Implementing Delaunay Refinement Using Ruppert's Algorithm

Alexander Hu
ID: 110544388

Introduction

Delaunay refinement to resize triangles is an important function used in applications like interpolation, the finite element method, and the final volume method. The goal of this refinement is to create triangles of a certain size and, more importantly, have specific angles that are suitable for future use.

This report will cover the design and implementation of Ruppert's algorithm, a famous algorithm discovered in the early 1990s by Jim Ruppert. The design of this algorithm revolves around the use of Lawson's Algorithm and Bowyer-Watson's Algorithm in order to insert points into a constrained delaunay mesh while maintaining the delaunay constraint. This implementation of the algorithm is designed without keeping the size of the triangle in mind, although a later section will cover how this could easily be implemented. Afterwards, the report will cover the test cases and experiments done with the algorithm as well as extended considerations on some of the inner workings of the algorithm.

Design

The basics of Ruppert's algorithm is as follows:

1. Provide a constrained delaunay mesh formed from some PSLG.
2. First, check each segment in the PSLG. If there are any points that encroaches the diametric circle formed using the segment AND the point is visible to the edge, split the edge into two subsegments by inserting the midpoint into the diagram, then performing Lawson's algorithm on the newly formed triangles until all triangles are confirmed Delaunay.
3. After that, start removing skinny triangles. When a skinny triangle is found, attempt to insert the circumcenter of the triangle into the mesh via Bowyer-Watson's algorithm. However, if the circumcenter encroaches at least one PSLG segment, instead split all segments that are encroached and form new triangles using the subsegments.
4. When there are no skinny triangles left in the mesh, the algorithm ends. The resulting mesh is delaunay, not simply constrained delaunay.

Although the high-level description of Ruppert's algorithm is quite straightforward and elegant, the details involved with each step proves to be a challenge to the unsuspecting. The specifics on some of the most important decisions regarding these details are covered below:

Data Structure

The usage of PyVista as the framework this algorithm is based on means it comes with its own data structures the algorithm has to work with. The setup work involves extracting the points, faces, and edges from the constrained delaunay mesh as well as the segments specified by the PSLG, which are also composed of edges.

The points are stored in a list of 3-tuples, where the indices of the tuple represent the x, y, and z coordinate of the point. The points are stored in a list specifically for the purpose of retrieving them based on their position, which serves as the point ID used by both the edges and the faces.

Each edge is represented by an immutable frozenset containing two point IDs. This is possible without any concern of duplicating two points as each edge must be composed of two distinct points, which ensures the point IDs in the set are distinct. Using a set allows for the possibility of edges being identified with either point as the source point as equivalent edges, as the source point of an edge is not integral to this algorithm.

Each face is represented by a 4-tuple. The first index represents the number of sides - as all the faces are triangles, this is always 3. The next three indices represent the point ID correlating to the index value of the point lying inside the list of points. The order of the face points does matter, as PyVista checks if the points are ordered in a clockwise or counterclockwise position in order to determine if it is hallowed out, for clockwise, or filled in, for counterclockwise. Each face stored should therefore be ordered counterclockwise.

To assist with the algorithm, the algorithm uses several global data structures. One main task these data structures cover is retrieval of commonly used information, specifically in regards to the correlation between faces and edges.

One necessary function is to find what faces are actually in the mesh currently, as inserting points and drawing new triangles also necessitates removing old ones. For the sake of avoiding the very tedious and slow task of actually removing the faces, instead a hashmap is used to store the face tuples and lazy deletion is utilized - active triangles are denoted with 1 as the value and deleted ones are denoted with 0.

Another common operation is checking what triangles contain an edge, specifically useful when trying to execute Lawson's algorithm. To facilitate this, a hashmap is used with the keys specified as the frozensets of edges. The values associated with each frozenset edge key is a list containing all the face tuples that contain the edge in question.

Thus, to check if an edge contains a face, simply access its corresponding list from the hashmap, check all the face tuples in the list, then finally return all face tuples that have a value of 1 in the faces hashmap.

Finally, there are two queues and a stack, all represented by lists, used in order to determine what needs to be considered next by the algorithm. One queue is for storing the frozenset edges representing encroached segments. The other is for storing face tuples representing skinny triangles, and for storing face tuples representing triangles. The stack is used to handle Lawson's Algorithm to determine which triangles need to be checked for flips. These data structures need to eventually be empty in order to progress on to the next stage or complete the algorithm.

Checking Edge Encroachment

The base problem of edge encroachment can be easily handled with the following steps:

1. Find the distance between the midpoint and one of the points of the encroached edge.
2. Check all points except for the ones on the edge, if performing an initial sweep, or the point to insert if trying to perform Bowyer-Watson's algorithm.
3. If any point has a distance less than or equal to the distance between the midpoint and one of the points of the encroached edge, it must also be on or within the diametric circle formed by that edge and therefore is encroaching the segment. Add that segment to the segment stack.
4. For every segment in the segment stack, split the segment into two subsegments by inserting the midpoint.

For the initial sweep, if there are no segments found to be encroached after checking all points, then the initial sweep ends. However, if there are any segments found, then after all those segments are split, another sweep to check if any points, including inserted midpoints, encroach any resulting subsegments is performed.

Ruppert's algorithm also adds a second condition when performing the initial sweep at the start of the function - points that are considered encroaching the segment must be visible by the segment. Specifically, if a line is drawn from the potential encroaching point to the midpoint of the edge, there should be no other segments from the PSLG that intersect that line.

One edge case considered was that the line could intersect a PSLG segment that lies within that line itself. Another edge case considered was that the line intersects one of the endpoints of a PSLG segment. For both of these edge cases, the algorithm implementation considers them to still be visible.

For the former point, the edge case also implies that there is another point lying on the enclosed PSLG that not only encroaches that segment, but is visible to the encroached segment. For the latter, an assumption is made that intersecting an endpoint means that endpoint also encroaches the edge and is visible to the encroached edge. This latter assumption is not entirely accurate and this behavior introduces a mild bug that will be covered in a later section.

Lawson's Algorithm Usage

Lawson's algorithm states that, given a convex 4-sided polygon formed by two triangles sharing an edge, if the triangles are not delaunay, then by connecting the two points that do not lie on the shared edge and removing the original shared edge would form two new triangles that are delaunay. This process is often described as "flipping" the shared edge.

This process is specifically for two reasons: First, a constrained delaunay diagram is not fully delaunay due to considerations made for the PSLG limitations. Second, inserting a midpoint to an encroached segment and drawing new triangles using that midpoint also does not guarantee that the new triangles are delaunay with each other or with pre-existing triangles in a mesh. Lawson's algorithm allows these issues to be resolved every time a midpoint is inserted into an encroached segment, ensuring that the mesh is delaunay upon completion.

The specific triangles that Lawson's algorithm checks in Ruppert's algorithm are determined by using a stack. Upon creating new triangles by splitting an encroached segment into two subsegments via an inserted midpoint, all new triangles created are inserted into the stack. The goal now is to empty this stack, which indicates no more triangles need to be checked with Lawson's algorithm.

The Ruppert algorithm implementation will pop a triangle, triangle A, from the stack. If the triangle is still active (it still has a value of 1 when accessed by the faces hashmap), it will first check if any of the points from adjacent triangles that do not lie on the edges of triangle A encroach the circumcircle of triangle A. If not, it will then check the adjacent triangles and see if some point on triangle A encroaches the circumcircles of the adjacent triangles. If still not, then all those triangles are delaunay and the next triangle from the stack can be popped.

If, however, any triangle's circumcircle is encroached by another triangle's non-shared edge's point, then those triangles must have their shared edge flipped by Lawson's algorithm. The two triangles are then added to the stack, as the new triangles formed by the flip may also require additional flips to triangles adjacent to them.

Bowyer-Watson Usage

The Bowyer-Watson algorithm is primarily used when trying to eliminate skinny triangles from the mesh. Skinny triangles are identified by checking if the ratio of the length of the shortest side of the triangle and the length of the circumcircle radius is larger than some predefined value. For Ruppert's algorithm, this predefined value is set as the square root of 2. Doing so ensures that no triangle remaining in the mesh in the end has an angle that is less than about 20.7 degrees.

This removal process attempts to insert the circumcenter of the skinny triangle into the mesh, which is executed by the Bowyer-Watson algorithm. However, the circumcenter is only inserted if it does not encroach any of the PSLG subsegments. If it does, then instead of inserting the circumcenter, all

subsegments encroached by the circumcenter is subject for splitting and added to the segment stack. If there are any segments in the segment stack, those are prioritize for splitting before attempting to remove any skinny triangles.

It is possible that, in the process of splitting an encroached segment, the skinny triangle in question is removed anyways (due to Lawson's algorithm). However, if it is not, then another attempt is made to insert the circumcenter of that triangle, which may or may not encroach any new subsegments.

The process of Bowyer-Watson's algorithm is quite simple. Check for all triangles that have a circumcircle encroached by the circumcenter of the skinny triangle and mark them for removal (this includes . Find the boundaries of all marked triangles by checking each triangle for edges that are not shared by any other marked triangle. Remove the triangles, insert the circumcenter, then form new triangles using the boundaries and the circumcenter inserted.

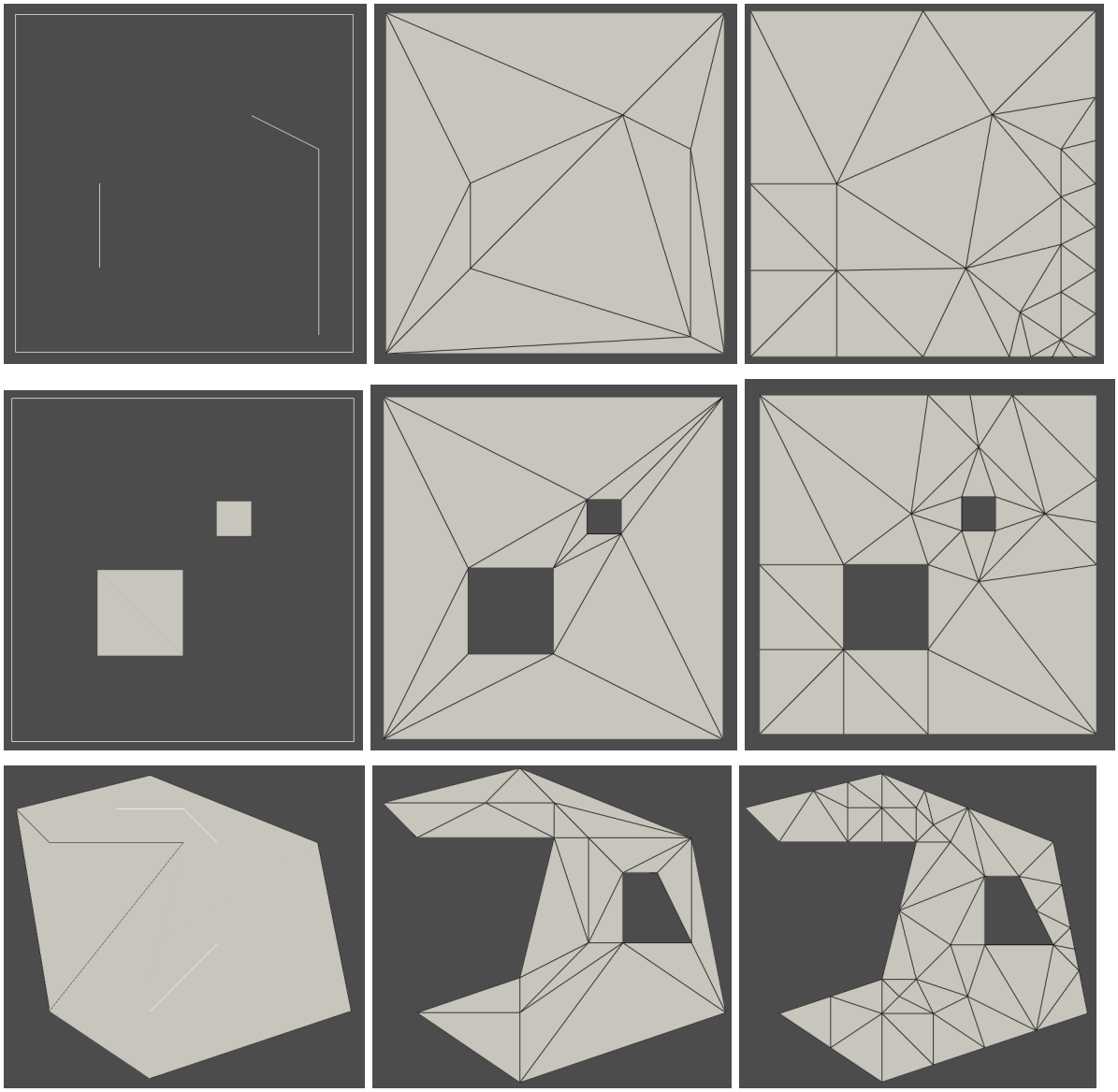
The original implementation of Bowyer-Watson requires a "super triangle" that encloses all points to be inserted. However, Ruppert's algorithm does not need to worry about this because any circumcenter of a skinny triangle that could be inserted has been proven to lie inside the area of the mesh in Ruppert's algorithm, provided that there are no encroached segments. This is why encroached segments are always prioritized over removing skinny triangles - otherwise, it is possible to try and insert a circumcenter outside the mesh boundaries, which would be incredibly problematic.

Implementation Choices

PyVista

Python, and by extension PyVista, was used because of the ease in utilizing the Python language itself. Although the performance would never match up to an optimized C++ program, the code implementation does not aim to be anything resembling optimize, rather it simply attempts to be a demonstration of how Ruppert's algorithm functions with various example inputs in practice.

Experimental Results



The algorithm was provided with three different PSLG inputs: A square enclosing only line segments, A square hollowed out by two other squares, and a concave polygon containing both line segments and a hollowed out triangle. The algorithm managed to successfully output the delaunay refined mesh for all three of these inputs without any issue. The visual results are shown above - the images on the left are the PSLG inputs, the images in the middle are the resulting constrained delaunay meshes, and the right images are the final refined delaunay meshes. Note that PyVista may not be able to accurately represent the PSLG of the diagram depending on how it was set up, but it manages to represent the constrained delaunay mesh properly given the PSLG input.

Extended Thoughts

Limitations of PSLG Input

In Ruppert's original paper, he stresses that connected edges of any PSLG input must form angles no smaller than 90 degrees. Further studies have shown that this restraint could be relaxed to 60 degrees, but there remains such a restraint due to how the algorithm could fail otherwise.

When angles less than 60 degrees are used in the PSLG, it is possible for Lawson's algorithm to attempt inserting midpoints in subsegments that are close to these problematic angles. In that situation, the inserted midpoints would then immediately encroach one of the subsegments forming the triangle, necessitating new midpoints. Thus, more and more midpoints would be inserted into smaller and smaller segments, causing the algorithm to never terminate.

Visible Points Edge Case Problem

Previously, it was mentioned that the implementation of whether or not a point is visible to an encroached edge had a bug attached to it. Specifically, the bug regards the possibility that the line passes through a point of the PSLG that also goes through a hollowed out section to reach the midpoint. Although the initial definition would declare that this is visible, it should be obvious to any reader that this really should not be the case.

Thankfully, this bug does not affect the termination of the algorithm, nor does it cause the final resulting mesh to not be delaunay. The end result generally would just be that additional midpoints are added to the mesh. It is possible that, similar to the earlier mentioned PSLG limitation issue, midpoints are added forever and the program fails to terminate, but that is the result of the limitations of the PSLG input itself.

Size of Triangle as a Restraint

It is mentioned that it may not be desirable for the triangles of a mesh to be too large. It is possible to deal with this however by specifying some sort of size constraint and, along with inserting the circumcenter of skinny triangles to remove them, adding the circumcenter of large triangles to the mesh.

Applying Ruppert's Algorithm in 3D

In the decades since, Ruppert's Algorithm has been considered 3-dimensional space. However, Ruppert's algorithm in its base form was unable to handle 3-dimensional space due to the inability of circumcircle checks to account for tetrahedron slivers. It also suffered issues regarding mesh boundaries and the potential for inserting points outside of them. Attempts to implement Ruppert's Algorithm in 3D therefore had to expend much effort to address these edge cases.

Conclusion

The implementation of Ruppert's algorithm proved to be a subject supported by many intricacies in its foundation to comprehend that allowed for its functionality in a simple and elegant fashion.

The implementation ultimately was relatively straightforward. However, there was a good degree of challenge in understanding why it was possible to perform certain actions in the algorithm without fear of edge cases. In addition, there was a good amount of problem solving involved with implementing the building blocks of the algorithm - of particular note was figuring out how to apply Lawson's algorithm and Bowyer-Watson's algorithm properly to perform Ruppert's algorithm.

For students in the future, implementing Ruppert's algorithm for delaunay mesh refinement would be a good challenge for someone who is more interested in programmatic problem solving while trying to avoid some of the more complex mathematics potentially involved with high level algorithms in graphics.

References

J. R. Shewchuk, 2001, Delaunay Refinement Algorithms for Triangular Mesh Generation
https://people.eecs.berkeley.edu/~jrs/papers/2dj.pdf

C. L. Lawson, 1971, Transforming Triangulations
https://www.sciencedirect.com/science/article/pii/0012365X72900933

G. L. Miller, S. Pav, N. Walkington, 2002, Fully Incremental 3D Delaunay Refinement Mesh Generation
https://www.researchgate.net/publication/2552561_Fully_Incremental_3D_Delaunay_Refinement_Mesh_Generation

Bowyer-Watson Algorithm
https://en.wikipedia.org/wiki/Bowyer%E2%80%93Watson_algorithm