Non-photorealistic Rendering in a Pointillistic Style

Kelly Wang and Keegan Dant
Rensselaer Polytechnic Institute

ABSTRACT
In this paper, we describe a method for a non-photorealistic rendering of an input 3D scene such that the final output is rendered in a pointillistic style. We achieve this through the use of raytracing, sending rays to collect color and position information from a scene and storing that information into a graph structure, then combining the nodes collected in the graph based on the colors of adjacent neighbors. The graph with combined nodes then has each stored position and color randomly processed. We present output images that have been filtered by our algorithm to evoke a pointillistic style.

1. INTRODUCTION

Realistic computer rendering has been a significant pursuit in various fields of entertainment for the last several decades. As a result of these efforts, many successful methods for simulating real life phenomena and generating life-accurate scenes have been achieved. At the same time, as more of these photorealistic methods are developed, it builds up a foundation from which a focus on non-realistic work is allowed to arise; in other words, as higher accuracy is achieved, the more the ability to break logical rules is attained. One such field is non-photorealistic rendering, or NPR, which takes input scenes or images, often realistic ones, and directly filters over them to output images that appear to be done in an artistic style.

1.1 MOTIVATION

Photorealistic rendering is certainly a powerful and useful tool, not just for aiding in mathematical or scientific simulation, but also has its own place in creative projects. However, not all creative projects require life-accurate renderings, and may at the same time require automation of some previously manual process to provide a certain style or look. NPR provides a solution for such efforts.

As will be discussed in further detail below, there are certain aspects used in traditional pointillism that have been defined as theory. As with many other accomplishments in computer graphics, there is a theoretical basis from which machine code can be written to replicate in simulations (notably physics simulations such as with cloth and fluid). We were interested in replicating an artistic style, especially one with some form of logical basis that would translate well to code. Pointillism made the most sense to us as it involves aspects of color theory and averaging based on neighbor dot positions and colors.

1.2 RELATED WORKS

Seo and Yoon's work in [1] presented the results of the analysis of notable pointillism painter Georges Seurat's painting and their work on a rendering algorithm that filters over 2D image inputs. They generate points recursively, storing them in a parent-child structure such that hue jitter and complementary color generation is possible in order to mimic Seurat's complementary color shading technique. Our implementation differs in that we directly render from 3D input and the data structure we store our points in, but their techniques aforementioned still apply.

Sugita and Takahashi in [2], like [1], also analyze Seurat's pointillism and the color theory it is based on. They render their final pointillistic image through five steps which include halo filtering, uniform randomly rendering pure colorist pointillism dots, rendering complementary colors, and generating an ebauche. All except for the ebauche step were applicable to our work. Their work also describes how to get the radius of a similarly colored area for recreating the halo effect, but not how to continue building it to create non uniform size points throughout the image as we wanted. The pointillistic halftoning algorithm they implemented influenced our choices for generating our point data structures around a certain radius and colored area.

2. ASPECTS OF POINTILLISM

Pointillism was a painting style devised by Georges Seurat and Paul Signac and was widespread during the post-Impressionist era. It consists of placing dots

instead of strokes on the canvas to take advantage of the viewer's optical ability to blend the spots of color into a cohesive image.

We focused on certain aspects of Seurat's style of pointillism, chromoluminarism and "halos." Chromoluminarism is the practice of applying patches of different colors in close proximity such that the colors are not mixed up close, but give the illusion of being mixed and are perceived as a different color at farther distances. This effect is intended to maximize the luminance and shimmeriness of the applied paint, and commonly utilizes the pairings of complementary and primary colors.

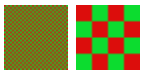

Figure 1. (Left) an image showing a murky yellowish-green. (Right) A close-up of the same image, revealing its composition of red and green.

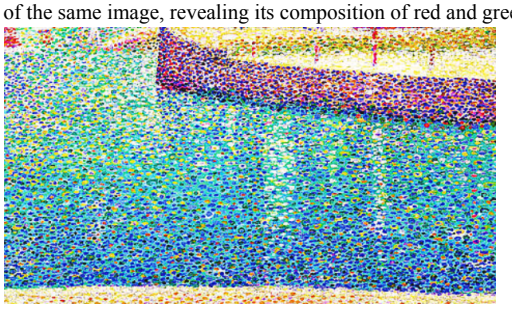

Figure 2. A close-up of "Entrance of The Port of Honfleur" by Georges Seurat, 1886, showcasing chromoluminarism.

The second effect found in Seurat's work that we focused on was the "halo" effect, where a separation between the painting subject and the background was achieved through a gradient starting from a subject's silhouette, and would often be a lighter color than the background, though dark halos sometimes appeared. This allowed for higher contrast and clarity of the painting subject.

Figure 3. A close-up of "A Sunday Afternoon on the Island of La Grande Jatte" by Georges Seurat, 1884-1886, showcasing the "halo" effect.

Our implementation would aim to replicate these effects.

3. IMPLEMENTATION

Although we reference Seurat's work for our implementation of pointillism, we also decided on certain differences from "traditional" pointillism in some ways, mainly, we aim for varying point sizes as well as a varying amount of space between points.

Our implementation consists of two main components. The first is the data structure used to contain image information, and the second is the algorithm used to combine the information stored in the graph so our criteria of varying point sizes and inter-point spaces can be fulfilled.

3.1 GRAPH DATA STRUCTURE

Our method of retrieving image information is through raycasting, and our inputs are 3D scenes set up through OpenGL. Our ray tracing algorithm sweeps the scene as follows:

ALGORITHM 1: Ray Tracing Sweep

```
div = 5;
for every row in screen_height / div
    for every column in screen_width / div
        // Construct a ray from the camera towards the
        world position of the specified row and column
        // retrieve the color returned from the ray hit
        // add the color and its associated row/column
        to a graph data structure
```

We will henceforth refer to the color and position added to the graph as a Point.

Our point graph has the underlying structure of a binary tree map. Each key in the map is a unique integer (ID) identifying the associated value (the added Point). We keep track of the largest ID that has been entered into the graph and increment it whenever a Point is created, ensuring each Point has a unique ID.

We choose a graph data structure because the combination algorithm requires knowledge of which Points in 2D space are neighboring a target Point. As such, we also keep track of edges, or the neighbors, as a list of IDs in the Point itself. The initialized graph will always have a grid structure where each Point's position corresponds to a Point within the screen grid, so neighbors are also deterministic upon graph initialization, and can be found using 2D array arithmetic:

ALGORITHM 2: Calculating Neighbors

```
rows = screen_height / div;
cols = screen_width / div;
set<int> tmp;
```

```
if (curr_id - cols >= 0)
    // up
    tmp.insert(curr_id - cols);
    // up left
    if (j != 0) tmp.insert(curr_id - cols - 1);
    // up right
    if (cols - j != 1) tmp.insert(curr_id - cols + 1);
if (j != 0) tmp.insert(curr_id - 1); // left
if (cols - j != 1) tmp.insert(curr_id + 1); // right
if (next_point_id + cols < rows*cols)
    tmp.insert(curr_id + cols); // down
    // down left
    if (j != 0) tmp.insert(curr_id + cols - 1);
    // down right
    if (cols - j != 1) tmp.insert(curr_id + cols + 1);
```

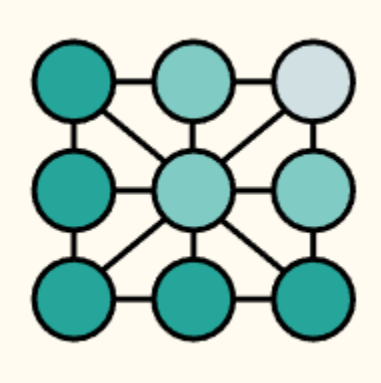

Figure 4. A visualization of initial nodes and edges/neighbors for a graph of size 3x3

3.2 COMBINATION ALGORITHM

The method we used to combine points in the point graph takes advantage of the underlying binary tree map for fast access, addition, and deletion of <ID,Point> pairs. It works as follows:

ALGORITHM 3: Combining Points

```
maxCombinations = 3;
threshold = 0.05;
while (points still combining)
   vector<int> addList;
   vector<int> deleteList;
   for (points in graph)
      if (point.timesCombined < maxCombinations &&
         point is not in deleteList)
         //Find the neighbor point that is closest in
         color using the distance formula and whose
         distance is less than threshold
         //Get average weighted position and color of
         the two points
         //Combine neighbor sets, excluding points
         being combined, and clean up all neighbors to
         match new Point index
   //Make new point and add to addList
   //Add old two points to deleteList
```

Both points being combined must have not been combined maxCombinations amount of times before and must not already be in the deleteList vector. Both vectors used are so that the point graph is not edited during one loop of the algorithm. This does have a bias to combine with points that have not been combined during the loop and we believe this may lead to some artifacting, as will be discussed in a later section.

3.3 POST-PROCESSING

Following the application of our combination algorithm, each Point in the graph has two effects applied to it.

3.3.1 POSITION JITTER

We add random jitter to each Point's position in the form of a random 2D vector. The magnitude is then scaled depending on the number of combinations a Point has undergone: Points that were combined one or less times received a much larger jitter than Points combined more than once.

3.3.2 COMPLEMENTARY COLORS

To properly account for the chromoluminarism effect found in Seurat's paintings, the color of each Point also has a random chance of being altered depending on its luminance (higher chance as color gets darker).

Specifically, the hue might stay the same, or be shifted in either direction such that the new hues are complementary and equidistant to the original hue. We also have a random chance of heightening the saturation by a random amount. The color is then decided as follows:

ALGORITHM 4: Complementary Colors

```
origColor = point.getColor();
// our representation stores color using RGB, so we
first convert from RGB to HSL
newColor = RGBtoHSL(origColor);
shade_prob = 1 - origColor.magnitude();
hueShift = 0;
if (RandInRange(0,1) < shade_prob)
    dir = RandInRange(0,1);
    // 45 is approximately a quarter of the color
    wheel, which fulfills the aforementioned criteria.
    if (dir > 0.5) hueShift = 45;
    else hueShift = -45;
newColor.hue += hueShift;
newColor.hue modulo 360;
if (RandInRange(0,1) > 0.5)
    newColor.saturation *= 1 + RandInRange(0,1);
    if (newColor.saturation>1) newColor.saturation = 1;
// we then convert from HSL back to RGB.
newColor = HSLtoRGB(newColor);
point.setColor(newColor);
```

3.3.3 FILE OUTPUT

In order to render with OpenGL we created a separate program due to the existing implementation of Homework 3's OpenGL interface. To pass our point graph to this separate program we wrote it to file. This file simply contains the width and height of the original image on the first line and then for each Point its RGB color, its x and y screen position values in the graph, and the number of times it was combined.

3.4 RENDERING WITH OPENGL

As stated previously a separate program was created to render our final image output. This program simply reads in the file written for the point graph, calculates the position transformation for rendering to an OpenGL window, and calculates the radius of every point based on its number of times combined and the width/height of the window along with an input parameter. Most of the OpenGL code used is boilerplate needed to create an OpenGL window and actively render. To place the circles on the screen glColor3f() provides the coloring to the next circle and the circle is drawn with glBegin(GL_TRIANGLE_FAN) and glVertex2f() using cartesian coordinates. glClearColor() is additionally used as a parameter for the background color of the image as later described.

4. RESULTS

All of the results we present here were rendered on a Lenovo Thinkpad T480s laptop with an Intel i7 4-core processor. We were successfully able to produce 2D image output of a given input image in a pointillist style while maintaining the composition of the input image and resolution. Our solution's performance time is as expected with very little time needed to create output. As well as images in a pointillist style, we were able to create other interesting output with several user guided parameters as shown in section 9.

4.1 PERFORMANCE

Our graph's ability to track IDs and each Point's ability to track its neighbors allows for fast access of a Point and initial population of the graph is relatively quick; a resolution of 1920x1080 is populated in approximately 4 seconds. With the same graph our combination algorithm is complete in approximately 30 seconds. Rendering with OpenGL happens in approximately 3 seconds.

4.2 ARTIFACTS

In the produced output artifacts may appear where points of similar colors combine into straight horizontal or vertical lines. We believe this is due to a bias in the combination algorithm, but it does not significantly affect the quality of our results.

Figure 5. An example of vertical artifacting

4.3 PARAMETERS

We have four parameters that can influence the look and quality of our output image. These are the combination color threshold, the base circle radius size, the max amount of combinations for each point, and choice of a white or black background.

Changing the threshold parameter and the max combinations will determine the amount of points generated in the point graph. The higher the threshold, and lower the combination max, the less combinations that will happen overall in the resulting image, leading to more circles being rendered.

The need for a combination max arose for images with large sections of uniform color so that a single large point would not emerge in the output. The base circle radius size parameter usually also goes hand-in-hand with the number of combinations; each time a point is combined its radius already increases.

To get optimal output, these parameters need to be tweaked for each individual input image, but they also allowed for us to produce results that did not match the pointillist style such as those in Figure 6 and 7. In general, the pointillistic style is limited the more homogeneous the colors in the source image are.

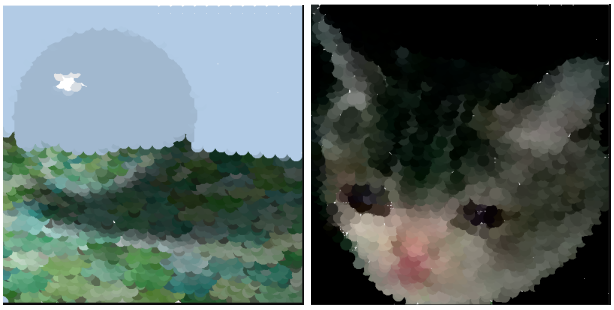

Figure 6 & 7. Examples of non-pointillism output

4.4 TECHNICAL CHALLENGES

We originally tried to add new OpenGL code to the source code we pulled from Homework 3. The intention was to add circles to the existing mesh renderer, but we were unsuccessful in incorporating the existing DrawCircle() function from OpenGL and the other option of adding circles in the form of extra mesh faces would have involved unnecessary math/work, considering the existing DrawCircle() function. Thus, we made a separate Visual Studio project with GLFW, which reads the file containing point graph data output by the renderer to output the final image. We also ran into issues setting up the GLFW libraries for the second program.

Our project required creating new .obj files for 2D images. The OpenGL camera placement was unintuitive and took a bit of maneuvering to properly set up.

5. LIMITATIONS AND FUTURE WORK

We completed all of our original main tasks for this project, but unfortunately did not get to our bonus tasks of deliberating creating the halo effect seen in Seurat's painting, and the ability to allow direct input of 2D images into the pointillistic algorithm instead of rendering a 3D scene and using ray tracing. Although in some output, using certain parameters, a halo effect is somewhat possible, if we were to continue this project we would want to work on both of these tasks.

Allowing direct 2D input, i.e. a .jpeg or .png image, would allow us to bypass the need for the Homework 3 code and place all of our project into a singular program. The process of switching between two programs to go from input image to output render is pretty streamlined, but it does become a limitation when trying to automate the process.

An additional limitation we have is the previously mentioned artifacting that may happen on input with large sections of homogenous color. With some tweaking the artifacting is barely noticeable in our image output below, but is still undesirable.

6. CONCLUSION

In this paper, we described a method for a non-photorealistic rendering of an input 3D scene such that the final output is rendered in a pointillistic style. We also achieved output that did not directly reflect pointillism, but created an artistic visualization of the input image. We have presented output images that have been filtered by our algorithm to evoke the pointillistic style popularized by Georges Seurat.

7. PROJECT ROLES

Keegan Dant created the combination algorithm used. Also debugged OpenGL and GLFW issues, implemented the process for writing the point graph to file and reading from the file to render circles in a separate program. Rendered all of the images.
Kelly Wang researched for Seurat's pointillism technique, created the graph data structure, set up new scene .obj files, wrote the post-processing function that applied the position jitter and chromoluminarism effect, and assisted in the debugging of the combination algorithm.
In total we spent ~35 hours on the total project, 20 of those for coding and 15 for the paper, presentation, and proposal.

8. REFERENCES

1. Seo, SangHyun, and KyungHyun Yoon. "Color juxtaposition for pointillism based on an artistic color model and a statistical analysis." The Visual Computer 26 (2010): 421-431.
2. Sugita, Junichi, and Tokiichiro Takahashi. "A method for generating pointillism based on seurat's color theory." ITE Transactions on Media Technology and Applications 1.4 (2013): 317-327.
3. Meier, Barbara J. "Painterly rendering for animation." Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. 1996.
4. "Chromoluminarism and Pointillism – New Techniques from Georges Seurat." Spoken Vision, Spoken Vision, 15 Aug. 2014, https://spokenvision.com/chromoluminarism-pointillism-new-techniques-georges-seurat/.
5. "RGB to HSL Color Conversion." *RGB to HSL Converter | Color Conversion*, RapidTables, https://www.rapidtables.com/convert/color/rgb-to-hsl.html.
6. "Crying Cat." Know Your Meme, Know Your Meme, 20 June 2018, https://knowyourmeme.com/memes/crying-cat.
7. Lnh. "This Is Fine." Know Your Meme, Know Your Meme, 12 May 2015, https://knowyourmeme.com/memes/this-is-fine.

8. Payne, Matt. “San Juan Sunrise Panorama.” Matt Payne Landscape Photography , 2016, https://www.mattpaynephotography.com/photo/san-juan-mountains-sunrise-panorama/. Accessed 21 Apr. 2023.
9. Mclean, Elspeth. “Mandala Stones Collection #3.” Elspeth Mclean, https://www.elspethmclean.com/mandala-stones-portfolio-page-1. Accessed 24 Apr. 2023.

9. IMAGES

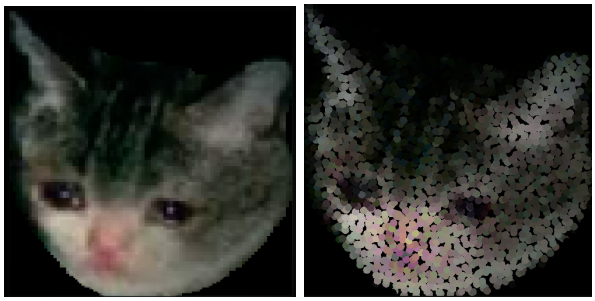

Figure 8 & 9. Popular meme *Sad Cat* was used as an input image, shown on the left. Right is the output with post processing on and on a black background. Some of the post processing effects can be seen closer to the nose.

Figure 10 & 11. Another popular meme, *This Is Fine*, was used as an input image, shown on the top. Bottom is the output with post processing on and on a white background. The post processing effect was very apparent here in the yellow parts of the fire and the back wall.

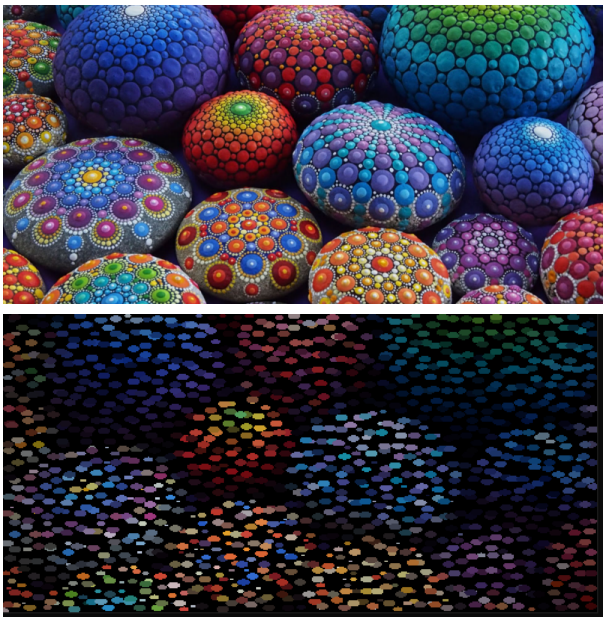

Figure 12 & 13. A mural created by Elspeth Mclean was used as an input image, shown on the top. Bottom is the output with no post processing and on a black background. We wanted to use an image with a large range of colors throughout, in both small and large sections. This output does not directly reflect all aspects of pointillism, but it showcases our technique’s ability to preserve detail.

Figure 14 & 15. A landscape photo of a San Juan sunrise taken by Matt Payne was used as an input image, shown on the top. Bottom is the output with post processing on and on a black background. The composition of the original photo was not kept as much as we would like, but it did produce a nice abstract piece of the original. We liked the yellow spread throughout the horizon introduced by complementary colors.

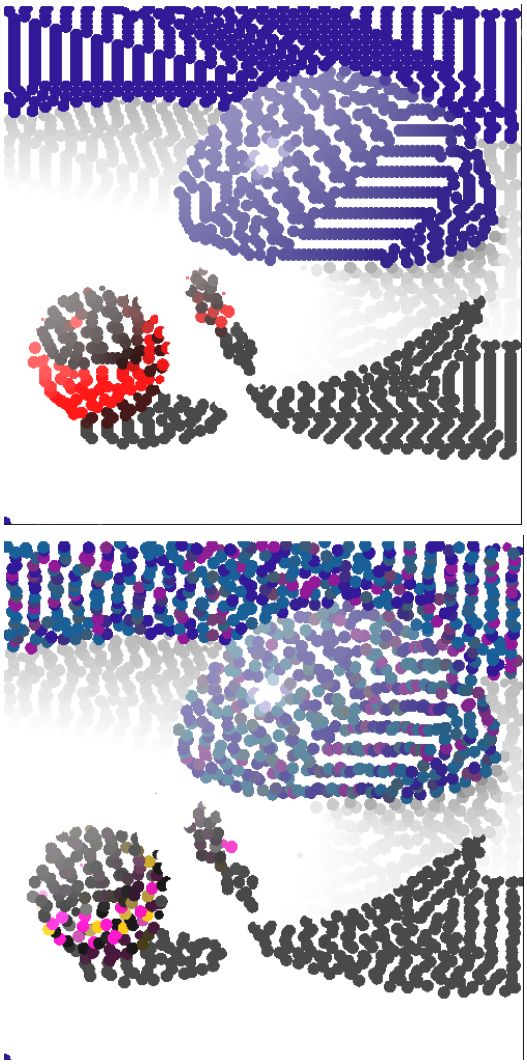

Figure 16 & 17. The reflective_spheres.obj provided for Homework 3 was used as input for both of these. Both images are on a white background, and the bottom has post processing turned on. We liked how the white of the background blended very well into the shadowed areas of the plane. These both show the artifacting that is possible, but the position jittering in post processing helps with this in the second image.

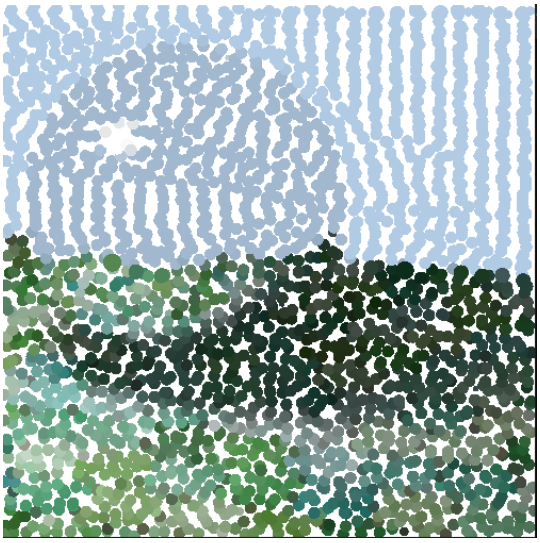

Figure 18. The textured_plane_reflective_sphere.obj provided for Homework 3 was used as input. Post processing was not turned on and it was drawn on a white background. The textured plane portion of the output image created some very interesting patterns in the image. The blue portions of the image also show artifacting previously discussed.

	Base Circle Radius	Combination Maximum	Combination Threshold
Sad Cat	3	3	0.5
This is Fine	5	1	0.25
Elspeth Mclean Mural	2	2	0.4
Matt Payne Landscape	3	3	0.05
Reflective Spheres	3	3	0.5
Textured Reflective Sphere	3	3	0.5

Table 1. Parameter details for the above image results.

10. BLOOPER IMAGES

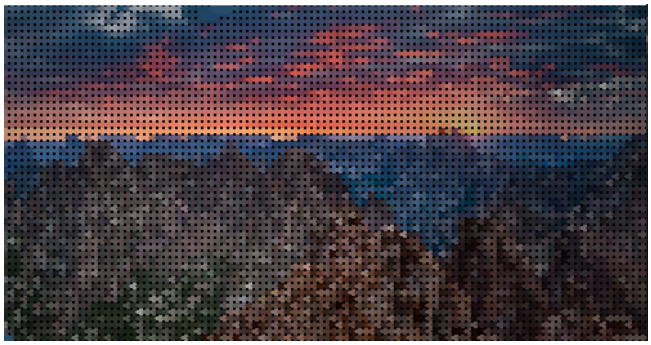

Figure 19. An example of blooper output when trying to find optimal parameters. This kept the composition of the image, but also introduced small black dots throughout, very similar to a pixelated effect.

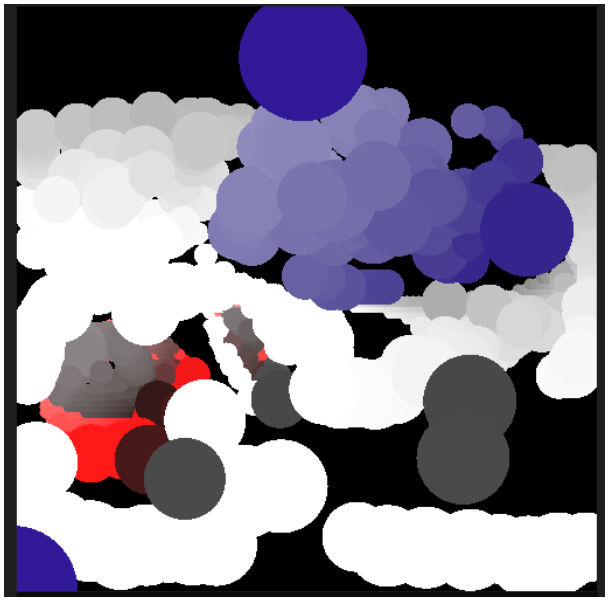

Figure 20. A blooper from before a max combination was implemented for each point. Very large circles were produced here where large areas of homogeneous color were present, namely the sky was turned into a single blue dot at the top.