An Implicit Surface Editor using Marching Cubes

A Final Project by Dylan Lawn and Jordan Faas-Bush

RPI Advanced Computer Graphics Spring 2019

1) Introduction

Defining geometry with implicit surface algorithms allows meshes to be created using constructive solid geometry. Through utilizing intersections the constructive interference between implicit surface definitions, we are able to create more natural transitions between shapes being added together. Using this technique, we endeavored to create a mesh editor that allows users to create organic shapes out of simple primitive shapes in an additive and subtractive manner. We created a file format for saving and loading geometry definitions losslessly, as well as the ability to export files in .obj format for usage in other applications.

1.1) Related works

There are several related works covering parts of this editor, but few that combine them. *Marching Cubes* by Lorensen and Cline discusses a method of triangulating meshes from constant density surface definitions, and was used to create meshes out of the final combined implicit surface definition data.

Other papers described metaballs and implicit surfaces. *A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space*, by Gilbert, Johnson, and Keerthi, describes an algorithm that can be used to efficient determine whether two implicitly defined convex surfaces are colliding with one another, and to what extent, using Minkowski sums. While less applicable to the development of the editor, the paper was essential for understanding implicit surfaces and inspiring this project.

Interactive Techniques for Implicit Modeling by Bloomenthal, Jules, and Wyvill, covers ways to more efficiently render metaballs, as well as design considerations for implicit surface editors such as skeleton construction and visualization techniques besides Marching Cubes that

can be used for more efficient rendering during real-time editing. While this paper opted to go with the tried and true method of marching cubes, *Interactive Techniques* provided alternatives which could be used in the future for alternate mesh generation.

2) Model representation

The model is stored in two forms, the first is the “perfect” model storing the representations of the implicit surfaces. The implicit representation is shown to the user as simplified colored objects in the scene and is used for designing the model. This model can be saved to a file formatted with the JSON standard which is platform independent and can be shared between any user of the software. This is the only representation that the user can directly modify and is where the bulk of the user’s designing takes place. This implicit data is then fed to the marching cubes algorithm which converts it to an instance of a Unity Mesh class, which is rendered to the screen as a standard triangle based 3D model. Aside from visualizing what the final result will look like, the Mesh class and representation is edited by the smoothing algorithm, and is eventually what gets exported to the .obj file format for use by other applications.

2.1) The Implicit Hierarchy

The implicit representation is stored as a hierarchy of Implicit Surfaces (although Implicit here is used perhaps a little too broadly). Each object is either additive or subtractive, which either adds to the resulting output mesh or subtracts from it. The hierarchy is made up of three types of objects -- the Implicit Sphere, the Implicit Cube, and the Implicit Intersection. Each of these objects has a local location, a local rotation, and a local scale. Each of these transformations are applied recursively on top of those above it in the hierarchy, allowing users

to create attached “limbs” and similar objects with positions dependant on those of the items higher in the hierarchy. This hierarchy can be saved to a JSON file that can then be loaded on that same computer or any other computer using the editor program.

2.2) The Implicit Surfaces

The first surface implemented in the editor and quite possibly the most useful is the implicit sphere. The Implicit Sphere contains the coordinates, rotation and scale as all implicit surfaces do, along with a radius (although because the sphere can be scaled, this has been set to a constant value of one). This surface is aptly represented in the model editor as a cube model with a radius of 1.

The implicit cube has side lengths of 1. Unlike the Implicit Sphere, the rotation of the cube matters even when it is not scaled. This surface is represented as a cube model in the model editor.

The Implicit Intersection is quite different from the other two object types. Rather than having an impact on the output mesh by itself, this object creates a modifier to the children under it in the hierarchy. The additive intersection object returns the intersection of any of its children as an additive influence. The subtractive intersection object takes that same result but uses it as a subtractive influence for those items higher up in the hierarchy. This object has no visual representation in the scene view of the editor, only appearing in the hierarchy view.

2.3 The Mesh Representation

The game engine Unity has a built in representation of a triangle mesh that stores vertices and triangles, and their associated information such as normals, bones, and vertex

colors. This was the obvious choice for mesh representation because of its easy integration into the Implicit Surfaces Editor. The representation did provide slight performance issues however due to its lack of connectivity information as is mentioned later in this paper. The mesh *can* be exported to an .obj file by the editor but the editor is unable to import them. However, there is no reason why the mesh cannot be rebuilt by the editor using the same input file and the same mesh building settings.

3) Features:

This editor is fully featured and is ready to be used to create models for other projects. While the output mesh is not always the most efficient, the simplicity of model creation may very well make up for it. The editor allows users to add additive or subtractive cubes, spheres, or object intersections to the scene, adjust their locations, scales, and rotations handles or an editor panel. The editor allows users to select multiple objects and edit them in bulk or individually using handles or using an editing panel. Users can convert the implicit surfaces into a triangle mesh using several adjustable modes, smooth the mesh, and then export it into an industry standard file format. The user can then save the editor file for use later or elsewhere.

This editor supports a complete workflow for the end user from inspiration to a resulting output model. While the editor does not yet support features such as UV unwrapping, rigging and mesh simplification, free (or paid) tools such as Blender or Maya are easily able to import the created .obj files and complete such tasks. Because of this, while the editor may not be able to create production ready models by itself it is more than ready to create models for prototypes or simpler applications, and the models created can be improved with easily accessible software.

3.1) Testing

The editor and algorithms were tested through the creation of numerous models. The simplest test case which uses most of the features of the editor is creating a “moon” pockmarked with craters. This simply tests the additive and subtractive surfaces.

A more complicated test is creating a more complicated shape such as a giraffe or a bunny. These can combine additive and subtractive shapes for features such as eye-sockets or limbs.

Additional tests involve creating things like dice to test using the intersection object, or simply some very complicated model to test all of the above.

Finally it’s important to test saving and loading the editor files, and then testing loading the exported models in outside programs such as Blender to ensure compatibility. The exported models were tested in the Blender modeling program as well as by loading them back into Unity 3D and succeeded.

4) Algorithms

4.1) Marching Cubes

For the rendering of implicit surfaces defined by the shapes defined in the editor, we utilize the Marching Cubes technique. The functions for all of the surfaces are evaluated on a three-dimensional grid of user-defined size and passed in to the March function. We included an option to either cache the result of the surface definitions into a list, or evaluate them during the iteration, allowing for a tradeoff between memory usage and iteration performance. The grid is iterated over per cell and evaluated at each point to find where the surfaces intersect the cube

at that point. A table of possible triangle configurations, and another of resulting edge layouts, is queried to determine the resulting triangulation of that grid cell. We utilize bit flags to store the result of comparing each cube point against the isosurface and determine the correct vertices in the table. In order to avoid the issue of duplicate vertices, we keep a dictionary mapping existing vertex positions to indices in the vertex array, and only add unique positions to the array.

4.2) Smoothing

The original implementation of our algorithm utilized a fixed isosurface value of 0.5, which resulted in the mesh having a blocky appearance. In order to reduce this and to provide the user with more creative control, we added a smoothing algorithm. Iterating over each triangle position defined in the vertex array, we check the position of its neighboring vertices and find the average of their position. We then interpolate between the average neighbor position and the old vertex position by a user set value to get the new position of that vertex.

4.3) Additive/subtractive hierarchy

There are two hierarchy evaluation modes the user can switch between when evaluating the values at the points of the marching cubes algorithm. Both modes are evaluated recursively from items at the top of the hierarchy with no parents to the leaves. Each object first converts the coordinates from worldspace to local space by applying the inverted position transformation, rotation transformation, and scaling transformation to the queried point.

The first is a simple boolean system, resulting in each point being either inside or outside of each hierarchy cube or sphere. In this mode the intersection object simply returns whether or not the point in question is inside all of its children. The hierarchy evaluates from the root of the

tree to its leaves, with the highest elements taking precedence over the lower elements. Because of this, subtractive elements only apply to objects above them in the hierarchy, allowing for creating “eye sockets” and other similar structures. The benefit of this mode is that it allows creating models very similar to the representations the users move around in the editor. The drawbacks include the fact that this binary representation does not allow for a smooth interpolation of the vertex values in the marching cubes algorithm.

The second mode creates a smoother scalar field than the first by using a falloff function often used with metaballs instead of the binary inside/outside evaluation of before. The cube and sphere all have a falloff function from their edges. The intersection acts as a minimum function, returning only the minimum value of any of the children. The pros include smoother shapes and metaball evaluation, the cons include a disconnect from the editor UI and the mesh results, and that it often requires fine tuning the isosurface parameter in order to create useful results.

5) Editor user interface

The user interface is divided into four primary sections: the scene view, surface inspector, hierarchy, and settings menu. The scene view is the viewport into the world space containing the model being constructed. It can render the model through both representations of the raw individual implicit surfaces, as well as a marching cubes rendering of the resulting cumulative mesh. Users can translate and rotate the camera around the scene in order to edit the model from different viewpoints. Different surfaces can be selected with the mouse and can be manipulated with several different tools: position, scale, and rotation. “Gizmos” are rendered for each of these tools and can be used to manipulate the selected surface with the mouse.

The surface inspector is used to edit the position, scale, and rotation of the selected surface in the scene numerically. In addition, it can be used to change the type of shape the surface defines, as well as change its type between additive and subtractive, and edit its name.

The hierarchy panel is an important feature of the editor because it determines the order in which additive and subtractive surfaces are applied to the final mesh definition. In order to enable surfaces nested inside of spaces created by subtractive surfaces, subtraction is only applied to surfaces above the subtractive surface's parenting level.

6) Challenges:

The marching cubes algorithm constantly caused hiccups in the development of the editor. The first issues were with simply implementing it and the numerous cases. The next issues were with determining how the hierarchy should factor into mesh generation -- should subtractive and additive surfaces only apply to those items above them on the hierarchy or to every item? This difficulty is part of the reason why the final editor ended up with multiple different modes for generating the output mesh. After the initial marching cubes function started working the high resolution meshes generated sometimes had strange artifacts which were eventually determined to be caused by reaching the vertex limit of Unity's mesh representation (65,535 vertices) due to the initial implementation not checking for duplicate vertices and the high resolution. While these issues were eventually fixed they took time (and also created interesting failure cases).

There are only two known bugs in the program. The first is somehow caused by rotating parent objects and swapping back and forth between selecting the parent and child object. This somehow moves the child object to some strange location for some unknown reason. This problem can be solved by unparenting the child after the desired rotations have been achieved

(which has been programmed to keep the same world transformations). The second bug is an issue with the bounding boxes of the implicit shapes when rotated. When the shapes are rotated they can expand beyond the calculated bounding boxes which results in the generated mesh being cut off. This has been patched though by providing the user with a variable that signifies how much to expand the bounds by, allowing the user to expand the bounds manually if they are insufficient.

While it is not a surprise, the stretch goal of automatically rigging the model remained just that -- a stretch goal. It was only realized too late in the project that exporting the rig and weights wasn't as essential as simply creating them in the Unity Mesh representation which supports them by default. While the resulting rigged models would not have been compatible with modeling software, they likely would have been compatible with other Unity projects and certainly could have been animated in the Unity editor.

One of the core goals for the developers of this project was to create a complete standalone application that could be used outside of the Unity editor. This was almost entirely completed, with almost every feature accessible to the end user in a built version, with only unessential parts missing. While unessential, some of the missing variables like the marching cubes mesh offset would have been useful to expose to the end users in the built version. While it was a challenge to make this a completed application it was almost entirely surmounted.

7) Future Improvements:

This editor provides almost a complete workflow for end users. The next obvious steps are to complete the workflow. UV unwrapping is significantly more complicated than it first appears for arbitrary meshes, but due to the more regular nature of the marching cubes algorithm it could be simplified. Rigging and weight painting is also a very difficult problem, while

they do save time, much of the challenge for rigging comes from fixing the issues the automatic rigging created. Aside from additional features to complete the modeling pipeline, the marching cubes algorithm could be improved to better align to the surfaces, and more implicit shapes and surfaces could be added for additional modeling ability.

Appendix A: Division of work

All times are approximate

Dylan

Exporting meshes 2 hours

Marching cubes 8 hours

Jordan

Saving and loading editor files

Smoothing meshes 2 hours

Editor UI 7 hours

Hierarchy 2 hours

Implicit Surface Point Evaluation V1 4 hour

Implicit Surface Point Evaluation V2 5 hours

Bibliography

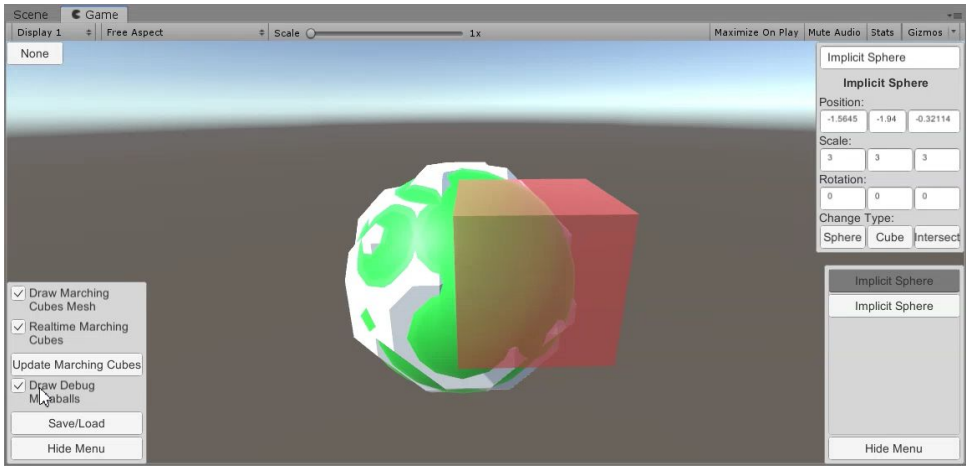
Lorensen, William E., and Harvey E. Cline. “Marching Cubes: A High Resolution 3D Surface Construction Algorithm.” ACM SIGGRAPH Computer Graphics, vol. 21, no. 4, 1987, pp. 163–169., doi:10.1145/37402.37422.

Gilbert, E.g., et al. “A Fast Procedure for Computing the Distance between Complex Objects in Three-Dimensional Space.” IEEE Journal on Robotics and Automation, vol. 4, no. 2, 1988, pp. 193–203., doi:10.1109/56.2083.

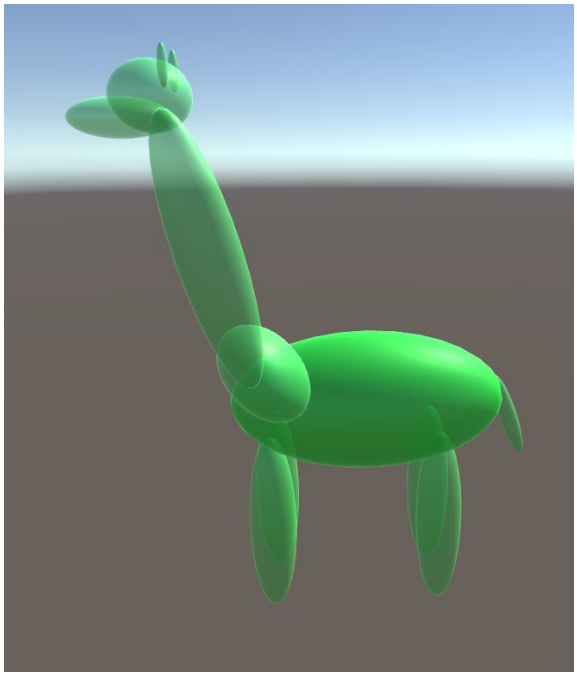
Bloomenthal, Jules, and Brian Wyvill. “Interactive Techniques for Implicit Modeling.” Proceedings of the 1990 Symposium on Interactive 3D Graphics - SI3D '90, 1990, doi:10.1145/91385.91427.

Figures:

Success: test case 1 of collisions in a moon. One of the first successful mesh generation results using a subtractive mesh to remove from an additive mesh



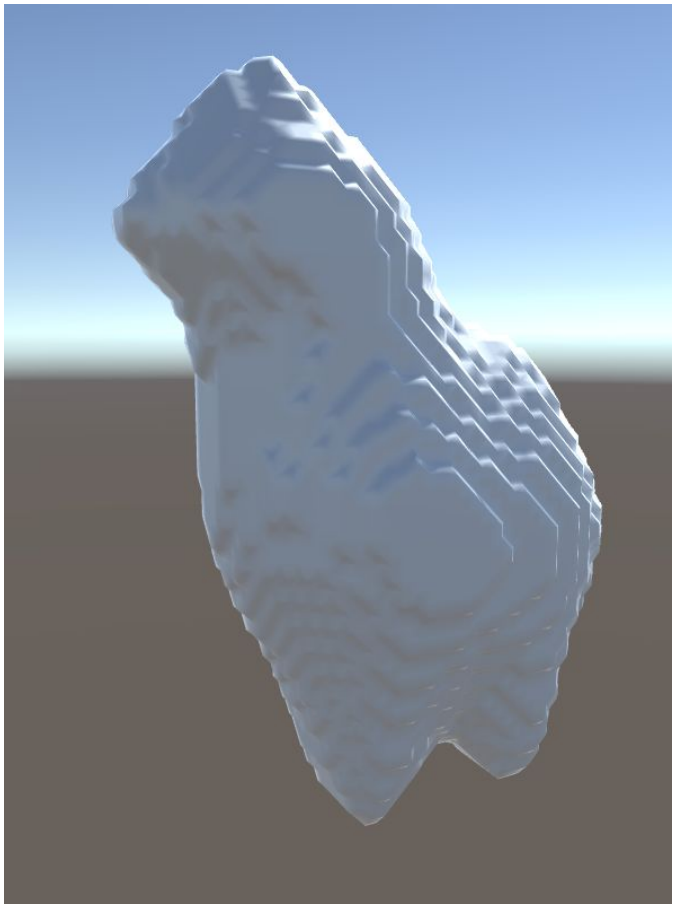
Success: Test case 2 generating a creature mesh. This figure shows the created giraffe using the default implicit surface visualization



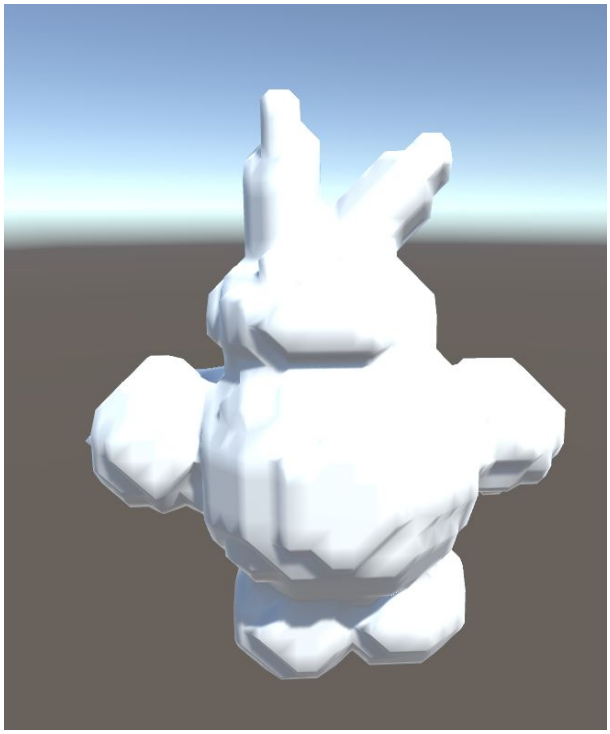
Success: Test case 2 generating a creature mesh. This first figure shows using the binary mesh mode:

Success: Test case 2 generating a creature mesh. This next figure shows using the implicit surface mode:

Success: The above giraffe with a larger isovalue:



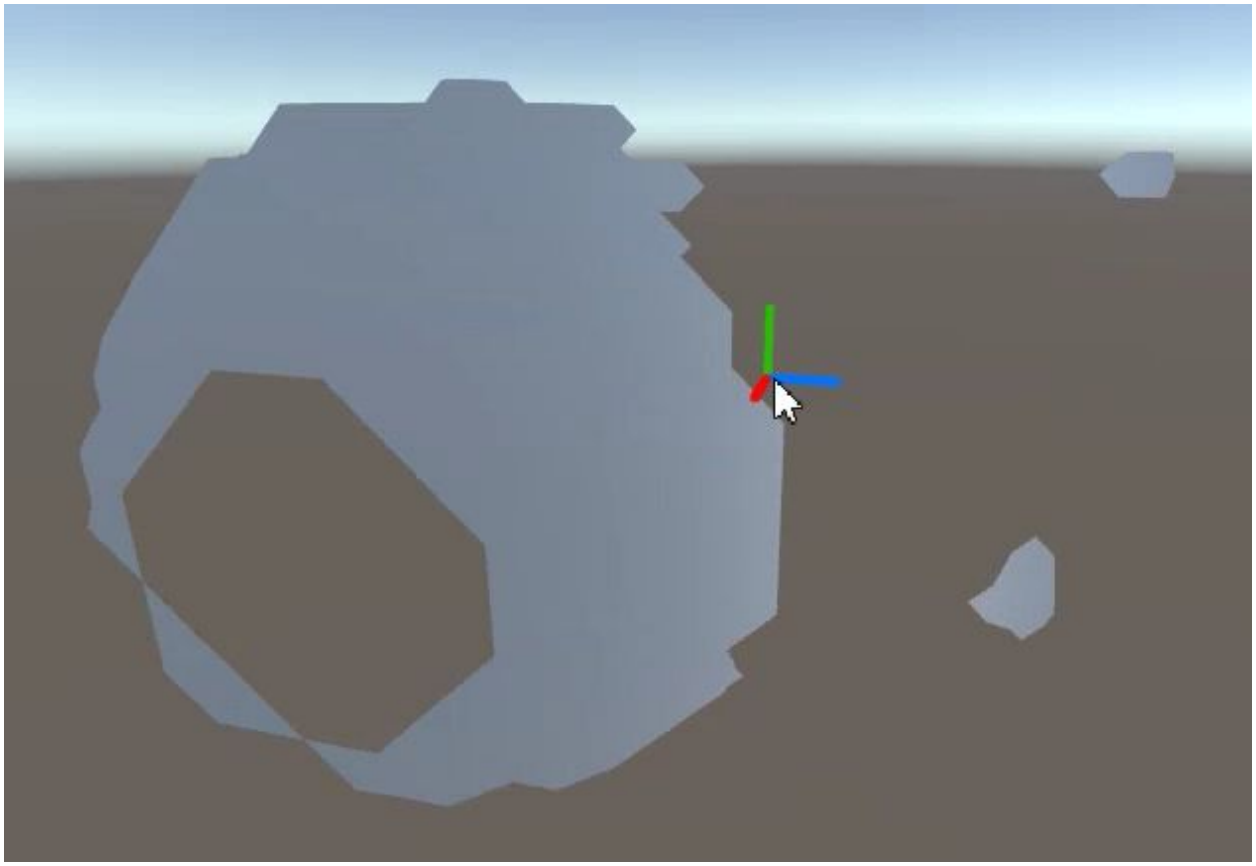
Success: A bunny model



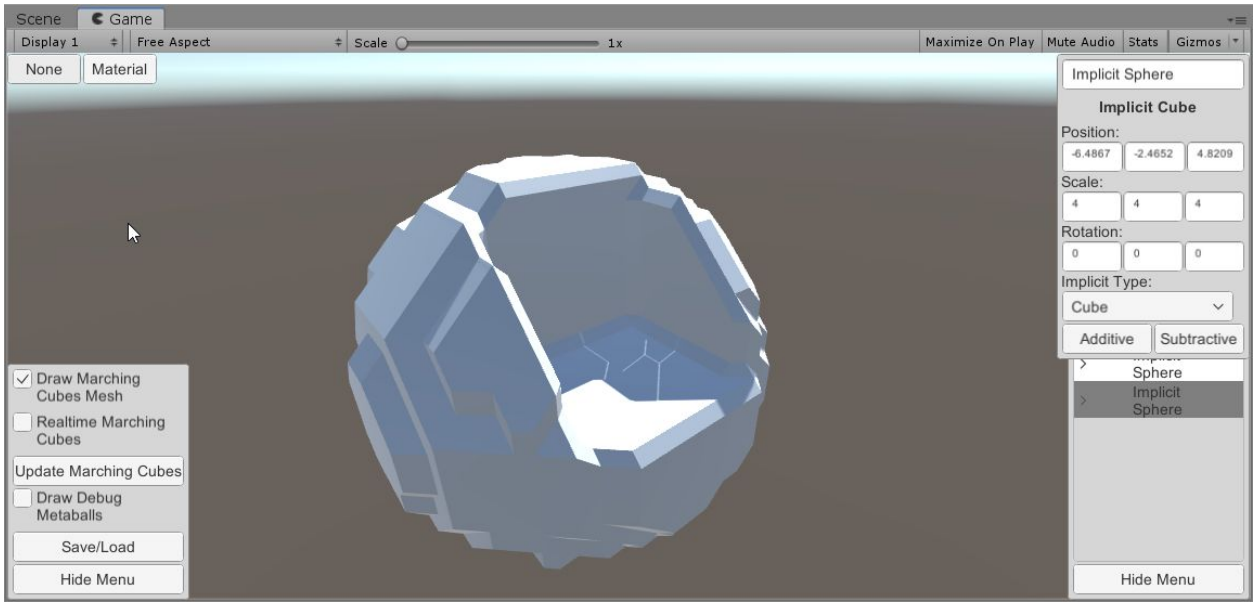
Success: An implicit intersection used to create a rounded dice

Success: a smoothed dice:

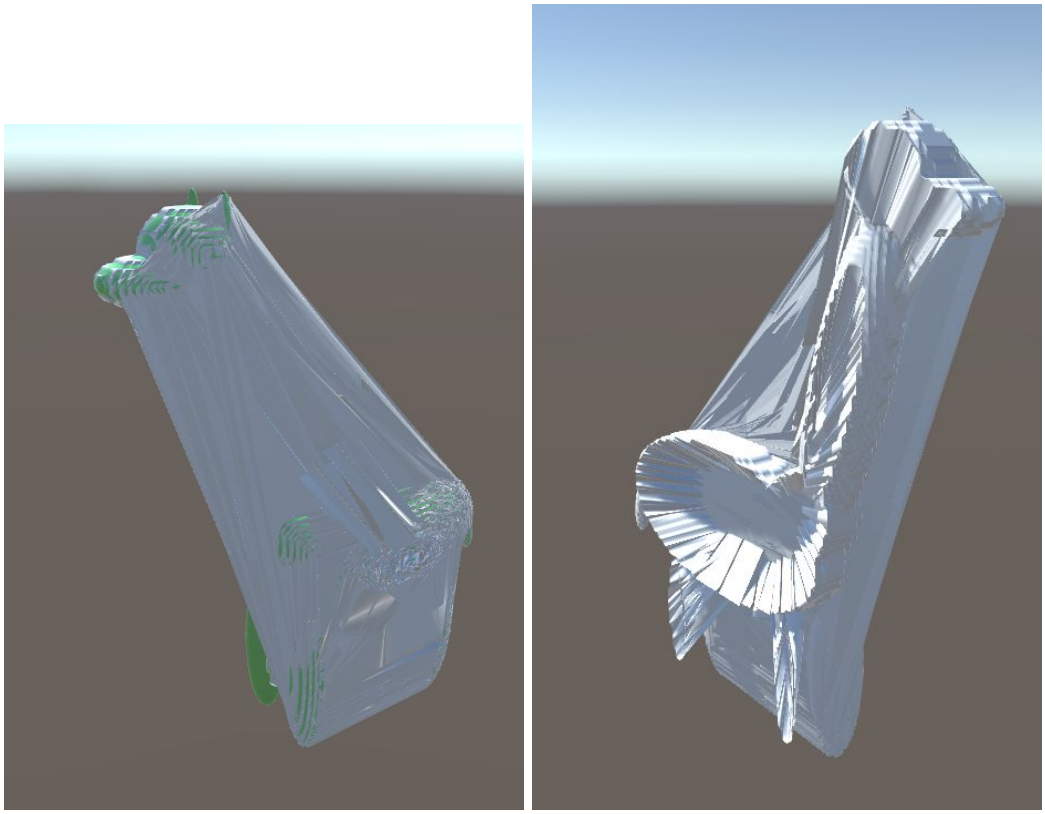
Blooper: failure to recalculate normals after mesh generation



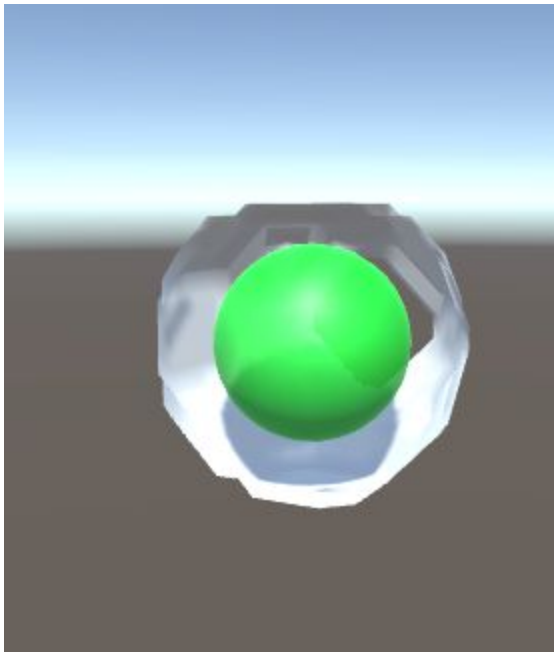
Minor blooper: prior to implementation of vertex sharing, light could appear through “cracks” between the triangles



Failure cases: These meshes had too many vertices for the Unity Mesh representation:



Failure case: an incorrect evaluation function for the implicit surfaces results in the mesh being built inside out



Failure case: A weird result from the implicit surface function at a low resolution