Isosurface Stuffing Project Report

Stephen Carey, Computer Science 839

1: Project and Methodology

My final project for CS 839 was an attempt to implement Labelle and Shewchuk's 2007 paper on Isosurface Stuffing. The algorithm fills an isosurface with tetrahedra whose dihedral angles are provably bounded between 8 and 160 degrees (although these bounds may be slightly adjusted by a change in parameter). Such tetrahedra are ideal for finite element simulation methods, whose performance is faster with fewer tetrahedra, and more accurate with higher-quality tetrahedra. My project demonstrated this by running the CS 839 Ground Collision demo on tetrahedra generated by my implementation of Isosurface Stuffing. While reading the Labelle and Shewchuk paper, which can be found here: https://people.eecs.berkeley.edu/~jrs/papers/stuffing.pdf , I identified three sequential goals for improving a tetrahedral stuffing. The first, and the only one I achieved, concerns stuffing near an object's isosurface, where all the "interesting" features are found. Most of the 2007 paper is concerned with generating high quality tetrahedra near the surface; there I successfully mirrored Labelle and Shewchuk's technique.

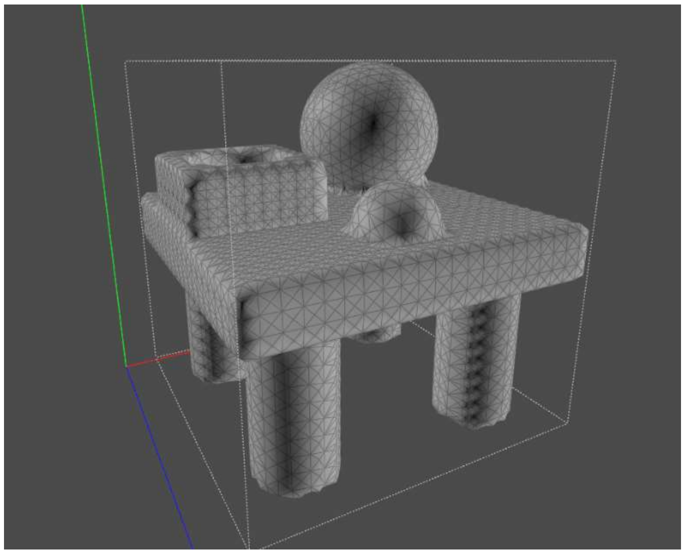
Figure 1: a mesh generated on a 50 x 50 x 50 BCC lattice

The Isosurface Stuffing algorithm accepts a signed distance function, $f(R^3) \to R$, which is continuous and defined in all space. F's magnitude at any point is equal to that point's distance from an isosurface: when on the isosurface, $f = 0$. F is positive outside of the isosurface of the object that will be meshed, and is negative inside the object. The zero-surface is filled in four steps from a BCC lattice. First, choose a subset P of the BCC lattice which contains every point inside the surface, every point on the surface, and every outside point connected to a point inside the surface. P therefore contains all points in the BCC lattice which have non-positive values in f, and every nearby point. No other points on the BCC lattice need be considered. Labelle and Shewchuk mention that this set could be generated by a depth first search from one or more seed points. For ease of programming, I followed their implementation and instead checked all BCC points within a bounding box. At higher BCC resolutions, the depth first search method would become more worthwhile: I found the algorithm impractically slow on my Acer Aspire E5-576 for any resolution higher than 100 x 100 x 100.

While generating P, the algorithm records every edge of the BCC grid whose endpoints are contained by P but whose values of f have a different sign: in other words, every edge that crosses the surface. It then approximates the cut point where the edge crosses the surface; where $f = 0$. The approximation uses a binary search: starting with a positive and negative side, it generates the midpoint and check its f value. If positive, the midpoint replaces the positive endpoint; if negative, the midpoint replaces the negative endpoint. This continues until *f*'s magnitude is below a certain tolerance: then the midpoint is recorded as a cut point. Although I stored the lattice points of P in a list, I decided to store cut points in an std::map. The value is the cut point itself; the key is the pair of endpoints.

Once all the cut points are generated, the algorithm checks each lattice point's 14 edges for a cut point. If a lattice point is too close to one of its cut points, keeping both points would result in a tetrahedron with an unacceptably small angle. In that case, the algorithm snaps the lattice point's spacial location to the nearest cut point, and all cut points formerly connected to that lattice point are deleted. The parameter, α, which defines how close is "too close", controls the quality of the tetrahedra. Labelle and Shewchuk provide a comprehensive analysis of how different α choices affect the final output; I used their provided value with no modification.

The final step of isosurface stuffing is to generate 1-3 tetrahedra within each BCC tetrahedron from P using precomputed stencils based on the *f* value of lattice endpoints. If all four BCC lattice points have zero or negative *f* values, the entire lattice is within the surface, and the BCC tetrahedron is output with no modification. If one or more of the lattice points has a positive *f* value, landing outside the isosurface, the algorithm generates new tetrahedra using the inner lattice points and the cut points that have already been calculated for the edges. These new tetrahedra are constructed from a group of pre-computed stencils; my implementation uses several large "if-else" trees to determine which stencils to use and how to generate the tetrahedra. My algorithm generates the list of particles and surface particles that the CS 839 demo will use simultaneously with these tetrahedra. Each lattice and cut point has an "index" field. Whenever I generate a tetrahedron, I check whether its points have assigned indices. If they do, nothing else need happen: the algorithm outputs those 4 indices to m_meshElements. If any points do not yet have an index, I first assign them the next available index, and add them to the end of m_particles. Cut points and lattice points with $f = 0$ values are surface particles: their indices are added to the m_surfaceParticles list. Once my stuffing algorithm finishes, the CS 839 demo has all the data in needs to run a simulation.

2: Misaligned Tetrahedral Faces Bug

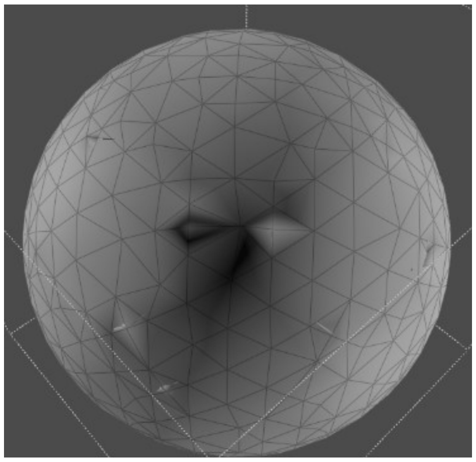
Figure 2: visual artifacts on the surface of a sphere

My algorithm does contain one bug that I was only partially able to fix. Occasionally, the provided stencils require the algorithm to split a quadrilateral into two triangles. This split can occur on either of the quadrilateral's diagonals: although the choice may improve tetrahedron quality, it is not required for any of isosurface stuffing's angle bound guarantees. The bug occurs when two stencils, operating on either side of the same quadrilateral, split the quadrilateral on different diagonals. The result is that four distinct and partially overlapping triangles generate instead of two copies of the same two triangles. This bug has no effect on simulation correctness: the correct particles still generate, and tetrahedra continue to completely stuff the surface. It does, however, break the assumptions of the CS 839 .usda file generator. The CS 839 demos do not record interior triangles in the .usda file in order to reduce unnecessary data storage. The demos determine whether a triangle is interior by counting how many times it appears in the mesh. Triangles which appear twice must have a tetrahedron on either side of their face, and must therefore be interior triangles. Triangles which only appear once can be labelled exterior triangles, and are saved to the .usda file. The mismatched splitting of quadrilaterals causes the generated interior triangles to lack an exact mirroring on another tetrahedron, leading the demo to mark an interior triangle as exterior. When these interior triangles are rendered, they create visual artifacts.

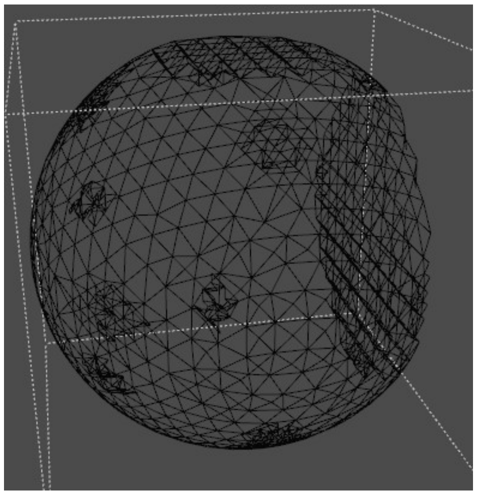
Figure 3: duplicated interior triangles

Labelle and Shewchuk provide non-ambigious specifications in their paper for how quadrilaterals ought be split. These specifications use the lengths of a quadrilateral's edges to determine which diagonal is preferred. If length isn't a sufficient metric, they furthermore introduce a parity rule that uses the evenness or oddness of the original BCC lattice grid coordinates to determine which diagonal should contain the split. My initial implementation did not follow this methodology, and so some of my earlier simulations (particularly the TableFalling.usda file included in this submission) are filled with these visual artifacts. After discovering the error, which is far more noticeable in a wireframe view of the mesh, I attempted to add these parity rules, and did indeed fix most of the visual artifacts. Some of my more recent mesh generation tests, however, have continued to display interior triangles. I think the problem comes from a flawed implementation of Labelle and Shewchuk's parity rules, but I haven't been able to find the actual error.

3: Signed Distance Functions

Both Labelle and Shewchuk's algorithm and my implementation of that algorithm treat the signed distance function, *f*, as a complete black box. Its operation has no effect on the output of isosurface stuffing, and my code provides the algorithm with an abstract class that can be extended by anything. Admittedly, if *f* is costly to evaluate, the algorithm might be sped up by replacing the binary search for cut points where $f = 0$ with taking the simple median between positive and negative valued lattice points. That decision would negatively effect the fidelity of generated meshes to the provided surface, but it may be worthwhile in some cases. Regardless, the ability to use any *f* opens the door for many practical improvements to isosurface stuffing algorithms.

3.1: Using Geometric Primitives

My implementation limited itself to analytical signed distance functions originating from geometric primitives. https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm contains a library of useful functions that can cover a surprising extent of objects. In my own work, I wrote functions for three objects: boxes, spheres, and cylinders. The signed distance functions for these objects are exact, bounded, and easy to calculate. Combining primitives into more complicated objects is also easy. I implemented three such binary operations. Union and Intersect take the respective min and max of the distance functions of each of their components. Subtract takes the max of the first distance function and the negative of the second distance function. With these operations, I was able to create and mesh fairly complicated objects, such as the desk shown in figure 1. Adding more primitives such as cones, tori, rods, and polyhedra, would be trivial.

3.2: Possible Extensions

More advanced use of the Isosurface Stuffing Algorithm would use a standard discretization of the signed distance function, where each lattice in a grid has a pre-computer value for *f*, and intermediate points are calculated as the interpolation of nearby grid points. More intriguing is the possibility of using a sparse representation of *f*, such as OpenVDB. Labelle and Shewchuk's algorithm should also work for level set functions in general, not just signed distance functions. The only place where the algorithm uses the magnitude of *f* is in the binary search to find cut points. That search may converge more slowly for a level set that isn't a signed distance function, but it should still converge. My implementation should similarly be able to handle general level set functions.

4: Potential Improvements to the Project

Labelle and Shewchuk offer an additional improvement to their isosurface stuffing algorithm. They consider the interior of an object as well, and provide a method for generated graded interior tetrahedra. Larger interior tetrahedra will generally improve the speed of a simulation without significantly reducing the quality. Labelle and Shewchuk use a modified quadtree to determine where large tetrahedra are permissible, and develop several tetrahedral stencils to handle transitions between different levels of detail in a BCC mesh. My implementation does not provide this optimization, but could easily be extended to do so. Such an improvement would be pair well with an attempt to use a sparse implementation of a signed distance function. Furthermore, my project only handles collisions with the ground plane. Providing foundations for more dynamic simulations would be an interesting extension to the work I've already done.

5: Included .usda files

SmallBounce.usda: a low resolution simulation of a bouncing ball.

BounceCutaway.usda: the same bouncing ball, slicing open the interior of the ball to display its BCC lattice.

HollowBall.usda: the same bouncing ball, hollowed out by a subtract binary operation.

TableFalling.usda: a more elaborate simulation with a higher resolution mesh; required 8 hours of simulation time.

Desk.usda: a single frame containing a detailed, high resolution mesh.