A Practical Analysis of Maze Geometry Representations

Jose Luchsinger

1 Background

Over the past couple of years, I have been part of a research group within RPI's chemical engineering department whose focus is on creating structures for use in porous media. In particular, they're interested in mazes and their potential utility as filters for particles in a fluid flow system [1].

My main responsibility in this group has been developing a real-time simulation of particle flow through these structures using the Unity game engine. The main purpose of the simulation is to allow researchers to quickly observe the effectiveness of certain maze configurations before replicating them in real life for use in physical experiments.

Currently, the simulation works by referencing fluid flow data that is generated using COMSOL, a set of proprietary multiphysics tools. Roughly, the workflow looks like this: the shape of the maze (where walls do or do not lie) is generated by a set of MATLAB scripts which output the shape of the maze as a plain text file as well as a DXF file, a format designed for CAD programs. COMSOL reads the DXF file and calculates the force of the fluid flow through the maze, outputting a heat map as well as a triangulation of the maze with associated physics data assigned to each face.

This triangulation is then fed back into MATLAB to convert it into a 2D table of coefficients for a polynomial fit of the maze, which is then referenced during the simulation to determine the force which will be applied to a particle at a given position. The exact process with which this data is referenced is described later in this report, but in theory, this allows force data to be queried in constant time, alleviating the computational cost of querying that other data structures require. By pre-processing the force data and referencing it in this way, more resources are free to handle the task of simulating the particles themselves, accomplished using Unity's default 3D physics system (an implementation of Nvidia's PhysX system).

The result is a simulation capable of simulating the movement of upwards of about 1500 particles on student-owned machines, which are typically low-end laptops. The team is satisfied with the current version of the simulation to begin using it in undergraduate labs this spring, however there are several

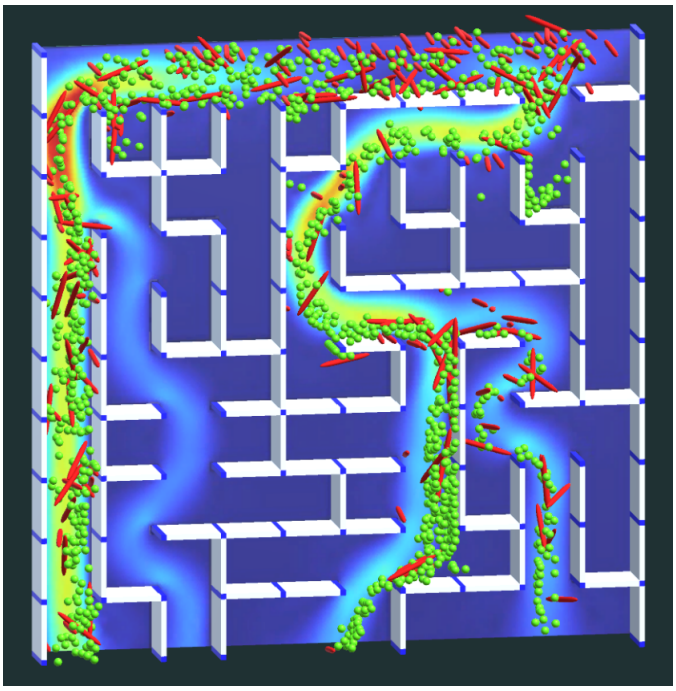

Figure 1: The Unity particle simulation.

issues with Unity which make switching to a new engine desirable. In particular, certain aspects of the engine which are designed for game development end up unused in the course of the simulation while still taking valuable processing resources, creating an arbitrary upper bound for certain aspects of the simulation, such as the maximum particle count or the maximum time scale value.

If a new engine is in the cards, then it makes sense to begin thinking about architectural changes that will maximize the limits of the simulation. The current system of storing fluid flow data is a compromise made to accommodate the limitations of the Unity engine, and though experiments to confirm the accuracy of the simulation are yet to be conducted, I still have concerns with this system's feasibility, especially in the long term when we expand to larger, more complex maze geometry. With regards to storing and accessing fluid flow data, this course has introduced us to several structures which are more than capable of increasing the fidelity of that data with minimal additional computational expense. The purpose of this report is to examine a set of these structures and compare them against each other to determine the limitations of each structure and decide which is best for handling fluid flow data in this new engine.

2 Data Structures in CFD Applications

2.1 Octrees

When choosing a structure to replace the existing maze representation, there exist several viable alternatives which are used regularly in the field of computational fluid dynamics. Orthtrees appear to be the most commonly used, having a large and active body of research revolving around them. For example, in a 2020 paper by Xu et al., researchers created a simulation of a single particle flowing through a fluid using an octree mesh as their background mesh [2]. The hardware target for the simulation they describe is quite different—a new mesh is created at each time step to accurately simulate the effect the particle has on the surrounding flow, which is only feasible on a supercomputer—however this paper and others like it demonstrate how orthtrees are a desirable structure to use in CFD applications, and in the context of simulating particle motion in fluid in particular.

2.2 Delaunay Triangulations

Though orthtrees appear to be the most obvious choice, I felt it necessary to look into the efficacy of other structures used for CFD applications in case anything existed that was better suited to this application in particular. There does exist a body of research regarding the use of Delaunay triangulations in CFD applications, albeit much smaller than that which exists for orthtrees. A 1992 paper by N.P. Weatherill describes a method of assembling Delaunay triangulations for use in CFD applications, showing that there exists some basis for using such triangulations as a structure for particle simulations [3]. However, there are some conflicting views on the matter: a NASA memorandum released earlier that year describes issues with Delaunay triangulations of CFD grids, particularly a distortion of the grid's shape during the triangulation process [4]. Despite these reservations, the most desired quality for any of these data structures is rapid query time. With the flat surfaces of the mazes being meshed, there should be little to no distortion during the meshing process. If, somehow, a Delaunay triangulation of the maze presents some advantage in querying time, I believe it would still be a valid option for meshing, hence its inclusion in this testing.

2.3 COMSOL

One other representation I was curious about was the mesh of the maze created by COMSOL. Currently, this is used in some capacity in the current simulation, if only as an intermediate step before being reduced to a polynomial fit. To take a closer look at COMSOL's output, I asked one of the members of the mazes research group with access to COMSOL to send

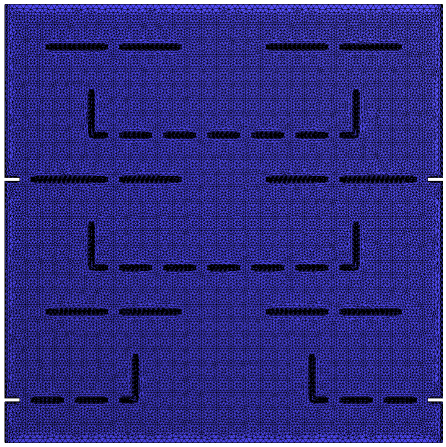

Figure 2: Maze mesh exported from COMSOL.

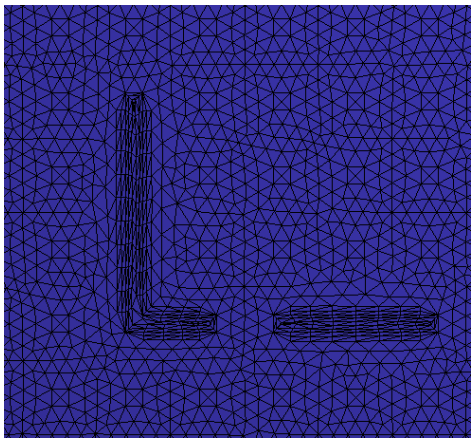

Figure 3: Close-up of COMSOL mesh.

me an exported mesh file. Thankfully, when exported from COMSOL, the mesh is recorded as a series of vertices followed by a series of triangles, much like an OFF file. After writing code to import it, the mesh itself appears to be some sort of triangulation of the maze. Because COMSOL is closed-source, I don't have access to documentation specifying how exactly mesh the triangulated, though based on the shape of the mesh, I suspect it's an quadtree-based polygonal mesh with a unique split predicate that still creates fine uniform meshing in open areas of the maze. Another issue is that no distinction is made between triangles from walls and triangles from open maze channels. One could write code to associate certain triangles with walls based on COMSOL's velocity data by using CGAL's property map feature, however the velocity data is exported as binary, meaning there's no way to import it into CGAL like there is with the mesh. COMSOL's documentation doesn't specify a particular reason for the velocity data to be stored this way, through the reason is likely to reduce the size of the data (which would likely be more apparent for the large meshes COMSOL is built for). Due to the amount of uncertainties with COMSOL and the additional work required to import its data, and the fact that I don't have a copy of COMSOL myself to generate meshes with, I decided to omit this representation from testing.

3 Method

3.1 Testing Environment

All tests were performed on a laptop with an Intel i7-8550U CPU running at 4Ghz. The Unity simulation was tested within the Unity 2020.3.18f1 editor on Windows 10, while all other tests were performed on Arch Linux using CGAL version 5.6. The CGAL implementation was created without any sort

of multi-threading, while queries in the Unity simulation take place on the main thread. The particles in the Unity simulation were disabled for testing.

3.2 Maze Generation

The plain text maze shape file used was created by the Maze Research group's MATLAB scripts. The file was used to construct an outline of the maze using CGAL's surface mesh package. Four vertices are created around the point where two walls meet, and then vertices are connected based on the placement of maze walls. It should be noted that this results in some unnecessary vertices, like those that lie on the walls of longer channels.

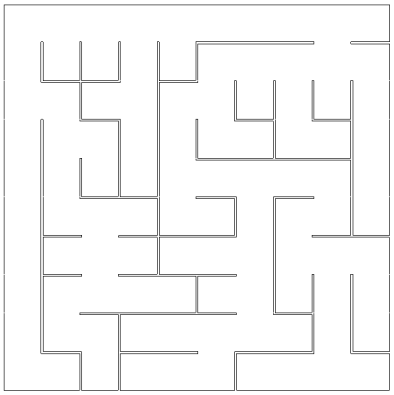

Figure 4: Outline of the maze as a CGAL surface mesh.

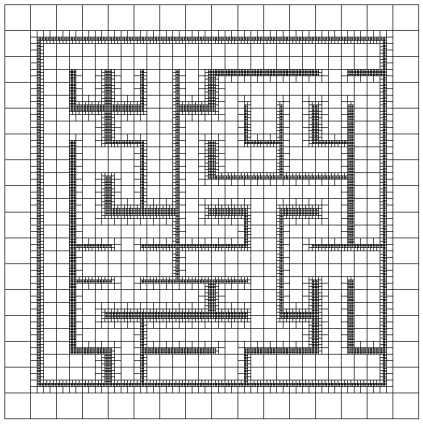

Figure 5: Quadtree representation of maze.

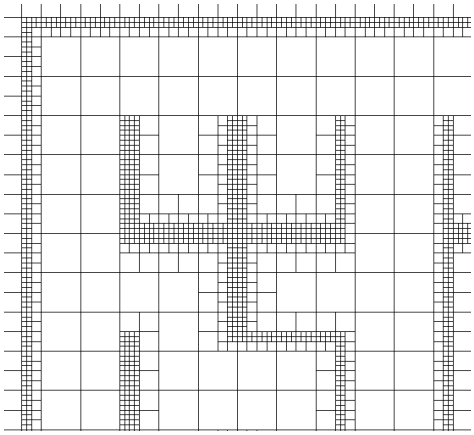

Figure 6: Close-up of upper-left corner of quadtree.

3.2.1 Quadtree

The quadtree structure was created using CGAL's orthtree package. CGAL allows the user to define a custom split predicate, which I used to split nodes whenever they intersected with any element of the maze geometry (e.g. when an edge crosses a node). The split predicate will split up to a certain node size (1cm), at which point it will stop splitting. The end result is a quadtree with 12869 nodes and a depth of 8, with its deepest nodes placed along the edges of the maze geometry.

3.2.2 Triangulations

Three triangulations were created for testing, all of which were built using CGAL's constrained triangulation package. In all cases, triangulations were first constrained by the edges of the maze geometry, ensuring no triangles crossed over maze walls. The first triangulation created was a basic constrained triangulation using default predicates and default traits. The resulting triangulation is well-formed in some areas (well-formed meaning triangles don't cross between maze cells), however some triangles don't respect cell boundaries, becoming long and thin. The second triangulation was a constrained Delaunay triangulation, again constructed with default traits and predicates. Predictably, this triangulation looked much better, consistently respecting cell boundaries. The third triangulation is a subdivision of the second, repeatedly adding additional vertices at the centroid of each triangle, and is intended to mirror the quantity of nodes created by the quadtree.

The first two triangulations had a triangle count of 722. The subdivided Delaunay triangulation had a triangle count of 6498 triangles.

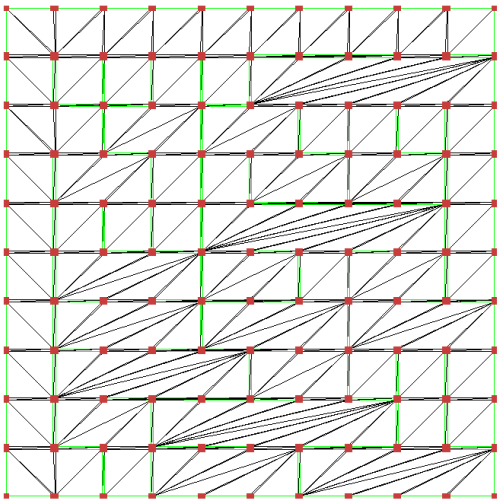

Figure 7: Constrained triangulation of maze.

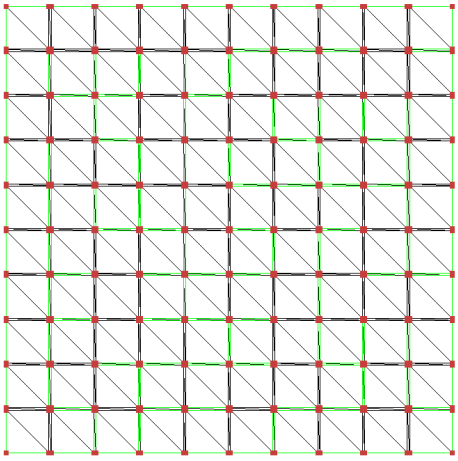

Figure 8: Constrained Delaunay triangulation of maze.

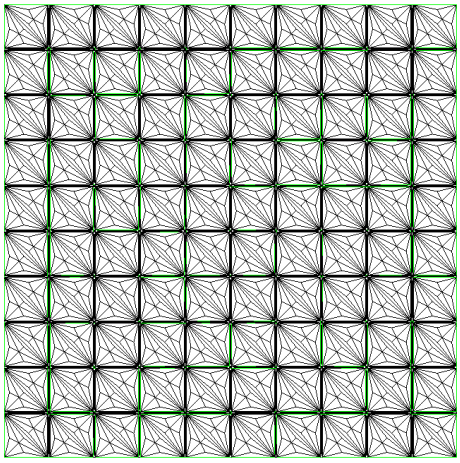

Figure 9: Subdivided Delaunay triangulation.

3.3 Data Structure Querying

Before any testing is done, a set of n random 2D points is created, all within the bounding box of the maze. Points can be placed within open channels or within walls. After the points are generated, they are all queried sequentially. In the CGAL implementation, the time in milliseconds to query n points is measured using std::chrono, while in the Unity implementation, the time is measured using C#'s System.Diagnostics.Stopwatch.

For each random point, CGAL::locate() is called without any additional parameters such as starting face. According to CGAL's documentation for the Triangulation_2 class, locate is performed via a line walk starting from the infinite face of the triangulation, taking $O(\sqrt{n})$ on average and $O(n)$ in the worst case [5]. Neither the documentation for the constrained triangulation class nor the constrained Delaunay triangulation class, both of which inherit from Triangulation_2, indicate that they make any changes to this location method. For CGAL's orthtree class, locate appears to be implemented traditionally, traversing the orthtree to find the deepest cell containing the query point. Assuming that's all CGAL does, the search should take $O(\log n)$ time.

For the Unity simulation, each query point is passed to a function that converts the point in world space to a position in the coefficient table. This is done by dividing each value of the point by the height or width of a maze cell. Typically, this conversion is followed by a series of constant time operations that returns the force applied to the particle, but for the sake of testing, this functionality was removed. This operation takes place in constant time and, in theory, it should be the fastest of any of the data structures presented here.

4 Results

Query Count	Unity	Quadtree	CT	CDT	CDT (subdiv.)
100	365	2	4	2	5
1000	3685	21	46	20	48
10000	37698	102	205	91	380
100000	384389	934	2917	924	5482
1000000	3735439	12763	29139	12646	51307

Table 1: Query time for each data structure in milliseconds.

4.1 Unity

Of all the tests I performed, the Unity simulation presented the most unexpected results. Though querying points in this structure should have been

trivial, the Unity simulation took the longest out of any of the structures to finish its tests, in fact being the only structure with total times exceeding 1 minute. One of the contributing factors of this may be that these queries are happening on the Unity's main thread, along with whichever processes Unity places on that thread. The issue with that, however, is that there are no other expensive operations occurring on that thread. In any case, the additional processing required by Unity appears to be a primary factor in preventing the Unity simulation from simulating higher particle counts.

4.2 Quadtrees

Quadtrees appear to be the clear winner among all of the data structures examined during testing. Besides the Unity simulation's query times falling as flat as they did, this was the anticipated outcome. Orthtrees are a popular data structure in CFM applications for a reason, and as both prior research and this testing has shown, one of the primary reasons is their highly desirable balance between fidelity and query time. This testing has confirmed that if any data structure were to replace the one currently used in the Unity simulation (or be the basis for flow data querying in the new simulation), it would be some form of orthtree.

4.3 Triangulations

Also as anticipated, all types of triangulations lagged behind quadtrees in query time. Given the differing time complexity of their locate functions, such a disparity makes sense. The more interesting aspect of the results to me is the difference in time between the basic constrained triangulation and the constrained Delaunay triangulation. Given the method of location for triangulations, a time loss for an ill-formed triangulation was expected, but not to this degree. The subset of thin triangles in the basic constrained triangulation managed to double the required query time from the constrained Delaunay triangulation. I'm left wondering if this is purely due to the shape of the triangulation, or if the time loss is in part caused by a poor choice of starting vertex on the part of CGAL::locate. Of interest is the query time of the non-subdivided Delaunay triangulation, which is on par with the query time of the quadtree despite the difference in time complexity. Of course, this similarity disappears when we attempt to scale the element count of the Delaunay triangulation to match the quadtree, in which case we see query times around 5 times as long as the quadtree.

5 Further Work

For the future development of maze research projects in general, the results of this testing have confirmed that orthtrees are a highly desirable structure for use in real-time simulation. That being said, one thing that should be investigated further before any implementation is memory usage. I doubt the memory usage of an orthtree is prohibitive, seeing as these tests could be run on a laptop without much issue. If and when the simulation expands to use 3D data, it may become a bigger issue.

The most immediate concern these results raise is how to minimize query time in Unity. It may be worth investigating Unity's multi-threading system to see how much the query time can be reduced in the existing simulation, however that would still be limited by the time Unity takes to get that work back into the main thread. I intend to look into improvements for Unity further, especially since the simulation will be actively used by students this upcoming semester, however these results seem to all confirm that the best course of action is to start from scratch and create an application purpose-built for particle simulation.

References

[1] Guo, Alex, et al. "Transport in mazes; simple geometric representations to guide the design of engineered systems." Chem. Eng. Sci., vol. 250, 15 Mar. 2022, p. 117416, doi:10.1016/j.ces.2021.117416.

[2] Xu, S., Gao, B., Lofquist, A., Fernando, M., Hsu, M.-C., Sundar, H., & Ganapathysubramanian, B. (2021). An octree-based immersogeometric approach for modeling inertial migration of particles in channels. Comput. Fluids, 214, 104764. doi: 10.1016/j.compfluid.2020.104764.

[3] Weatherill, N. P. (1992). Delaunay triangulation in computational fluid dynamics. Comput. Math. Appl., 24(5), 129–150. doi: 10.1016/0898-1221(92)90045-J.

[4] Posenau, Mary-Anne K. and David M. Mount. "Delaunay triangulation and computational fluid dynamics meshes." 8 Jan. 1992, https://ntrs.nasa.gov/citations/19920021663.

[5] CGAL 5.6 - 2D Triangulations: CGAL::Triangulation_2<Traits, Tds >Class Template Reference. (2023, December 12). Retrieved from https://doc.cgal.org/latest/Triangulation_2/classCGAL_1_1Triangulation__2.html.