grid cell 2 intersects triangle 1, and so forth.

3.1.3 – Querying the Compact Uniform Grid

Once the vertices of the mesh have been moved and remeshed and the uniform grid has been constructed and populated with tetrahedrons from the old mesh, we relabel the Delaunay cells in the new mesh. The relabeling is performed by iterating over the Delaunay cells in the new mesh—recall that a Delaunay cell is a triangle or tetrahedron in a Delaunay mesh with its associated material label. For each Delaunay cell, we compute the reverse-advected circumcenter and use it as a query point. We compute which grid cell contains the query point in the same way that we computed the grid cells containing the Min and Max points for the tetrahedrons' bounding boxes in equations (3) and (4). Once we know the index of the grid cell containing the query point, we iterate from $L\big[C[i]\big]$ to $L\big[C[i + 1]\big]$. If the indexed tetrahedron contains the query point, we assign the old Delaunay cell's material value to the new Delaunay cell in the new mesh. If none of the indexed tetrahedrons contain the query point, or if the grid cell overlaps no tetrahedrons, then the new Delaunay cell is assigned the material value "air."

3.2 – Parallelization

Our work includes parallelization using the OpenMP API [Barney]. The construction of the uniform grid includes two doubly nested for-loops. These loops were not parallelized because the contents of the inner for-loops must be synchronized, making parallelization of the construction of the grid difficult to perform in parallel with the simple locking mechanisms that OpenMP provides. The overhead involved in a more complex locking system, such as one involving an array of mutexes where each thread acquires a lock associated with the modulus of the location it is updating, proved too much for the small reduction of lock contention for the meshes we worked with. However, the querying/relabeling step requires no writes to the grid, and each circumcenter's new material is written exactly once. The relabeling loop is, therefore, embarrassingly parallel, naturally lending itself to OpenMP constructs.

Exploiting other types of SIMD parallelism using CUDA on the GPU is not easily achieved because of our choice to use CGAL. CGAL's objects and data structures are not designed to run on the GPU—for example, none of the objects' or data structures' constructors or functions are implemented as GPU kernels--so conversion to GPU-ready data is required. Future work might explore a GPU implementation based on a different computational geometry library, but that is beyond the scope of this thesis.

To test our algorithm for speed and consistency, we designed two experiments. The first experiment, weathering, uses the algorithm also used by Tychonievich and Jones for spheroidal weathering, which normalizes the mean curvature[1] across the surface of an object. This first experiment was further broken down into two sub-experiments: 1) weathering various objects over 200 frames, and 2) weathering the same shape with various resolutions—numbers of tetrahedrons. The second experiment, colliding objects, uses a simple function to move two objects toward and through each other.

4.1 – Weathering

The purpose of the weathering experiment is to show that our algorithm performs faster than the previous fastest method while still producing similar results. As Figure 15 shows, the results of our algorithm are still consistent with those produced by Tychonievich and Jones. Here we weathered a cube to a sphere, the expected result for uniform spheroidal weathering.
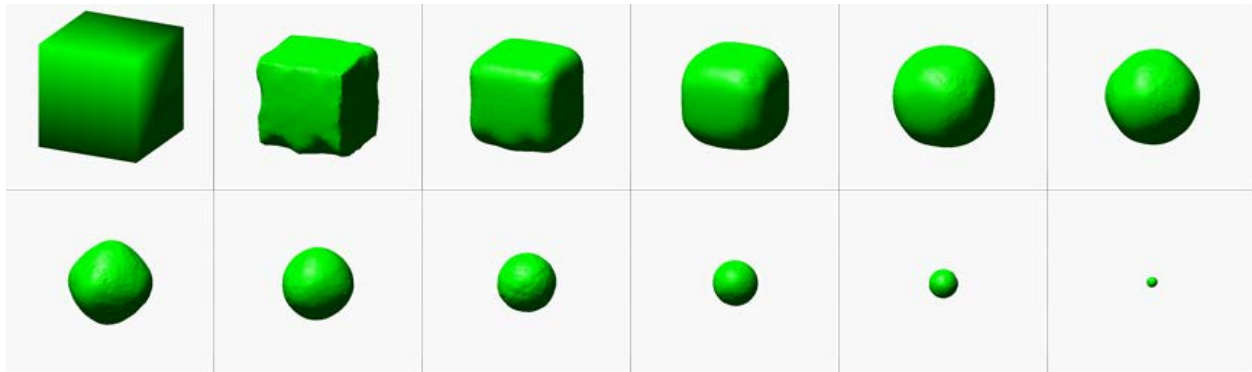

Figure 15 - Weathering a uniform cube into a sphere.

---

[1] Let p be a point on the surface S. Consider all curves C_i on S passing through p. Every such C_i has an associated curvature K_i given at p. Of those curvatures K_i, at least one is characterized as maximal κ_1 and one as minimal κ_2, and these two curvatures κ_1, κ_2 are known as the principal curvature of S. The *mean curvature* at $p \in S$ is then the average of the principal curvatures, hence the name: $H = \frac{1}{2}(\kappa_1 + \kappa_2)$.

Figures 16 through 18 show similar results for non-uniform spheroidal weathering of a Y, a cow, and two tori. Here, non-uniform weathering means that the object being weathered has non-uniform durability, so "softer" areas of the surface weather faster than "harder" areas.
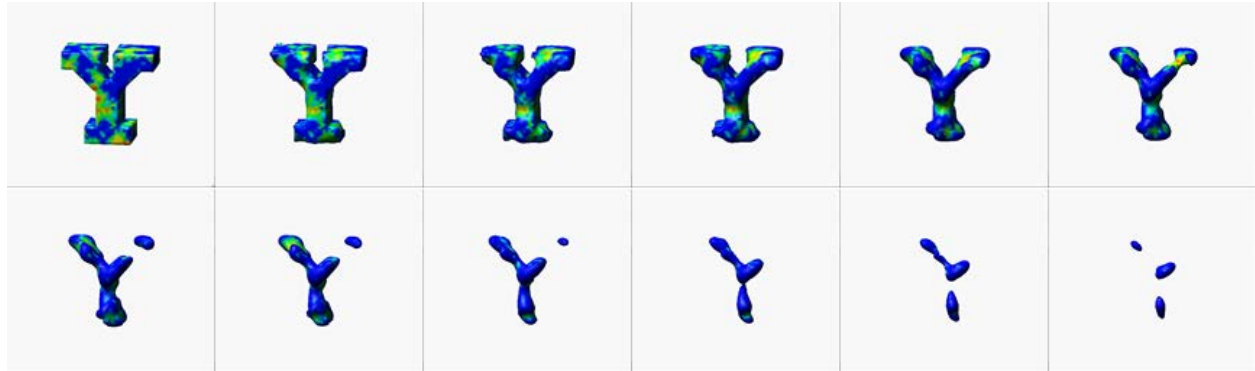

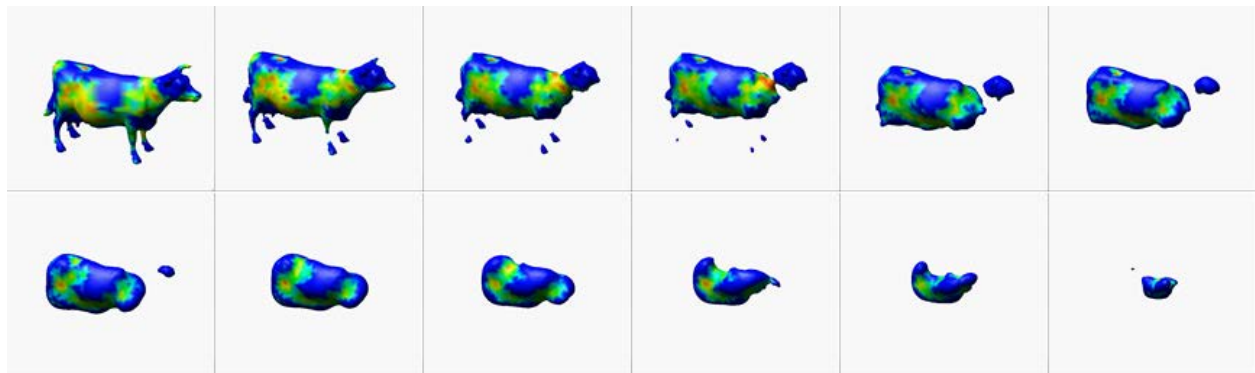Figure 16 - Non-uniformly weathering a Y.

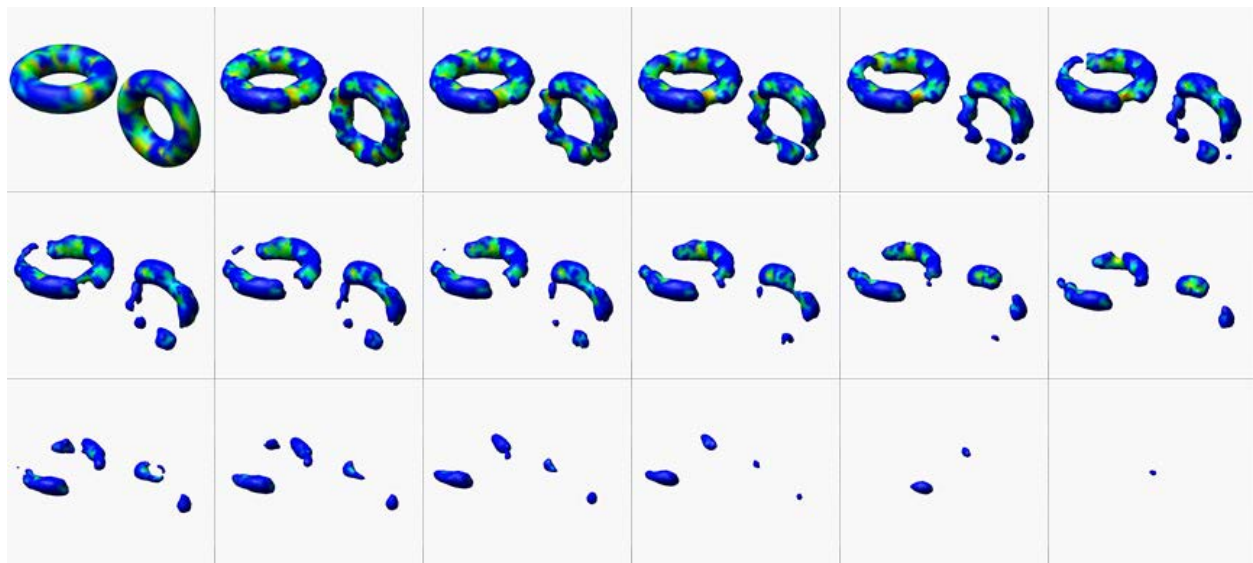
Figure 17 - Non-uniformly weathering a cow.

Figure 18 - Non-uniformly weathering two tori.

Our experiments with weathering show that using the compact uniform grid improves speed performance of the relabeling step for a Delaunay deformable mesh compared to methods based on the Delaunay hierarchy. Since Tychonievich and Jones' algorithm spent 70% of its compute cycles in the relabeling step, improving relabeling performance also improved overall performance of the deformation algorithm.

On average, our serial implementation achieved a speedup of 1.874 over Tychonievich and Jones' algorithm, and our parallel implementation achieved a speedup of 3.365 on 4 parallel cores. Figure 19 shows the total runtimes for Tychonievich and Jones' work as well as our work, both in serial and in parallel, over the entire experiment for weathering the two tori.
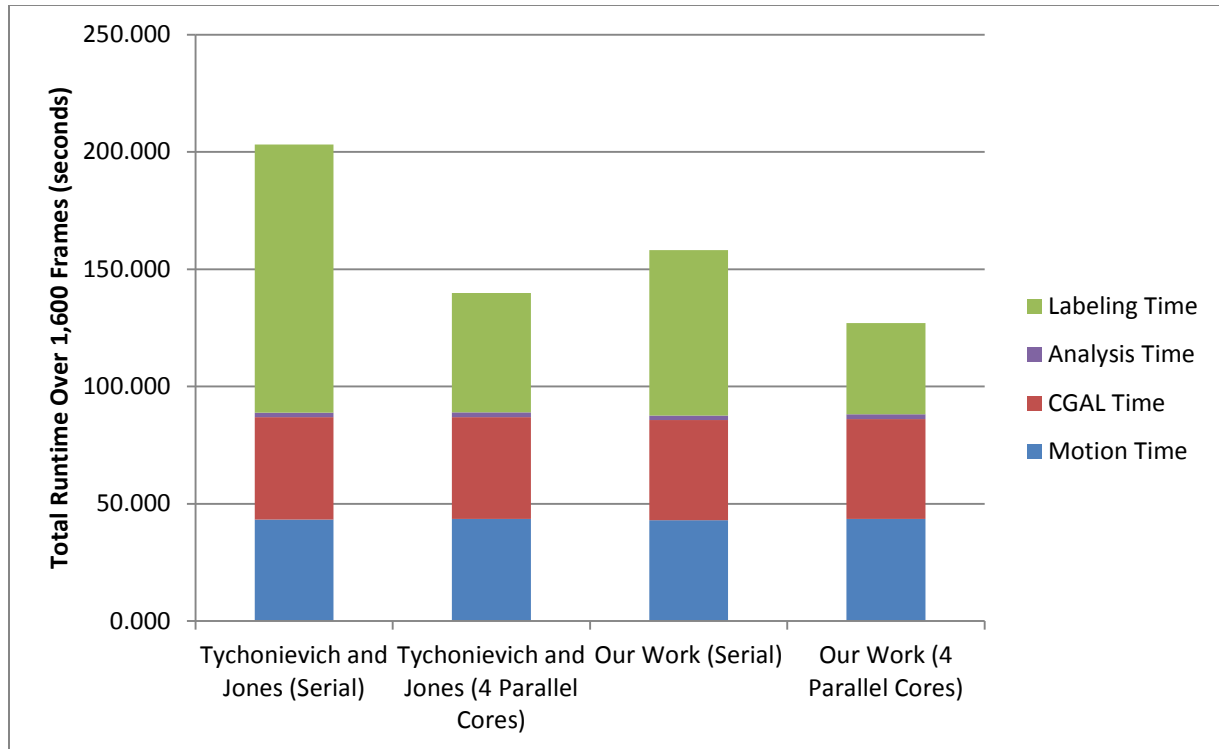
Figure 19 - Total runtimes for weathering two tori.

Figures 20 through 22 show the results of the second part of the weathering experiment, which
involved comparing the runtimes for the first 200 frames of weathering the Y but using different
numbers of tetrahedrons—effectively changing the level of detail of the object. Figure 20 shows
that the total runtime for all computations other than the relabeling step remained fairly
consistent for our work versus Tychonievich and Jones' algorithm. In Figure 21 we show that the
relabeling step of our algorithm scales much better than the previous fastest method. Finally,
Figure 22 shows that enhancing the performance of the relabeling step had a noticeable impact
on the performance of the overall algorithm.
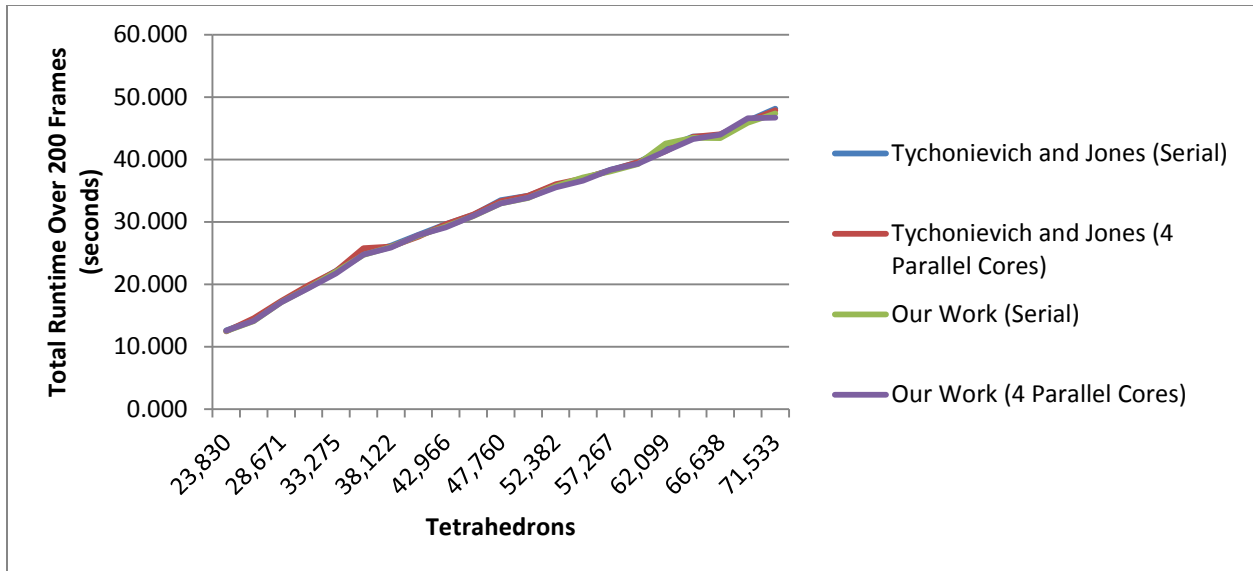
www.manaraa.com

Figure 20 - Total runtime over 200 frames with increasing levels of object detail for all computations other than the relabeling step when weathering the Y.
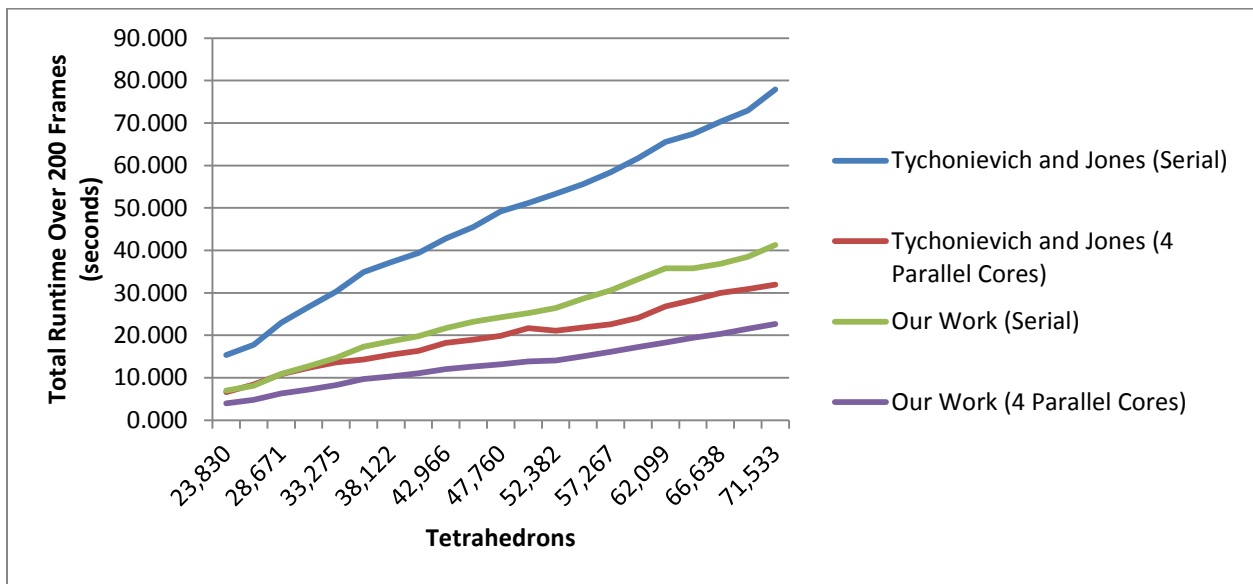

Figure 21 - Total runtime over 200 frames with increasing levels of object detail for the relabeling step when weathering the Y, showing that our algorithm scales better than the previous fastest method.
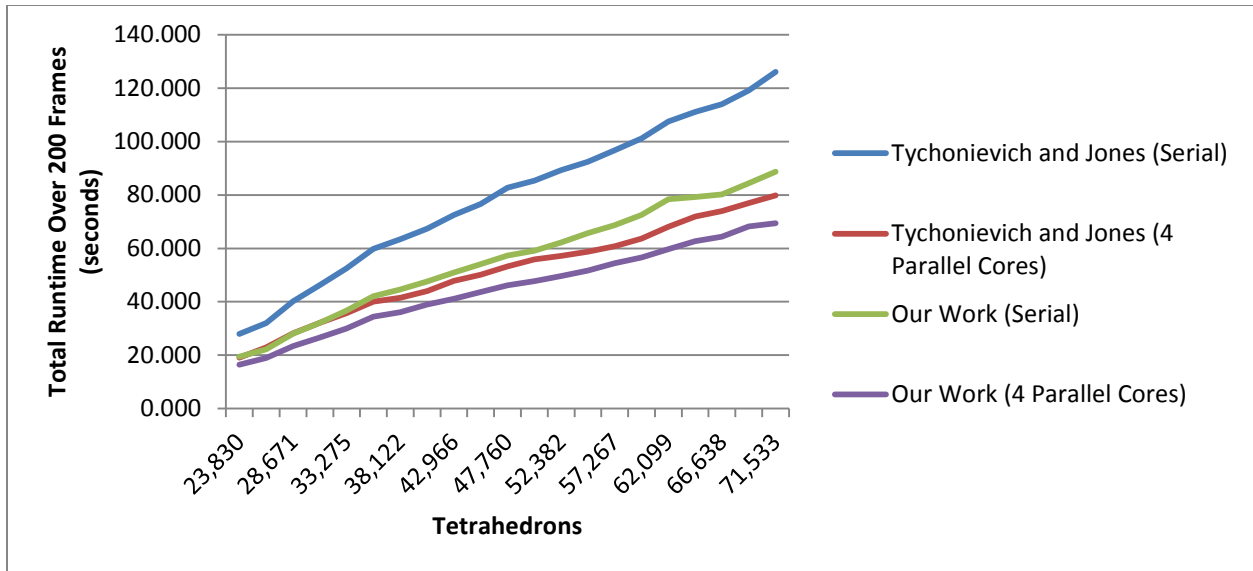
Figure 22 - Total runtime over 200 frames with increasing levels of object detail for all computations when weathering the Y, showing that enhancing the performance of the relabeling step had a noticeable impact on the performance of the algorithm as a whole.

4.2 – Colliding Objects in Space

The purpose of the colliding objects experiment was to show that our algorithm can handle significant topology changes, beyond what has been demonstrated using previous methods. It should be noted that these experiments were *not* intended to be physically based simulations of real-world objects colliding—mass was not conserved, and no collision detection was performed. The only source of deformation of the objects is a function that moves the objects toward each other and resolves shared vertices to remain with one of the objects.

As Figures 23 through 25 show, our algorithm is able to keep the objects closed and labeled while performing significant topology changes including closing holes, merging objects, creating holes, and splitting objects.
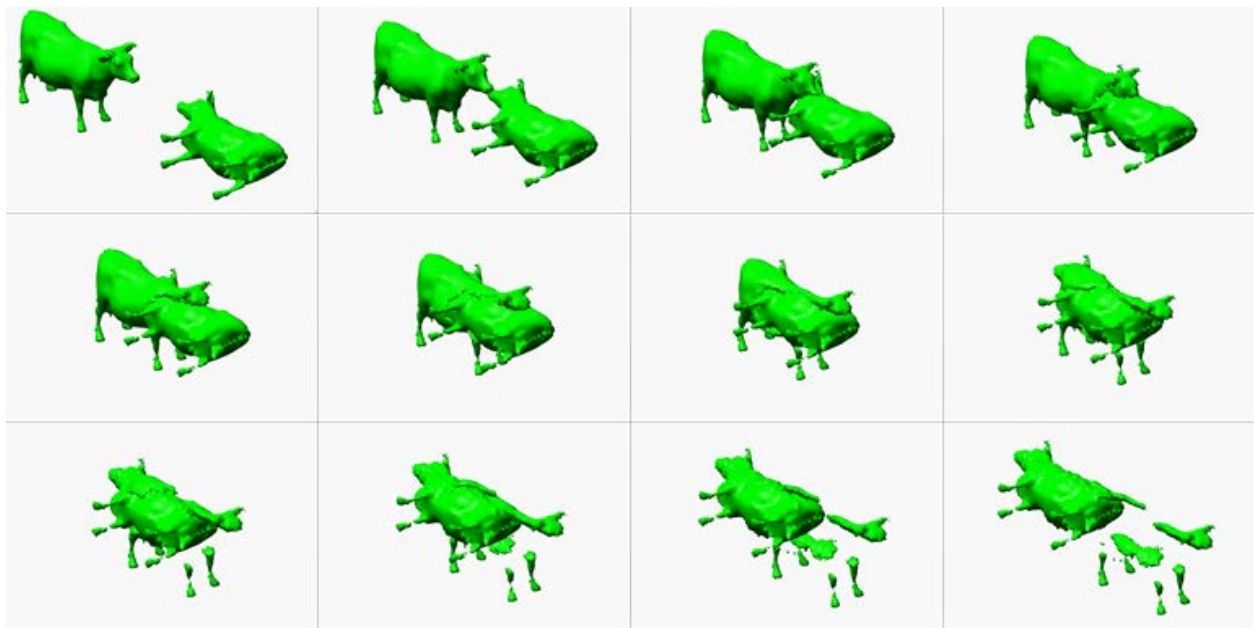
32

Figure 23 - Two cows colliding in space. We used a simple function to move the two objects toward each other, which resolves shared vertices by moving the shared vertices with the cow that starts on the right.
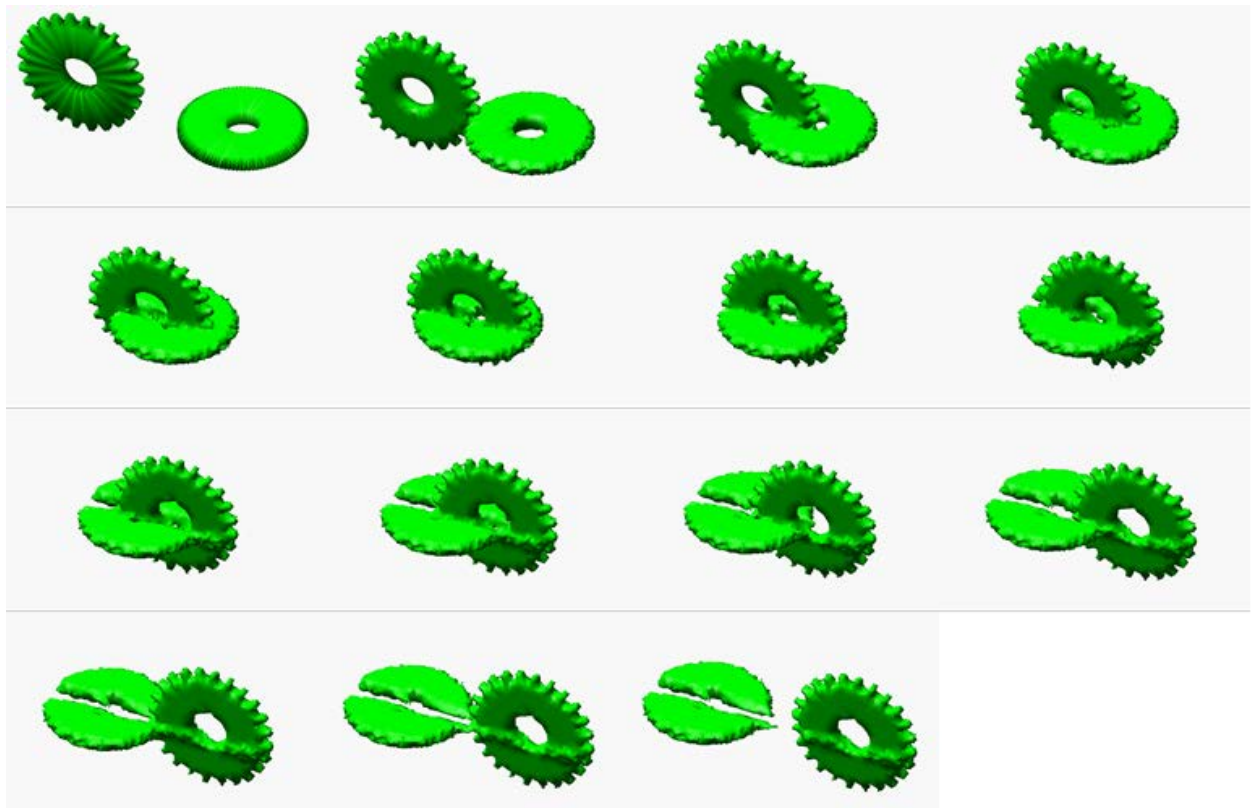

Figure 24 - Two gears colliding in space. We used the same function as from the colliding cows, but this time the function resolves shared vertices by moving them with the gear that starts on the left. The extra "matter" that accumulates where the gears intersected is the vertices from the horizontal gear that became shared and therefore moved with the vertical gear.
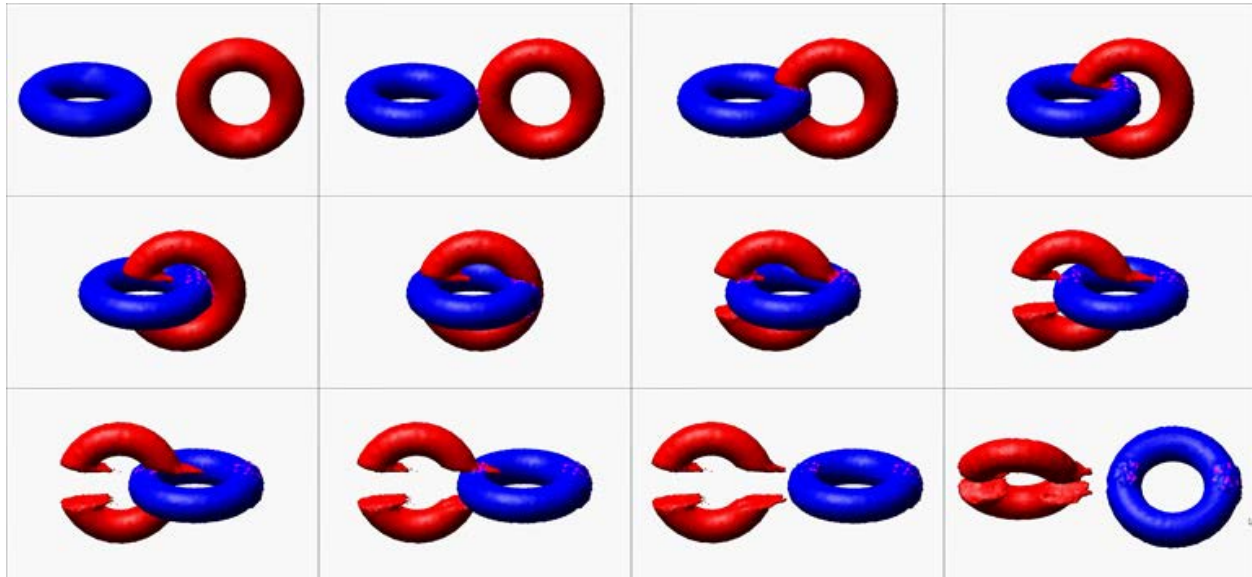
Figure 25 - Two tori colliding in space. Each torus is identified by a separate color, and vertices currently previously shared are colored magenta. This experiment specifically shows the Lagrangian nature of our algorithm since the vertices carry their information with them.

Note that in Figure 24, the accumulation of vertices across the area where the gears intersected is a result of the resolution of shared vertices—we simply chose to move shared vertices with the vertical gear. This is also what causes the split in the horizontal gear.

In Figure 25 we gave each object a color that is carried with the vertices of the objects. Note that shared vertices are colored magenta, even after the objects have split. This shows the Lagrangian nature of the algorithm as the vertices carry information with them.

Figures 26 through 28 show the total runtimes for each of the collision experiments. Note that for the colliding cows and gears the relative speedups are similar to the weathering experiments. However, the colliding tori ran quite slowly using Tychonievich and Jones' algorithm.
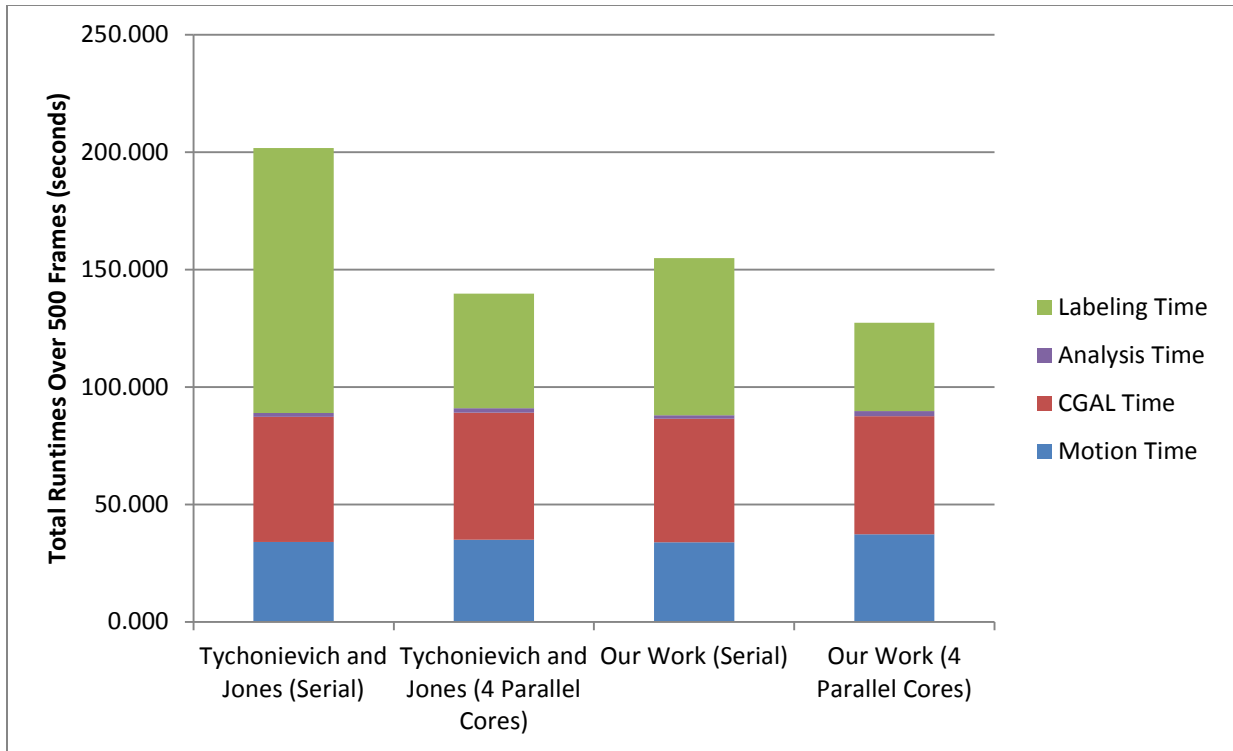
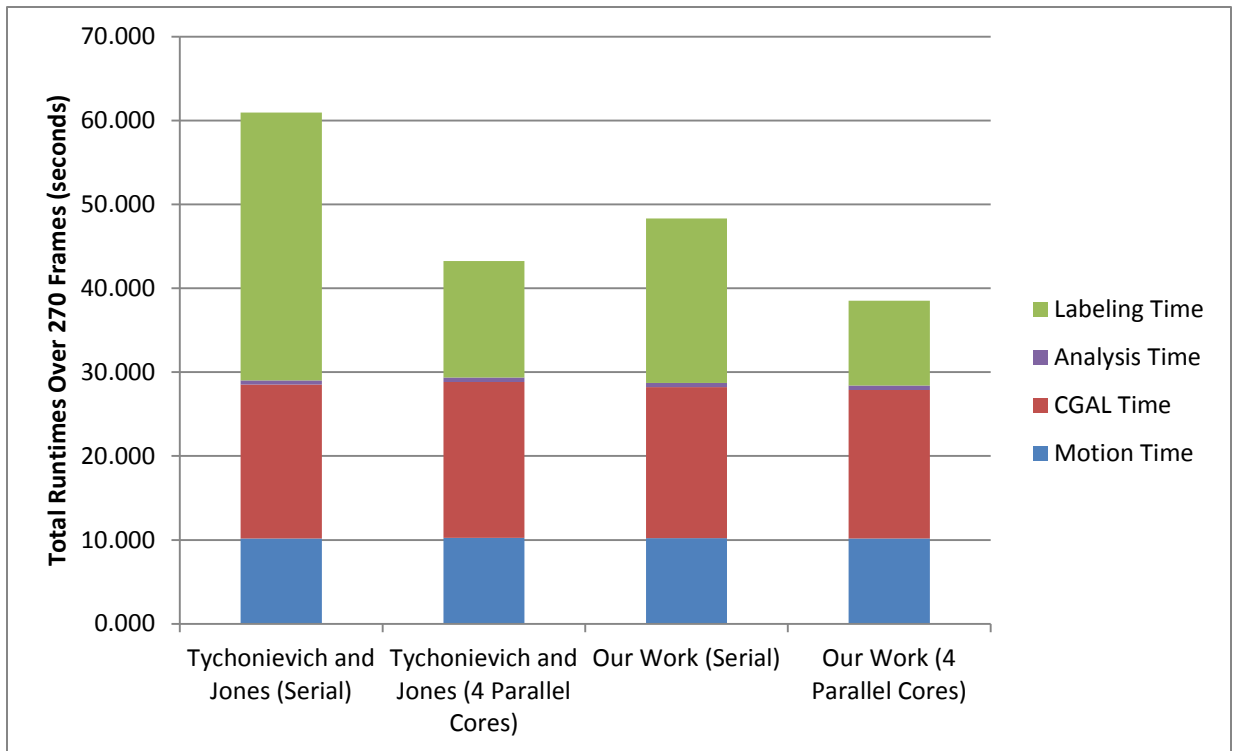Figure 26 - Runtime results for colliding cows.

Figure 27 - Runtime results for colliding gears.
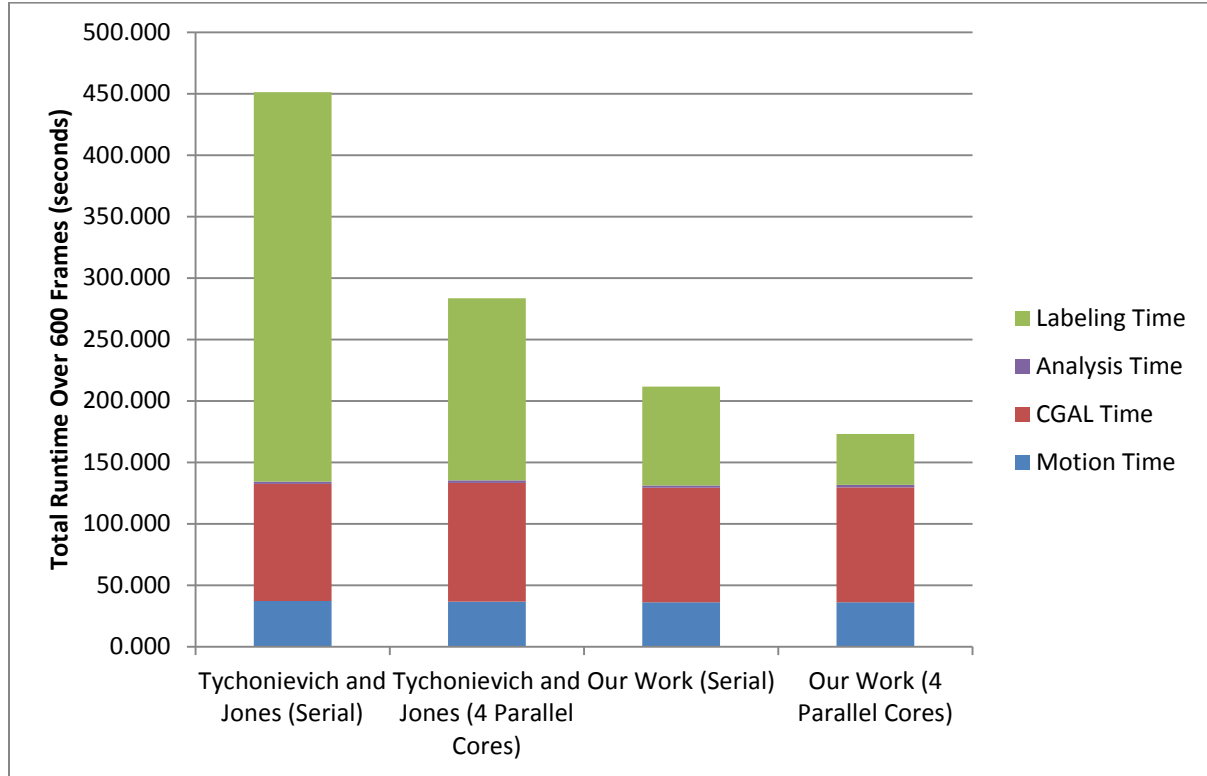
Figure 28 - Runtime results for colliding tori. Notice that Tychonievich and Jones' algorithm ran very slowly with this experiment.

The average speedup of our serial algorithm over Tychonievich and Jones' algorithm for the cows and the gears is 1.658. The average speedup of our parallel algorithm over Tychonievich and Jones' algorithm is 3.074. For the colliding tori, the serial speedup is 3.934 and the parallel speedup is 7.635.

4.3 – Analysis

Since Tychonievich and Jones' algorithm spent 70% of its compute cycles in relabeling, optimizing the relabeling step naturally has a noticeable impact on the overall algorithm. Our average overall speedup for all experiments is 1.455 in serial and 1.779 in parallel. It is interesting to note that parallelizing Tychonievich and Jones' algorithm produced a speedup of

36

1.452, which is comparable to the speedup we achieved simply by using the uniform grid in serial.

The compact uniform grid is a viable alternative to walking the Delaunay hierarchy because the computations involved in creating and traversing the grid are faster and less complex. The compact uniform grid also decreases the number of tetrahedron inside/outside tests needed to find the Delaunay cell containing a reverse-advected circumcenter. For example, in weathering the Y over 20 frames our algorithm performed over 24 million inside/outside tests for tetrahedrons, but Tychonievich and Jones' algorithm using CGAL's Delaunay hierarchy performed over 35 million similar tests.

Our algorithm does not produce *exactly* the same relabeling as Tychonievich and Jones' algorithm—and consequently, it doesn't produce exactly the same triangulation for each frame. This is due to the fact that Tychonievich and Jones use CGAL exclusively for their relabeling step, which was designed to use high levels of precision. Our relabeling step, on the other hand, uses standard 64-bit, double-precision floating point numbers. Our tetrahedron inside/outside test can report a false positive or false negative when the query point is very close to one of the faces of the tetrahedron. However, since the purpose of this thesis is not exact precision but rather to produce similarly plausible results as Tychonievich and Jones, we ignore this issue. Careful profiling determined that CGAL's inside/outside test actually runs slightly faster than ours does despite being more accurate. But our grid-based algorithm requires fewer inside/outside tests as well as having constant-time lookup to find the grid cell that contains a query point and remains faster overall.

In our experiments, every frame of the deformation animations is literally a test case. Therefore, instead of simply running a handful of experiments, we technically ran thousands, and each frame is handled correctly—it is visually reasonable and it is computed faster.

CHAPTER 5 – CONCLUSIONS AND FUTURE WORK

5.1 – Conclusions

Replacing the Delaunay hierarchy with a uniform grid in the relabeling step for deformable Delaunay meshes gives an average speedup of 2.145 over the previous fastest method. Furthermore, parallelizing the relabeling step on 4 cores achieves an average speedup 3.979 over the relabeling step in the previous fastest work. Using a uniform grid also preserves the ability to relabel in the presence of topology changes.

5.2 – Future Work

It is important to point out that our work created the uniform grid after CGAL had already stored the mesh in its own data structure. For further optimization and speedup, we would prefer to use a specialized implementation of a Delaunay triangulation that is stored automatically in a compact uniform grid. Alternatively, future work might involve integrating a uniform grid within CGAL.

Because our algorithm uses the compact uniform grid, it no longer depends necessarily on the Delaunay mesh. Other triangulation algorithms that use valid—non-degenerate—tetrahedrons would be equally valid. However, to our knowledge no triangulation algorithm exists that converts a point cloud into a solid partitioned into tetrahedrons other than the Delaunay triangulation as used in CGAL. One candidate is the triangulation algorithm developed by VanderZee et al. [2008], which uses tetrahedrons for simple shapes such as parallel planes and pyramids. However, as it stands VanderZee's algorithm is not capable of handling the large complex objects used in our experiments. Further development of VanderZee's algorithm may

www.manaraa.com

yield an alternative approach that does not rely on the Delaunay property. Our experience with CGAL leads us to believe that replacing not only the triangulation algorithm but also the internal data structure and data types with more specialized data types would achieve significant further speedup.

As noted previously, the colliding tori experiment performed poorly using Tychonievich and Jones' algorithm. We were unable to account for this difference. Different kinds of simulations appear to achieve different kinds of speedup. A precise characterization for what simulations get what kinds of speedup would be a good direction for future work.

Our algorithm does not yet fully support multiple interacting material types. However, our experiments with the colliding objects show that it is possible. Other questions to be explored with multiple interacting materials include how to handle physically colliding objects and how to obtain conservation of mass and momentum.

REFERENCES

Alliez, P., Cohen-Steiner, D., Yvinec, M., Desbrun, M.: Variational Tetrahedral Meshing. In: ACM Transactions on Graphics, Volume 24 Issue 3, pp. 617-625 (2005)

Autodesk: Maya 2011 (2011). http://usa.autodesk.com/adsk/servlet/pc/index?id=13577897&siteID=123112

Autodesk: Mudbox 2011 (2011). http://resources.autodesk.com/med/Autodesk_Mudbox

Barney, B: OpenMP. https://computing.llnl.gov/tutorials/openMP/

Barreira, N., Penedo, M. G.: Topological Active Volumes. In: Pattern Recognition, Volume 43 Issue 1 (2010)

Beneš, B., Těšínský, V., Hornyš, J., Bhatia, S.: Hydraulic Erosion. Computer Animation and Virtual Worlds, Volume 17 Issue 2, May 2006 (2006)

Blender Foundation: Blender v2.49b (2009). http://www.blender.org/

Bundysoft: Large 3D Terrain Generator v2.9 (2010). http://www.bundysoft.com/L3DT/

Cgal, Computational Geometry Algorithms Library v3.7 (October 2010). http://www.cgal.org

Devillers, O.: The Delaunay Hierarchy. In: International Journal of Foundations of Computer Science, Volume 13 Issue 2, pp. 163-180 (2002)

E-on Software: Vue9 xStream (2010). http://www.e-onsoftware.com/

Fürnstahl, P.: Consistent Mesh Partitioning using Tetrahedral Meshes. Master's thesis, Graz University of Technology (2005)

Ito, T., Fujimoto, T., Moraoka, K., and Chiba, N.: Modeling Rocky Scenery Taking into Account Joints. In: Proceedings of Computer Graphics International, pp. 244-247 (2003)

Jones, M.D., Farley, M., Butler, J., Beardall, M.: Directable Weathering of Concave Rock using Curvature Estimation. In: IEEE Transactions on Visualization and Computer Graphics, Volume 16, No 1, pp. 81-95 (2010)

Lagae, A., Dutré, P.: Compact, Fast and Robust Grids for Ray Tracing. In: Eurographics Symposium on Rendering, Volume 27 Issue 4, pp. 1235-1244 (2008)

Lorensen, W.E., Cline, H.E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In: ACM SIGGRAPH Computer Graphics, Volume 21 Issue 4, pp. 163-169 (1987)

OpenGL, Open Graphics Library. http://www.opengl.org/

Pons, J.P., Boissonnat, J.D.: Delaunay Deformable Models: Topology-Adaptive Meshes Based on the Restricted Delaunay Triangulation. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 1-8 (2007)

Quad Software: Grome 2 (2009) http://www.quadsoftware.com/index.php?m=section&sec=product&subsec=editor

Rosenberg, Johannes: GeoControl 2 (2008). http://www.geocontrol2.com/e_index.htm

Sederberg, T.W., Cardon, D.L., Finnigan, G.T., North, N.S., Zheng, J., Lynche, T.: T-spline Simplification and Local Refinement. In: ACM Transactions on Graphics, Volume 23 Issue 3, pp. 276-283 (2004)

Sederberg, T.W., Zheng, J., Bakenov, A., Nasri, A.: T-splines and T-NURCCs. In: ACM Transactions on Graphics, Volume 22 Issue 3, pp. 477-484 (2003)

Tychonievich, L.A., Jones, M.D.: Delaunay Deformable Mesh for the Weathering and Erosion of 3D Terrain. In: The Visual Computer: International Journal of Computer Graphics, Volume 26 Issue 12, pp. 1485-1495 (2010)

VanderZee, E., Hirani, A. N., Guoy, D.: Triangulation of Simple 3D Shapes with Well-Centered Tetrahedra. In: Proceedings of the 17th International Meshing Roundtable, Part 1, pp. 19-35 (2008)

World Machine: World Machine 2(2008). http://www.world-machine.com/index.php