CS 633: Computational Geometry Fall 2017

Voronoi Stippling

Julian Easley *Project Assignment 2*

1 Summary of the Two Methods

1.1 Hedcuter Method

First, the method starts by creating n sample points that are spread out evenly. It evenly spreads out the sample points by using a gaussian distribution. By default the method creates 1000 sample points.

After creating the n sample points, the method computes the weighted centroidal voronoi tessellation, which is computed by using the Lloyd's method. This algorithm iteratively generates what is called a Centroidal Voronoi Diagram. Unlike a standard Voronoi Diagram, the sites lies exactly in the centroid (the exact center) of its Voronoi region. This algorithm basically moves the sites of a given Voronoi Diagram so that it becomes a Centroidal Voronoi Diagram.

In order to compute the centroid, it computes the density first. It computes the average density of the cell by dividing the sum of the densities of each pixel in the cell by the number of pixels in the cell. Once the density is compute, it calculates the centroid by multiplying the position of each pixel, in the voronoi, by the density and dividing by the number of pixels in the cell.

The algorithm more specifically is given points not converged to centroids and it computes the Voronoi Diagram from those points. After that it computes the centroids of each Voronoi region and calculates the manhattan distance between the old position of the site and the new position of the site, which is the centroid. Then it moves the site to the centroid position. Afterwards, it averages all distances for all the sites to their centroid and when the average distance is greater than the maximum site displacement (default is 1.01) and the number of iterations is less then the maximum number of iterations (default value is 100) then it repeats the steps above and computes the Voronoi Diagram with the newly moved sites and calculates the centroids again. Otherwise, the algorithm stops.

1.2 Voronoi Method

First, similar to Hedcuter, the method begins by generating n stippler points spread out evenly. It evenly spreads out the stippler points by using a uniform distribution and blackish points have a higher probability of being picked. By default, it creates 4000 stippler points.

It then creates n sample (stippler) points and constructs the Voronoi Diagram of the stippler points. Then it finds the centroid of each cell. The difference between the methods this time is the way the density is calculated. The Voronoi Method computes the average density of a cell by dividing the cell into m tiles (default value is 5 tiles), and sampling one pixel per tile and computes the sum of the densities of the sample pixels divided by the number of tiles in the cell. After computing the

density, it calculates the centroid by multiplying the position of each pixel, corresponding to a tile in the voronoi cell, by the density and dividing by the number of tiles.

The algorithm then checks if the current average displacement of stipplers reached the threshold (default value is 0.1). If it did not then it repeats the steps above and calculates the new Voronoi Diagram and the new centroids. Otherwise the algorithm stops.

2 Comparison of the Two Methods

The two methods method differ in lots of ways by mainly by the following:

(1) By default the Hedcuter method samples only 1000 random points and the Voronoi method samples 4000 random points. Even if you make them generate both 4000 points they will still not result in the same image due to mostly the randomness.

(2) The way they generate the random points are different since the Hedcuter method randomly creates points all over the image whereas the Voronoi method generates points that are closer to black than white.

(3) The Voronoi method shows more depth in the image than the Hedcuter method. The radius of the points, in the areas that are closer to black, are longer compared to Hedcuter method. Look at Figure 1 and 2 as examples.

(4) The radius of hedcuter points are so small that I have to manually increase the size using the radius attribute in order to see the points more clearly. In Figure 1, I had to set the radius to 4 in order to see the points more clearly.

(5) Another major difference is the Hedcuter method has some of the site points on the white areas of the image where as in the Voronoi method this does not occur.

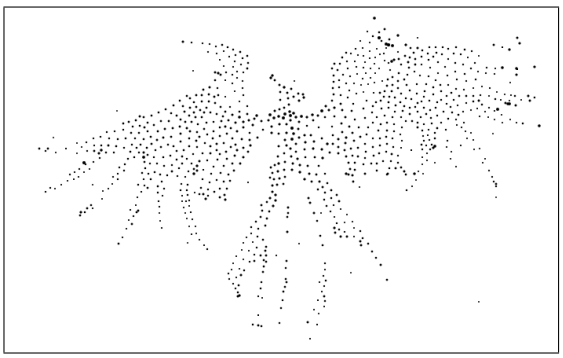

Figure 1: Hedcuter method with 1000 sample points and radius = 4

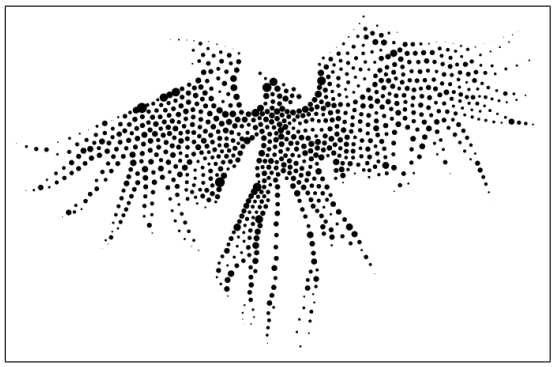

Figure 2: Voronoi method with 1000 sample points

1. **Do you get the same results by running the same program on the same image multiple times?**

   No for both methods, because the initialize of the sample points are always random so there will always be a little difference between one run of a method compare to another one. One run might have a lot of overlapping and other might not. One run might not put enough points to get a certain detail of the image compare to another run and etc.

2. **If you vary the number of the disks in the output images, do these implementations produce the same distribution in the final image? If not, why?**

   No, as explained above, even when both methods create the same number of disks (sample points), they won't result in the same distribution as shown in
   Figures 1 and 2. The reason this occurs is because the two methods calculate the centroid differently resulting a different output. Also the scaling of the radius is way more dynamic in the Voronoi method.

3. **If you vary the number of the disks in the output images, is a method faster than the other?**

   Yes, the Voronoi method is so much faster than the Hedcuter method. For example, when running the fairyeyes image in both methods, it took 11.65 seconds using the Voronoi method with 1000 sample points and it took 91.7865 seconds using the Hedcuter method with 1000 sample points. If you increase the number of sample points, the Hedcuter method slows down even more compare to the Voronoi method.

4. **Does the size (number of pixels), image brightness or contrast of image increase or decrease their difference?**

From 1180x1258 Einstein Image **From 583x600 Einstein Image**

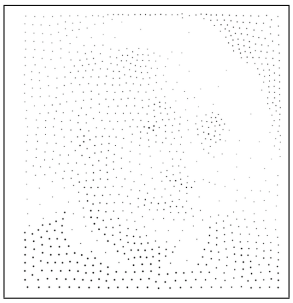

Figure 3:
Hedcuter method with 1000 disks

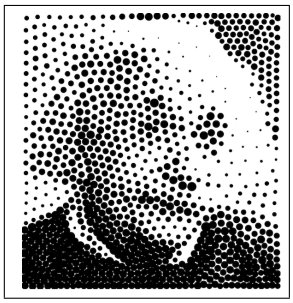

Figure 4:
Voronoi method with 1000 disks

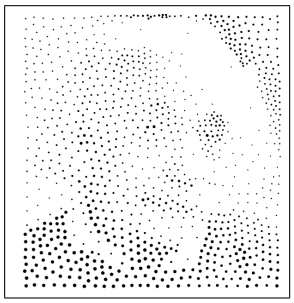

Figure 5:
Hedcuter method with 1000 disks

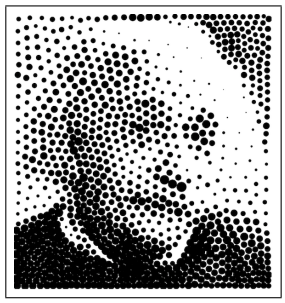

Figure 6:
Voronoi method with 1000 disks

As you can see from Figures 3 and 4, the difference in the radius of the disks is very drastic between the two methods and very noticeable when the image is large. The same difference occurs with Figures 5 and 6 using a scaled down version of the image but Figures 4 and 6, which both use the Voronoi method, result in almost the exact same

image when thats not the case with Figures 3 and 5. Therefore, scaling of the image does not effect Voronoi method but does effect the Hedcuter method.

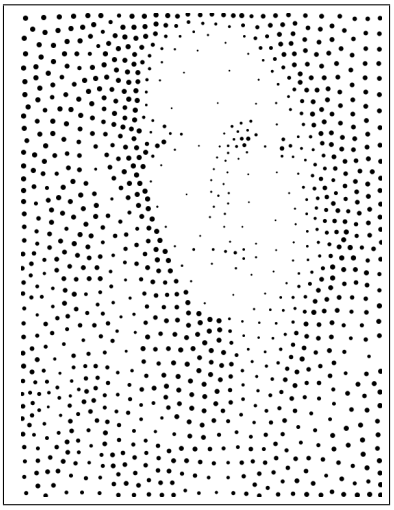

Figure 7: Hedcuter method with 1000 sample points and radius = 4

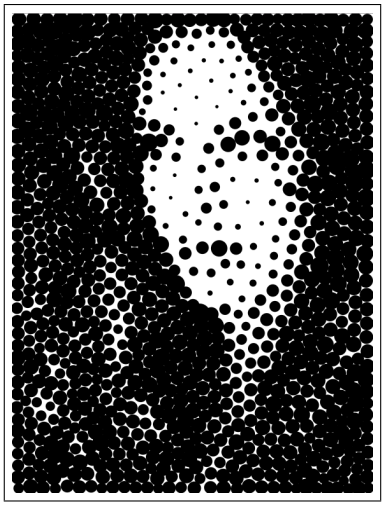

Figure 8: Voronoi method with 1000 sample points

When stippling the fairyeyes image, the difference between the two methods is very noticeable. Figure 7 and 8 proves that the higher the contrast in the image the more transparent the difference is between the two methods.

5. **Does the type of image (human vs. machine, natural vs. urban landscapes, photo vs. painting, etc) increase or decrease their difference?**

   The type of image does not effect the difference between the two methods. The Voronoi method will always result in the better stippling of the image.

6. **Are the outputs of these stippling methods different the hedcut images created by artists?**

   The main difference between using a stippling method and a professional artist who makes stippling drawing is the computer program is bounding by how many points it will make where as an artist can make as many points as it wants. Also an artist tends to make the points all the same size unlike the Voronoi method. One other difference between an artist doing the stippling is they can choose where to put the points whereas the position of the initial points of the stippler method is random and some of the initial point can end up being useless and overlap with other points.

3 Improvement of Hedcuter Method

3.1 Improvement 1: Initialization of the Sample Points

As stated above, the Hedcuter method starts by creating n sample points that are spread out evenly. It evenly spreads out the sample points by using a gaussian distribution. The Voronoi method, on the other hand, begins by generating n stippler points spread out evenly. It evenly spreads out the stippler points by using a uniform distribution and *blackish points have a higher probability of being picked*.

In the Hedcuter algorithm, the sample points are initialized in the file *hedcut.cpp* and in the method called *sample_initial_points*. In this method there is a comment that states: "decide to keep basic on a probability (black has higher probability)" but that not what the program does. The program just generates random points everywhere. I modified the code by using the same way the Voronoi method chooses blackish points more often than whitish points.

For example, Figures 10 and 11 show the initial 1000 sample points for Figure 9 using the Hedcuter method before and after my modification.

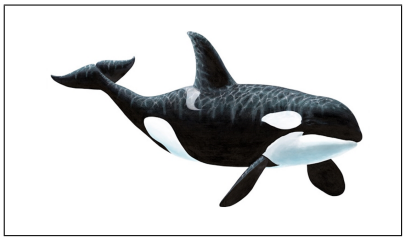

Figure 9: Whale Image

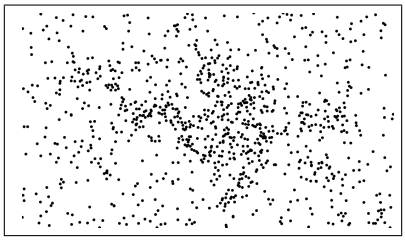

Figure 10: Before

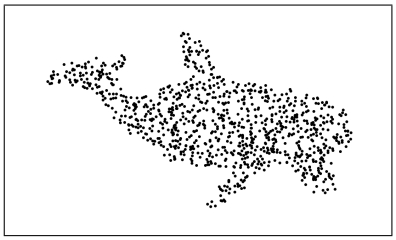

Figure 11: After

Code Changes in *hedcut.cpp*:

```
void Hedcut::sample_initial_points(cv::Mat & img, int n, std::vector<cv::Point2d> & pts)
{
    //create n points that spread evenly that are in areas of black points...
    int count = 0;

    cv::RNG rng_uniform(time(NULL));
    cv::RNG rng_gaussian(time(NULL));
    cv::Mat visited(img.size(), CV_8U, cv::Scalar::all(0)); //all unvisited

    while (count < n)
    {
        //generate a random point
        int c = (int)floor(img.size().width*rng_uniform.uniform(0.f, 1.f));
        int r = (int)floor(img.size().height*rng_uniform.uniform(0.f, 1.f));

        //decide to keep basic on a probability (black has higher probability)
        float value = img.at<uchar>(r, c)*1.0/255; //black:0, white:1
        float gr = fabs(rng_gaussian.gaussian(0.8));
        if ( value < gr && visited.at<uchar>(r, c) ==0) //keep
        {
            count++;
            pts.push_back(cv::Point(r, c));
            visited.at<uchar>(r,c)=1;
        }
    }
```

Figure 12: Before

```
void Hedcut::sample_initial_points(cv::Mat & img, int n, std::vector<cv::Point2d> & pts)
{
    //create n points that spread evenly that are in areas of black points...
    int count = 0;

    cv::RNG rng_uniform(time(NULL));
    cv::RNG rng_gaussian(time(NULL));
    cv::Mat visited(img.size(), CV_8U, cv::Scalar::all(0)); //all unvisited

    boost::mt19937 rng; //added
    boost::uniform_01<boost::mt19937, float> generator( rng ); //added

    while (count < n)
    {
        //generate a random point
        int c = (int)floor(img.size().width*rng_uniform.uniform(0.f, 1.f));
        int r = (int)floor(img.size().height*rng_uniform.uniform(0.f, 1.f));

        //decide to keep basic on a probability (black has higher probability)
        float value = img.at<uchar>(r, c)*1.0/255; //black:0, white:1
        //float gr = fabs(rng_gaussian.gaussian(0.8));
        float gr = ceil(generator()); // random number between 0 and 1

        if ( value < gr && visited.at<uchar>(r, c) ==0) //keep
        {
            count++;
            pts.push_back(cv::Point(r, c));
            visited.at<uchar>(r,c)=1;
        }
    }
```

Figure 13: After

3.2 Improvement 2: Disk Radius more Dynamic

As mentioned above the diameter of the disks in the Hedcuter method were extremely small and I had to manually increase the radius as an argument in order to even see the disks clearly. Even if I increased the radius enough manually, all the points even the insignificant ones are increased in radius.

I modified the code by making the disk radius more dynamic through the size of the voronoi cell and the average color of the voronoi cell. The bigger and the more black the voronoi cell is then the bigger the radius of the disk. This was done by using the Voronoi method code as reference.

For example, Figures 15-17 show stippling of Figure 14 using the Hedcuter method with 1000 disks before my modification and after. Notice in Figure 16 that all the points are increased in size, compared to Figure 15, unlike in Figure 17. More specifically, if you look at the points on my face, in Figure 17 they are smaller compare to Figure 15, which makes the image look better.

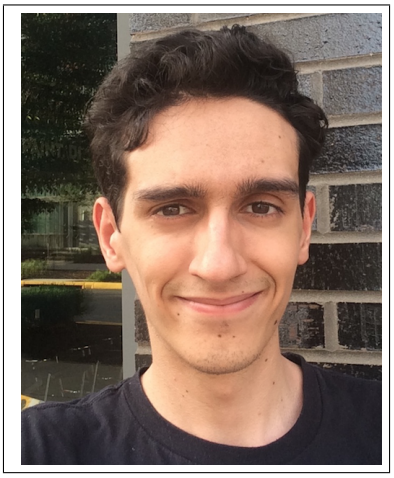

Figure 14: Original Image of Me

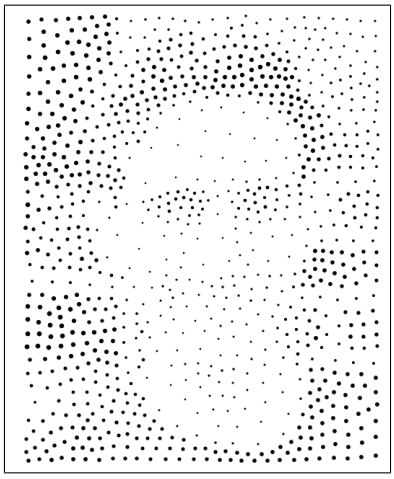

Figure 15: Before (radius = 4)

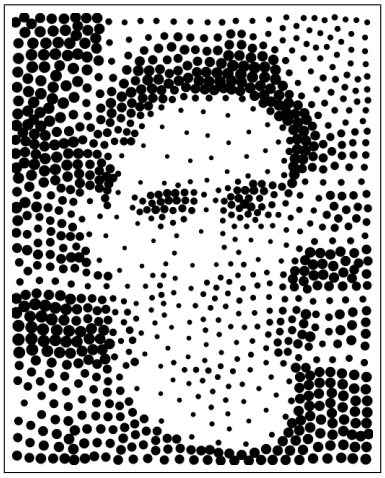

Figure 16: Before (radius = 10)

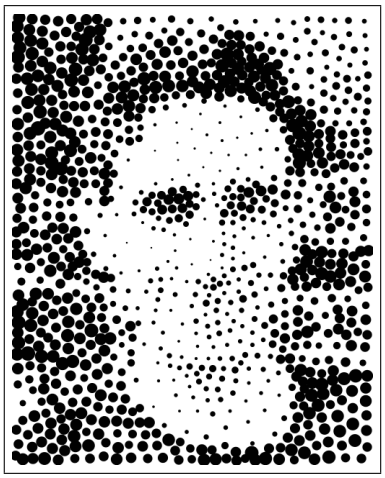

Figure 17: After

Code Changes in *hedcut.cpp* in the *create_disks* method:

```
void Hedcut::create_disks(cv::Mat & img, CVT & cvt)
{
    cv::Mat grayscale;
    cv::cvtColor(img, grayscale, CV_BGR2GRAY);

    disks.clear();

    //create disks from cvt
    for (auto& cell : cvt.getCells())
    {
        //compute avg intensity
        unsigned int total = 0;
        unsigned int r = 0, g = 0, b = 0;
        for (auto & resizedPix : cell.coverage)
        {
            cv::Point pix(resizedPix.x / subpixels, resizedPix.y / subpixels);
            total += grayscale.at<uchar>(pix.x, pix.y);
            r += img.at<cv::Vec3b>(pix.x, pix.y)[2];
            g += img.at<cv::Vec3b>(pix.x, pix.y)[1];
            b += img.at<cv::Vec3b>(pix.x, pix.y)[0];
        }
        float avg_v = floor(total * 1.0f/ cell.coverage.size());
        r = floor(r / cell.coverage.size());
        g = floor(g / cell.coverage.size());
        b = floor(b / cell.coverage.size());

        //create a disk
        HedcutDisk disk;
        disk.center.x = cell.site.y; //x = col
        disk.center.y = cell.site.x; //y = row
        disk.color = (black_disk) ? cv::Scalar::all(0) : cv::Scalar(r, g, b, 0.0);
        disk.radius = (uniform_disk_size) ? disk_size : (100 * disk_size / (avg_v + 100));
```

Figure 18: Before

```
void Hedcut::create_disks(cv::Mat & img, CVT & cvt)
{
    cv::Mat grayscale;
    cv::cvtColor(img, grayscale, CV_BGR2GRAY);

    disks.clear();

    //create disks from cvt
    for (auto& cell : cvt.getCells())
    {
        //compute avg intensity
        unsigned int total = 0;
        unsigned int r = 0, g = 0, b = 0;
        for (auto & resizedPix : cell.coverage)
        {
            cv::Point pix(resizedPix.x / subpixels, resizedPix.y / subpixels);
            //total += grayscale.at<uchar>(pix.x, pix.y);
            r += img.at<cv::Vec3b>(pix.x, pix.y)[2];
            g += img.at<cv::Vec3b>(pix.x, pix.y)[1];
            b += img.at<cv::Vec3b>(pix.x, pix.y)[0];
        }
        //float avg_v = floor(total * 1.0f/ cell.coverage.size());
        r = floor(r / cell.coverage.size());
        g = floor(g / cell.coverage.size());
        b = floor(b / cell.coverage.size());

        //create a disk
        HedcutDisk disk;
        disk.center.x = cell.site.y; //x = col
        disk.center.y = cell.site.x; //y = row
        disk.color = (black_disk) ? cv::Scalar::all(0) : cv::Scalar(r, g, b, 0.0);
        //disk.radius = (uniform_disk_size) ? disk_size : (100 * disk_size / (avg_v + 100));
        disk.radius = (uniform_disk_size) ? disk_size : (cell.radius * disk_size); //added
```

Figure 19: After

Code Changes in *wcvt.h* in the first *move_sites* method (move_sites(cv::Mat & img, VorCell & cell)):

```
//update
float dist = fabs(new_pos.x - cell.site.x/subpixels) + fabs(new_pos.y - cell.site.y/subpixels); //manhattan dist
cell.site = new_pos;

//done
return dist;
```

Figure 20: Before

```
//update
float dist = fabs(new_pos.x - cell.site.x/subpixels) + fabs(new_pos.y - cell.site.y/subpixels); //manhattan dist
cell.site = new_pos;

//width of disks
float closest = numeric_limits<float>::max(),
      distance;
float x0 = new_pos.x, y0 = new_pos.y,
      x1, x2, y1, y2;

for (std::list<cv::Point>::iterator iter = cell.coverage.begin(); iter != cell.coverage.end(); ) {
    x1 = iter->x; y1 = iter->y;
    ++iter;
    x2 = iter->x; y2 = iter->y;

    distance = abs( ( x2 - x1 ) * ( y1 - y0 ) - ( x1 - x0 ) * ( y2 - y1 ) ) / sqrt( pow( x2 - x1, 2.0f ) + pow( y2 - y1, 2.0f ) );
    if ( closest > distance ) {
        closest = distance;
    }
}
cell.radius = 6 * (1 + 2*closest) * total / cell.coverage.size();

//done
return dist;
```

Figure 21: After

Code Changes in *wcvt.h*:

```
struct VorCell
{
    VorCell(){}

    VorCell(const VorCell& other)
    {
        site = other.site;
        coverage = other.coverage;
    }

    cv::Point site;
    std::list<cv::Point> coverage;
};
```

Figure 22: Before

```
struct VorCell
{
    VorCell(){}

    VorCell(const VorCell& other)
    {
        site = other.site;
        coverage = other.coverage;
        radius = other.radius; // added
    }

    cv::Point site;
    std::list<cv::Point> coverage;
    float radius; // added
};
```

Figure 23: After