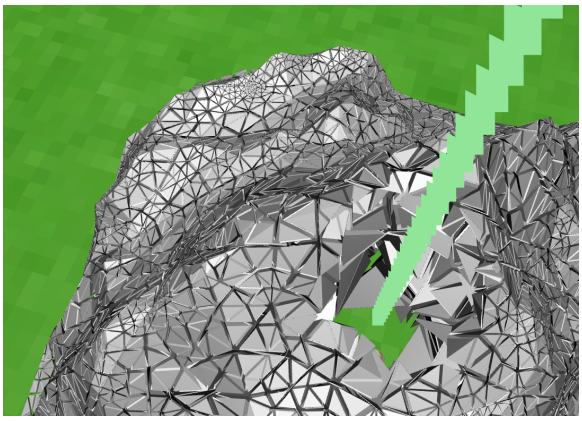
Particle-Based and Preprocessed Deformation using Tetrahedrons

Michael Peters, Marcus Panozzo



ABSTRACT

We present an application which simulates both pre-processed and real-time tetrahedral mesh deformation. We use TetGen to generate quality tetrahedral meshes. Our preprocessor, PLYBuilder can remove spherical portions of the tetrahedral mesh using what we call, cookie-cutter spheres. PLYBuilder also outputs the preprocessed meshes in PLY format for view with ParaView and CSV format for RenderLite. RenderLite visualizes the tetrahedral meshes in real-time and can perform a particle based deformation system to visualize drilling a hole or slicing a portion off of a mesh.

1. Introduction

Tetrahedrons have been used in a variety of ways to represent the internal volume of a bounding mesh. Our approach uses tetrahedrons as progressive "chunks" of the mesh that can be removed when degraded. Our simulator seeks to take advantage of this volumetric structure to simulate a progressive degradation of a mesh. We will demonstrate the following two ways of degrading the mesh:

1) Cookiecutter spheres with PLYBuilder
2) Particle system simulation with RenderLite

2. Related Work

Other works such as Huang's and Kenwright's have efficiently generated mesh deformations using a combination of interpolation and a basis mesh. These algorithms have also been shown to run effectively in real time. Our approach aims to use a similar basis mesh to Huang and Kenwright. However, we do not interpolate the exterior faces while deforming the mesh. This step applied on top of our process to further improve the smoothness of the deformation. We decided to leave it out for the sake of a simpler application.

3. Tetrahedralization with TetGen

TetGen, developed by Hang Si, can produce high quality tetrahedral meshes for any closed polyhedral mesh. It implements Hang Si's Constrained Delaunay tetrahedralization algorithm as discussed in his paper from Weierstrass Institute for Applied Analysis and Stochastics.

3.1 Removing Bad Tetrahedrons

TetGen provides two sliders to control the quality of the output mesh. The first of these sliders controls the aspect ratio of the tetrahedron. A tetrahedron's aspect ratio refers to the length of its longest edge divided by the length of its shortest edge. The best aspect ratio for a quality tetrahedron is 1 [3]. That is, all lengths of a tetrahedron's edges are the same. A lower aspect ratio tetrahedralization will contain fewer artifact-producing elements. These "bad" tetrahedrons are classified as needle, spindle, wedge, cap, and sliver tetrahedrons.

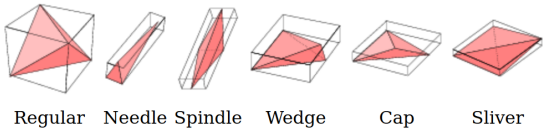


Figure 1: Tetrahedron Shapes
Source: http://tetgen.org

During deformation, "bad" tetrahedrons should be avoided as they can create unexpectedly large changes or artifacts in the resultant mesh.

TetGen provides a quality-determining slider proportional to aspect ratio. This slider controls the minimum radius-edge ratio of the tetrahedron. The radius-edge ratio refers to the length of the radius of the smallest circumscribing sphere to the length of the smallest edge. Regular tetrahedrons have the smallest radius-edge ratio at $\sqrt{6}/4 \sim 0.612$ [3]. The radius-edge ratio is very effective at filtering out needle, spindle, wedge, and cap tetrahedrons. However, it is not effective at filtering out sliver tetrahedrons. This anomaly can be seen in the following diagram:

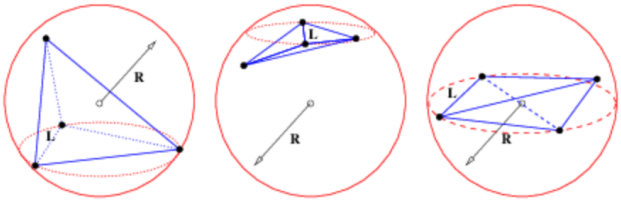


Figure 2: Radius-Edge Ratio Visualization
Left-to-right: Regular, Wedge, and Sliver
Source: http://tetgen.org

As shown in the figure, the sliver tetrahedron on the right maintains a similar radius-edge ratio to the more regular tetrahedron on the left. In a perfect scenario, the regular tetrahedron would be selected over the sliver tetrahedron. However, the radius-edge ratio does not provide this level of quality control

3.2 Maximum Area

The second of TetGen's tetrahedralization quality sliders controls the maximum volume of a single tetrahedron within the output mesh. TetGen uses smaller tetrahedrons to represent detailed surfaces of a mesh. However, this could lead the interior of the mesh to have much larger tetrahedrons than the exterior.. Constraining the maximum volume can help to maintain a consistent size of tetrahedron within the mesh. Without a consistent volume, larger tetrahedrons in lower-detail portions of the mesh can cause jumpy deformations as depicted in figure 3. Figure 4 depicts a properly area-constrained tetrahedralization.

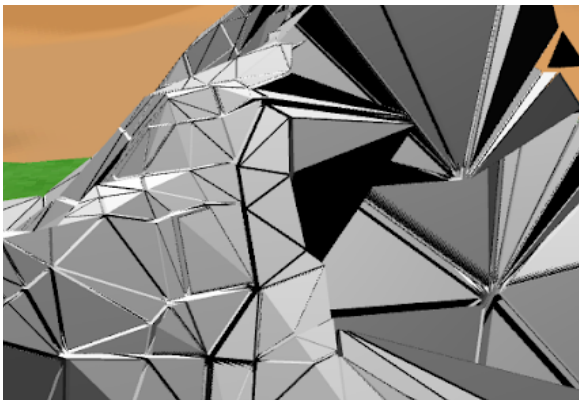


Figure 3: Without area constraints

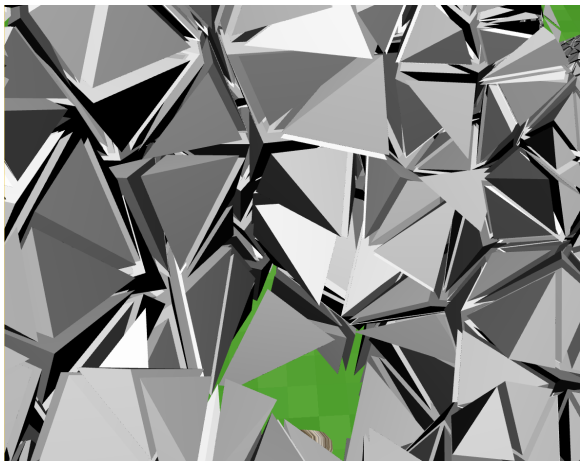


Figure 4: With area constraints

4. PLYBuilder

The first purpose of PLYBuilder was to generate tetrahedral mesh files viewable in the mesh viewing program, ParaView. ParaView can open .ply files containing a combined list of vertices and edges. TetGen has the option to output .ply files. However, TetGen's .ply file output only contains the exterior faces of the mesh and does not contain the structure for the internal tetrahedrons.

4.1 Tetrahedron Visualization

TetGen's outputs more than the external face.ply file. It provides a variety of data files including vertex data for each tetrahedron, a list of tetrahedral vertex elements, a list of neighboring tetrahedrons, and a list of exterior faces in non .ply format. PLYBuilder loads the vertex data file and the elements data file. It processes these files to determine the faces of the internal tetrahedrons and outputs a .ply file containing this data. The resulting PLYBuilder .ply file can be visualized in ParaView. With some minor transparency, it is easy to see the internal tetrahedral structure of the meshes as shown below.

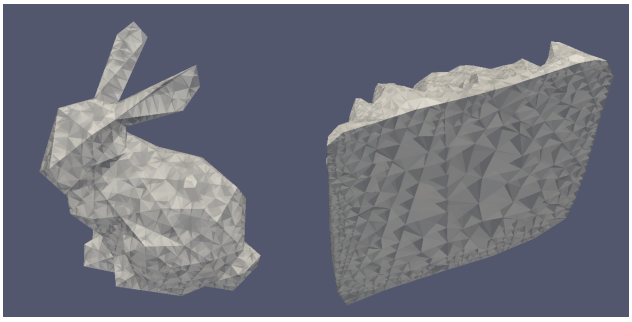


Figure 5: Tetrahedral Visualization in ParaView

4.2 Random Deformation

As a proof-of-concept, we determined that PLYBuilder could be used to randomly deform our tetrahedralize meshes. To create a random

deformation, we selected a ratio of tetrahedrons to keep and randomly selected them from the list of elements. The culled tetrahedrons were skipped and not added to the resulting .ply file. An example of this can be seen in the following figure.

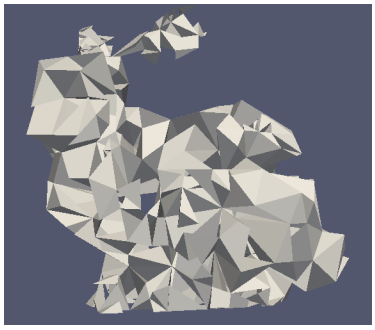


Figure 6: 50% Deformed Bunny

4.3 Cookie Cutter Spheres

The random generation inspired us to go further with pre-processing deformation. It could be useful to cut out portions of a mesh to create dents or other surface features. To do this, we created spheres within the mesh. Any tetrahedrons intersecting with these spheres would be removed from the resulting mesh.

A naive approach to sphere-tetrahedron collision would entail intersecting the sphere with each of the tetrahedron's face planes. However, for improved performance, we implemented an alternative sphere-tetrahedron intersection method that uses a bounding sphere around the tetrahedron and a projected point. This method takes advantage of the convexity of the sphere and tetrahedron shapes. First, we computed the maximum radius from the centroid *ct* of each tetrahedron's vertices, *rt*. Then we computed the distance *d* between *ct* and the center of the sphere *cs*. If the sum of the radius of the sphere *rs* and *rt* is greater than *d*, then the tetrahedron and sphere cannot intersect. Further, if *d* is less than or equal to *rs,* the tetrahedron must intersect with the sphere as its centroid intersects with the sphere. Otherwise, when $rs < d < rt + rs$, we project a point *p* to $rs * (rs - dt) / \| rs - dt \|$. This can be described as projecting to the point on the sphere closest to the centroid of the tetrahedron. If this point is on the internal side of all four faces of the tetrahedron, the sphere intersects with the tetrahedron. If not, the final stage of our sphere collision algorithm calls for intersection with each of the vertices *v* of the tetrahedron to be intersected with the sphere. If $\| v - cs \| < r$, the tetrahedron intersects with the sphere. Otherwise, the tetrahedron does not intersect with the cookie cutter sphere, and will be exported by PLYBuilder. With the high-quality, low aspect ratio tetrahedrons generated by TetGen, the final step of our algorithm can be skipped to produce similar results with slightly improved runtime. However, the results will be slightly more jagged. This effect is exaggerated with tetrahedral meshes containing more needle and spindle tetrahedra. The full and partial versions of the cookie cutter algorithm's results are shown below.

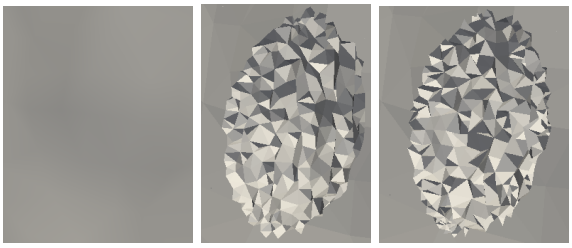


Figure 7: Cookie Cutter Cutouts
Right-to-left: No cookie cutter, full cookie cutter, cookie cutter with last step removed

5. RenderLite

RenderLite is our lightweight rendering program we created in order to display our simulations. Written in C++ and using the OpenGL rendering pipeline, it is capable of loading all necessary geometries, textures, and shaders into one place, where they are processed, and utilized when drawing to the screen.

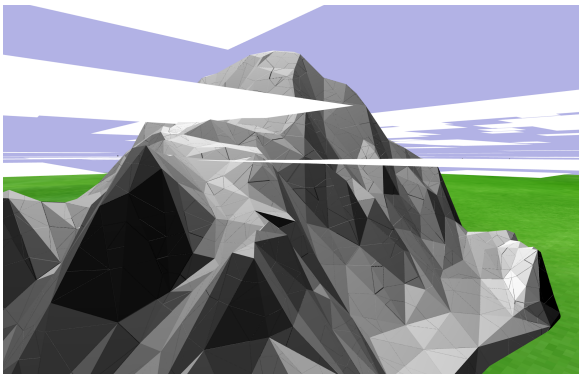


Figure 8: RenderLite rendering a simple mountain tetrahedral mesh.

Runtime of the RenderLite program consists of several stages. First, all essential objects are initialized, such as a window context (for drawing to the screen). Then, shader programs are compiled based on those specified by an xml file, allowing us to specify many shaders programs, with any variety of stages, to be compiled upon runtime without including extra program logic.

Most importantly, traditional meshes and then, the tetrahedral meshes, are loaded into program memory. After loading these into program memory, this data is passed to OpenGL buffers. We referred to these references to stored OpenGL data as GLHandles. These allow for instancing of objects on the screen.

5.1 Loading Shader Programs

For ease of updates, shader programs are stored in text files, which are then referenced in an XML file. The XML file stores a schema containing specifications for all the shader programs to be compiled. This schema includes the specified stages, such as a Vertex shader stage, a Geometry shader stage, or a Fragment shader stage. This file is read and a corresponding hashmap of compiled GLSL shader programs is stored for use by RenderLite.

5.2 Loading Traditional Meshes

In the context of our project, "traditional meshes" refers to non-tetrahedral meshes. These are loaded into memory using an external library called Assimp. The mesh vertex data is then placed into OpenGL buffers to be drawn later.

5.3 Loading Tetrahedral Meshes

Tetrahedral meshes are specified using two csv files. These files can be generated using PLYBuilder. One contains the vertex positions themselves, while the other contains a specification for constructing a tetrahedron using four numbered vertices (which were specified in the vertex file). This has the benefit of allowing us not to duplicate vertex data when reading from file memory, ensuring space is conserved when copying data into OpenGL buffer memory.

Similar to loading traditional meshes, tetrahedron vertex data is stored in an OpenGL memory buffer to be called upon later.

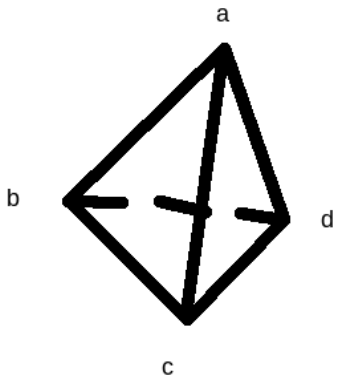


Figure 9: Sketch of a Tetrahedron. The four vertices would be stored in memory

Handles to the tetrahedron's data are then packaged into Tetrahedron objects, which store unique positions, a centerpoint, a transformation matrix (for storing any transformations), and a health value.

Tetrahedrons use a health based system to determine when they should be removed. Once the health of a tetrahedron object reaches below zero, it is removed from the drawing calls. A tetrahedron's health is initially determined by its volume when it is imported from the data file.

5.4 Rendering Loop

Once all external assets have been loaded, RenderLite runs in a rendering loop, taking user inputs, updating the window when resized, and drawing to the OpenGL context with the latest scene information. Most importantly, the rendering loop is where computations for the interaction between tetrahedrons and particle systems are done.

6. Particle Systems

A particle is simply a point in space which is managed by the ParticleEmitter class. Tetrahedrons are reduced in health when they come into contact with a particle. The ParticleEmitter class keeps track of attributes such as locations for particles, the current time of the simulation, and an updated list of which particles are in play.

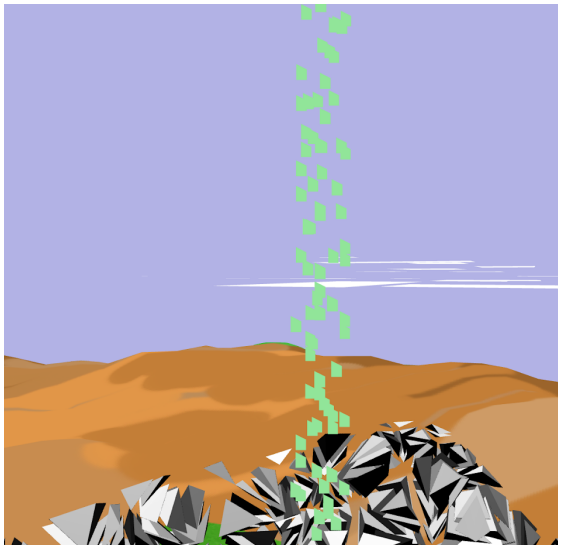


Figure 10: Screenshot of RenderLite particle emitter in action.

6.1 Particle Emitter

The ParticleEmitter class is responsible for the generation, modification, and storage of particles. Particles are generated in a cylindrical volume. The generation sequence occurs in a chronological order, generating sections of the full stream segments at a time. The duration of the stream, as well as how many particles to draw per section can be specified during construction of the object.

Particles are pre-determined as a specified initial direction and position. They are only active (and visible) once the ParticleEmitter's internal timekeeper reaches a certain threshold. This allows us to visualize new particles without needing to generate

them during the real-time rendering. This was done to improve the performance of the program.

6.2 Particle Appearance

We decided to render the particles as points in OpenGL memory buffers. The attributes of the particles (i.e. size, shape, color, texture) are determined inside the geometry shader program. For our simulations, we chose to render square-like patches. The color is added using a simple single-color fragment shader.

The flexibility of the shader compilation system allows us to create multiple variants of particle emitter shaders very easily. Particles with different colors can simply be made by using a different shader program.

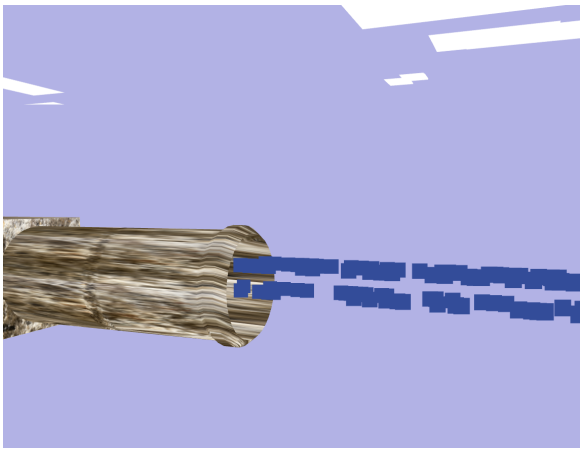


Figure 11: Modified particle shader program, illustrated moving horizontally and in blue.

7. Terraformer

The Terraformer class is responsible for performing the Tetrahedron health reduction and removal of damaged tetrahedrons. During execution of the rendering loop, the Terraformer performs proximity calculations using the particles in play from the ParticleEmitter and the tetrahedron positions passed in earlier.

7.1 Particles in Play

Upon each tick of the particle emission undertaken in the main render loop, the particles which “are in play” (are active) are updated by the Terraformer. Time governs whether a certain section of the ParticleEmitter’s particles are drawn or not.

7.2 Proximity Check

After each call to update the particles in play, the proximity of a tetrahedron to nearby active particles takes place. This proximity check simply compares the centroid of the tetrahedron to the position of the particle. If the two positions are within a specified range of each other, then the tetrahedron's health is lowered. If the health goes below a certain threshold, it is marked as damaged and will no longer be rendered. Efficiency of this method is O(n*m), where n is the number of tetrahedrons and m is the number of particles in play.

8. Future Work

PLYBuilder's cookie cutter feature is useful to cut out portions of a mesh. However, these cutouts are typically relatively jagged, even with very low radius-edge lengths from TetGen. PLYBuilder could implement a tetrahedral mesh simplification system by simplifying the jagged tetrahedrons into a smoother surface. The initial surface tetrahedrons could be determined using the external faces list generated by TetGen. Then, each time an external face is removed by the cookie cutter sphere, its revealed tetrahedrons could be marked as having an exterior face. Finally, after cookie cutting out the sphere, the marked faced tetrahedrons could be simplified together to produce a smoother mesh that could then be re-fed through the tetrahedralization pipeline.

The primary future work for RenderLite includes fixing vertex scaling issues (which cause the spacing in-between tetrahedrons) as well as improving the performance of the particle simulations. Due to the method of activating particles based on time, as the simulation runs on, the number of frames per second decreases significantly. This could be mitigated by choosing a better data structure for storing particles, such as a spatial tree. The performance could also be improved by implementing a hierarchy of tetrahedron collisions. Further, the realism could be improved by implementing a progressive shader for the tetrahedral meshes.

9. Team Workload

Michael Peters researched TetGen to create the tetrahedral meshes. He also created the PLYBuilder program itself. He assisted in creating properly formatted input files for RenderLite to be able to load the tetrahedrons. He also developed the cookie cutter sphere preprocessor step within PLYBuilder.

Marcus Panozzo wrote RenderLite and sub-components part of the visual application, such as the previously mentioned ParticleSystem and Terraformer. He also created some of the original mesh models such as the mountain.

10. Acknowledgments

We would like to thank Professor Cutler for providing feedback and guidance on parts of this project.

REFERENCES

[1] SCHEWCHUCK, JONATHAN RICHARD. Constrained Delaunay Tetrahedralizations and Provably Good Boundary Recovery. *IMR*, 2002

[2] SI, HANG. A new constrained Delaunay tetrahedralization algorithm. *Weierstrass Institute for Applied Analysis and Stochastics*, 2020.

[3] SI, HANG. TetGen: A Quality Tetrahedral Mesh Generator and 3D Delaunay. *WIAS-Software,* 2020

[4] HUANG, JIN. Efficient Mesh Deformation Using Tetrahedron Control Mesh. *ACM Solid and Physical Modeling Symposium*, 2008. pp 241-247.

[5] KENWRIGHT, BEN. Free-form tetrahedron deformation. *International Symposium on Visual Computing,* 2015. pp 787-796.

[6] V. KRS, et al. Wind Erosion: Shape Modifications by Interactive Particle-based Erosion and Deposition. *ACM SIGGRAPH / Eurographics Symposium on Computer Animation,* 2020.

[7] KAMARUDIN, N H, et al. “Assembly Meshing of Abrasive Waterjet Nozzle Erosion Simulation.” *IOP Conference Series: Materials Science and Engineering*, vol. 290, 2018, p. 012069.