Winding Number Computation for Triangle Soups and Point Clouds

Erwin Pimentel Daniel Scrivener Stanley To

Boston University
Spring 2022

Problem Statement

We aim to explore novel approaches to mesh voxelization, specifically with the use of generalized winding numbers. In our work, we build upon prior course experience using a naive raycasting method for voxelization, in which we cast a ray from each voxel center and count the number of surface intersections as a means of achieving inside-outside segmentation [5]. This implementation lacks a proper user interface, making it difficult to interact dynamically with meshes and change voxelization parameters. Furthermore, this method proves to be computationally expensive, with an execution time on the degree of hours for certain high-resolution voxelizations. Thus, our objectives for this project are to (1) extend our naive implementation to the OpenFlipper interface to allow for users to engage with voxelization of meshes using different parameters and (2) leverage generalized winding numbers for inside-outside segmentation to improve efficiency and robustness.

Background

From character animation to physically-based simulation, mesh boundaries are often used to represent solid 3D objects in the field of computer graphics. Inside-outside segmentation is a crucial part of interacting with meshes, as it is frequently used for a handful of applications: generating signed distance fields, voxelizing volumes, and recovering smooth surfaces from point clouds. However, meshes often contain defects such as open boundaries, self-intersections, and non-manifold pieces due to the fact that many meshes are constructed from imperfect datasets or with artistic aims that do not align with strict geometric guidelines [1][2]. As such, it is necessary to account for these categories of mesh aberrations. Generalized winding numbers extend our understanding of the classic 2D winding number in order to account for the aforementioned defects in oriented triangle meshes.

The traditional winding number is a signed, integer-valued property of a point p with respect to a closed curve C; intuitively, it can be thought of as the number of times C wraps around p in the counterclockwise direction [4]. More formally, the winding number is equal to the signed length of the projection of C onto the unit circle centered at p. The winding number can be generalized to three-dimensional space by describing it as the signed surface area of the projection

of a continuous surface onto the unit sphere centered at p [2]. In the discrete case of polygonal meshes, this method of calculating the winding number involves summing the signed solid angles subtended by each surface patch.

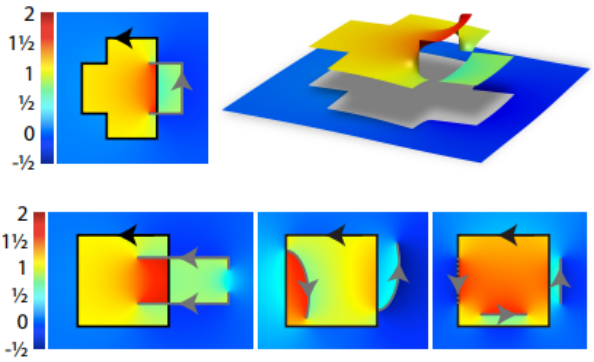


Figure 1: The winding number intuitively captures self-intersections (top) and holes (bottom). [2]

Importantly, the generalized winding number is harmonic everywhere except across the boundary, which allows it to gracefully handle holes, non-manifold attachments, and self-intersections [2]. The harmonic property also allows us to experiment with and fine-tune the threshold value in order to achieve a visually desirable inside-outside segmentation.

Methodology

Groundwork

With an eye toward voxelization, our initial work consisted of porting parts of a suitable C++ mesh processing framework provided by Professor Whiting through *CS581: Computational Fabrication* to OpenFlipper. Said framework provides built-in methods for generating a mesh representation from a set of points with associated information regarding their solidity. Once this had been accomplished, other functions were written to work within the overall software pipeline: voxelization query points are determined by dividing the axis-aligned bounding box of each mesh into a number of cubic segments matching the user-specified resolution. The center of every such segment corresponds to a query point. Data structures were designed to store the mesh elements: a framework for representing the voxel grid was similarly adapted from *CS581*.

Rather than dealing with the performance overhead involved in constructing a matrix and computing its determinant with libraries such as *Eigen3*, we instead laid out all calculations as making use of standard C++ arithmetic operators. Operations between vectors are facilitated by the ACG library packaged with OpenFlipper.

OpenMesh, another bundled library, was used to import and manage mesh data. We were able to feasibly traverse surface patches and points by leveraging the performance of its proprietary halfedge data structure. As voxelization was intended to be a visual demonstration of the underlying mathematical process, we did not use OpenMesh to emit the updated mesh object due to the

massive space complexity of our mesh outputs. However, the resulting meshes can be imported into OpenFlipper via OpenMesh in the same way as any other Wavefront object (.obj).

The plugin interface was modeled after the DGPExercises plugin designed for *CS599 A1: Geometry Processing*, which in turn exposes the Qt framework on which OpenFlipper's interface is built. This provides a convenient method for the user to generate a signal to initiate either voxelization method and for specification of parameters (resolution of voxelization, solidity threshold). We chose to allow manual specification of these parameters as a means of giving the user control over the results: voxelization resolution determines how many cubic elements will be used to subdivide each spatial axis, whereas the solidity threshold determines the winding number value required for a voxel to be marked as belonging to the mesh interior.

Winding Number Computation for Triangle Meshes

Having laid the groundwork for voxelization, the first major project milestone was to implement the foundational winding number algorithm for triangle soups as introduced by Jacobson et al. [2]. This method of computing the winding number for a given query point involves determining the signed solid angle subtended by every discrete surface patch:

$$w(q) \approx \sum_{i=1}^{m} \frac{1}{4\pi} \Omega_i(q)$$

Figure 2: Discretization of the winding number for triangle soups.

for m surface patches where q is the query point. In our implementation, surface patches correspond to elements of the triangle mesh, meaning that all solid angles could be computed via a standard series of calculations. The following formula was used to determine the tangent of the signed solid angle, whose magnitude could be determined using the standard atan function [3].

$$\tan(\frac{\Omega(q)}{2}) = \frac{\det(\begin{bmatrix}\mathbf{a} & \mathbf{b} & \mathbf{c}\end{bmatrix})}{abc + (\mathbf{a}\cdot\mathbf{b})c + (\mathbf{b}\cdot\mathbf{c})a + (\mathbf{c}\cdot\mathbf{a})b}$$

$$\mathbf{a} = v_0 - q, \mathbf{b} = v_1 - q, \mathbf{c} = v_2 - q, a = \|\mathbf{a}\|, b = \|\mathbf{b}\|, c = \|\mathbf{c}\|$$

Figure 3: Calculation of the signed solid angle.

Winding Number Computation for Point Clouds

The work of Barill et al. in generalizing the notion of winding numbers to point clouds provided the direction for our improved implementation [1]. Notably, a formula for the contribution of individual points toward the winding number was derived from the underlying definition in Figure 2 as follows:

$$w(q) \approx \sum_{i=1}^{n} a_i \frac{(p_i - q) \cdot n_i}{4\pi \|p_i - q\|^3}$$

Figure 4: Discretization of the winding number for point clouds.

This sum is computed for all points p_i in a set of n points oriented with respect to a query point q. a_i represents the geodesic Voronoi area of the 1-ring neighborhood around the current p_i, which can be approximated without connectivity information by constructing a ring from the k-nearest neighbors with similarly-oriented normals. Given that our aim was to improve upon the running time of the approach described by Jacobson et al., we elected to treat the Voronoi area as a constant for every point rather than employ another library, TRIANGLE, for the task of estimating the Voronoi area. This provided us with intuition regarding the sensitivity of the method to the accuracy of this parameter, which Barill et al. found to be of lesser importance [1].

Data Collection

One of our goals was to analyze and compare the running times of various voxelization algorithms on standard datasets. To this end, we added timing controls to our code using the standard C++ *chrono* library in order to obtain runtimes accurate to one thousandth of a second. All tests for which execution time was measured were conducted on an ARM-based MacBook Pro (M1 Pro, 16 GB LPDDR5). Since the OpenFlipper framework can be ported to many different systems, some visual results were obtained on a desktop running Windows (Intel i7-11700F, 16 GB DDR4). Though we did not attempt to compute standard deviation across multiple trials, we can report informally that identical tests were run on separate occasions with only milliseconds of difference between their execution times.

Results

Execution time

The processing time of both algorithms as well as the baseline raycasting method was measured for a variety of standard meshes. Some meshes were provided as part of *CS599*; some were obtained from the Stanford 3D Scanning Repository; others were modeled in Sketchup. We selected a wide variety of meshes with interesting geometric features and varying polygon counts. Our results are as follows:

Voxelization by Raycasting Technique (res = 32)		
Mesh	No. Triangles	Processing Time (ms)
Icosahedron	20	139
Sphere	960	7123
Teapot	2464	17659
Bunny #1	1000	7279
Bunny #2	10000	71963
Bunny #3	69664	506532 ($\sim$ 8 min)

Voxelization by Direct Evaluation of Winding Number (Trimesh)			
Mesh	No. Triangles	Resolution	Processing Time (ms)
Bunny #1	1000	32	7370
Bunny #1	1000	64	59277 ($\sim$ 1 min)
Bunny #1	1000	128	471976 ($\sim$ 8 min)
Teapot	2464	32	17660
Max	20196	32	143033 ($\sim$ 2 min)
Armadillo	345944	32	2698200 ($\sim$ 44 min)

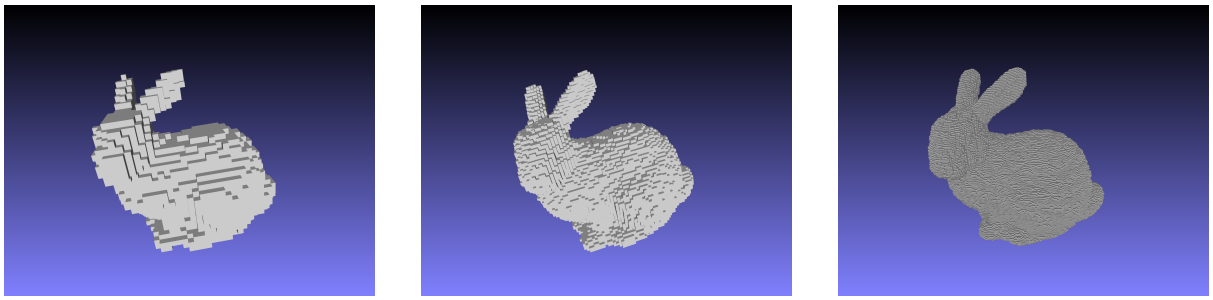
Voxelization by Direct Evaluation of Winding Number (Point Cloud)			
Mesh	No. Vertices	Resolution	Processing Time (ms)
Bunny #1	502	32	444
Bunny #1	502	64	3462
Bunny #1	502	128	27852
Bunny #3	34835	32	30016
Teapot	1292	32	1115
Max	10227	32	8777
Armadillo	172974	32	150593 ($\sim$ 2 min)

Crucially, the point cloud algorithm is much faster than other methods at the expense of accuracy, as indicated by the visual results. The triangle mesh algorithm and raycaster are approximately even in terms of running time.

Both experimental algorithms lend themselves to parallelization due to the fact that the contribution from each surface patch or point can be calculated independently [1]. This optimization would greatly improve the execution time despite requiring additional computing resources.

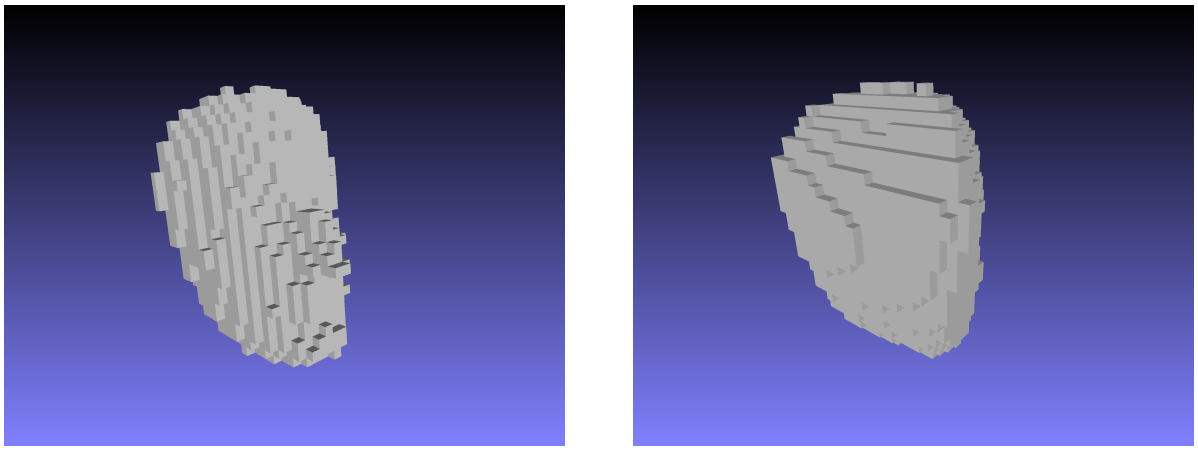
Visual Results

A major advancement of both experimental algorithms is their ability to produce reasonable results from meshes with aberrations such as self-intersections and holes. Whereas raycasting requires a watertight mesh due to its sensitivity to the randomly-selected ray, the experimental algorithms process primitive elements individually without regard to their connectivity. Well-conditioned, detailed triangle meshes such as the Stanford bunny provide superb results with all three methods, but the point cloud algorithm provides an edge in terms of performance.



(a) res = 32 (b) res = 64 (c) res = 128

Figure 5: Comparison of trimesh algorithm run on *bunny.obj* for three different resolutions (thresh = 0).



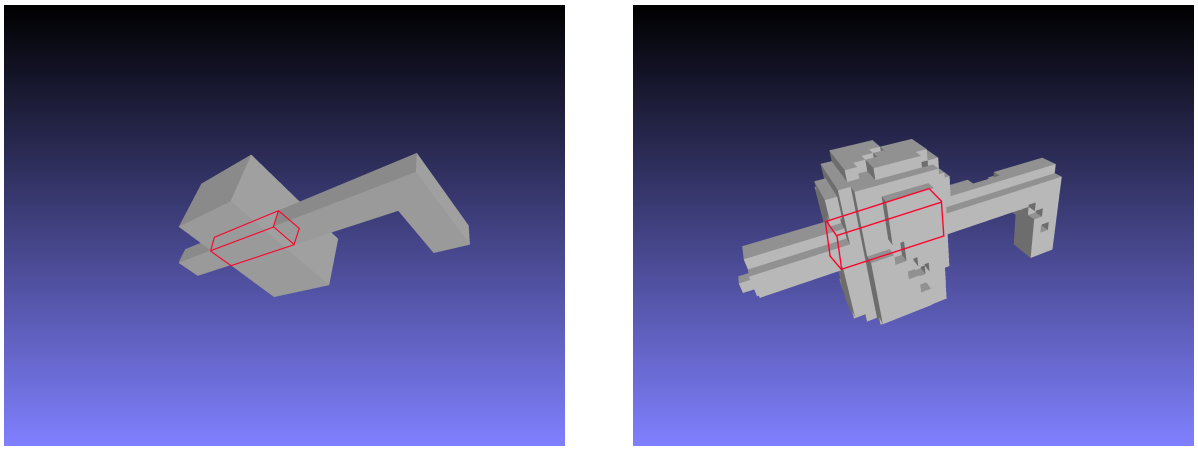
(a) Front (b) Back

Figure 6: Trimesh algorithm run on *max.off*, a mesh with an open boundary. Note the gradual tapering effect on the open half of the mesh.

Some limitations of the experimental algorithms also make themselves apparent in these results, signaling directions for future improvement. When operating on meshes with fewer primitive elements, the winding number calculation tends to be less granular, providing subpar surface representation. This was observed in the case of *block_intersect.obj*: despite correctly modeling the self-intersected interior, a large amount of noise is generated around the mesh boundary. Furthermore, approximating the Voronoi area as a constant has consequences for the point cloud method, which tends to produce physically exaggerated results in areas with high principal curvatures. Prominent examples of this defect can be seen in the Stanford bunny's right ear and the spout of the Utah teapot, both of which are slender features with sharp curves.

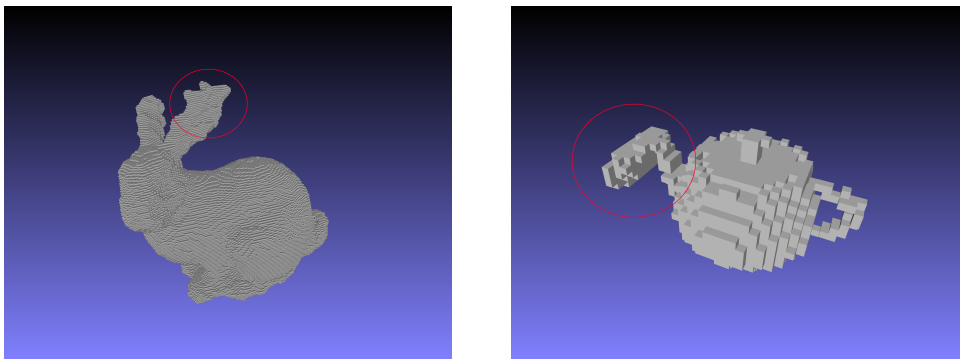
Assessment and Future Work

Overall, we were pleased to provide a functional implementation of two research algorithms despite the apparent limitations in our work. Notably, the recursive tree-based algorithm — a hallmark of the Barill et al. publication — is conspicuously absent from our implementation for a number of reasons. First, we encountered difficulties using the available libraries for constructing a bounding-



(a) Raycasting (b) Trimesh winding number

Figure 7: Comparison between raycasting algorithm and trimesh winding number algorithm on *block_intersect.obj*, a mesh with a self-intersection in the outlined area. This area is correctly marked as solid by the experimental algorithm yet left completely empty by the raycasting approach. Large amounts of noise in the winding number calculation are present on the boundary.



(a) *bunny.obj* (b) *teapot.obj*

Figure 8: Point cloud winding number algorithm run on two different meshes. Note regions where the features of the original model are greatly exaggerated or distorted.

volume hierarchy, likely due to inexperience in managing different parts of a complex geometry processing framework like OpenFlipper. Our next option was to design our own data structures as a means of subdividing the mesh boundary, but this too posed issues regarding the variable scaling of meshes as well as determination of nearest neighbors. Second, one key limitation of the "fast" winding number algorithm is the need to construct a hierarchical representation of the mesh before any useful computation can take place. Given that the point cloud algorithm is meant to prioritize efficiency, this approach seemed antithetical to our goals. It is worth noting that certain preprocessing information — namely, the hierarchical data structure — could be stored on individual meshes in order to improve the performance of subsequent calculations on the same mesh. However, as the general use-case for winding number determination lends itself to computations on multiple meshes, this did not seem like an appealing compromise.

Estimation of the Voronoi area surrounding each point is the other key omission in our implementation: the effects of treating this quantity as a constant are shown through our visual results.

Our theoretical approach to this problem was to capture sets of similarly-oriented nearest neighbors in the form of a directed graph. A rudimentary triangulation of the 1-ring neighborhood could then be generated from this subset of points. However, the time complexity involved in constructing this directed graph would likely be far greater than that of the winding number algorithm. One interesting compromise could be made between the triangle mesh and point cloud algorithms by (a) calculating the Voronoi area around each point using built-in mesh connectivity (in the case of triangle meshes only) and (b) treating triangle meshes as point clouds otherwise.

Finally, various quality-of-life improvements could be made to the plugin interface. Notably, the plugin does not emit an updated visual representation of the mesh after voxelization occurs. This is deliberate: the imported *CS581* framework generates the voxelized mesh somewhat inefficiently with many redundant faces and vertices. OpenFlipper seemed to encounter difficulties rendering these meshes: however, the tools of OpenMesh could likely be used to eliminate redundant elements prior to mesh generation.

References

[1] Barill, G., Dickson, N., Schmidt, R., Levin, D. I., and Jacobson, A. Fast winding numbers for soups and clouds. *ACM Transactions on Graphics* (2018).

[2] Jacobson, A., Kavan, L., and Sorkine, O. Robust inside-outside segmentation using generalized winding numbers. *ACM Trans. Graph. 32*, 4 (2013).

[3] Van Oosterom, A., and Strackee, J. The solid angle of a plane triangle. *IEEE Transactions on Biomedical Engineering BME-30*, 2 (1983), 125–126.

[4] Whiting, E. Cs480: Polygons, 2021.

[5] Whiting, E. Cs581: Solid modeling, 2022.