Master Project - Michael Zinsmaier

Michael Zinsmaier
Matr.: 01/637211
Universität Konstanz
FB Informatik/Informationswissenschaften
michael.zinsmaier@uni-konstanz.de

ABSTRACT

In this report, I discuss the benefits of using density fields to visualize and explore large graphs. I give a detailed guide on how to exploit modern graphics hardware in order to get smooth and accurate results. I present a realtime normalization algorithm that automatically adjusts the calculated results to the screen colorspace while the user interacts. I show the advantages of multi-rendering to preserve class information during the process. Finally, I motivate the benefits of the approach on three examples.

1. INTRODUCTION

Graph visualizations and scatterplots are common tools to present data sources like the Iris data set, social networks or structures of XML files. Furthermore, there is a direct representation of matrices, as graphs, and they are useful for the formulation of various abstract problems. Graphs are usually formalized as $G(V, E)$ with a set of vertices V representing the data members and a set of edges E that defines the relationships between them. Scatterplots on the other hand can be seen as graphs with $|E| = 0$ and one formulation can be used for both approaches.
Being useful in many fields a lot of research has been done to improve the visualization of graphs. This research is divided into two parts: I) layouting and II) presenting.

Layouting tries to improve the perception of graphs or to highlight some of their properties by moving vertices and/or edges. There exists a variety of layout algorithms for different purposes and different graph scales. But the application of such algorithms is only possible if the position of the moved object does not encode data. It is especially prohibitive on scatterplots but also on road maps and many other graphs.

Presenting graphs deals with the visual representation of graphs on paper or on screen. Graphs are normally represented with discrete objects, most often circles or quads for vertices and lines for edges. Several parameters like the size of objects, transparency, coloring, etc. can be adjusted to improve the perception of graphs. However, large graphs tend to clutter and are hard to read. Common technics for reducing the amount of visual objects, are node merging and edge bundling (which is related to layouting) but the algorithms are computational expensive and are limited in the scale of reduction.

I propose a continuous drawing method that uses Gaussian functions to generate a density field of vertex influences. With this approach arbitrary large graphs can be drawn without neglecting the influence of a single node. Furthermore varying the bandwidth of the basis-functions provides a natural interface to explore data sets. However, computational costs remain to be a limiting factor but the use of massive parallel processors (GPU) can push the borders far enough. My implementation uses *geometry shaders* and *floating point textures* to take advantage of modern hardware.

2. RELATED WORK

Early work on density based graph visualization already focused on Gaussian kernels and two dimensional graphs but was limited through insufficient graphics hardware. GraphSplatting [5] is an early attempt that experiments with height fields, contours, and color mapping to visualize the obtained density field. Furthermore, the authors map an additional scalar field to the visualization by adding high frequency noise. The intensity of the noise expresses the scalar values. Texture blending is used for hardware acceleration, but to keep the accumulated values in the margins of 8 bit color resolution, the input values have to be scaled by an iterative algorithm and loose precision.
Lampe and Hauser [2] recently applied these ideas to the visualization of streaming data, like the commercial air traffic in US. They define the *line kernel* which effectively extends *Kernel Density Estimation* [4],[3] to edges by applying the Gaussian kernel function to the convex combination of two consecutive points. An interesting feature is that the integral of these kernels sums up to a third data value which proves especially useful to integrate informations about the time spent between two points. To read out data, users can build the integral over a certain area. The authors use a similar GPU approach as GraphSplatting [5] to enable real time data streaming but overcome the necessity of scaling through the usage of floating point textures.

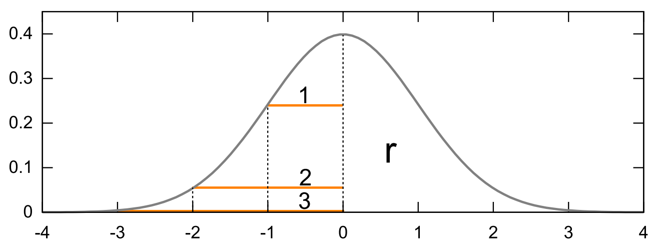

Figure 1: Radius of influence r of 1, 2 and 3 on $gauss_{1D}(x, \sigma = 1)$

While using similar technics as [5] and [2] my focus lies on the interactive visualization of large static graphs and scatterplots. Therefore I present a hardware accelerated algorithm that determines a suitable normalization factor in a few micro seconds and can thus respond to changes in the density field in real time. Also, I propose the usage of multiple distinct density fields in one visualization, to deal with class information, like in the Iris data set.

3. THE DENSITY FIELD

In this section I briefly define the generation of a continuous $2D$ function from a graph $G(V, E)$, named henceforth *Density Field.*

Let g be a graph with n vertices $v_i, i \in \{1, 2, .., n\}$. Each vertex v_i has a position $p_i = (x_i, y_i) \in R \times R, i \in \{1, 2, .., n\}$.

Definition:

$$D_f(x, y, \sigma) = \sum_{i=1}^{n} f(x, y, p_i, \sigma) \tag{1}$$

Now the *influence* has to be added to the nodes. For that f is chosen as *gaussian kernel*:

$$f(x, y, p_i, \sigma) = \frac{1}{2\pi\sigma^2} \cdot e^{-0.5(\frac{x - x_i + y - y_i}{\sigma})^2} \tag{2}$$

The result is a continuous $2D$ field, constructed by summing up n Gaussian basis functions. Sigma (σ) remains as a free parameter and can be used to control the smoothness of the field.

4. IMPLEMENTATION

The density field D has to be sampled to get a scalar value at each pixel. The most stable way to achieve this, would be to calculate the contribution of all vertices for the center of each pixel. While the introduced error would be minimal, the method requires solving an equation with n subterms for each pixel. To increase performance, the fast decreasing *bell* shape of the Gaussian function can be exploited by adding a radius of influence r to control the number of subterms. The bell will be cut off at r (Fig. 1).

4.1 Using the Hardware

Sampling D consists of many identical and independent subproblems, thus parallelization of tasks can be used to speed up computations. Using a common PC the CPU or the GPU can be used to calculate solutions to the equations. While CPUs can solve less than 100 problem instances in parallel, this number is much higher using a GPGPU approach. An efficient range query data structure is required to take advantage of r. However, such a data structure is difficult to implement on the GPU.

$$\frac{1}{2\pi(a \cdot \sigma)^2} \cdot e^{-0.5(\frac{x+y}{a \cdot \sigma})^2} = \frac{1}{a^2} \cdot \frac{1}{2\pi\sigma^2} \cdot e^{-0.5\left(\frac{\frac{1}{a}(x+y)}{\sigma}\right)^2}$$

Figure 2: Scaling sigma with a factor a equals scaling (x, y) with $\frac{1}{a}$ and adding a factor of $\frac{1}{a^2}$. If only the relation of values matters the factor can be omitted.

4.2 Texture Blending

To avoid these difficulties the problem can be translated into graphics and the blending abilities of *OpenGL* can be used. Instead of the *pull* approach, where each pixel *asks* for the contribution of the vertices, a *push* approach, where each vertex *spreads* its contribution to the pixels can be used.

The push approach:

- take all vertex positions p_i
- generate a quad with side length $r * \sigma$ and center p_i
- apply a texture with precomputed solutions of the Gaussian function
- accumulate all textures into a frame buffer object (FBO)
- convert the results to the screen colorspace

This approach can be implemented in standard *OpenGL* and works without range queries. Also, the solutions of the Gaussian function can be stored in textures and the evaluation of one pixel simplifies from solving eq. 1 to the accumulation of precomputed values. Furthermore, changes of sigma can be mapped to changes of the texture size (Fig. 2). Thus, it is enough to store the standard normal distribution in the textures. These advantages come at the costs of applying the textures to the quads and the introduction of additional discretization errors. Because GPUs have efficient texture support the negative performance implications are minimal, but discretization errors can be harmful.

Introduced errors:

- texture to quad mapping => approximation of the Gaussian function
- limited resolution of textures
- limited resolution of the FBO
- no contribution of points with $distance(p_i) > r$
- conversion to the screen

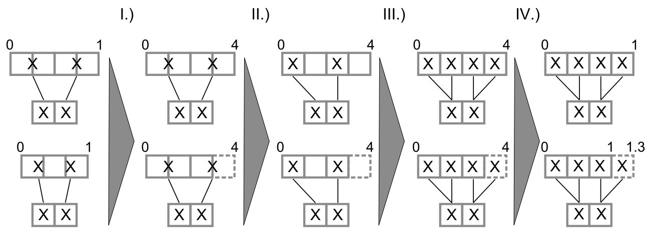

Figure 3: Reducing a parent texture of width 4 (upper example) or width 3 (lower example) with a kernel of size 2 in one dimension.

4.3 Normalization

To display the accumulated results values have to be mapped from the interval $[0, maxValue]$ into the colorspace of the display. A direct approach would be to map to the interval $[0, 1]$ and use the values to address one color channel. The $maxValue$ of accumulating n nodes has the following upper bound (eq. 3):

$$\begin{aligned} upperBound &= max(gauss_{2D}(x, y, \sigma = 1.0) * n) \\ &= max\left(\frac{1}{2\pi}e^{-\frac{x^2+y^2}{2}} \cdot n\right) \ , x = y = 0 \\ &\approx 0.159 \cdot n \end{aligned} \tag{3}$$

However, using $upperBound$ as $maxValue$, wastes big parts of the limited screen colorspace, as the points most likely will not have the same position. To get a better $maxValue$, one could try to estimate the distribution of points, but since the data is irregular and the user can interactively change the zoom level and the field of view a pleasant result cannot be achieved (Fig. 6 (left)). The best response to these problems is to scan over the result and search for the exact $maxValue$.

4.4 MaxValue Calculation

The implemented search algorithm uses the GPU to calculate the maximal value on many small rectangles in parallel and fills a smaller texture that contains only those maximal values. After reaching a certain size, the CPU jumps in and scans the remaining texture sequential.

let	k	be the kernel size
	w	be the target texture size
	pw	be the size of the texture that should be reduced

Then Figure 3 shows the process of rendering in one dimension. The target texture has size $w = 2$, the kernel has size $k = 2$ and the parent texture has either width $pw = 4$ (upper example) or $pw = 3$ (lower example). Notice that the parent texture width pw can be smaller than $w * k$ because w is calculated as follows:

$$w = \left\lceil \left(\frac{pw}{k}\right) \right\rceil \tag{4}$$

The depicted algorithm in more detail:

1. Switch from texture to kernel space. This means converting the *OpenGL* texture coordinates of the target texture from $[0, 1]$ to $[0, w \cdot k]$. One pixel of the target texture belongs to k pixels in the kernel space. The original texture coordinates were centered on each pixel, the obtained coordinates represent the center of a group of k pixels.

$$texCoord * w * k$$

2. Move $-\frac{k}{2}$ to the left border of the kernel window and then $+\frac{1}{2}$ to the center of the first kernel member. (Notice that the length of one pixel in kernel space is exactly 1)

$$-\frac{k-1}{2}$$

3. Add an offset of $\{0, 1..k - 1\}$ to sample the pixels of the kernel.

$$+offset$$

4. Divide the coordinates through pw to switch back to texture coordinates. The new coordinates possibly leave $[0, 1]$ but match exactly the pixel centers of the parent texture. To deal with values greater than 1, *GL_WRAP* can be set to *GL_CLAMP_TO_EDGE*

$$*\frac{1}{pw}$$

Altogether the shaders have to solve the following formula:

$$\begin{aligned} maxValue &= max_{offset}(value(k, w, pw)) \\ &= max_{offset}((texCoord * w * k) - \\ &\quad \frac{k-1}{2} + offset) * \frac{1}{pw}) \end{aligned} \tag{5}$$

4.5 Discretization

Using a set of colors to represent discrete intervals (Tab. 1), results in something similar to contour lines which is easy to read and gives a good intuition of the real data underneath. Therefore a sequential single hue scheme from colorbrewer [1] can be used for that. Additionally, vertex class information can be preserved through blending, using different color channels or multiple render passes. With that information, different hues can be used during discretization. However, this requires appropriate color schemes that can be intermixed if clusters overlap. The examples (Fig. 7) are produced with a single render pass and a red, green and blue color scheme.

interval	from	to
0	0.00	0.04
1	0.04	0.07
2	0.07	0.12
3	0.12	0.20
4	0.20	0.31
5	0.31	0.45
6	0.45	0.61
7	0.61	0.77
8	0.77	0.91
9	0.91	0.97
10	0.97	1.00

Table 1: Color intervals based on $gauss_{1D}(x, \sigma = 1.0)$ with $x \in [0, 2.575]$ (99% radius)

5. TACKLING ARTEFACTS

Discretization errors can cause visual artefacts like jagged lines, holes and noise. Apart from perceptual disturbance, such errors can suggest incorrect data properties to the user and therefore have to be removed. This section describes the problems that occurred during the master project and explains how to resolve them.

5.1 Quadratic Textures

Textures are quadratic, the sampled function is not. Sampling the Gaussian function in a certain range gives a circle of data values and should be represented by a circle of texture values. Setting texture edges to zero solves this problem (Fig. 4).

5.2 Radius of Influence

If the radius of influence r is too small, significant function values get ignored. The accumulation of textures makes things even worse because small values can add up and become visual important. There is a trade off between performance and big sampling ranges that can not be solved perfectly. However, sampling the function in the interval $[-4\sigma, 4\sigma]$ works well in the experiments.

5.3 Frame Buffer Object Resolution

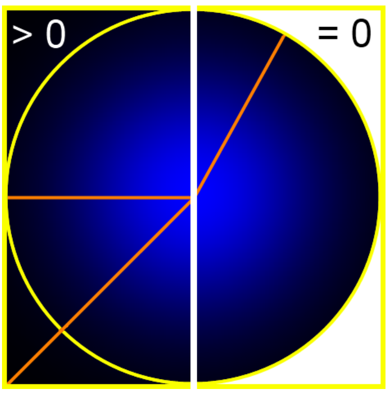

Figure 4: Sampling without edge postprocessing (left side) leads to different sample ranges. Setting edges to zero solves the problem (right side)

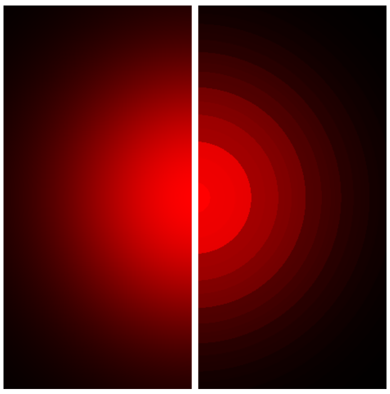

Figure 5: One kernel (left side) and the artefacts through blending 1000 kernels with the same center (right side) using a FBO with 16 bit resolution (half float)

The resolution of the FBO has the strongest influence on the results. If it is chosen too low, rounding errors occur during the accumulation of textures (Fig. 5). The shown artefacts depend on the artificial distribution of the example. Even small distances between the vertices will hide the patterns. Nevertheless, the accumulation errors exist and cause *Moiré Patterns* and other side effects. While color resolution was limited to integer formats or half precision floats in the past, single and even double precision is available on modern GPUs. Experiments showed that single precision floats are sufficient to eliminate the visual artefacts.

6. RESULTS

In this section, I present the results of applying the algorithms I) to an artificial hierarchical dataset, II) to the Iris data set. I highlight the implications of the normalization algorithm and multi-rendering.

The Hierarchical data is generated by drawing n_0 points from a normal distribution centered at $\frac{screenWidth}{2} / \frac{screenHeight}{2}$ with $\sigma_{distribution} = \sigma_0$. Each of these points becomes the center of a new cluster with n_1 points and $\sigma_{distribution} = \sigma_1$. This process is iterated until the desired number of levels is reached and the points from the last level form the graph. The user can now interactively change the free parameter σ and discover the different levels. However, if *maxValue* is not adjusted accordingly, on most levels only the borders of the clusters can be seen. The inner borders diminish, due to the insufficient mapping to the screen color space (Fig. 6 left). Applying the normalization algorithm, enables adjusting *maxValue* automatically, and subculsters and heat spots become visible (Fig. 6 right).

The Iris data set is a well known multivariate data set that consists of four attributes and three classes. The attributes *Sepal Length*, *Sepal Width*, *Petal Length* and *Petal Width* separate the three species of Iris flowers. *Sepal Lenght* and *Sepal Width* is chosen for rendering because the advantages of multi-rendering are emphasized by the large overlap between two of the clusters. One could just form a graph g from the 150 items of the data set and render them in

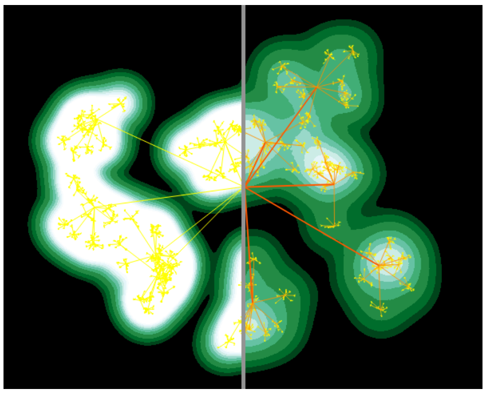

Figure 6: An artifical data set without dynamic normalization (left side) and with our normalization algorithm (right side). Inner clusters and heatspots become visible.

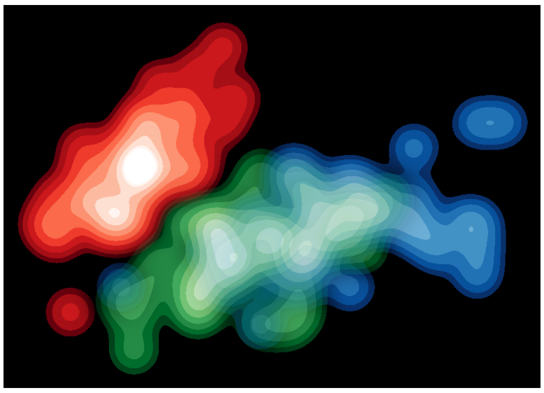

Figure 7: The Iris data set rendered into three density fields. The fields are displayed with a red (setosa), green (versicolor) and blue (virginica) color scheme.

on pass. However, it is difficult to separate the clusters for versicolor and virginica in this case. Instead class information is used to generate multiple density fields. The effects of this method can be seen in Figure 7 in the lower left corner. The red and blue point are separated from the green cluster, not only by their color, but also by their shape. They form distinct, disconnected clusters.

7. REFERENCES

[1] C. Brewer. Colorbrewer 2.0 @ www.colorbrewer.org, 2009.

[2] O. D. Lampe and H. Hauser. Interactive visualization of streaming data with kernel density estimation. In *Proceedings of the IEEE Pacific Visualization Symposium (PacificVis 2011)*, pages 171–178, March 2011.

[3] E. Parzen. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):pp. 1065–1076, 1962.

[4] M. Rosenblatt. Remarks on some nonparametric estimates of a density function. *The Annals of Mathematical Statistics*, 27(3):pp. 832–837, 1956.

[5] R. van Liere and W. de Leeuw. Graphsplatting: Visualizing graphs as continuous fields, 2001.

7.1 The Push Approach

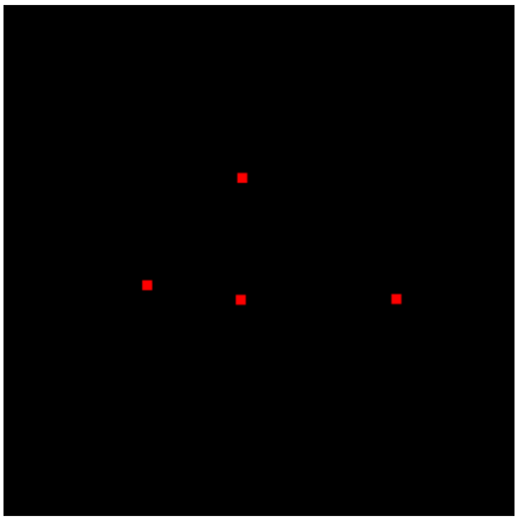
Figure 8.a: A set of four points that shall be converted into a density field.

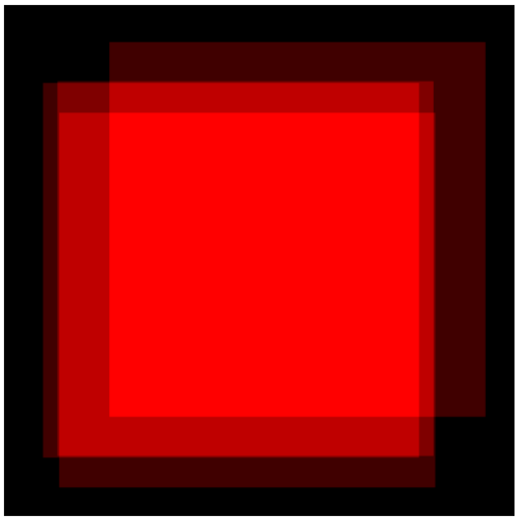
Figure 8.b: Converting the points into four quads. (symbolic figure not the same points)

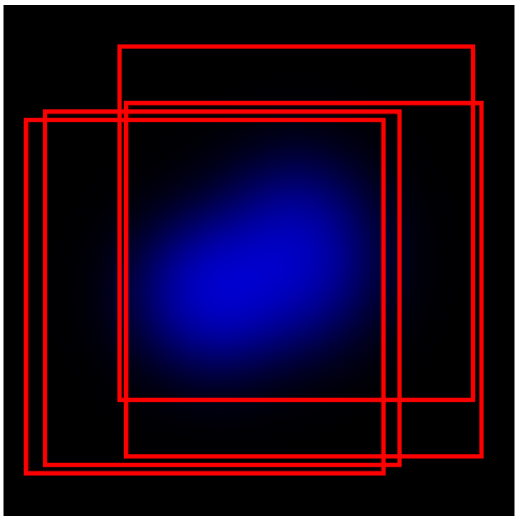
Figure 8.c: Applying the Gaussian texture and normalizing the values. (symbolic figure not the same points)

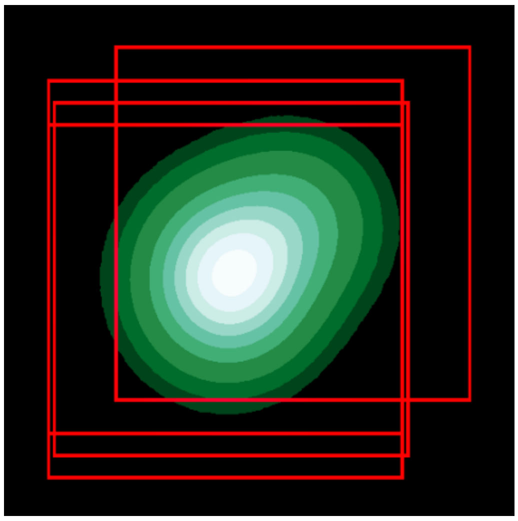
Figure 8.d: Displaying the result as discrete colors.(symbolic figure not the same points)

7.2 FBO Artefacts

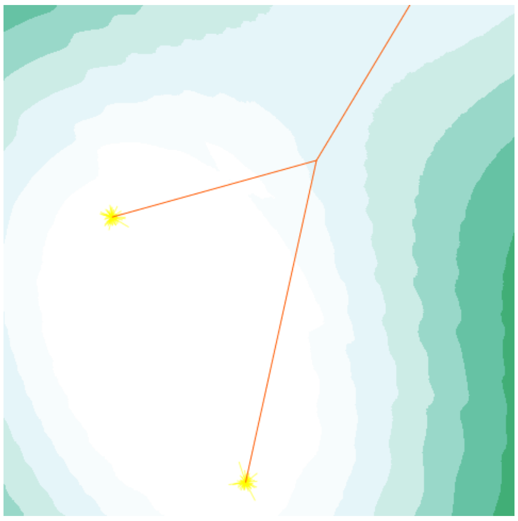

Figure 9.a: The visible effect of choosing the FBO resolution too low.

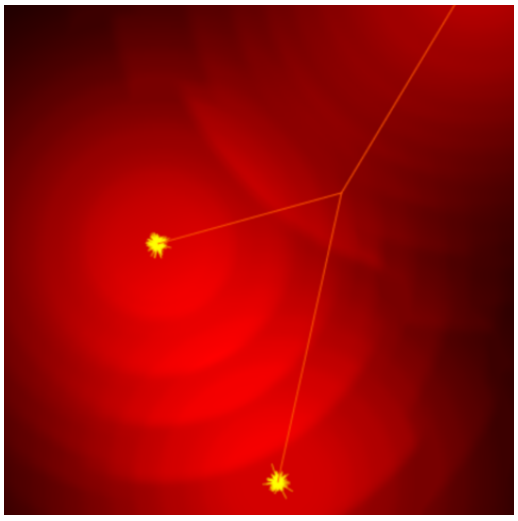

Figure 9.b: Without discretization the Moire patterns are visible (increased contrast).

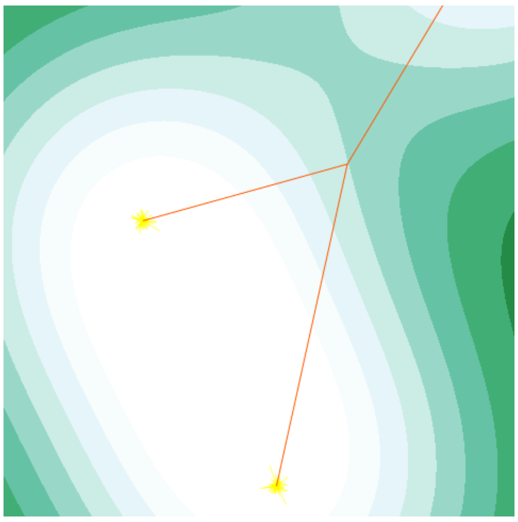

Figure 9.c: Increasing the FBO resolution solves the problem.

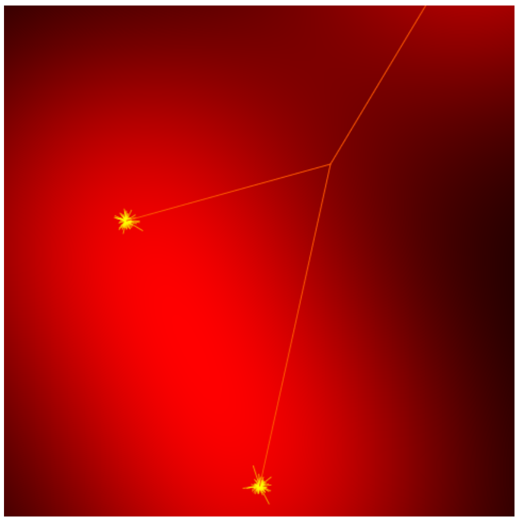

Figure 9.d: Switching back to the *continuous* view shows smooth kernels as intended.

7.3 Zooming on Hierarchical Data

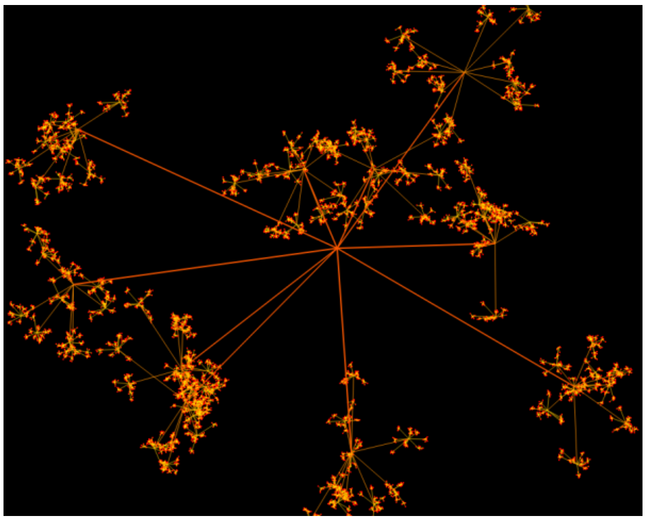

Figure 10.a: The data points (red dots) and the hierarchy tree with their parents (orange)

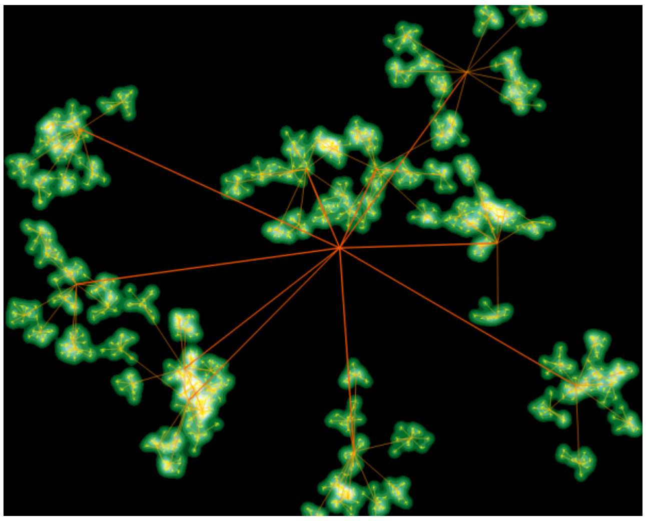

Figure 10.b: Low level clusters generated with a small σ value.

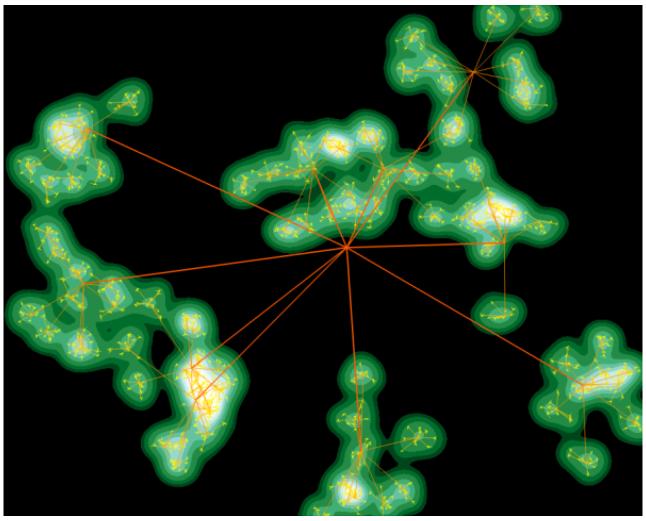

Figure 10.c: Increasing the σ value merges more and more data points into the clusters.

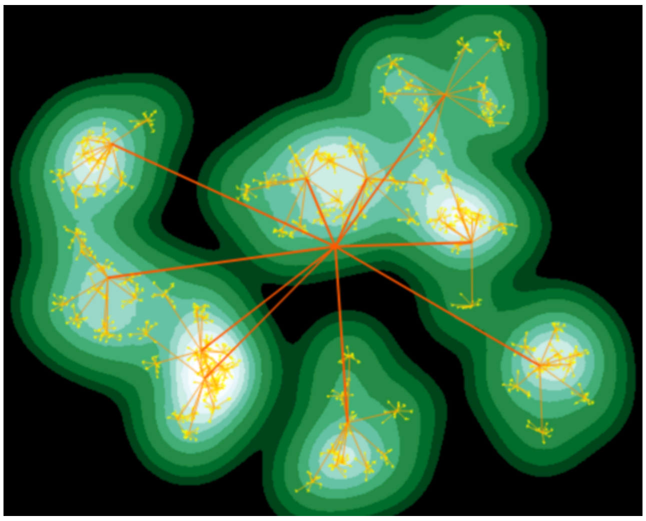

Figure 10.d: Most clusters merged. Subclusters remain visible due to the normalization algorithm.

7.4 Multiple Density Fields

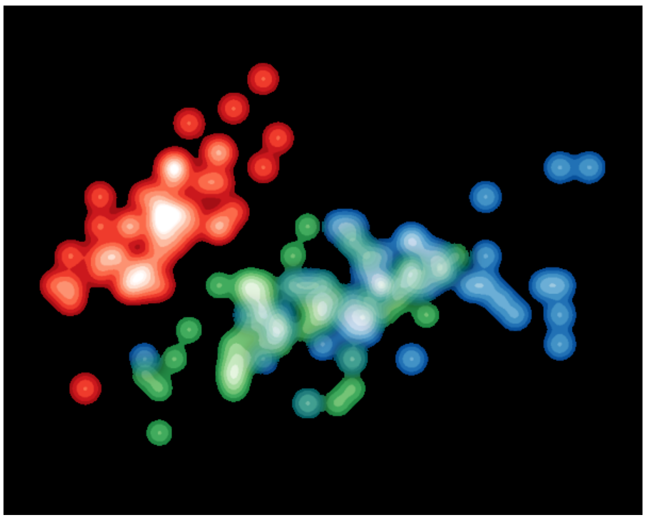

Figure 11.a: The single values of the Iris data set. *Sepal Length* versus *Sepal Width.*

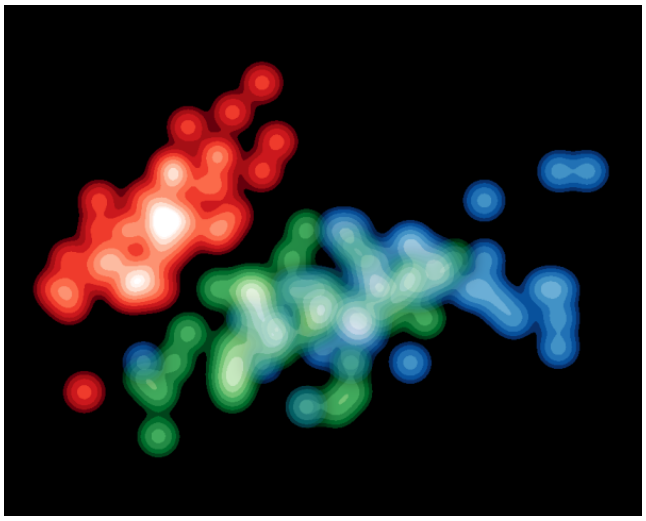

Figure 11.b: Nearby data points merge and form small clusters.

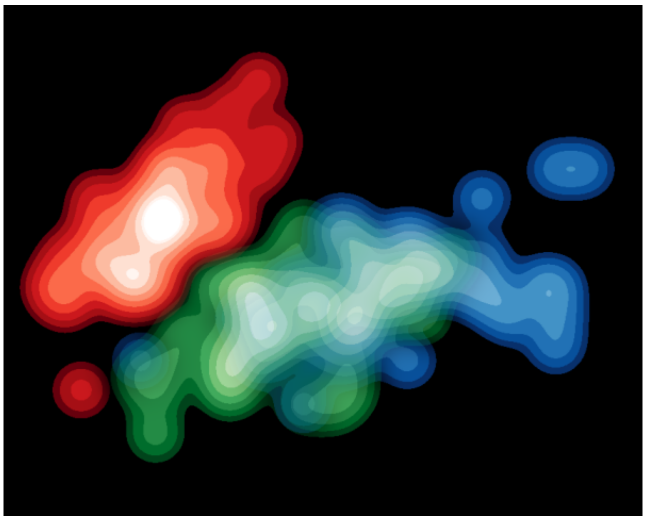

Figure 11.c: The clusters merged but the single values in the lower left corner are still disconnected. Also one can see two red heat spots.

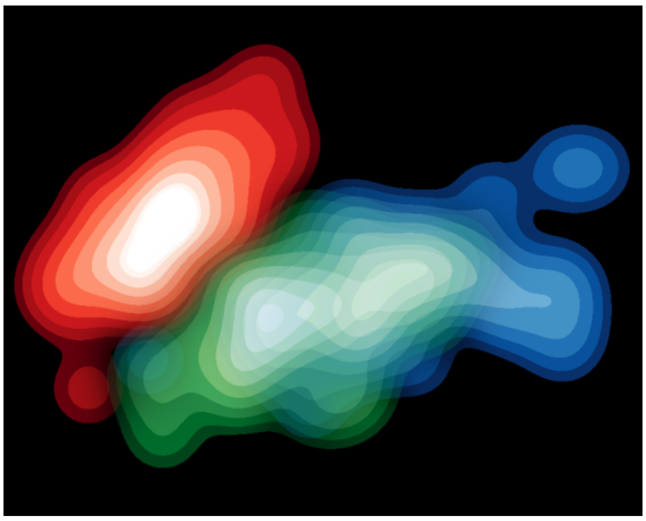

Figure 11.d: Although the green and blue cluster have a large overlap the cores of their clusters can still be seen.