Fall 2017

Voronoi Stippling Report

Anthony Sermania *Computational Geometry*

1 Summary of the two methods

1.1 hedcuter method

The hedcuter method is as follows:

Hedcut:

1. Create a set of n random initial points (keep on random probability, favoring darker areas)
2. Compute the Weighted Centroidal Voronoi Tessellation (CVT)
3. Generate Disks

Computing The CVT:

1. While (we have fewer iterations than the limit AND the max-displacement is greater than some threshold:
   (a) Computer the Voronoi
   (b) Shift Voronoi cell sites towards cell centroids (and record displacement

Compute the Voronoi

1. For each input site
   (a) redirect site to corresponding position in resized image (fake enhanced resolution)
   (b) determine distance based on color of site
   (c) add site to open list/heap
2. while heap is not empty
   (a) cell ¡- max heap element (pop)
   (b) if distance has been updated or cell has been visited
      i. skip
   (c) for each neighbor of cell
      i. if cell.distance+neighbor.distance ¡ neighbor.distance
         A. set neighbor.distance ¡- cell.distance+neighbor.distance
         B. make neighbor.root ¡- cell.root

C. push neighbor into heap

3. For each cell in image
   (a) determine root of cell
   (b) add cell to cell.root's coverage
4. Remove all cells with coverage.empty

Move voronoi cell sites to their new locations (and record displacement)

1. For each cell
   (a) calculate new position as average of positions in coverage
   (b) Generate high res img
   (c) newpos ¡- (0, 0), total ¡- 0
   (d) for each cell c in cell.coverage
      i. newpos += c.distance*c.pos
      ii. total += c.distance
   (e) newpos /= total to normalize
   (f) newpos /= subpixels to redirect to original image
   (g) store manhattan distance from new to old position
2. return the max displacement (manhattan distance) or average of displacements (depending on termination mode)

Generate Disks

1. Collect the darkness/color of the area in a voronoi cell
2. Create a disk at cell site (center)
3. Color/darken cell according to the average color/darkness of the pixels i the cell

1.2 voronoi method

This one was a little bit tougher to work through mentally. At the highest level it uses Lloyd's Algorithm as so:

While generating points (xi) not converged to centroid

1. compute voronoi diagram of xi
2. compute centroid Ci
3. move generating point xi to centroid Ci

The Voronoi diagram computation is done differently here, utilizing an entire Voronoi Generation class that appears to implement a half-edge mechanism for doing it's work after some initial set up. It is extremely well commented; rather than just essentially replicating the comments, here is an overview of the half-edge voronoi generation:

1. While there are events:
    (a) If the lowest site is lower than the lowest vector intersection process the site
        i. Get left and right half edges, plus the edge of the left halfedge (or the bottom site if there's no edge)
        ii. create a bisector for the left edge (or bottom site) and if it intersects the left edge shuffle the vertices so the left edge and bisector share a vertex.
        iii. Create a new half edge right of the first (only at the moment) bisector
        iv. if this new bisector intersects with the right halfedge, add the bisector halfedge to the ordered linked list of verts
        v. get next site
    (b) else process the lowest vector intersection
        i. Get the halfedge for the intersection vector, as well as the surrounding halfedges
        ii. get the sites of the of the exterior halfedges
        iii. consider a circle passing through the triple of sites obtained
        iv. get the vertex that caused the event
        v. set the endpoints of the left and right half edges to be this vertex
        vi. mark the lowest half edge for deletion
        vii. clear events for right halfedge
        viii. mark right halfedge for deletion
        ix. if the site to left of the event is higher than site to the right, swap them
        x. create an edge between the two sites
        xi. create a half edge for this edge
        xii. insert the new bisector to the right of the left halfedge
        xiii. cleanup (reposition half-edges not intersecting the bisector that should)

That still got pretty verbose, but it's not quite done. Once this routine has run, the generator that ran it is used to collect the edges before completing (this process is straightforward and is not detailed here).

Now it moves to step 2, redistributing the stipples (bringing them closer to the centroids of the Voronoi cells). This is done in parallel where applicable.

1. Do some setup (make pairs of points and edge lists)
2. For each pair:
    (a) Calculate Centroid
    (b) update Radius and vertex X/Y positions to be that of the calculated centroid

(c) determine displacement of point

3. average displacement of all points

Calculating a Centroid: According to the paper, *Weighted Voronoi Stippling* by Adrian Secord, there are some integrals being done here. The implementation provided however seems to work around that, operating primarily on distance constraints:

1. Compute the Clip Lines
2. For all points in this space if a point is inside it's polygon, it contributes to the area density
3. If a cell is not completely white, shift the centroid according to the weighted-spot:area density ratio (that's a weird way to put it, but it's what's happening)
4. Iterate over the available edges and determine the nearest and farthest, using one of those distances to determine the size of the stipple (nearest if NoOverlap, Farthest otherwise) and scale by the area:max-area ratio
5. Return the pair of the original point and the radius.

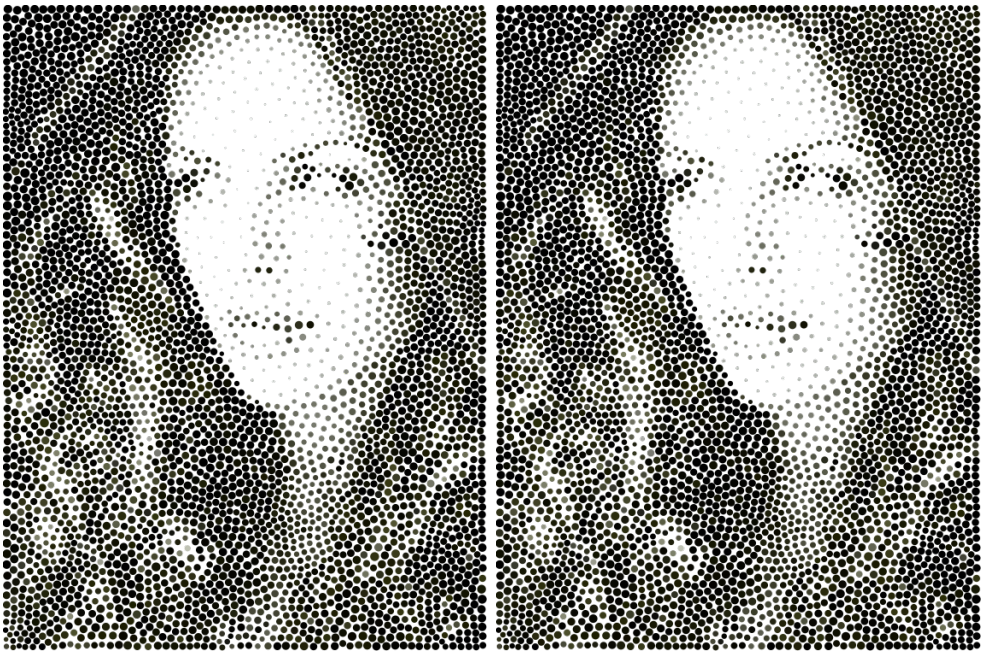
2 Comparison of the two methods

Both methods saw fair success. I will however claim the voronoi code to be superior overall.

Let's start with hedcut though. For one thing, hedcut was not consistent across multiple runs with the same parameters. Take for instance the following images:

We can see here that the two images, run with the same parameters (radius 4, 10,000 stipples) produced different stipplings. By contrast, here are two images from the voronoi code with the equivalent parameters (4,000 stipples, color, overlap)

And two more from voronoi, this time with no overlap.



It is worth noting I discovered later that hedcut might by consistent with sufficiently many iterations (see Improvements).

In terms of speed, hedcut may be faster in the simplest circumstances, coming out at around the same computatin time for 10,000 stipples as voronoi did for 4,000 and 8,000, however voronoi never seemed to vary much with number of dots- just by image. Hedcut on the other hand does take longer the more dots there are. The two images below are both 100,000 stipples. The image on the left is the hedcut image, created in 16.22 seconds; the one on the right is from voronoi, created in 4.43 seconds (not substantially slower than the 8,000 or 4,000 stipple runs for voronoi).

Beyond that time difference, the level of image fidelity with this many stipples is vastly superior in the voronoi image. I suspect the difference in speed comes from the use of so many loops in the hedcut code. The difference in quality is undoubtedly due to the stronger implementation of varied stipple sizes (based purely on the voronoi cell size) in the voronoi code. The hedcut code uses a base radius which is scaled, resulting in much weirder stipple sizes.

3 Improvement of hedcuter method

While I technically made two improvements to hedcut, I can only claim one of them was significant. The insignificant one was some compiler optimizations; this ended up not amounting to any consistently measurable improvement, and moreover wasn't a significant revamp to the code, so I would understand considering it poorly. Given that I was working on Windows in the end, and this required some working knowledge of Visual Studio (2013), maybe it's worth something? I ultimately did not try a third attempt due to time- much of which was wasted either trying to get the project to run (copy-paste .dll saved the day) or interviewing CS 110 students one at a time.

I did however make one very significant change that yielded marvelous improvements. There was a point in move-sites where the code was creating a higher resolution version of our source image to do centroid-relocation math in before redirecting the result back to the source image resolution. Being located in move-sites, it was being run for every voronoi cell, every iteration of the algorithm.

This struck me as highly inefficient. So I relocated it to just before we began running the algorithm in full- we never change the source image, so the higher res version should never change either. I was initially confused because I was mistakenly attributing the benefits of greater iterations to this change, but I did discover my folly.

Figure 1: Original Code. Left is 1 iteration; Right is 3 iterations. A fair improvement can be seen by increasing the number of iterations.

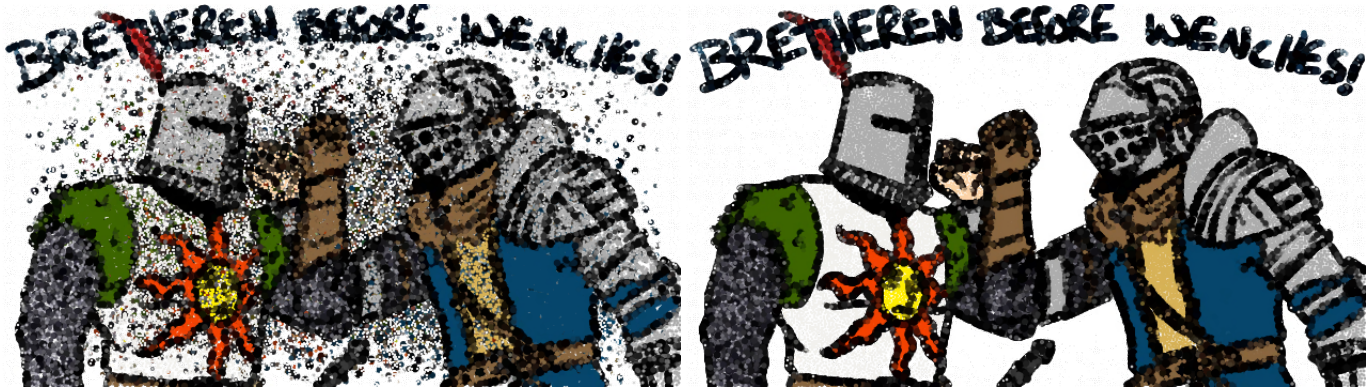


Figure 2: The revised version. As before, left is 1 iteration and right is 3. The images are quite similar to their original counterparts. It is possible the hedcut algorithm gains consistency with increasing iterations- a costly but evidently worthwhile gain.

So, No significant changes in the images from this. However, consider for a moment the following tables, indicating run times for various runs using the sunbros.png image for 1 or 3 iterations, before and after the change.

Run #	Image	stipples	# Iteratior	Version	Time
1	Sunbros	100000	1	1	17.668
2	Sunbros	100000	1	1	17.077
3	Sunbros	100000	1	1	17.053
4	Sunbros	100000	1	1	17.705
5	Sunbros	100000	1	1	18.462
6	Sunbros	100000	3	1	50.184
7	Sunbros	100000	3	1	46.953
8	Sunbros	100000	3	1	48.774
9	Sunbros	100000	3	1	50.331
10	Sunbros	100000	3	2	0.318
11	Sunbros	100000	3	2	0.3
12	Sunbros	100000	3	2	0.353
13	Sunbros	100000	3	2	0.381
14	Sunbros	100000	3	2	0.301
15	Sunbros	100000	1	2	0.099
16	Sunbros	100000	1	2	0.097
17	Sunbros	100000	1	2	0.099
18	Sunbros	100000	1	2	0.11
19	Sunbros	100000	1	2	0.107

Versions	Status	Iterations	Average Time	% Improvement (over Plain)
1	Plain	1	17.593	0
1	Plain	3	49.0605	0
2	Relocated ImgRez	1	0.1024	99.41795032
2	Relocated ImgRez	3	0.3306	99.32613814

A 99 percent improvement! That image rescaling must have been taking a hefty toll to run for so many iterations, seemingly unnecessarily!

After seeing the improvements made here, I decided to accept the negligible impact of the compiler optimizations; this now performs even better than the voronoi implementation in terms of run time.

I did make several other efforts that ultimately were not completed. I made a few attempts to determine the stipple size solely using the voronoi cell area, but concluded that I would have to backtrack a litte further than I was comfortable doing and risk breaking other key parts of the code. I also swapped the random number generator for the uniform gaussian to the included MT19937 generator during the sample point creation. This had no noticeable difference however (pictures omitted because I'm already lost in ths LaTeX thing).