Solid Freeform Fabrication 2019: Proceedings of the 30th Annual International Solid Freeform Fabrication Symposium – An Additive Manufacturing Conference
Reviewed Paper

Using Parallel Computing Techniques to Algorithmically Generate Voronoi Support and Infill Structures for 3D Printed Objects

T. Williams, S. Langehennig, Prof. M. Ganter, Prof. D. Storti
Dept. Mechanical Engineering, University of Washington, Seattle, WA, 98195

Abstract

Many methods of 3D printing rely on support and infill structures in order to produce quality parts. This paper formulates an algorithm that produces support and/or infill structures based on Voronoi cells for objects described by a function or a closed triangulated mesh. The algorithm utilizes Voronoi structures with a high degree of customization provided to the end user, and takes advantage of parallel computing to cut down on the computation time required to generate these structures. The aforementioned method is novel because it uses Voronoi structures as supports and combines support and infill generation into a single process, displaying the flexibility of Voronoi foam structures in 3D printing applications. The primary focus is the implementation of the algorithm itself and the customization capabilities it provides.

Introduction

3D printers (in one form or another) have existed since 1980, and a majority of them have some reliance on support structures. When working with fused filament fabrication (FFF) 3D printers, vat photopolymerization (VPP) 3D printers, and material jetting 3D printers, sacrificial material must be provided to support the sections of the object that have large bridges or overhangs. Support material often takes one of two forms: a linear (or serpentine) pattern, or a treelike structure. Both of these structures provide the printer with a form of scaffolding on which to build overhangs. Fig. 1 shows the 3DBenchy model with linear and treelike support structures generated and rendered in FlashPrint, in addition to the Voronoi-based support structure this paper is focused on [1, 2].

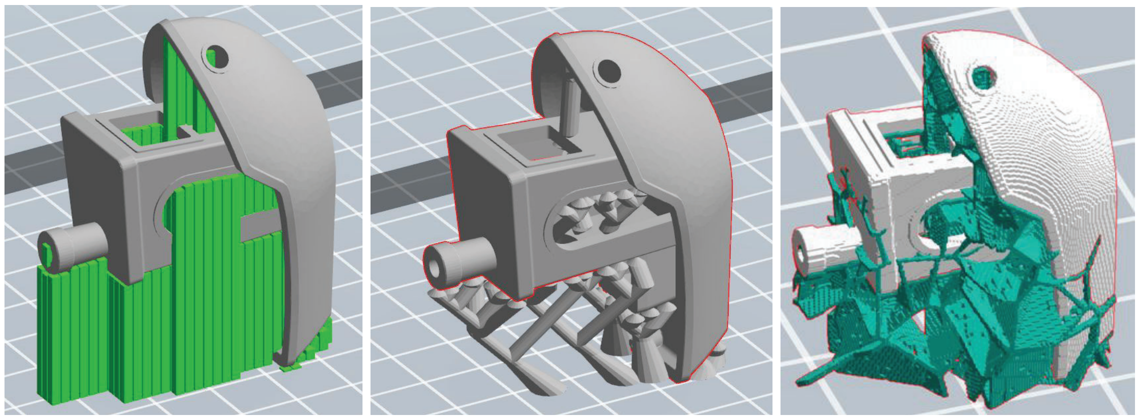

Figure 1: *Linear Supports (Left), Treelike Supports (Middle), Voronoi Supports (Right)*

Many 3D printers, such as the FlashForge Dreamer or the Prusa i3 MK3 MMU2, have the capability of printing in multiple separate materials, allowing supports to be made of a material that can be removed using a solvent. For example,the model can be made of PLA or ABS, while the supports can be printed using PVA, which can be dissolved in isopropyl alcohol. The usage of multiple materials allows for the supports to be directly connected to the model, producing a higher-quality surface finish than one would get by attempting to break the supports off of the model manually. Multiple material usage also eases the process of support removal. The manual labor of removing supporting material has been reduced to submerging the entire print into a bath of the support material's required solvent.

On a similar note, much thought has been put into the infill patterns used for 3D printing. 3D printing a part at full density is often a waste of both material and print time, so an 'infill' structure is generally used to fill in the object enough to provide structural support, but not so much as to spend a disproportionate amount of time and material. Most modern slicing software packages, such as Slic3r, Cura, and FlashPrint, offer a variety of infill patterns including hexagonal or linear grid, but design of infill patterns remains an active area of study.

Recently, studies have demonstrated that Voronoi foam structures can be used as effective infill structures [3-6]. By definition, Voronoi foam structures are geometric constructs based off of point fields. Each cell of the foam contains all points in space that are closer to that cell's seed point than any other cell's seed point, with boundary walls separating the cells [7]. A 2D example is provided in Fig. 2. As the seed points can be distributed non-uniformly, Voronoi foams, have the benefit of allowing for a gradient of infill percentages to be used, as the cells can be distributed in higher density to produce a more rigid section, lower density to produce a more flexible section, or anywhere in between [3-5]. The fact that the distribution of the seed points can be determined in any number of ways provides a much higher level of control to the designer while setting up their 3D prints. These properties can be useful in producing parts with material properties and/or functionality that would be difficult or impossible to achieve with standard infill techniques, let alone traditional manufacturing processes.

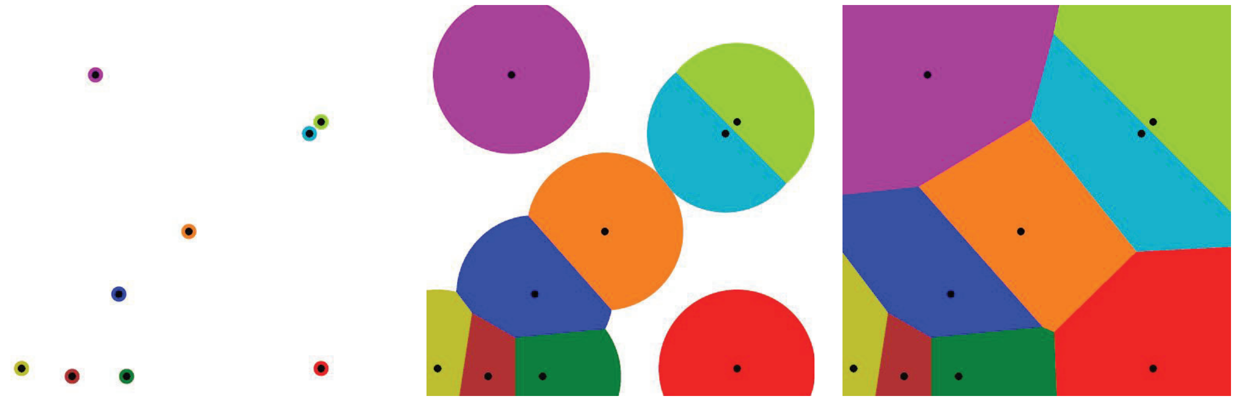

Figure 2: *A process of generating 2D Voronoi cells, black dots represent the seed points. The cells can be geometrically defined by drawing expanding circles from the seed points [8]*

Voronoi structures are often produced in discrete, or voxel, space. As such, this can lead to incredibly long computation times if performed on a single processor in serial mode. Due to these issues, the algorithms covered in this paper are often written to take advantage of parallel computing. Parallel computing is the concept of running several independent tasks simultaneously to speed up computation time. This is in contrast to serial computing, where all tasks are performed sequentially. CUDA, a parallel computing platform, was developed by Nvidia and released in 2007. CUDA allows the programmer to work directly with certain Nvidia graphics cards and take advantage of the high processor count to run large numbers of tasks at the same time, so long as the tasks are not interdependent. Parallel computing has powerful applications in voxel modeling and functional representation, as each voxel's value can often be calculated solely using the defining function, independent of the values of the other voxels in the model.

In this paper, we will be exploring a combination of algorithmically generated infill and support structures, Voronoi foam structures, and parallel computing. This paper introduces a parallel computing algorithm that generates support structures for 3D models that utilizes the properties of Voronoi foams. The same algorithm can also be used to generate infill structures, and this usage is also discussed in the Methods of Implementation.

Proposed Alternative

As an alternative to modern support structures and infill patterns, we propose the usage of a Voronoi-inspired 3D structure to increase the level of customization that is available to the end-user. As mentioned earlier, Voronoi cell structures are based on sets of discrete points. Once the points have been placed, the cell walls are defined as the planes of equal distance between point pairs. All volume within a given cell is closer to that cell's seed point than any other seed points. The seed points can be generated with the assistance of a weighting function to help with strategic point placement, decreasing the likelihood of material being placed where it is not needed.

Once the points have been generated, there are many ways to actually produce the Voronoi cell structure [7,9-10]. This paper utilizes a discrete approximation method that utilizes the jump flooding algorithm run on a parallel computing system [9].

Methods of Implementation

Supports:

In order to implement Voronoi cell support material, it is simpler to first convert the entire model into a form that is more convenient to work with for our purposes. We convert the triangulated model (in the form of an STL or PLY file) into a voxel model through a slicing and voxel-filling algorithm written by Christian Pederkoff [11]. The resulting voxelized model is termed 'Model A', and is defined as a 3D array with each cell in the array defining the qualities of a single voxel. The value of a cell defines its location within the context of the model; negative values are inside the model, positive values are outside the model, and zero values are on the surface of the model. As visible in Fig. 3, there is minimal loss of detail at the current resolution of 175x175x288 voxels.

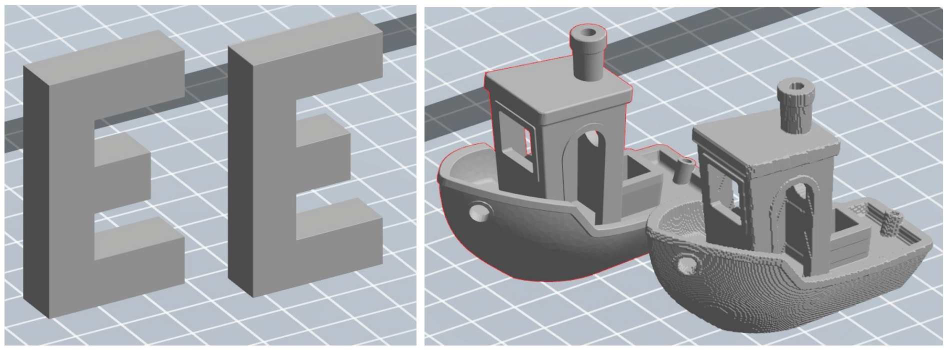

Figure 3: *Left to Right: Original E Model, Voxelized E Model (Model A), Original Benchy, Voxelized Benchy [1]*

From model A, a ‘shadow’ is generated by **projecting** the voxels down along the vertical axis. The projection process produces a solid block of material with the original model trapped within, we will call this array ‘Model B’, as shown in Fig. 4. Bolded algorithms are further explained in the appendix, allowing for the Methods section to describe the basic structure of the process, while the intricate details are left for the curious reader.

Next, model A is removed from model B via **boolean subtraction** [12]. The subtraction process leaves a solid support below every overhang on the model. Fig. 4 shows the updated model B, with the green material displaying the support block.

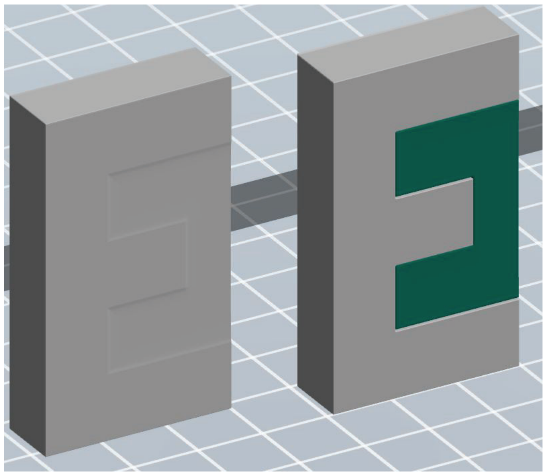

Figure 4: *Model B before (Left) and after (Right) boolean subtraction*

Model B is then modified to be a **depth distance field**, and from here we can generate the seed points for our Voronoi cells. An array of equal size to array B is generated, we will call this ‘Point Field C’. The separate array is generated to preserve the information from model B for future use. Point field C will then be **populated with random points**, weighted to lead to a larger concentration of points where the distance field in Model B has a lower value, in this case near the underside of the object.

Now that Point Field C has been generated, we have to produce the walls of the Voronoi cells. As there can easily be upwards of three or four hundred seed points, a direct distance calculation run on each cell of the 3D array to measure the distance to each of the seed points would be fairly computationally expensive, even in parallel. Instead, we can use the **jump flooding algorithm** (JFA) [9]. The JFA produces a 4-dimensional array that we will call ‘Model D,’ where every cell contains the coordinates of the seed point it is closest to.

With Model D, we can define one Voronoi cell as a collection of cells in the array that all contain the coordinates of the same seed point. However, the model we want is just the cell walls. In order to extract this information, we will need to produce another 3D array of equivalent size to point field C, henceforth known as 'Model E.' Each cell in Model E checks the first three values of the corresponding vector in Model D, and then compares those three values to the equivalent values stored in all adjacent cells in Model D. If any of the adjacent voxels have values that differ from the values of the initial voxel, the cell in Model E takes the value of -1, otherwise it takes the value of 1. In short, if that voxel is at the border of a Voronoi cell, it takes on a negative value. The result of this process can be seen on the left in Fig. 5.

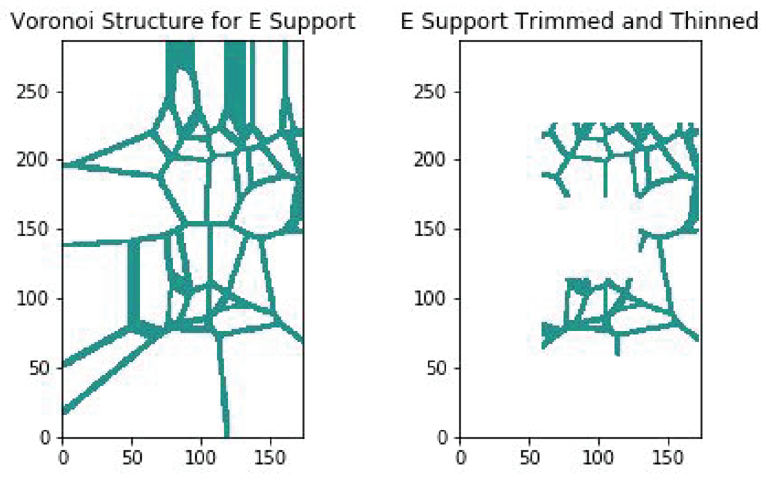

Figure 5: *Slice of the Voronoi structure Model E (Left), and Voronoi structure after intersecting Model B and thinning (Right), both taken halfway along the Y axis of the model.*

Now that we have our Voronoi cell wall structure, we can see that the voxel model in Model E is currently two to three voxels thick in most places. The thickness of the model can be changed by calculating its **signed distance field** (SDF) and adding a constant to the value of each voxel [9]. A positive constant will decrease the thickness by twice its value, while a negative constant will increase the thickness by twice its value, as subtracting a larger constant will make more voxels drop below zero. The additive process modifies the thickness of the entire object at a low computational cost.

Next, the **boolean intersection** is calculated between Model B and Model E, producing a Voronoi support structure that exists exclusively below the original model, removing any material that would intersect with the original object or extend beyond the 'shadow' of the original object [12]. The intersection operation lowers wasted material, decreases print time, and ensures that the support structure does not overlap or intersect with the model itself. The results are shown on the right in Fig. 5 and on the left in Fig. 6, as the green portion of the model.

In order to speed up the process of dissolving the supports from the model, the cells need to be perforated to allow for fluids to access the insides of the cells, increasing the accessible surface area of the Voronoi cell supports. The perforation process can be done by enlarging each of the seed points in Point Field C (Using the same SDF technique as was used for the cell walls) and projecting them along all three axes in both positive and negative directions. The modified Point Field C is then boolean subtracted from Model E, ensuring that each Voronoi cell has at least six holes for fluid to pass through. The results of this step are shown on the right in Fig. 6, as the green portion of the model. The perforations also have the benefit of allowing resin to drain out of the supports if used on a VPP printer.

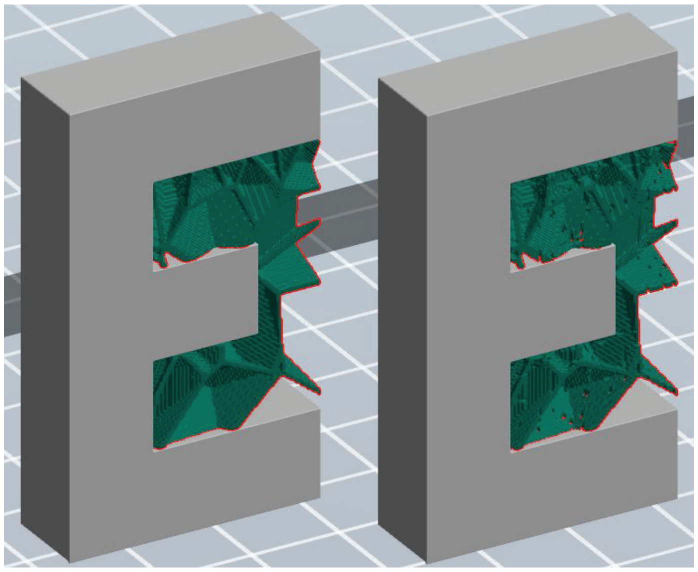

Figure 6: *Voronoi support structure (Left), Voronoi support structure with perforations (Right)*

Finally, to increase surface quality, a sacrificial layer is generated. Our thin, sacrificial layer is offset from the underside of the object by two voxels, providing a platform to ensure that the entire model is being supported. The layer is created by making a copy of Model A, translating the copy several units down in the X axis, and **boolean intersecting** the copy with Model B, shown in the left of Fig. 7 [12]. The sacrificial layer is then **boolean unioned** to Model E to complete the support structure, shown on the right of Fig. 7 [12].

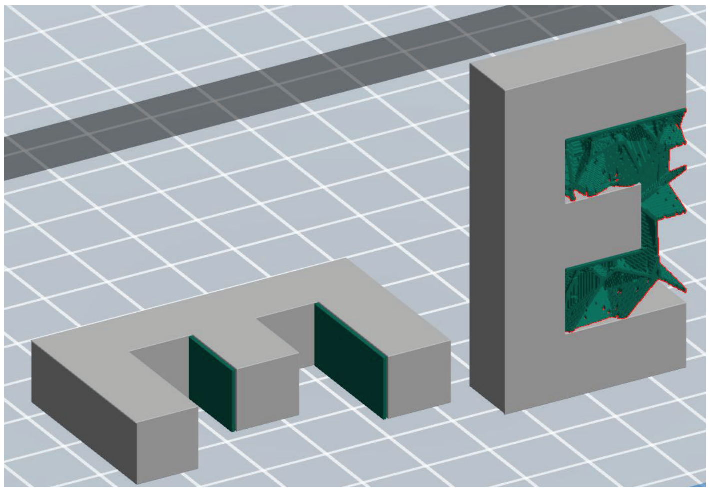

Figure 7: *The sacrificial contact layer (Left, reoriented for clarity) and the completed Voronoi support structure (Right)*

Now that the support structure has been fully generated, the last step is to scale it to the proper size and export it back into a triangulated mesh, in this case through the application of a marching cubes function packaged with the scikit-image library [13]. A complete flow chart of this process is available in Fig. 8, with each labeled array defined in Table 1.

Table 1: Definitions of Named Matrices	
Model A	Voxelized Model of Input Mesh
Model B	Depth Distance Field of Support Block
Point Field C	Weighted Random Point Field
Model D	Voronoi Foam
Model E	Trimmed and Thinned Voronoi Foam

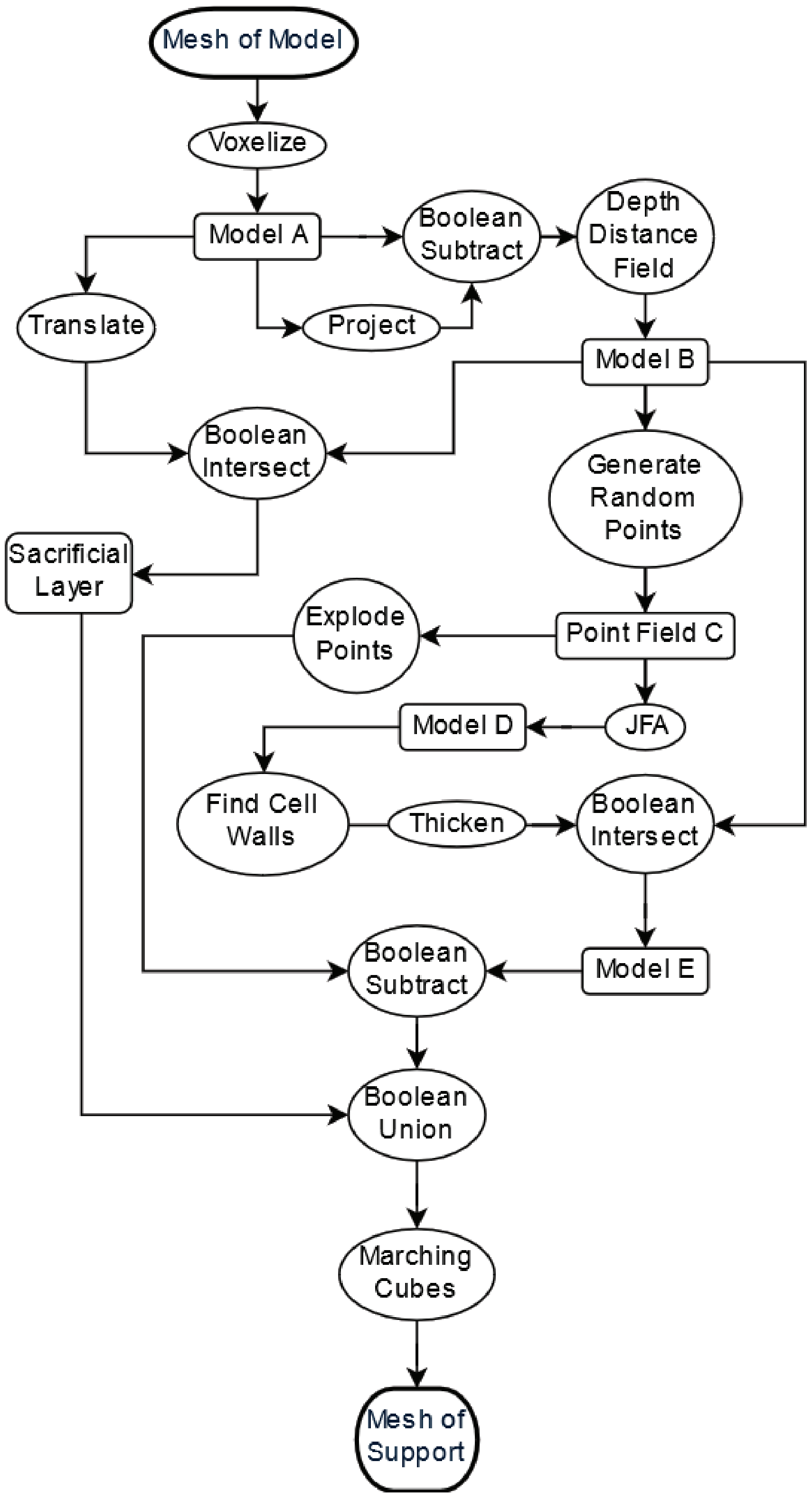

Figure 8: *Data Flow Diagram for Support Generation*

Infill:

In order to produce the infill structure, a very similar procedure is followed. First, a distance field must be calculated to provide the weighting information for seed point generation. In this case, the SDF of Model A is calculated and used as the input for the seed point generation. Next, the JFA is used to produce the walls of the Voronoi cells and the thickness is adjusted using the SDF of the cell walls to ensure that the support structure is thick enough to print properly, but not so thick that it takes more layers than needed. The results of this process can be seen in Fig. 9.

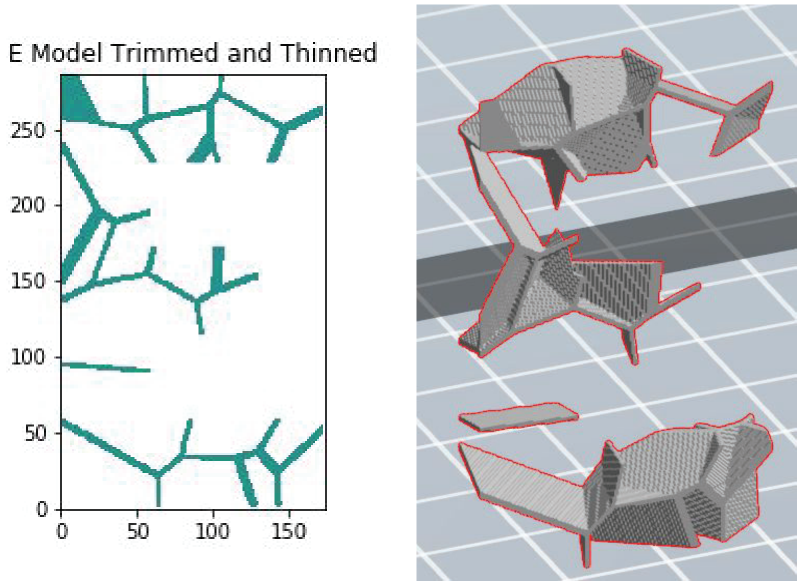

Figure 9: *Slice of the infill structure (Left), half of 3D model (Right)*

In order to ensure that the Voronoi structure remains inside the model, we then need to perform a boolean intersection between the freshly generated cell walls and Model A. The resulting trimmed cells make up the complete infill structure, but this leaves us without the 'skin' of the model. This can be remedied by producing a 'shell' of Model A and boolean unioning the shell with the infill structure. The shell can be obtained by taking the SDF of Model A and setting all negative values with a magnitude above a user-defined threshold to 1. The resulting model now has a facade that matches that of the original object, complete with the Voronoi infill structure. The results of this process can be seen in Fig. 10.

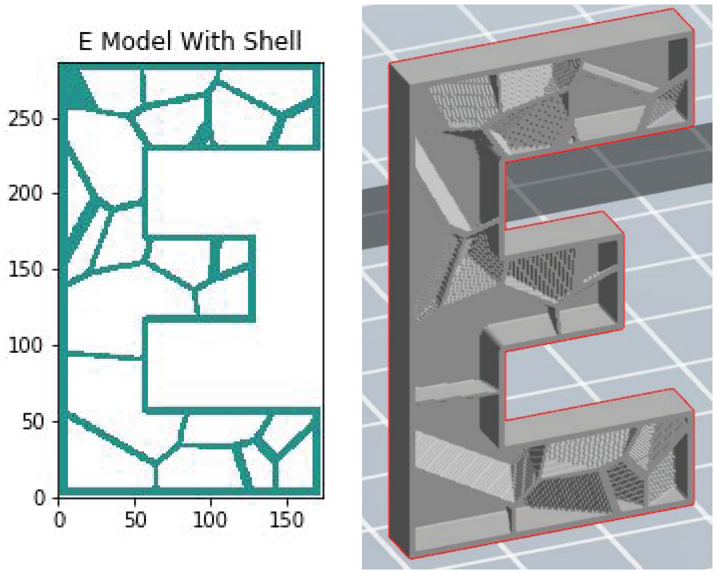

Figure 10: *Slice of the infill structure with a shell (Left), half of 3D model (Right)*

At this point, both the support structure and the model with infill have been generated. If the supports are intended to be broken away, we can boolean union the resulting matrices together to obtain a single model, otherwise both the support and model arrays can be kept separate, as visible in Fig. 11. The final models are then sent through the marching cubes algorithm to generate STL or PLY files, then can be imported into a slicing software for printing.

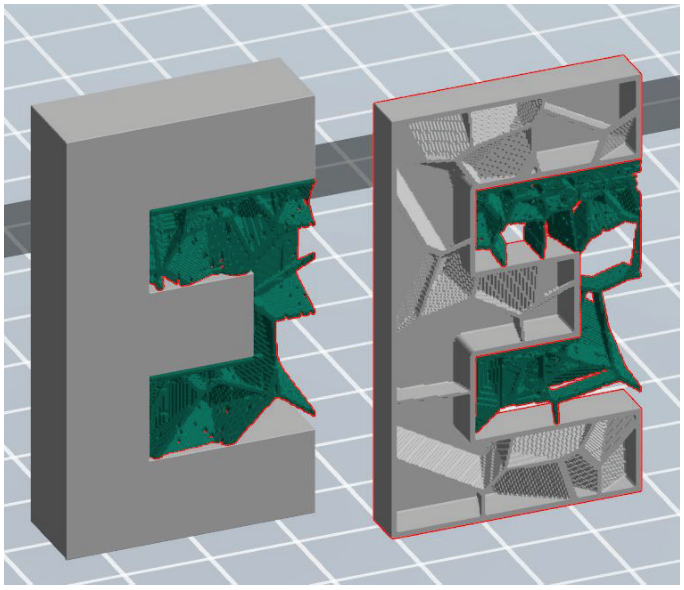

Figure 11: *Model with support and infill, next to sliced in half for internal view*

We can see in Fig. 12 that the prototype model discussed up to this point is capable of being printed using a multi-material 3D printer.

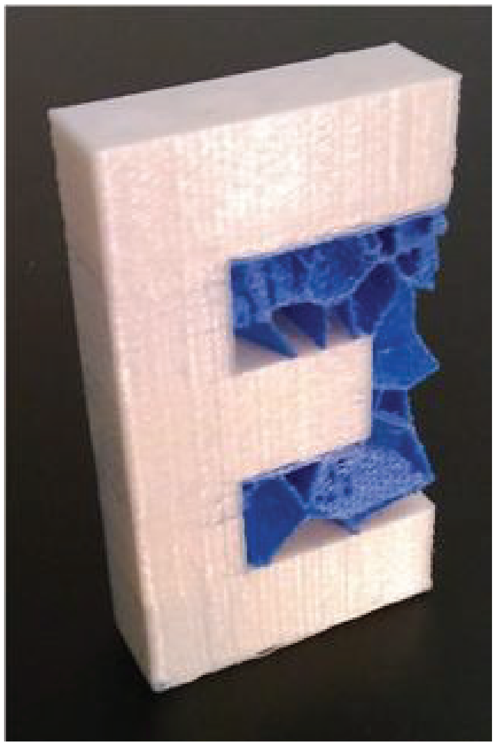

Figure 12: *A printed model with Voronoi supports*

In addition to the traditional methods of 3D printing, which usually require a meshed model to function properly, the model can also be exported as an image stack. Image stacks can be used by certain VPP printers as input data, as the images are projected into the resin directly as a way to selectively cure the material. One of the benefits of having the model in a voxel state is that each voxel can define the material type of its relevant pixel in the image stack.

In order to take advantage of the current multi-material VPP technology, we can choose different types of materials to make up the support and the model itself. As the different materials can be cured by different wavelengths, we can set all support pixels to be colored to match the cure wavelength of the support material, while the model pixels can be colored to match the cure wavelength of the model material [14]. Fig. 13 shows several images from one of these potential image stacks. The support material is colored blue, the model material is colored red, and the empty space is left as black.

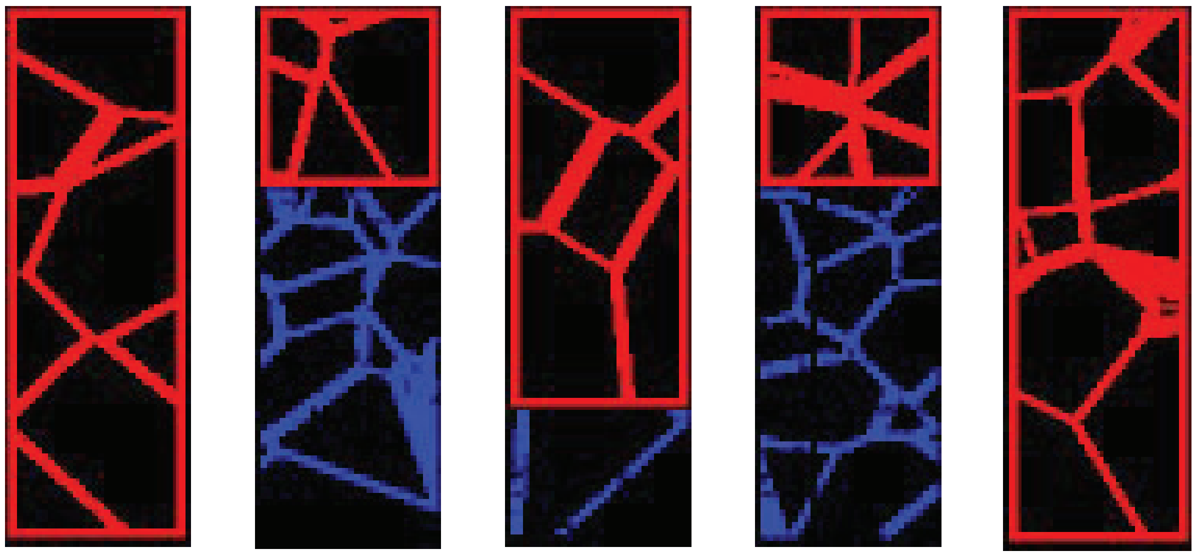

Figure 13: *Sample images from image stack used for VPP process, taken from layers 50, 100, 150, 200, and 250 (Left to Right)*

Discussion

This method of infill and support generation comes with the benefit of being incredibly customizable. The thickness of the cell walls, the number of cells, and the point field density distribution are all adjustable. The level of control that the user has over the infill and support generation is well beyond what can usually be found in commercial slicing software, and provides the option of having a gradient of infill and support density, a feature that allows for structures that have smooth transitions of material properties.

In addition to the letter E model that was used to display the methods, additional models have also been tested, including the ubiquitous 3DBenchy model shown in Fig. 14 [1]. The support generation algorithm was able to deal with the challenges of the curved surfaces, overlapping overhangs, and uneven contact surfaces without difficulty, displaying the robustness of this method.

Preliminary tests of these infill and support structures also show promise, as seen in Fig. 12. Though more difficult to manually remove than treelike supports, the Voronoi structures do leave a high-quality surface finish and have shown a higher success rate, with the treelike supports occasionally failing mid-print due to wobble.

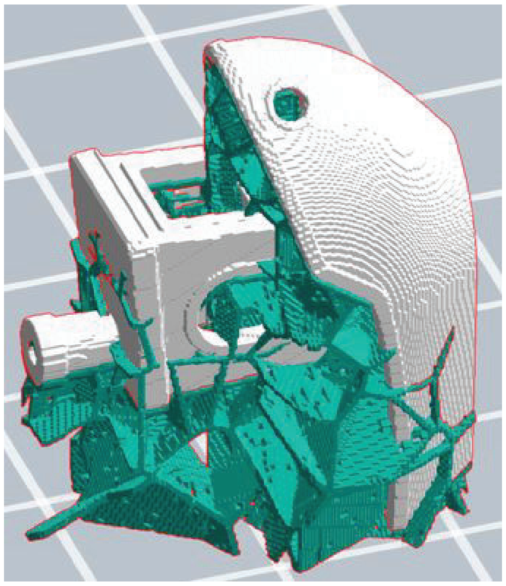

Figure 14: *Voronoi support structure applied to a more complex model*

Currently, there are several issues that can arise with this form of support and infill generation. First, due to the voxelized intermediate steps, the resulting support and infill structures have particularly strange surface textures. The textured finish is caused by a slight loss of information in the voxelization process, which then gets propagated along the process and magnified by the final marching cubes technique, giving the strange finish. If the finish is problematic for the parts intended use, a Laplacian smoothing operation can be used on the resulting meshes to give a smoother finish.

Another issue comes from the thickness of the shell and the walls of the Voronoi structures. Care must be taken to ensure that the shell of the resulting model is thick enough for the slicing software to register that it must apply multiple perimeters. Current testing has shown that a shell thickness of approximately four voxels leads to favorable results, and that a Voronoi foam thickness of approximately two voxels leads to an infill structure that is one to two perimeters thick. This voxel thickness ensures that the structure is printed thick enough to function properly, but no thicker than needed.

Due to the quirks of utilizing randomness to place the seed points for the structures, this algorithm often generates asymmetrical support structures for symmetrical models. More testing must be done to determine the benefits or drawbacks of these asymmetries. If a symmetrical infill or support structure is desired, a solution to this problem would be to generate the structures for only half the model, then mirror the results and union the halves back together to obtain the desired structure.

Lastly, no structural testing has been performed at this point. Other work has been done on the mechanical properties of 3D printed Voronoi structures, showing that they allow for gradients in part strength, a property difficult to achieve with current methodologies [3-5].

Regarding time investment, standard infill and support patterns do generate much faster than the techniques discussed in this paper. Both treelike and linear support structures can usually be generated in less than a second on comparably sized models, while Voronoi supports of sufficient resolution can take upwards of two to three minutes, depending on the resolution.

Due to the high complexity of the cross sections, Voronoi support and infill structures do take longer to print than traditional structures, such as the grid and hexagonal patterns. However, optimizing these structures for print time goes beyond the scope of this paper.

Parallel Benefits:

As mentioned, this software was written to take advantage of parallel computing technology. This is primarily due to the large arrays that are being managed throughout the process. To observe the benefits of using parallel computing, a version of the software was stripped of the GPU components and run in serial, and timing was analyzed between the two versions on 10x10x13, 20x20x30, and 30x30x46 arrays. Table 2 shows the average time it took for the software to develop both infill and support material for the E model at the corresponding resolutions.

Table 2: Processing Time

Model Size	Serial Computing	Parallel Computing
10x10x13 array	21.3 seconds	4.7 seconds
20x20x30 array	283.2 seconds	4.7 seconds
30x30x46 array	1308.4 seconds	5.3 seconds

As can be seen by the data, parallel computing has much better scaling, allowing for orders of magnitude better times than the serial implementation. Computation time for a 250x250x413 array was also measured in parallel, taking an average of 166.6 seconds. Serial implementation speeds were not measured at this size.

Of course, part of the reason for the particularly long computation times for the serial computation times is that the JFA is not at all optimized for serial implementation [9]. Changing to a distance field algorithm optimized for serial computation would make for an excellent first step in adapting this methodology to work on non-parallel computers.

Conclusion

The Voronoi support and infill generation algorithm performs as intended; producing printable, customizable infill and support structures based on the principles of a Voronoi foam. As the algorithm used to place the seed points can be easily modified, it is not difficult to edit the generation procedure to get a structure well-fitted to the model and its intended use. An additional benefit of the resulting model is the flexibility afforded by the voxel representation. The model and its supports are generated in voxel space, which can be easily exported as a traditional mesh file, such as an STL, or as an image stack. Generated image stacks can be harnessed by cutting-edge printing technology to produce multi-material prints on VPP 3D printers [14]. The full benefits of the support and infill generation algorithm are still being explored, but current results show promise in printability and stability.

References

[1] Creative Tools, Sweden, 2016, "#3DBenchy - the jolly 3D printing torture-test" http://www.3dbenchy.com/

[2] FlashForge, China, 2019, "FlashPrint" Version 3.28.0 http://www.flashforge.com/flashprint/

[3] Martínez, Jonàs, Dumas, Jérémie, and Lefebvre, Sylvain. "Procedural Voronoi Foams for Additive Manufacturing." *ACM Transactions on Graphics* Vol. 35 No. 4 (2016): DOI 10.1145/2897824.2925922. https://dl.acm.org/citation.cfm?doid=2897824.2925922.

[4] Martínez, Jonàs, Hornus, Samuel, Song, Haichuan, and Lefebvre, Sylvain. "Polyhedral Voronoi diagrams for additive manufacturing." *ACM Transactions on Graphics* Vol. 37 No. 4 (2018): DOI 10.1145/3197517.3201343. https://dl.acm.org/citation.cfm?doid=3197517.3201343.

[5]Martínez, Jonàs, Song, Haichuan, Dumas, Jérémie, and Lefebvre, Sylvain. "Orthotropic k-nearest foams for additive manufacturing." *ACM Transactions on Graphics* Vol. 36 No. 4 (2017): DOI 10.1145/3072959.3073638. https://dl.acm.org/citation.cfm?id=3073638

[6] Lu, Lin, Sharf, Andrei, Zhao, Haisen, Wei, Yuan, Fan, Qingnan, Chen, Xuelin, Savoye, Yann, Tu, Changhe, Cohen-Or, Daniel, and Chen, Baoquan. "Built-to-Last: Strength to Weight 3D Printed Objects." *ACM Transactions on Graphics* Vol. 33 No. 4 (2016): DOI 10.1145/2601097.2601168. https://dl.acm.org/citation.cfm?doid=2601097.2601168.

[7] Preparata, Franco P. and Shamos, Michael I. *Computational Geometry: an introduction*. Springer-Verlag Berlin, Heidelberg (1985)

[8] Jacobr. (2015). *euclidean (minkowski p=2) +source code*. [image] Available at: https://en.wikipedia.org/wiki/Voronoi_diagram#/media/File:Voronoi_growth_euclidean.gif

[9] Rong, Guodong and Tan, Tiow-Seng. "Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform." *Symposium on Interactive 3D Graphics and Games.* Pp 109-116. Redwood City, CA, March 14-17, 2006. DOI 10.1145/1111411.1111431. https://dl.acm.org/citation.cfm?id=1111431

[10] Hoff, Kenneth, Keyser, Joh, Lin, Ming, Manocha, Dinesh, and Culver, Tim. "Fast computation of generalized Voronoi diagrams using graphics hardware." *Proceedings of the 26th annual conference on Computer graphics and interactive techniques.* pp. 277-286 Los Angeles, CA, August 8-13 1999. DOI. 10.1145/311535.311567

[11] Pederkoff, Christian, 2019 "Turn STL files into voxels, images, and videos" https://github.com/cpederkoff/stl-to-voxel

[12] Ensz, Mark, Storti, Duane, and Ganter, Mark. "Implicit Methods for Geometry Creation." *International Journal of Computational Geometry & Applications.* Vol. 8 No. 3 (1998): pp. 509-536 https://www.worldscientific.com/doi/abs/10.1142/S0218195998000266.

[13] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu and the scikit-image contributors. "scikit-image: Image processing in Python." PeerJ 2:e453 (2014) https://doi.org/10.7717/peerj.453

[14] Schwartz, J. J., and Boydston, A. J. "Multimaterial actinic spatial control 3D and 4D printing." *Nature Communications*. Vol. 10 No. 1 (2019): Article 791 DOI 10.1038/s41467-019-08639-7. https://doi.org/10.1038/s41467-019-08639-7

The code described in this paper is freely available for download at:
`https://github.com/tjwill95/voronizer`

Appendix

Several algorithms are mentioned in the Methods that were not fully described in order to preserve the flow of the section. In order of appearance, here is a description of the process for each of the referenced algorithms in the following format:

```
Operation (Inputs: A, B, C; Outputs: X, Y, Z):
PSEUDOCODE
```

I will be using Python as a baseline for my pseudocode, meaning that the index of an array is dictated by square brackets (u[i,j,k] = element stored at index i,j,k), a colon selects all values within that index, (u[i,:,:] = 2D array), and the # symbol is used to dictate comments.

Some algorithms require several functions. The main algorithm will be **in bold**, and there will be a break indicated by --- between algorithms.

```
Project Down Vertical Axis (Input: VMI; Output: VMO):
#Assume X is the vertical axis.
VMO = copy of VMI
#This keeps from accidentally overwriting anything in the input
#array, as some languages transfer pointers to arrays instead
#of the entire array itself.
X = length of first index of VMO
#Need to know the lowest index that the part occupies, xMin
xMin = -1 #Priming the loop
i = 0     #Priming the loop
While xMin < 0:
     if min(VMO[i,:,:]) > 0:
          Increment i by +1
     Else:
          xMin = i
#Go down from the top of the array down until we hit xMin.
i = X-2
#Want to start 1 down from the top, as we know that no model can
#be beyond the top of the array.
While i ≥ xMin:
     Project Kernel(VMO,i)
     #Done in parallel, exact setup depends on the language
     Decrement i by 1
Return VMO

Project Kernel(VMO,CurrentX):
j,k = thread indices
if VMO[CurrentX+1,j,k] > 0:
     VMO[CurrentX,j,k] = -1
END
```

Boolean Operations:

All input matrices must have the same dimensions. All boolean operations function on a very similar idea, so we can construct a single kernel and modify the inputs as needed. For our purposes, let's design the kernel for the Boolean Union. As this kernel is going to be applying to all cells, and none of the cells have any interdependence on the matrices that they're in, this can be applied all at once via parallel computational methods.

```
Boolean(VM1,VM2,VMO):
     i,j,k = thread indices
     VMO[i,j,k] = min(VM1[i,j,k],VM2[i,j,k])
END

Union(Input: VM1,VM2; Output: VMO):
     VMO = Boolean(VM1,-1*VM2)
     #Done in parallel, exact setup depends on the language
Return VMO

Intersect(Input: VM1,VM2; Output: VMO):
     VMO = -1*Boolean(-1*VM1,-1*VM2)
     #Done in parallel, exact setup depends on the language
Return VMO

Subtract(Input: VM1,VM2; Output: VMO):
     #VM1 is the base, VM2 is the tool
     VMO = -1*Boolean(-1*VM1,VM2)
     #Done in parallel, exact setup depends on the language
Return VMO
```

```
**Depth Field** (Input: VMI; Output: VMO):
#Assume X is the vertical axis.
VMO = copy of VMI
#This keeps from accidentally overwriting anything in the input
#array, as some languages transfer pointers to arrays instead
#of the entire array itself.
X = lengths of first index of VMO
i = X-2
#Sets up the last index, top layer known to be outside the
#object as a buffer, second layer is where the model may begin.
While i ≥ 0:
     Depth Kernel(VMO,i)
     #Done in parallel, exact setup depends on the language
     Decrement i by 1
Return VMO

Depth Kernel(VMO,CurrentX):
j,k = thread indices
if VMO[CurrentX,j,k] ≤ 0:
     if VMO[CurrentX+1,j,k] > 0:
          VMO[CurrentX,j,k] = 0
          #Value above this index is positive, meaning this
          #index is the surface of the model
     else:
          VMO[CurrentX,j,k] = VMO[CurrentX+1,j,k]-1
          #Value above this index is negative, meaning this
          #index is within the model, and the index above has
          #already been set to the proper value.
END
```

```
**Point Generation**(Input: VMI, Threshold; Output: VMO):
#VMI is a distance/depth field of the object, with low magnitude
#values near the surface.
```

```
Random = Array of the same size as VMI with random numbers
between 1 and 0 for each cell.
VMO = Copy of VMI
#This keeps from accidentally overwriting anything in the input
Point Generation Kernel(VMI, VMO, Random, Threshold)
#Done in parallel, exact setup depends on the language
Return VMO

Point Generation Kernel(VMI, VMO, Random, Threshold):
i,j,k = thread indices
if VMI[i,j,k] < 0 and Random[i,j,k] < -1*Threshold/VMI[i,j,k]:
#The current function favors being close to the surface of the
#model, but could be modified to instead be entirely random, or
#to any other weighting function one would like.
     VMO[i,j,k] = -1
else:
     VMO[i,j,k] = 1
END
```

```
Jump Flood Algorithm(Input: VMI; Output: VMO):
#VMI is the object itself.
X,Y,Z = Lengths of the first, second, and third indices of VMI
VMOR = New array of dimensions (X,Y,Z,4), values set to 1000
VMOW = New array of dimensions (X,Y,Z,4), values set to 1000
#These act as read and write matrices, so that the Jump Flood
#Kernel has a stable array to read data from, and eliminates
#the possibility of a racing condition.
Populate Kernel(VMI, VMOR)
#Done in parallel, exact setup depends on the language
Number of Iterations = log base 2 of max(X,Y,Z), rounded up to
the nearest integer
While Number of Iterations > 0:
     Step Size = 2 ^ (Number of Iterations - 1)
     #Sets up the step size for this iteration
     Jump Flood Kernel(VMOR, VMOW, Step Size)
     #Done in parallel, exact setup depends on the language
     VMOW, VMOR = VMOR, VMOW
     #Switches the read array and the write array to ensure
     #that the kernel is reading the most recent data.
     Number of Iterations = Number of Iterations - 1
Last Iterations = 2
While Last Iterations > 0:
     Step Size = 2 ^ (Last Iterations - 1)
     Jump Flood Kernel(VMOR, VMOW, Step Size)
     VMOW, VMOR = VMOR, VMOW
     Last Iterations = Last Iterations - 1
#To improve accuracy, two more rounds of the Jump Flood
#Algorithm are done at the step sizes of 2 and 1, technically
#making this algorithm the JFA+2 described in [6].
Return VMOR

Populate Kernel(VMI, VMO):
i,j,k = thread indices
if VMI[i,j,k] ≤ 0:
     VMO[i,j,k,:] = [i,j,k,0]
END

Jump Flood Kernel(VMOR, VMOW, Step Size):
i,j,k = thread indices
M,N,P,D = VMOR[i,j,k,:]
#Extracts the coordinates stored in this cell and stores them in
#M, N, and P, then stores the distance from this cell to the
#stored coordinates in D.
CCI = 0
#Checked Cell Index
While CCI < 27:
     Checked Cell Coordinates = (i+((CCI//9)%3-1)*Step Size,
                                 j+((CCI//3)%3-1)*Step Size,
                                 k+(CCI%3-1)*Step Size)
     #This gives us the coordinates of each cell that has to be
     #checked for this cell in this iteration.
     M1,N1,P1,D1 = VMOR[Checked Cell Coordinates]
     #Extracts the coordinates stored in the values of the
     #checked cell
     D1 = sqrt((i-M1)^2+(j-N1)^2+(k-P1)^2)
     #Distance from current cell to coordinates stored in
     #checked cell
     if D1 < D:
          M,N,P,D = M1,N1,P1,D1
     #If the distance to the new coordinates is shorter than the
     #distance to the old coordinates, this cell stores the new
     #coordinates and the updated distance.
VMOW[i,j,k] = M,N,P,D
END

--
```

Signed Distance Field(Input: VMI; Output: VMO)

```
VMO = Copy of VMI
PJFA = Jump Flood Algorithm(VMI)
NJFA = Jump Flood Algorithm(-1*VMI)
#The Jump Flood Algorithm only fills in the positive values, so
#to get the distance field for negative values as well, we need
#to invert the model, such that positive is inside the object
#and negative is outside the object.
VMO = SDF Kernel(PJFA, NJFA, VMO)
Return VMO

SDF Kernel(Positive Input, Negative Input, VMO):
i,j,k = thread indices
Positive Distance = Positive Input[i,j,k,3]
Negative Distance = -1 * Negative Input[i,j,k,3]
if Positive Distance > 0:
     VMO[i,j,k] = Positive Distance
else:
     VMO[i,j,k] = Negative Distance
END
```