Poisson Surface Reconstruction using Walk on Spheres

Sasha Mishkin

1 OVERVIEW

The goal of this project was to implement an algorithm that reconstructs an implicit surface from a point cloud using Poisson Surface Reconstruction[1], but do so using the Walk on Spheres method[2]. I was able to write a solver that, at least for the small, sparse point clouds that I tested, generates surfaces that match my intuitive expectation of how the surfaces should look.

2 SOFTWARE USED

- The DIRT renderer, to contain and run my code
- libigl, the C++ geometry processing library, to run Marching Cubes
- MeshLab and Blender to visualize meshes (All meshes shown in this report were visualized using Blender)

2.1 Code locations

Solvers are defined in the files `include/dirt/wos.h` and `src/wos.cpp`. Testing, mesh generation, and input file parsing take place in `src/wos_testing.cpp` and can be executed by running `build/wos_test`.

3 SURFACE RECONSTRUCTION SOLVER

The central component of my project is a Poisson Equation solver that uses the Walk on Spheres algorithm to reconstruct a surface from a point cloud.

The input to the solver is a point cloud, consisting of point positions and outward normals. Upon being initialized, the solver creates a box-shaped boundary that tightly fits around around the point cloud. If the box is empty, though, I expand it to have length 1 in each dimension where it would have been empty. This specific size was chosen arbitrarily.

Then, the `evalPoint` function takes in a sampler, a point within the boundary, and a variance parameter. It runs a recursive helper function a set number of times (according to the `numSamples` class member variable) and averages the results.

The recursive helper is the core of the Walk on Spheres algorithm. It computes the distance from the input point to the surface boundary and returns the value $-\frac{1}{2}$ if the point lies very close to the boundary. Otherwise, the algorithm samples a point on the boundary of the largest sphere around it as input to the next recursive call.

It also samples a point on the interior of said sphere and calculates the source term and Harmonic Green's Function at that point, to add to the recursive result.

The result is a scalar that serves as an indicator function: Negative values represent points oustide the surface, positive values represent points inside the surface, and magnitude doesn't matter.

3.1 Source Term

The source term of the Poisson equation for surface reconstruction is the divergence of the vector field that is zero everywhere except at the points in the point cloud[1], and takes on the value of the

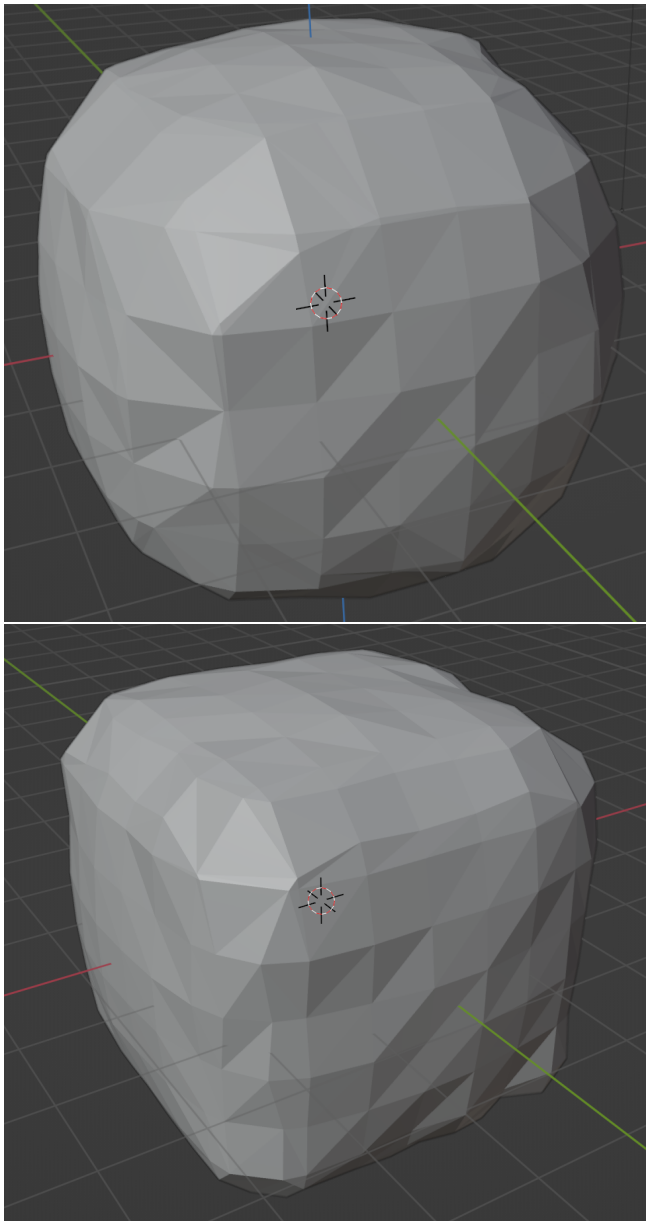

Figure 1: Reconstructed surfaces with the same variance parameter. The former is from a point cloud consisting of the centers of all faces of a cube. The latter is from a point cloud that *also* includes the vertices of this cube. The latter result is noticeably sharper.

normals at those points. If the point cloud in question is P, K is the Gaussian kernel, and $n(y)$ is the normal of point y, then the divergence at point x ends up being equivalent to:

$$\sum_{y \in P} \nabla K(x, y) \cdot n(y).$$

Specifically, $K(x, y)$ is differentiated with respect to x. The helper function `f` determines this source term.

3.2 Variance parameter

The Gaussian kernel takes in a free parameter sometimes referred to as b, representing variance or scale of the space involved. The `evalPoint` function in the solver takes in a value of b as an argument. Adjusting b, referred to as `variance` or `B` in the code, is largely a matter of trial and error, and can generate surfaces with different appearances, as shown in Figure 2.

4 ALTERNATIVE SOLVERS

I wrote two additional variations on the main solver.

4.1 Spherical Boundary Solver

One such variation represents the surface's boundary as a sphere instead of a box. This also means that the solver runs closest-point queries with respect to this bounding sphere. Figure 3 shows visual differences between the box and sphere boundary shapes.

4.2 2D Solver

I also wrote a 2D version of the WoS algorithm, with a rectangular boundary. There were several small changes required to implement this:

- Modifying the closest-point query to work with 2-dimensional boxes
- Modifying the calculation of the source term to use 2-dimensional vectors
- Changing the terms corresponding to the sampled ball's volume and Harmonic Green's Function.

To test the solver, I generated a 2D scalar field of the solver's output values, and extruded those values into 3D. I then generated a mesh around that 3D field.

5 MESH GENERATION, VISUALIZATION, AND FILE I/O

Although these steps aren't a part of the core algorithm behind my project, making these features integrate smoothly with my code took up quite a lot of time and effort.

5.1 Mesh Generation

In order to interpret the results of a surface reconstruction algorithm, we need a method to visualize the implicit surface that the function outputs. To do this, I relied on an external geometry processing library, `libigl`. I installed `libigl` and integrated it into Dirt's codebase and wrote a function (`marchingCubesTestBox3D()` and its variations) to preprocess WoS outputs into the format that `igl`'s marching cubes function takes as input.

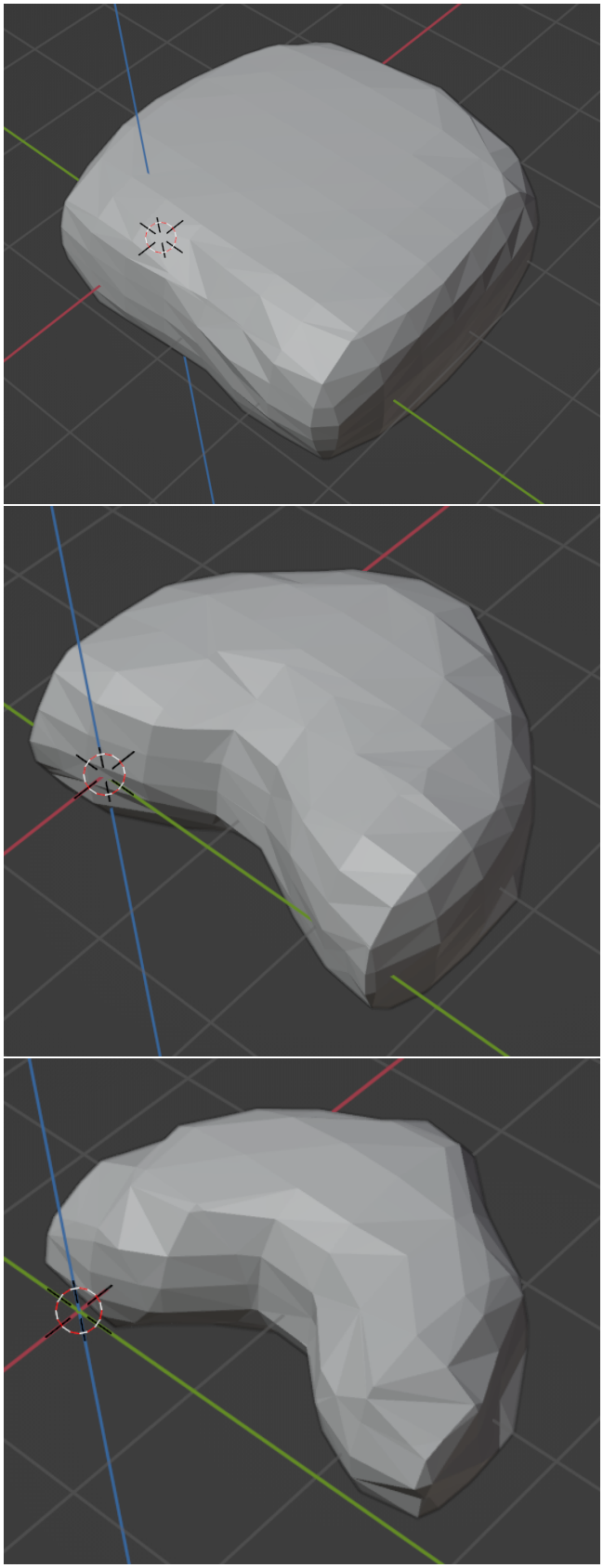

Figure 2: Reconstructed surfaces with `variance` set to 1, 0.5, and 0.4 respectively, from the same point cloud.

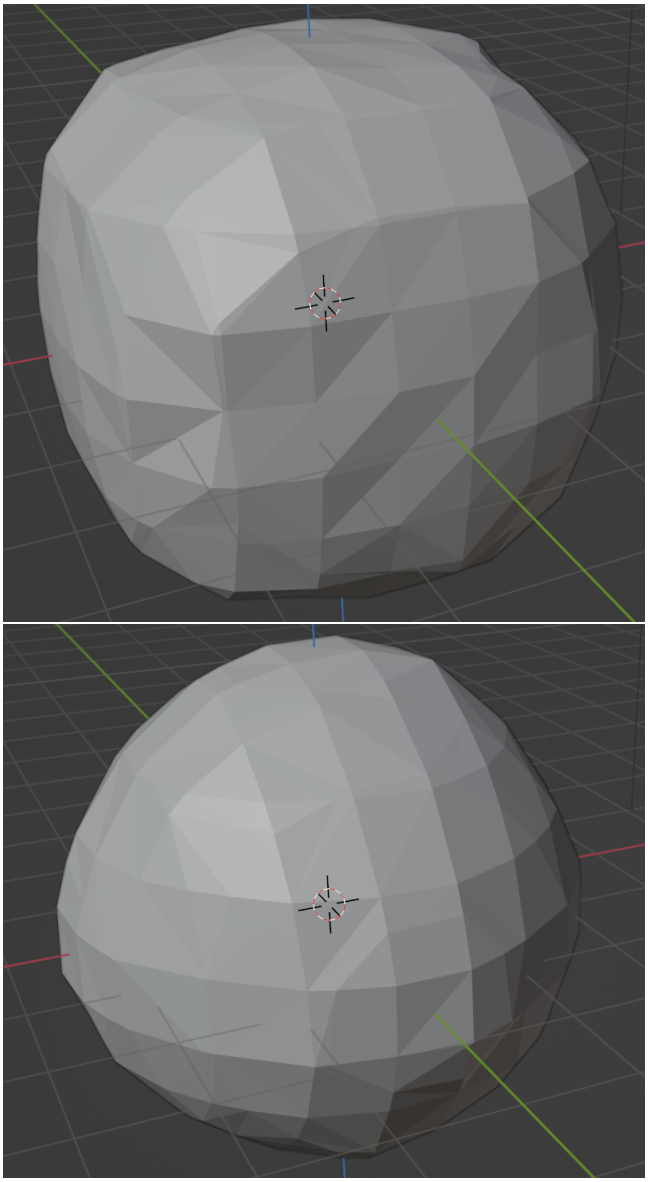

Figure 3: Identical point clouds rendered with the same variance, but using a box-shaped and spherical boundary (respectively).

5.2 File Parsing

Mesh Output. The Marching Cubes function provided by `libigl` returned two matrices, containing vertex positions and vertex indices for each face (respectively). I processed this data into an `.obj` file, since this seemed to match the output's format well. I then finally imported this file into a program (such as Blender or MeshLab) that could display meshes.

Point Cloud Input. I also wrote a simple parser (in the `main` function of `wos_testing.cpp`) that reads vertex position and normal data from a file specified in the command line. Unfortunately, I couldn't find a way to directly parse from exported `.obj` files (from, say, Blender) without heavily modifying the structure of how the file organizes vertex normal values. See inline documentation in the code for further details on input parsing if needed.

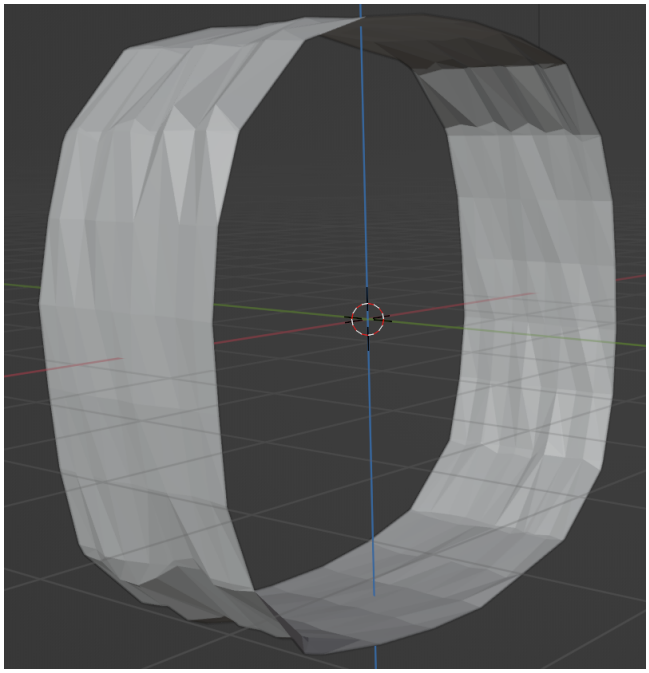

Figure 4: Mesh generated by the 2D solver, with a simple point cloud as input (just the centers of each side of a square centered on the origin). Variance is set to 1.

6 TESTING

In addition to testing correctness of the WoS algorithms by generating various meshes, I also used various other testing methods. I created unit tests to make sure that I created boundaries around point clouds correctly and that my closest-point queries functioned how they should.

Also, I wrote functions that ran the WoS algorithm on each point in a 3D grid and output the resulting indicator function values into a `.csv` file. This allowed me to visualize the results as scatterplots in `plot.ly` and double-check the values against what I expected. This also helped me check whether various bugs originated from my algorithm itself, or from issues with setting up the mesh generation and visualization pipeline.

7 LIMITATIONS AND FUTURE WORK

Point Cloud Generation. I found it difficult to find and generate non-trivial point clouds to use as test cases, especially since mistakes with calculating point normals were difficult to catch and could lead to the resulting mesh looking very wrong. Free point clouds online are very big and lack normal data, so I currently cannot make use of them as test cases.

Runtime. Generating a mesh requires looping over all points in a three-dimensional voxel grid, which means that the program scales cubically with respect to the desired mesh resolution. Additionally, every time the WoS algorithm's recursive helper function is called, we loop over every single point in the point cloud to compute the source term for a given sample. This limits us to using relatively small mesh resolutions and sparse point clouds.

A possible way to make the algorithm run faster would be to implement some kind of acceleration structure for determining the source term, to avoid looping through the entire point cloud repeatedly.

Raytracing. An interesting future step would be to implement a ray-intersection algorithm with the implicit surfaces that the code generates. This approach would allow us to raytrace the implicit surface directly, without a mesh generation step. It would be implemented using some form of bisection search to locate the distance at which a ray would reach a point where the implicit function is zero.

Screened Poisson Equation. There are other forms of surface reconstruction that use the Screened Poisson Equation. A possible future step is to write another solver that uses this approach.

ACKNOWLEDGMENTS

To Nicole Feng for help with deriving a formula for the source term.

REFERENCES

[1] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. 2006. Poisson Surface Reconstruction. *Eurographics Symposium on Geometry Processing* (2006). https://www.cse.iitd.ac.in/~mcs112609/poission.pdf

[2] Rohan Sawhney and Keenan Crane. 2020. Monte Carlo Geometry Processing: A Grid-Free Approach to PDE-Based Methods on Volumetric Domains. *ACM Trans. Graph.* 38, 4, Article 1 (July 2020), 18 pages.