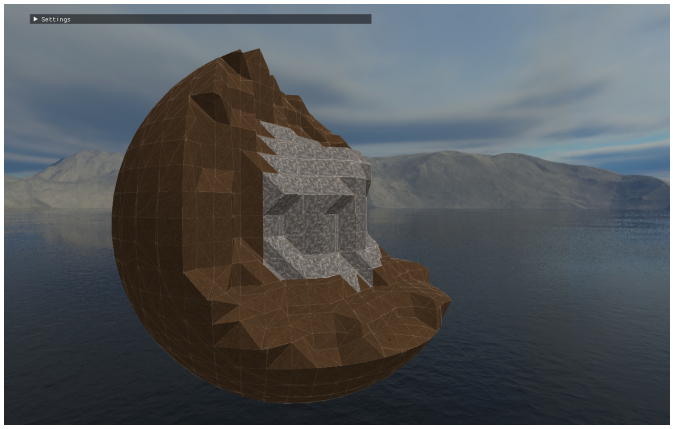
Destructible Meshes with Different Materials Using Marching Cubes

Eddie Krystowski & Joe Krystowski

Rensselaer Polytechnic Institute



Abstract

Traditionally, mesh destruction and deformation are both computationally intensive and complex. We present an approach to destroying and deforming meshes in real time by representing them using a signed distance field and rendering them using marching cubes. We aim to provide an algorithm that would allow destructive "forces" to affect parts of the mesh differently depending on the material that piece of the mesh is made out of.

1 Motivation

Deforming or destroying parts of an arbitrary mesh introduces a number of problems with regards to deciding how to represent the mesh beneath the "outer layer", which is typically not defined in a standard mesh. This is especially difficult when the mesh is composed of different materials which behave differently under force. We present a solution using marching cubes and a signed distance field to create an implicit representation of the mesh which gives a fast and accurate representation of the interior of the mesh. This representation can then be manipulated to simulate an approximation of deforming or destroying the mesh.

2 Related Work

The backbone of our algorithm is marching cubes, an algorithm proposed in“Marching Cubes: A High Resolution 3D Surface Construction Algorithm” by Lorensen and Cline [1], which creates triangle meshes of constant density surfaces from given 3D data. Marching cubes creates a logical cube occupying a 3D unit of space, defined by eight vertices. For each of the eight vertices, one of the 256 triangulation cases is identified, and subsequently is reduced through complementary and rotational symmetry to one of 14 cases (the 15th case is no triangulation, so it is excluded). However, as discussed by Nielson and Hamann in “The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes” [2], the original implementation proposed by Lorensen and Cline suffers from some erroneous results such as parts of the isosurface potentially containing holes.

The reason for this mostly comes down to the original paper reducing the 256 triangulation cases to 14 cases (with symmetry). Many others have explored various avenues to remedy these deficiencies, usually by either reducing to more than 14 cases, examining the triangulation case of adjacent cubes, or, in the case of Neilson and Hamann, to perform bilinear interpolation over each cube face to determine what triangulation case to use. While each of these have merit, we felt given the time constraints and added complexity of many of these solutions, we would instead just triangulate all 256 cases and perform no reduction to symmetrical cases.

Fortunately, the tedious triangulation of all 256 cases has been succinctly presented by Paul Bourke in “Polygonising a scalar field” [3]. Using the triangulation table provided in this document, we can perform fast and accurate triangulation of each cube with no holes in the isosurface.

3 Algorithm

3.1 Signed Distance Field Generation

The marching cubes algorithm requires an underlying scalar field to sample, also known as a signed distance field (henceforth referred to as SDF). For initialization of the SDF we opted with a simple approach, where the user will specify a function defining the implicit surface. This function should have the following properties:

f(x, y, z) < 0: (x, y, z) inside isosurface

f(x, y, z) = 0: (x, y, z) on isosurface

f(x, y, z) > 0: (x, y, z) outside isosurface

It is important that the function is continuous, especially along and inside the

isosurface to ensure proper interpolation between SDF values.

For example, for a sphere of radius r, the implicit function would be:

$$f(x, y, z) = x^2 + y^2 + z^2 - r^2$$

which trivially comes from solving the standard sphere equation for $f(x, y, z) = 0$.

One advantage to this approach is flexibility. It is very easy to define the implicit surface with this method, and by using some clever constructive geometry one can even produce complicated shapes beyond simple primitives.

Initially, we planned on implementing the fast marching method, as described in "3D Distance Fields: A Survey of Techniques and Applications" by Jones, Bærentzen, and Šrámek [4] for initializing and maintaining the SDF. However, due to time constraints caused by the difficulty of the other pieces of the algorithm, our algorithm simply samples the implicit function at each 3D grid point to build the SDF.

3.1.1 Defining Multiple Materials

In our implementation, different materials are defined by their own function. We do not use isolevels to define where different materials are because this limits the definition of materials to be dependent on the base shape, and it would generally prevent us from exposing the lower level materials for editing. Also, additional materials will take precedence over the "base" material which is the default defining the entire isosurface in the previous section. Materials are assigned when the cubes are generated, and the SDF generation functions are used to classify each cube as a certain material. For each material, we define a texture coordinate offset for the texture atlas, as well as the toughness of the material. The tougher the material, the more resistant vertices of that material are to edits. The material of an entire cube is determined by the material of the first vertex in that cube, mostly due to the fact that it made texturing each face easier. However, it is possible to adapt our implementation to define the materials for each vertex in each cube, although we feel this is somewhat redundant since each vertex will always be the first vertex in some cube.

3.2 Marching Cubes

The following section describes the marching cubes algorithm which is performed simultaneously across groups of cubes, each cube being processed independently from others (see section 3.4 Multithreading).

3.2.1 Cube index

After initializing the SDF, we iterate over each sampled point in the SDF and generate a 3D grid of cubes, each cube made up of 8 vertices, corresponding to 8 indices in the SDF. Each vertex has a vertex index, ranging from 0 to 7. For each cube, we calculate a *cube index*, an 8-bit number representing which triangulation case the cube represents. This is done by setting the n-th least significant bit to 1 if the vertex at index n is outside the isosurface, and 0 if the vertex at index n is inside the isosurface. In total, the *cube index* ranges from 0-255, and represents one of the 256 triangulation cases for marching cubes. We can save lots of time by recognizing that cubes with *cube index* 0 or 255 are either completely inside or outside the isosurface, respectively, therefore they do not need to be considered for triangulation.

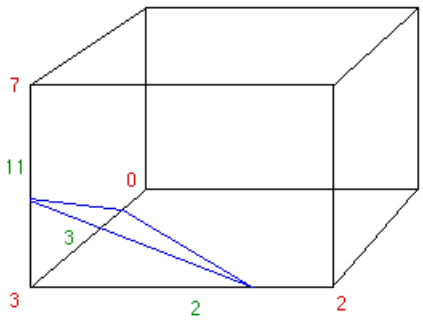


Figure 1. Cube triangulation example (Borke, "Polygonising a scalar field" [3])

As an example, in this image, the cube would have *cube index* = ***0000 1000***, or 8 in decimal, since vertex 3 is the only vertex outside the isosurface.

3.2.2 Edge table

For each cube, we need to know what edges have points on the isosurface, and thus would contribute points to the final triangulation. For this, we perform a lookup in the *edge table* [3], passing in the *cube index* of our cube. The edge table returns a 12-bit number, where similarly to the cube index, bit n is set to 1 if edge index n has points on the isosurface, and is 0 otherwise. Each of the 256 entries in the edge table is succinctly represented using a 3 character hex number, since one hex character is 4 bits. Referring to the example in Figure 1, we would expect that the edge table returns a number where the 11th, 3rd, and 2nd bits are set.

3.2.3 Linear Interpolation

Linear interpolation is subsequently performed on each edge that is marked to have a point on the isosurface by the edge table. This ensures we don't simply assume the point on the isosurface is directly on the midpoint of the line - doing so will lose a lot of detail and will not look good when

rendering with lighting. It is important to note that sampling the SDF in a denser 3D grid does not replace the need to perform linear interpolation, you would still end up with solely 45 degree faces - albeit much smaller ones.

We interpolate the "isosurface intersection point" *I* along an edge (p, q) using a standard linear interpolation:

$$I_x = \frac{p_x + (isovalue - v_p)(q_x - p_x)}{v_q - v_p}$$

$$I_y = \frac{p_y + (isovalue - v_p)(q_y - p_y)}{v_q - v_p}$$

Where p and q are the endpoints for the edge, and v_p and v_q are the SDF values at points p and q. Additionally, *isovalue* lets one determine what the boundary is for being classified as inside or outside the isosurface. As discussed earlier, when generating the SDF (and in the remainder of the algorithm), our algorithm expects the isovalue to be 0.

3.2.4 Generating Triangulation Data

For OpenGL to properly render the resulting mesh and cull back faces, we need to ensure we are using the proper and consistent winding order when loading the triangle vertex data into the vertex buffer. Fortunately, the *triangulation table* provided by Borke [3] includes a consistent winding order for each triangulation case. Given a *cube index*, the *triangulation table* will return groups of 3 edge indices, each group specifying a triangle to make using the isosurface points on each indicated edge, and the order being the exact order the vertices should be loaded into the vertex buffer. Additionally, encountering a -1 in the *triangulation table* means that there are no more triangles left to parse.

Returning back to the Figure 1 example, we see looking up cube index 8 in the triangle table gives

{**3**, **11**, **2**, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}

Since there is 1 group of 3 edge indices, there is only 1 triangle. For the sole triangle, we see that we must insert vertex data corresponding to the isosurface intersection point for edge 3, then 11, and finally edge 2. Note that position data is not the only data loaded into the vertex buffer. The additional vertex data sent to the shader pipeline includes normals for directional lighting, texture coordinates of a texture atlas which gives a visual differentiation between different materials, and the "health" of the vertex.

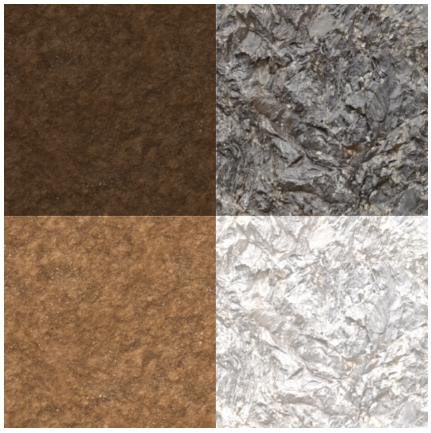


Figure 2. A texture atlas containing textures for dirt and stone materials, and their highlighted versions. Highlighted textures are stored at a vertical offset for ease of switching.

3.3 Editing SDF in O(1) time

Despite not having a specific data structure backing the SDF, we are still able to make edits to the SDF in constant time. This is possible due to the fact that we do not care about the accuracy of parts of the SDF that we can not see in the final marching cubes output. That is, we only need to worry about the correctness of cubes with vertices both inside and outside of the mesh. Therefore, in order to "destroy" a vertex on the mesh, we only need to edit the corresponding point, and those adjacent to it, in the SDF. This can be done in constant time by converting the world position of the location we want to edit into the closest corresponding point on the SDF, then accessing this point and its surrounding points in all eight cubes they are a part of.

3.4 Multithreading

Due to the fact that we are using a triangulation table to calculate the triangle faces generated for each cube, we can generate the triangle faces in parallel without having to worry about holes in the output. In our implementation, we break up our cubes into equal parts depending on the number of threads available, then send each batch of cubes to a new thread to be processed in parallel. One limitation to this approach is the fact that all threads write into the same buffer of triangle data, requiring a mutex lock. Additional speedups could be found if each thread can freely write their data at any time. One difficulty we encountered attempting to solve this problem ourselves is allocating the correct amount of buffer space for each thread. In the worst case, each cube can contain up to 5 triangles according to our triangulation table, and reserving space for 5 triangles per cube produces its own performance and memory concerns. In practice, we found that it was possible to reduce this allocation to an average of 1 triangle per cube, especially for simpler and smaller meshes. However, we did not like this approach because it made

the program theoretically unstable, especially once the mesh was being edited. In hindsight, we possibly could have created a vector for each thread and then copied the results from each thread into the VBO.

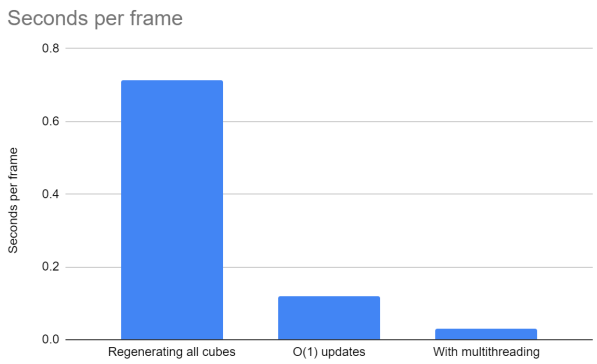


Figure 3. (Left) Seconds per frame regenerating all cubes on an edit. (Middle) Using described O(1) edit. (Right) O(1) edit with multithreading

4 Visualizations

This section describes various visualization aides we implemented to enhance the user experience.

4.1 Vertex Highlighting

In order to help visualize which parts of the mesh the bounding sphere was intersecting for editing, we added an option to enable highlighting of the selected vertices. When enabled, we capture the vertices in the bounding sphere and add an offset to their texture coordinates which adjusts their texture to the highlighted version of their current texture in the texture atlas. We decided to go with adding an additional "highlighted" version of each material's texture in our example since it would reduce the size of the vertex data, however if this is not possible it would also be just as effective to modify the directional lighting value in the fragment shader to give a highlighted look as well. Unfortunately, because highlighting the selected faces does require updating the VBO, this does have performance impacts which we will discuss later.

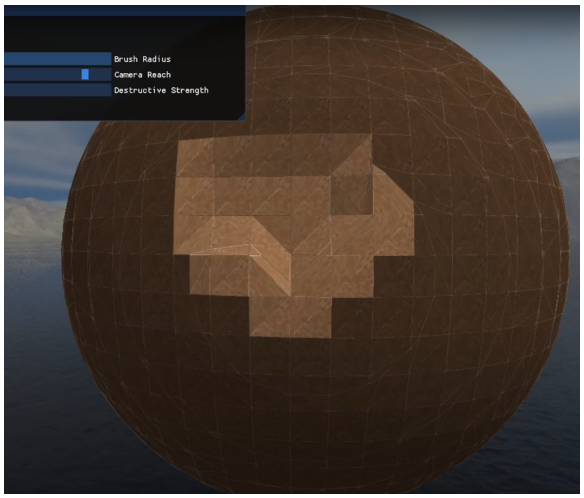


Figure 4. Vertex highlighting on a sphere.

4.2 Health Shader

In our implementation, each vertex has a health value which is lowered by a variable

amount when editing depending on the strength of the material at that vertex. In order to visualize the “damage” being done to a vertex, we added the ability to enable a health shader. The health of a face is defined by the health of the vertex with the lowest health on the face. For each vertex, we pass along the health as vertex data into the shader. In the fragment shader, we do linear interpolation on a red and green color depending on the health at that vertex, and mix the resulting color with the usual output of the fragment shader to achieve a red or green tint on the texture of that face. Note that each vertex is included in multiple different faces, and may have different health values for each face it is included in.

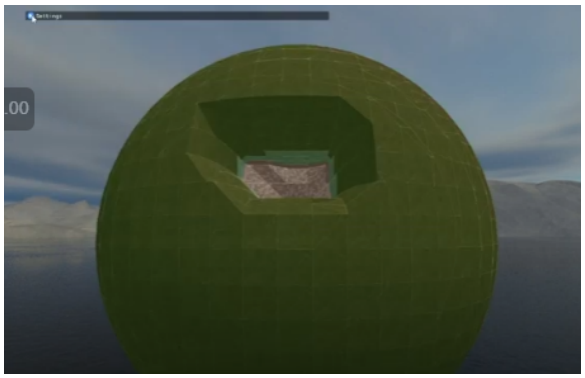


Figure 5. Health shader enabled on a sphere with dirt and stone materials.

5 Results

5.1 Destroying With Multiple Materials

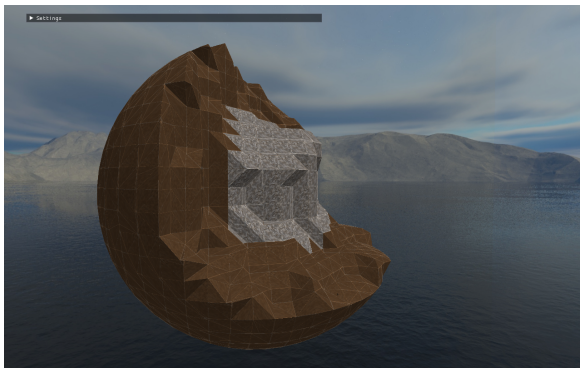


Figure 6. A sphere with dirt and stone materials after applications of destructive forces.

In Figure 6, we define a spherical mesh with a dirt exterior and stone interior. The stone on the inside is more resistant to the destructive forces, causing it to remain mostly intact while the dirt exterior has been destroyed.

5.2 Rendering Complex Shapes

Our marching cubes implementation is capable of rendering more complex implicit surface functions, including surfaces with holes in the center (torus, Figure 7) or surfaces that are not closed (wineglass, Figure 8). Both of these implicit surface functions were taken from Wikipedia [5]. We did not specify multiple materials for these isosurfaces, although we could have.

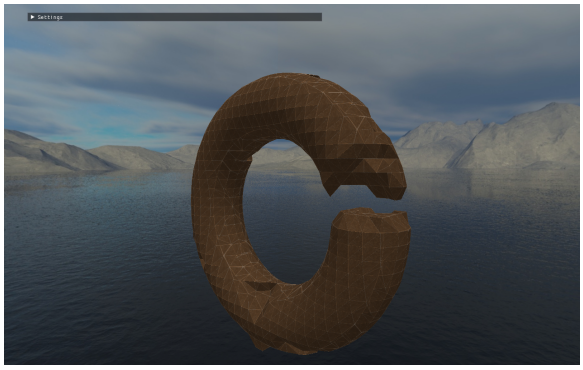


Figure 7. Rendering of a torus isosurface using marching cubes which has been partially destroyed using our editing algorithm.

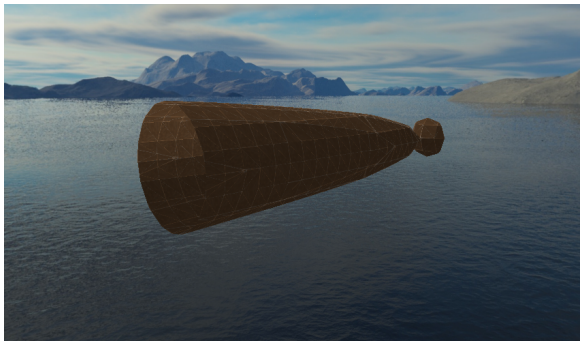


Figure 8. Rendering of a wineglass isosurface using marching cubes. There is an opening in the mesh where you would drink from

6 Limitations and Bugs

6.1 45 Degree Angles

While destroying pieces of the mesh works as expected, there are some limitations/bugs to the representation of the resulting mesh which causes the modified regions to have a significant number of 45 degree angles, which should be able to be avoided using signed distance fields, as is done on the initial outside surface of the isosurface. The main factor causing this issue is that when we use our bounding sphere to make edits on the mesh, we did not take into account the shape of the bounding sphere when updating our SDF. Instead, when a vertex is deleted, we erroneously made the assumption that this vertex and the interior vertex were equidistant from the surface. While this does produce a decent result, the more appropriate solution should be that for vertices along the edge of the bounding sphere, we should examine how close the vertex is to the surface of the bounding sphere and use this information to create a more accurate SDF value for the surrounding vertices. This would allow us to remove many of the 45 degree angles introduced through destroying vertices, and the resulting deformation in the mesh would more closely resemble the shape of the bounding sphere used to destroy the vertices.

6.2 Highlighting Mode Performance

Another limitation is the significant performance impact of the highlighting option for meshes with a large number of faces. While this option is very helpful for

users, we did have to add the ability to disable it for users wishing to work in real time with larger meshes. When a triangle face is highlighted or un-highlighted, we need to update the vertex data for those triangles. In our current implementation, it is not possible to change the vertex data for just one triangle, so we regenerate all vertex data for the triangles and the VBO, which can be a prohibitively expensive computation to do every frame for large meshes.

7 Future Work

7.1 Level of Detail (LOD)

Given that performance is greatly affected by meshes with a large number of triangles, even if you are not making edits or highlighting vertices, in the future we are interested in implementing some kind of tesselation shader or other LOD system for rendering our implicit surfaces.

7.2 Fast Marching Method

We would also like to eventually implement the fast marching method to generate and update the SDF. This can also potentially allow us to have users load in a mesh from a .off file or some other format, and generate an SDF based off of that mesh. This could be helpful since it would be difficult to determine an accurate looking implicit surface function for a given mesh without external tools.

7.3 Generative Editing

Given that we can destroy the meshes now, future work could involve adding the ability to build onto the mesh instead. We ran out of time to implement this ourselves, although it is definitely feasible using this project as a foundation. The result would essentially be a basic sculpting tool using marching cubes.

8 Appendix

In total, this project took around 50 hours of work from each of us. Most of the work was done in a pair-programming style, so nobody in particular was in charge of a single section of code, except for the CPU multithreading implementation entirely done by Joe. In the meantime, Eddie worked on an ultimately scrapped GPU multithreading implementation using compute shaders, which unfortunately we ran out of time to get working.

9 References

[1] William Lorensen and Harvey Cline, 1987. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm", SIGGRAPH ‘87.

[2] Gregory Nielson and Bernd Hamann, 1991. “The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes”. Visualization ‘91.

[3] Paul Borke, 1994. https://paulbourke.net/geometry/polygonise/

[4] Mark Jones, J. Andreas Bærentzen, and Miloš Šrámek, 2011. “3D distance fields: a survey of techniques and applications”, Transactions on Visualizations and Computer Graphics 2011.

[5]”Implicit Surface” https://en.wikipedia.org/wiki/Implicit_surface

[6] Joey de Vries https://learnopengl.com/