Ezra Davis
ezra.davis@yale.edu
Computational Complexity CPSC 568
Dana Angluin
Final Course Project
Spring 2018
May 15, 2018

Analysis of *Procedural Voronoi Foams for Additive Manufacturing*

1. Introduction

In this document, I will examine the computational complexity of a set of algorithms intended to create varying density flexible foam for 3D printed objects. I will employ a variety of analysis techniques and approaches to analyze computational efficiency in terms of time and space as described in Arora & Barak [2009], Leighton [1991], and Boppana, & Sipser [1989].

There are many reasons one would want to create a 3D printable object that is softer in some areas than others; it could be used in a variety of applications such as prosthetics where you want to control flexibility and bendability or a soft toy that keeps its shape in some parts more than others.

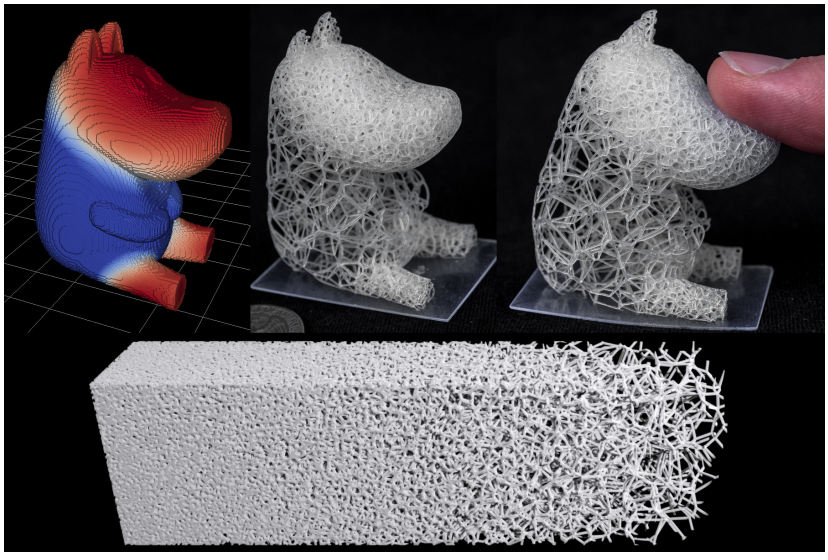


Figure 1. Example of flexible 3D printed toy model [Martínez et al., 2016]

Summary of the Martínez et al. paper

The recent 2016 paper, *"Procedural Voronoi Foams for Additive Manufacturing"* [Martínez et al., 2016], describes one highly parallel method of taking a 3D model and converting it to a 3D printable set of voxels to create an object with varying flexibility. Researchers have been using highly parallel graphics hardware to calculate Voronoi diagrams for many years [Hoff et al. 1999]; however, Martínez et. al [2016], in addition to introducing a new algorithm for determining a Voronoi edge beams, use their method in a novel and interesting application: creating a Voronoi foam for 3D printing.

Martínez, et al. (2016) lay out three simple algorithms for generating a 3D foam based on the desired density at a particular location. For an open cell Voronoi foam (the type of foam described in the paper), the foam's geometry (where the structural beams are) is given by a set of points ("seeds") in the local area, which can be generated pseudo-randomly. The first algorithm gathers all the seeds in a certain area; another algorithm determines which areas influence a particular point; and then the last algorithm determines if a given 3D point ("voxel") is part of the foam (and thus should be printed in plastic). The size of the area that can influence a particular voxel is determined by the foam's minimum density, which is set before calculations begin. Because the algorithms solve for each voxel separately, it is easy to solve for many voxels concurrently, which the authors take full advantage of – even implementing the algorithms on a graphics card to make it calculate blazingly fast.

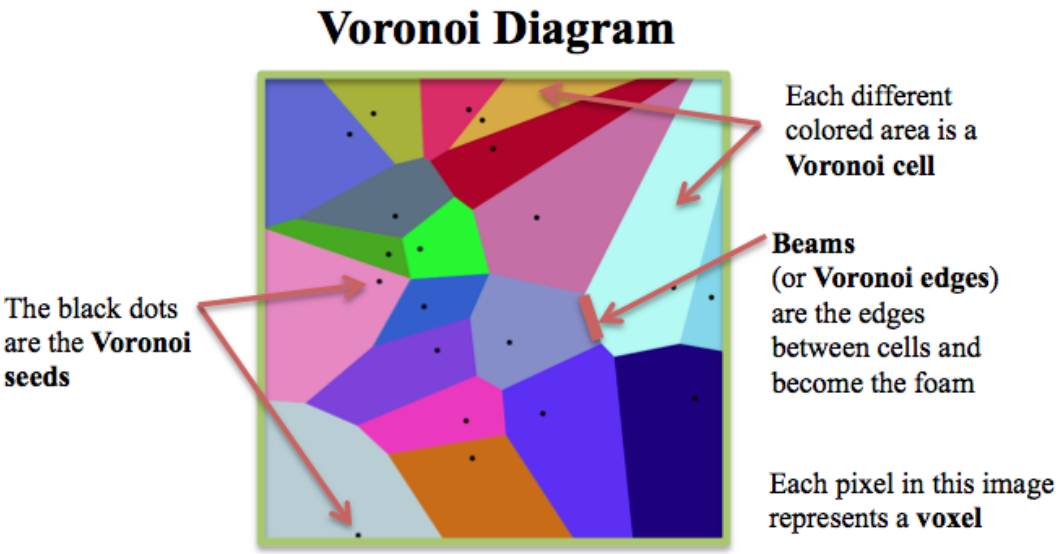
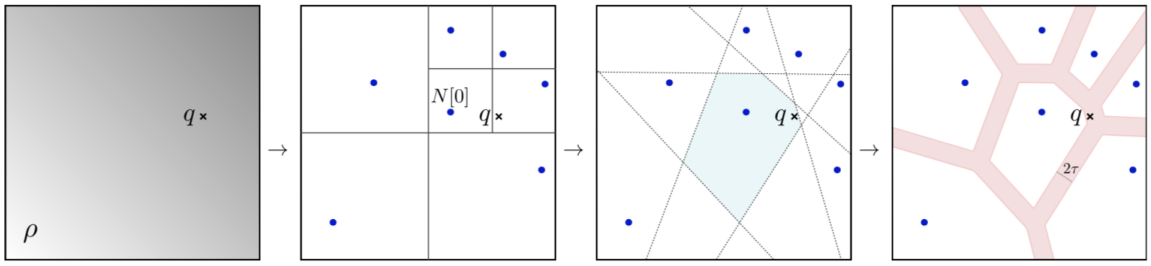


Figure 2. Voronoi Diagram
Euclidean Voronoi diagram illustration courtesy of Ertl [2015]

The algorithms described by Martínez et al. are optimized for performance on a highly parallel machine and are inefficient on a traditional computer processor that does not support parallel processing, resulting in a large amount of repeated computation.

Knowing the complexity of computationally expensive and/or parallel algorithms such as those in “Procedural Voronoi Foams for Additive Manufacturing” is especially important for determining the practicality of those algorithms compared to traditional non-parallel approaches.



(a) *Density field.* (b) *Gather seeds.* (c) *Evaluate structure.* (d) *Voronoi edge beams.*

Figure 2: *2D overview of Algorithm 1. (a) Input query point q and density field ρ. (b) The seeds that could contribute to the Voronoi cell of N[0] (which is the seed closest to q) are gathered (Algorithm 2). (c) The bisectors of the seed pairs influencing q are computed. (d) Finally, the algorithm checks whether q lies inside a beam of radius τ along the Voronoi edge.*

Figure 3. Algorithms described by Martínez et al. [2016]

2. Parallel generation of the Voronoi foam interior

The three core algorithms described by Martínez et al. [2016] define the process of generating an open cell Voronoi foam of varying density. Together, the three core algorithms determine whether or not a single voxel (discrete position in three-dimensional space) should be printed or not. The algorithms operate on a each voxel in the object individually, and are thus called many times to get the final 3D foam. A single call to the algorithms is an exemplar of a decision problem, and the full execution fits nicely into the NC complexity class of highly efficient parallel computable problems [Greenlaw et al., 1995]. Despite the fact that the algorithms use pseudorandom number generators, they do not (and in fact shouldn’t) use true randomness meaning that the algorithms do not belong in the RNC complexity class as defined by Greenlaw et al. [1995].

2.1 Overview of the three Algorithms described by Martínez et al. [2016]

Martínez et al. [2016] divides the problem into three nested parts, starting with the overarching decision problem (Algorithm #1: EvalStructure - *Should I print some plastic there?*) and finishing with a low level algorithm (Algorithm #3: SubdivideCell - *Where are the Voronoi seeds in this cube?*).

- **Algorithm #1 EvalStructure:** Evaluates the structure - Iterating through each of those Voronoi seeds in neighboring grid cells and determining if our query voxel is part of a beam (and thus should be printed)
- **Algorithm #2 GatherSeed:** Determines which grid cells could potentially influence our query voxel, and then calling a SubdivideCell to determine where those Voronoi seeds are.
- **Algorithm #3 SubdivideCell:** Generates Voronoi seeds (centers of the hollow parts of the foam) in a cubic region of space (referred to as a "grid cell"). Subdivdes the grid cell in a pseudorandom method based on the density in the grid cell.

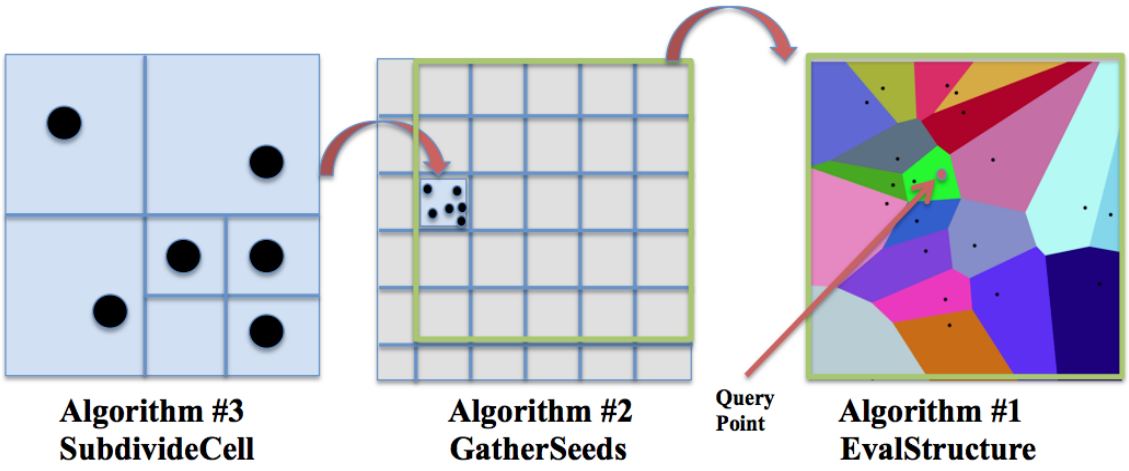


Figure 4. Illustration of the three algorithms in the Voronoi foam in the order of operations used for the complexity analysis.
Algorithm #3 is first used to determine the point in a grid cell.
Algorithm #2 is used to determine which grid cells could possibly affect our query point.
Algorithm #1 determines if the query point is in a beam of the Voronoi foam.

Euclidean Voronoi diagram illustration courtesy of Ertl [2015]

2.2. Order of Operations and Nesting of Algorithms

The original paper explains the algorithms in the order they are numbered: starting with the overarching decision problem and finishing with a low level algorithm. For purposes of computational complexity analysis, I would argue that it makes more sense to evaluate these algorithms in a different and un-nested order. I investigated the three algorithms starting with the simplest and building up to the overarching problem.

Martinez et al. [2016] Nested Order of Operations

Algorithm #1: EvalStructure →
- Algorithm #2 GatherSeeds →
  - Algorithm #3 SubdivideCell (called 125 times)
  - Perform calculations in Algorithm #2
  - Algorithm #3 SubdivideCell (called less than 125 more times)
- End algorithm #2→
- Perform calculations in algorithm #1
- End algorithm #1

Complexity Analysis Project's Order of Operations

Algorithm #3 SubdivideCell (generates Voronoi seeds without calling on other algorithms)
Algorithm #2 GatherSeeds (which calls on SubdivideCell)
Algorithm #1 EvalStructure (which relies on Gatherseeds)

Figure 5. Comparison of Order of Operations between the Martínez et al. (2016) and the order used for the complexity analysis project

2.3 Analysis of the Three Algorithms

Algorithm #3 SubdivideCell

- ***Algorithm #3 SubdivideCell:*** *Generates Voronoi seeds (centers of the hollow parts of the foam) in a cubic region of space (referred to as a "grid cell"). Subdivdes the grid cell in a pseudorandom method based on the density in the grid cell.*

SubdivideCell is a straightforward recursive algorithm that divides a grid cell until the density of a subcell indicates that there is only one seed in each subcell, and psuedorandomly places a Voronoi seed in each of the subcells. The algorithm thus returns a list of Voronoi seeds. This encourages the Voronoi seeds to be roughly evenly spaced, which is a desirable behavior. Note, this algorithm is turned into an iterative algorithm in the actual implementation to avoid stack overflow on a GPU, but this is largely unimportant from a complexity standpoint. It uses a (presumably not secure) pseudorandom number generator seeded with the current grid cell's center to get repeatable results.

Martínez et al. [2016] take advantage of the fact that our final object doesn't need to have truly random Voronoi seed locations, and divides the volume up into grid cubes. Each grid cube *must* have at least one Voronoi seed (providing a minimum density for our object), which means that each voxel in the final foam relies on at most a constant number of grid cells, which I will talk about more in depth below.

The algorithm makes recursive calls to at least $\log_8(\text{density})$ depth (some variation due to the density varying within the cell is expected but is unlikely to be too significant) with 8 recursive calls for each instance of the function. The recursive calls result in a total of $\rho * \log_8(\rho)/8$ calls where ρ is the highest density in that grid cell. And the overall complexity of SubdivideCell is $O(\rho * \log(\rho))$.

Algorithm 3: SUBDIVIDECELL

Input: Density field ρ, cell center c, cell size l.
Output: A set of seed, with a density driven by ρ

$N \leftarrow \emptyset$;
$t \leftarrow l^3 \times \rho(c)$; // target number of seeds in current cell
if $t \leqslant 2^3$ **then**
    $I = RandomPermutation(subcells)$;
    $n_{min} = \lfloor t \rfloor$; // minimum number of samples to draw
    for $i \leftarrow 0$ **to** $n_{min} - 1$ **do**
        $N \leftarrow N \cup \{RandomSampleInSubcell(I[i])\}$;
    $p \leftarrow random(0, 1)$;
    if $p \leqslant (t - n_{min})$ **then**
        $N \leftarrow N \cup \{RandomSampleInSubcell(I[n_{min}])\}$;
else
    for $subcell \in currentCell$ **do**
        $N \leftarrow N \cup$ SUBDIVIDECELL$(\rho, subcell.center, l/2)$;
return N

Figure 6. Algorithm #3 SubdivideCell Pseudocode [Martínez et al., 2016]

Algorithm #2 - GatherSeeds

- ***Algorithm #2 GatherSeeds:*** *Determining which grid cells could potentially influence our query voxel, and then calling a SubdivideCell to determine where those Voronoi seeds are.*

Martínez et al. [2016] use the fact that SubdivideCell will always return at least one Voronoi seed. Since a grid cube *must* have at least one Voronoi seed, each voxel is influenced by only those in grid cells that are in the nearby neighborhood. A voxel is influenced by the same seeds the Voronoi cell it is within (which corresponds to the closest Voronoi seed). A Voronoi seed's cell can only be influenced by a seed in a grid cell that is within the surrounding nearby cells (up to a little less than the closest 125 cells). These 125 cells are called the two-ring neighborhood of that Voronoi seed.
The nearest Voronoi seed to our query point can be anywhere within a two-ring neighborhood.

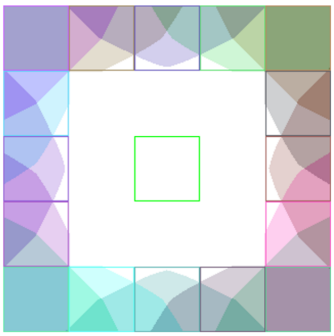


Figure 7. Algorithm #2 GatherSeeds illustration.
A 2D cross section of a two-ring neighborhood of a grid cell.
A Voronoi seed in the cell outlined in green could be influenced by a point in any of the non-shaded areas. [Martínez et al., 2016]

The upshot of this, is that we have a minimum and theoretical maximum number of grid cells that could influence our voxel. 125 is the minimum (simply assuming that the Voronoi seed closest to the query point is within the same grid cell) to a maximum of roughly 125 + 125 – 27 = 223 grid cells.

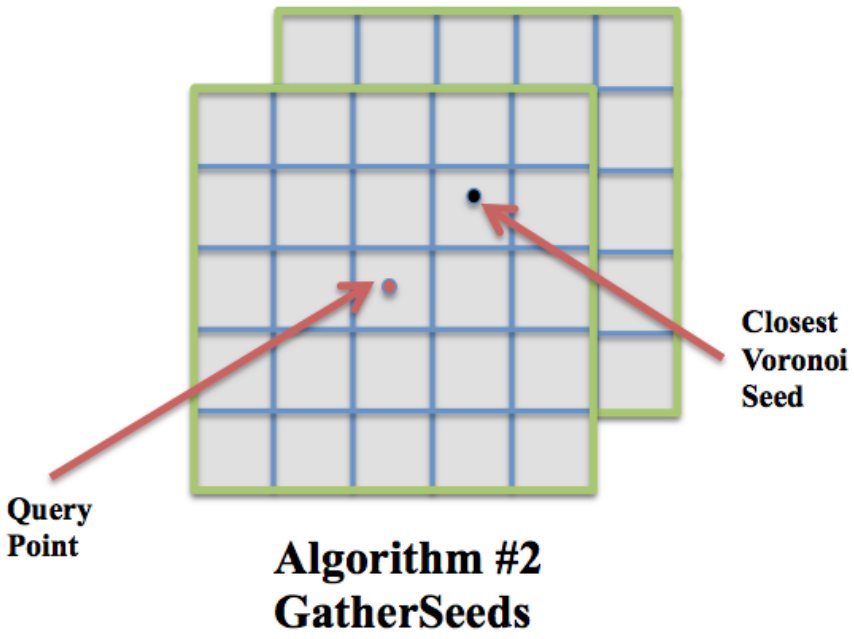


Figure 8. Algorithm #2 GatherSeeds illustration:
A possible set of grid cells collected by the algorithm

We could trim down the number of cells that we investigate (generate seeds for) significantly by determining if those cells had any location that could interfere with our query point, but EvalStructure (the algorithm that calls GatherSeeds) works fine when given the Voronoi seeds from a few extra grid cells, even if it becomes somewhat slower.

While visiting each cell in a two ring neighborhood, the algorithm keeps track of the closest Voronoi seed which takes a negligible amount of time, on the order of $O(\rho)$ (each Voronoi seed need only be checked once) - less than multiplying a relatively small constant time with the time spent finding Voronoi seeds in each cell.

We can easily show that GatherSeeds has the same asymptotic behavior as SubdivideCell - $O(\rho * \log(\rho))$, as it cannot call SubdivideCell more than a constant number (223) times.

Aside: One would expect that GatherSeeds would vary exactly the same with respect to density as SubdivideCell, but because more Voronoi seeds means that the closest seed must be closer to the query voxel, this isn't quite the case. If each of the grid cells contain at least 8 Voronoi seeds, the closest Voronoi seed to the query point must be in a one-ring neighborhood of its grid cell, meaning that GatherSeeds queries SubdivideCell at most $5^3 + 5^3 - 4^3 = 186$ times. This does not affect the asymptotic behavior of GatherSeeds however, as the closest Voronoi seed to the query point can always be in any of the cells in the one-ring neighborhood of the query point's grid cell.

Algorithm 2: GATHERSEEDS(ρ, q)

Input: Density field ρ, query point q.
Output: Set of seeds possibly influencing q

```
1  N ← ∅ ;
2  Visited ← ∅ ;
3  cq ← GridCellEnclosing(q) ;
4  closest ← ∞ ;
5  for cell ∈ TwoRingNeighborhood(cq) do
6  |  Visited ← Visited ∪ {cell};
7  |  seeds ← SubdivideCell(ρ, cell.center, cell.length);
8  |  N ← N ∪ seeds;
9  |  for s ∈ seeds do
10 |  |  if ||s − q|| < ||closest − q|| then
11 |  |  |  closest ← s;
12 cs ← GridCellEnclosing(closest) ;
13 for cell ∈ TwoRingNeighborhood(cs) \ Visited do
14 |  N ← N ∪ SubdivideCell(ρ, cell.center, cell.length);
15 return N
```

Figure 9. Algorithm #2 GatherSeeds Pseudocode [Martínez et al., 2016]

Algorithm #1 EvalStructure

- **Algorithm #1 EvalStructure:** Iterate through each of those Voronoi seeds in neighboring grid cells and determining if our query voxel is part of a beam (and thus should be printed)

The EvalStructure algorithm determines if the query voxel is actually within a beam of the foam. The algorithm compares each point
The query voxel is contained (or borders) the Voronoi cell defined by the Voronoi seed that is closest to the query voxel.

Intuitively, this algorithm has a $O(n^3)$ where n is the number of seeds (which has a linear relationship with the desired density ρ at high densities), but the average running time is significantly better, especially if the Voronoi seeds are sorted by distance to the closest Voronoi seed (which takes $O(n \log n)$ time).

If the external for loops run to completion (which they do for all query points that are not in a beam), they run for $\frac{n^2}{2}$ times. The if statement (on line 6 of the pseudocode in figure 10) will rarely evaluate to true, and never will if

$$dist(N[0], q) + \tau < \min\left(dist(N[i], N[0]), dist(N[j], N[0])/2\right.$$

the beam between $N[0]$, $N[i]$, and $N[j]$ cannot be more than halfway to $N[0]$ from either of the other Voronoi seeds, and thus cannot reach the query point if it is very close to reach the query

point if the query point is $N[0]$). The situation above is especially common when the density is high and thus the query point must be especially near to $N[0]$.

Using the reasoning above, I would argue that the average running time of EvalStructure is bounded by $O(\rho^2)$.

Algorithm 1: EvalStructure $\mathcal{F}_{\rho,\tau}(q)$

Input: Density field ρ, beam radius τ, query point q.
Output: Voxel state $\in \{0, 1\}$

```
1  N ← GatherSeeds(q) ; // seeds influencing q
2  for i ← 1 to |N| do
3      for j ← i + 1 to |N| do
4          bl ← BisectorLineEquation(N[0], N[i], N[j]);
5          pl ← ClosestPointOnLine(q, bl);
6          if ‖q − pl‖ ⩽ τ then
7              accept ← true;
8              for k ← 1 to |N| do
9                  if ‖N[0] − pl‖ > ‖N[k] − pl‖ then
10                     accept ← false;
11                     break;
12             if accept then
13                 return 1
14 return 0
```

Figure 10. Algorithm #1 EvalStructure Pseudocode [Martínez et al., 2016]

2.4 How density of the foam affects running time/space

The density function of the foam has a $O(\rho * \log(\rho))$ time relationship with the amount of time spent in the GatherSeeds algorithm and a $O(\rho^2)$ relationship with the time spent in the EvalStructure algorithm outside of GatherSeeds, which causes that part of EvalStructure to dominate the running time for large values of ρ. The desired density of the foam directly affects the running time of the algorithms by a factor of $O(\rho^2)$.

The space requirements of the three algorithms are very straightforward: All three algorithms must store a number of Voronoi seeds that correspond to a constant volume, and thus take up $\theta(\rho^2)$ space.

2.5 Comparison with traditional serial Voronoi diagram algorithms

I was inspired to analyze Martínez et al.'s paper [2016] because I had attempted to implement the algorithms myself. I originally chose to implement them in a serial programming language.

After implementing the three algorithms described in the paper serially, I discovered why the paper's authors made sure that the program could run in parallel; if you don't run the program in parallel, it takes a long time to generate each voxel (every point in 3D space).

I was curious if I could make my program faster without rewriting it in another language, so my first step was determining where my code spent most of its time. Using cProfile (a Python profiling tool), I discovered that the program's time was roughly evenly split between generating the points to determine if a voxel is in the foam or not and a small function that is used to find the bisector line between three seeds (points). This is used to figure out where the "beams" of the Voronoi foam are. The bisector line function is called for each pair of points generated, which makes a simple function (the most expensive operation it performs is a single cross-product) take so much time.

Next I investigated ways of making that function faster – the most straightforward method was using Numba, a Just In Time compiler for Python. Unfortunately, while it did improve the runtime, it doesn't improve the function dramatically enough without a major reorganization of my code, so I looked for other options.

Then I decided to draw multiple voxels at once, using a faster library (SciPy's API to qhull) to generate a 3D Voronoi diagram in less than one operation per pair of Voronoi seeds per pixel. I then used this to generate a 2D slice of my final model. This was still significantly slower than the original paper, but fast enough that I could generate 3D foams in a (nearly) reasonable amount of time.

2.6 Time comparison (both parallel and serial)

Traditional serial Voronoi diagram algorithms (such as those I used in my previous exploration of the problem) take $O(v \log v)$ to $O(v^2)$ in degenerate cases time to determine the locations of the Voronoi edge beams over a region [Rebay, 1993]. Algorithms such as Fortune's algorithm that always take $O(v \log v)$ unfortunately only work in two dimensions [Fortune, 1987]. v is the number of Voronoi seeds, which is the foam density times the volume. These Voronoi edges can be used to draw lines in a voxel grid very rapidly - only touching voxels that contain a line,

the proportion of which scales roughly with the desired density. If the size of the final volume is n by n by n, the foam can be drawn in roughly $O\ (n^3 * \rho * \log(n^3 * \rho) + n^3 * \rho)$ time.

The parallel Voronoi diagram algorithms described above however, takes $O(\rho^2)$ time for each voxel. On a serial machine (or small parallel one), the parallel algorithms take $O(N^3 * \rho^2)$ time, which, with the constant term, surprisingly ends up being slower than the serial version for most reasonable values of ρ. However, for a large parallel machine that can handle an arbitrary number of voxels at a time, the algorithm only takes $O(\rho^2)$ time, which is much better than the serial algorithm above (so long as the foam isn't incredibly dense).

2.7 Space complexity comparison

Both the parallel and serial algorithms must have enough space for all of the Voronoi seeds and all of the voxels. For some specific collection of Voronoi seeds the representation of the Voronoi diagram can $O(n^2)$ space, but in almost all cases, it takes up only $O(n)$ (where n is the number of Voronoi seeds) [Golin & Na, 2003]. Unlike many serial algorithms, the parallel algorithm can be run very incrementally, sacrificing time efficiency but allowing the parallel algorithm process to be very space efficient - with barely more memory than it takes to store the voxels.

3. Ancillary Algorithms

In addition to the three core algorithms described in detail above, there are some additional algorithms required to create a 3D printable result.

3.1 Parallel generation of external frame (foam intersecting edge of the model/mesh/object)

The paper by Martínez et al. [2016] is slightly less clear in indicating the exact implementation (and thus, performance characteristics) of the process of generating a frame for the Voronoi foam (preventing it from having loose ends or unsupportable portions of the foam).

Depending on the organization of the 3D model, I believe there are two clear possibilities for how to organize the algorithm:

Option 1. Determine if each voxel is part of the frame during the EvalStructure step.
Option 2. Instead of iterating through voxels, iterate through each part of the 3D model (whether that is each triangle, section of a curve, or part of a parametric function isn't too important for our runtime analysis)

Depending on the exact implementation, the amount of time spent in each of these options can vary wildly, so I did not pursue this for further analysis.

3.2. Detecting unprintable voxels

This is perhaps the simplest algorithm discussed in the paper: simply removing voxels that overhang too much compared to their neighbors. Each voxel only needs to examine their immediate neighbors and determine if they can remain attached. If a voxel cannot be supported by its neighbors, you need to recalculate any voxels that rely upon the newly removed voxel. This is easy enough to avoid in a serial program, but it is far more desirable to do a parallel program. Fortunately, the cases where an unsupported voxel causes its neighbors to become unsupported is sufficiently rare (<0.1% of the total volume according to Martínez et al. [2016]) that I'm not too worried about the additional runtime introduced by this recalculation, allowing us to perform almost a constant number of calculations for each voxel.

Figure 9. Photo of 3D printing of flexible foam

4. Conclusion

Overall Martínez et al. have offered up an elegant, practical, and fast solution to printing 3D foam structures with varying density properties. This is a promising area of research and parallelization has interesting impacts on determining the computational complexity of algorithms that solve problems with real world applications.

Further avenues for exploration

Some areas of further study would include a more detailed analysis of parallel vs. sequential approaches to more closely analyze effects on economy of time and space (potentially including average case running time, determining tighter bounds, including hardware concerns, or alternate implementations of the algorithms). An analysis of the possible choices for implementing the ancillary algorithms would also be an interesting avenue of further work.

6. References

Arora, S. & Barak, B. (2009). *Computational Complexity: A Modern Approach.* Cambridge University Press. 594p. http://theory.cs.princeton.edu/complexity/

Boppana, R. B., & Sipser, M. (1989). *The Complexity of Finite Functions*. https://www.researchgate.net/publication/242100641_The_Complexity_of_Finite_Functions

Ertl, B. (2015). Euclidean Voronoi diagram, via Wikimedia Commons [CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0)], Retrieived 2018 from https://commons.wikimedia.org/wiki/File:Euclidean_Voronoi_diagram.svg

Fortune, S. (1986). A sweepline algorithm for Voronoi diagrams, In *Proceedings of the second annual symposium on Computational geometry, SCG '86,* p.313-322. https://dl.acm.org/citation.cfm?id=10549

Golin, M.J. & Na, H-S (2003). On the average complexity of 3D-Voronoi diagrams of random points on convex polytopes, *Computational Geometry* 25, p197-231. https://home.cse.ust.hk/~golin/pubs/3D_Voronoi_I.pdf

Greenlaw, R., Hoover, J., & Ruzzo, W. L. (1995). *Limits to Parallel Computation: P-completeness Theory*. Oxford University Press. 311p. https://homes.cs.washington.edu/~ruzzo/papers/limits.pdf

Hoff III, K. E., Keyser, J., Lin, M., Manocha, D., & Culver, T. (1999). Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, p. 277-286. July ACM Press/Addison-Wesley Publishing Co. https://dl.acm.org/citation.cfm?id=311567

Leighton, F. T. (1991). *Introduction to parallel algorithms and architectures: Arrays· trees· hypercubes*. Elsevier. https://www.elsevier.com/books/introduction-to-parallel-algorithms-and-architectures/leighton/978-1-4832-0772-8

Martínez, J., Dumas, J., Lefebvre, S. (2016). Procedural Voronoi Foams for Additive Manufacturing. *ACM Trans. Graph.* 35, 4, Article 44 (July 2016), 12 pages. DOI = 10.1145/2897824.2925922 http://doi.acm.org/10.1145/2897824.2925922.

Rebay, S. (1993). Efficient Unstructured Mesh Generation by Means of Delaunay Triangulation and Bowyer-Watson Algorithm. *Journal of Computational Physics* Volume 106 Issue 1, May 1993, p.127. https://www.sciencedirect.com/science/article/pii/S0021999183710971