University of Bergen

INF319 - Project in Visualisation
Spring Semester 2020

Enhanced HexBin Plots

Maximilian Sbardellati
292332
msb001@uib.no

supervised by
Thomas Trautner
Stefan Bruckner

June 12, 2020

Contents

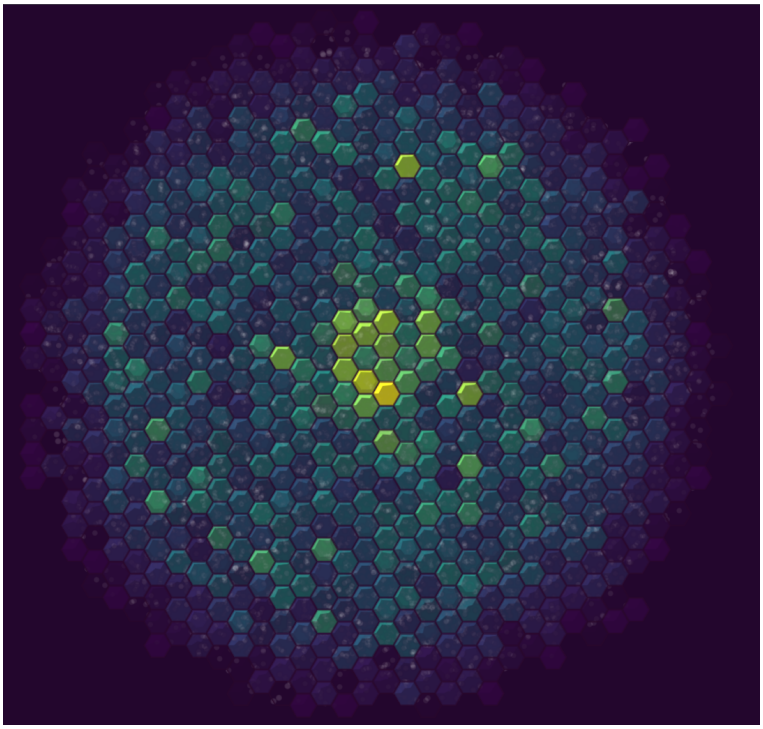

Figure 1: HexBin plot featuring tile opacity derived from tile discrepancy and cut of pyramids for 3D effect.

1 Introduction

This document features as a technical report on the project *Enhanced HexBin Plots*. The goal of this project was to develop and implement a visualisation is based on hexagonal binning of scatterplots. Since binning in general gives us a very abstracted view on the dataset, we tried to enhance it to also give more information about the underlying data distribution. We did this by incorporating two main ideas: First, we calculate the discrepancy measurement for each tile, set its opacity accordingly and blend it with the original data. This allows us to show interesting point distributions in details and hide not so interesting ones. The seconds idea is to create a 3D effect by intersecting a regression plane, that is fitted to our dataset using a Kernel Density Estimation (KDE), with a pyramid for each tile and render the cut of pyramid. The result of combining this two ideas is shown in Figure 1. In the following Section I will describe, how all the different features are implemented, before discussing how they are interconnected end dependent on each other in the rendering pipeline.

2 Features

This project is based on the *molumes* framework written in C++ and GLSL. We added a new Renderer class called *TileRenderer* to the project and implemented our visualisation there. In the following subsection I will explain how we implemented the different features of this project. The user is able to turn all this features on and of using checkboxes in the GUI (see Figures 9). All the different shader programs used are initialised into a dictionary where they can be queried by their name.

2.1 Point Circle Scatterplot and KDE

The *point circle scatterplot* is created by rendering the data-points as circles and blending them over each other (see Figure 2). The corresponding shader program is called *point-circle* and uses the shaders: *point-circle-vs*, *point-circle-gs*, *point-circle-fs*. Since a *KDE* is basically the same thing, we also use this shader program to compute the *point circle scatterplot* and *KDE* in the same rendering pass and save them in separate textures: *pointChartTexture*, *kdeTexture*. The vertex shader simply gets the x, y coordinates of the data-points as input and gives them to the geometry shader as they are. In the geometry shader we create a square geometry. The size of the square is the maximum of $pointCircleRadius * 2$ and $kdeRadius * 2$. *pointCircleRadius* and *kdeRadius*(Sample Radius) are user adjustable parameter and are found in the GUI (see Figure 9).

In the fragment shader we discard all fragments that are outside of the corresponding radius. For the *point circle scatterplot* we then compute an alpha value according to the position of the fragment in comparison to the circle center using a *smoothstep* function to fade it towards the outside. To compute the *KDE* we use a Gauss-function with a user adjustable *sigma* and *density multiplier* (see Figure 9).

We write into both textures using additive blending. For the *point circle scatterplot* we use the shader program *max-val* to compute the maximum alpha value of the texture and save it in the SSBO *valueMaxBuffer*. This maximum value is later used to normalize the color of the texture, since additive blending results in values > 1.

2.2 Tiling

The tiling of the scatterplot is done in 2 stages. First we count how many points are in each tile and the in a separate render pass, get the maximum count, saving it in the SSBO *valueMaxBuffer*. Second, we set the color of the pixels accordingly. We did *square* and *hexagonal* tiling. The size of the tiles is adjustable by the slider *Tile Size*. In the function *calculateTileTextureSize()* we transform the tile size from the slide into a size in World

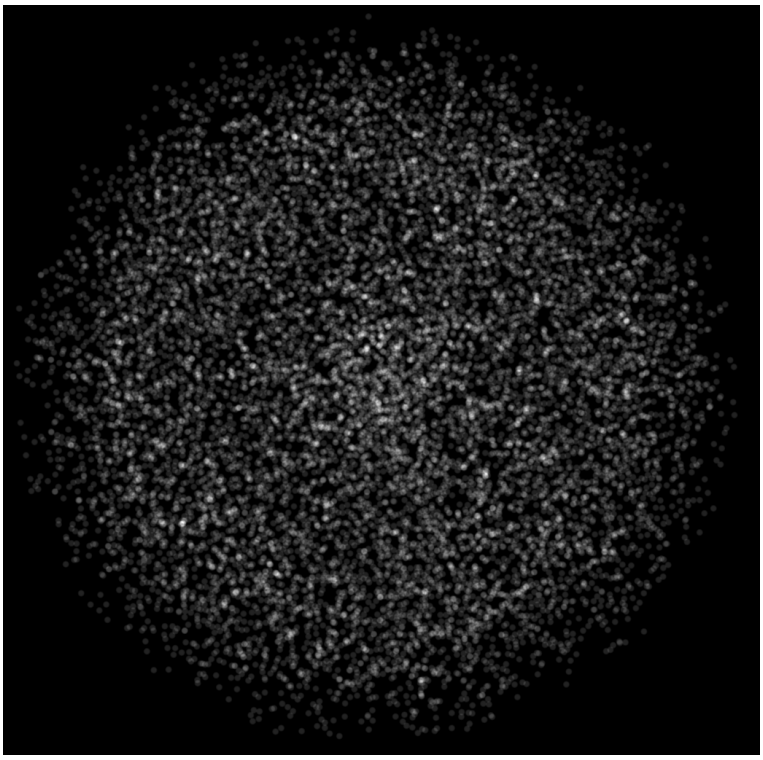

Figure 2: Point Circle Scatterplot

Space and use this to compute how many rows and columns of tiles we need. According to this we set the size of the textures *tileAccumulateTexture*, *tilesDiscrepanciesTexture* and the SSBO *tileNormalsBuffer*. To write into these smaller texture, we also set the *Viewport* to the resolution of the grid before doing the accumulation render pass and resetting it afterwards. We also calculate the bounding box of the tile grid which in most cases is a bit larger than the bounding box of the data-points.

In the accumulation vertex shader *square-acc-vs* and *hexagon-acc-vs* we map each data-point to a tile getting already the correct coordinates of the small texture. To keep these coordinates when sending them to the fragment shader, which just writes a value to the texture, we have to do a reverse viewport transformation before setting the *gl_Position*.

The coloring render pass the simply calculates for each pixel, to which tile it belongs and makes a texel fetch in the *tileAccumulateTexture* to get the correct color value (normalized using the maximum accumulation value), as seen in Figure 3.

2.2.1 Square Tiling

To compute the square tile coordinates of a data-point or a pixel, we simply divide the range of the tile grid bounding box into intervals of *tileSize* and check in which interval the point lies.

2.2.2 Hexagonal Tiling

To compute the hexagon tile coordinates of a data-point or a pixel, we need two separate steps. We first define rectangles which are half as wide and half as high as the hexagon as seen in Figure 4. We then map the point to rectangle coordinates and from there to hexagon coordinates. For a more details description look at the method *matchPointWithHexagon()* in the shader file *hexagon/globals.glsl*.

2.3 Grid

To render the grid around the tiles (see Figure 5), the shader program *grid* is used. Square and hexagonal tiles have separate implementations of this program, but the idea is the same in both of them. In the vertex shader we compute the left lower corner point of each square or the center point of each hexagon.

In the geometry shader we compute a triangulated square or hexagon using the previously computed point and the known size measurements of the current tile. The generated tile geometry is by *gridWidth* larger then the size of the coloured tiles. *gridWidth* is a user adjustable parameter (see Figer 9). In the geometry shader we also get the accumulation value for the

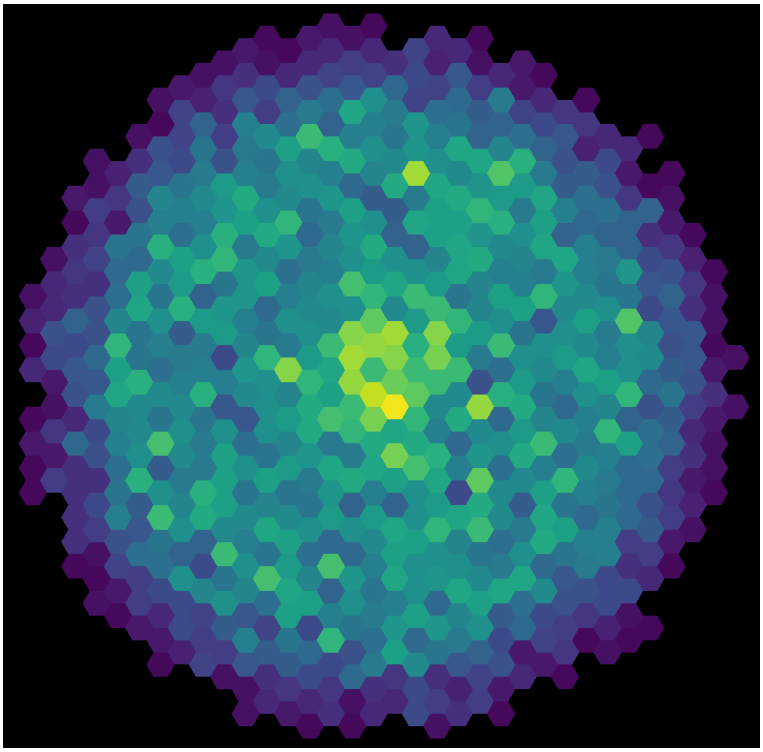

Figure 3: Hexagonal Tiling

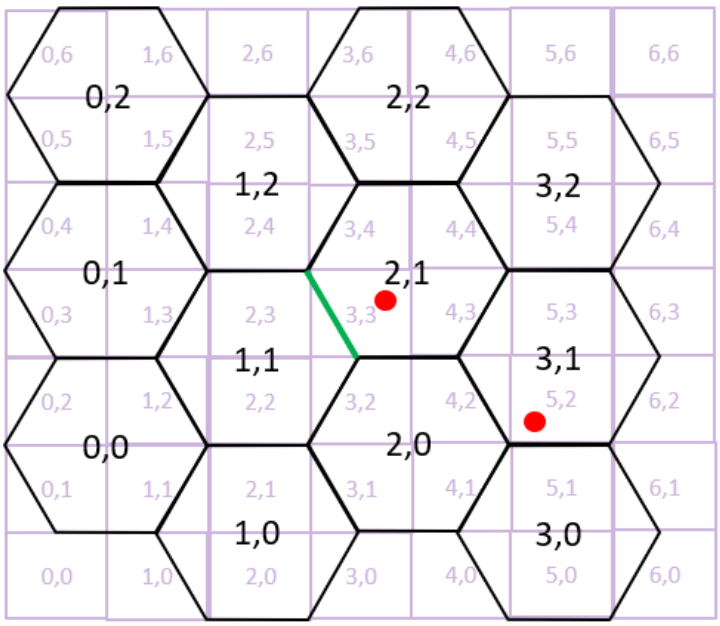

Figure 4: Sketch showing rectangle and hexagonal tiling and their relationship.

current tile and its neighbours. If the current tile is empty (no points were accumulated into it) we do no generate the geometry.

In the fragment shader we then give color to all fragments of the generated geometry, that are within *gridWidth* of the tile border and inside of the original tile, not the larger generated tile. Only if the neighbouring tile is empty (hence getting this values int he geometry shader), we also give color to the fragments outside of the original tile. We also fade the color towards the inside and outside for anti-aliasing. For a more detailed description look at the files: *square-grid-vs*, *square-grid-gs*,*square-grid-fs*,*hexagon-grid-vs*,*hexagon-grid-gs*, *hexagon-grid-fs*.

2.4 Regression Plane

There are three steps in computing the regression plane visualisation. First we need to compute a KDE. We already discussed how this is done in Section 2.1. Seconds we need to compute the pixel normals from this KDE and accumulate them into a buffer. This is done in the shader program *square-normals* or *hexagon-normals*. In the fragment shader of this program we fetch the KDE value of the current fragment and its neighbours from the *kdeTexture* and compute the pixel normal from this information. We then accumulate the normals (*vec4*) and the KDE value of the current fragment into the SSBO *tileNormalsBuffer* using *atomicAdd*. Since *atomicAdd* only

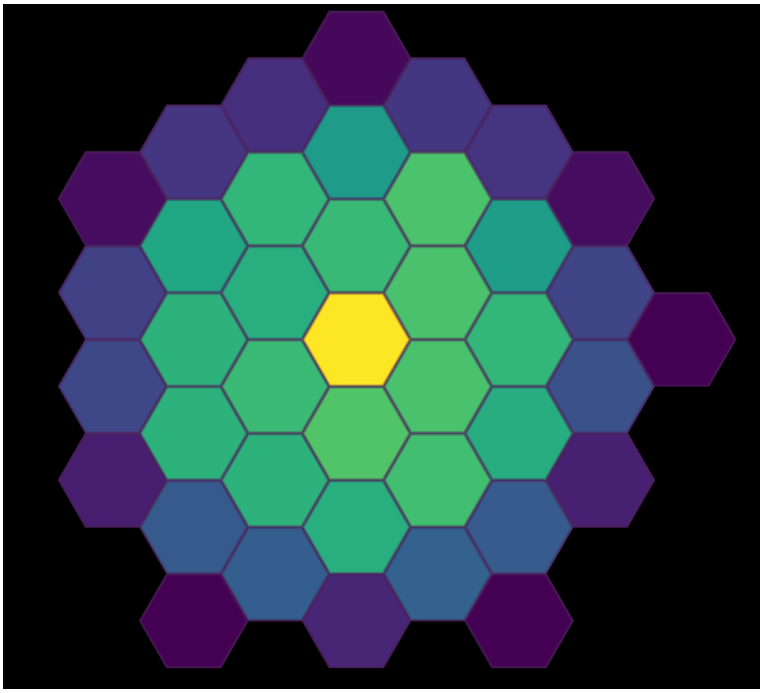

Figure 5: Grid

works on integers and we are working with floats, we multiply our float with a factor of 100 and then cast them to integers. When we read from the buffer again, we divide with 100 to get the original value with a precision of 2 digits after the decimal point.

The final colouring is done in the same fragment shader that performs the colouring of tiles in general. From the computations that were going on for the colouring, we already know in which tile we are currently located. First we compute the height of the given tile by getting the accumulated KDE value from the buffer and normalising it with the w value of the accumulated texture which represents the number of pixels inside of this tile (see Figure 6). The height of the pyramid is also dependent on the parameter *tileHeightMult*, which is adjustable by the user. Additionally it is possible to flip the pyramid to the negative z-axis using the parameter *invertPyramid*.

Next, we compute the normal of the regression plane:
$normalize(vec3(tileNormal.x, tileNormal.y, tileNormal.w))$. The height of the plane inside the pyramid depends on the user adjustable parameter *borderWidth* and the lowest intersection point between pyramid and plane, which is not allowed to be negative. Knowing the pyramid height and regression plan normal, we then compute the intersection points between regression plane and pyramid. Using the edges between the intersection points, we then check for each fragment, if it is in the intersection part or if it lies

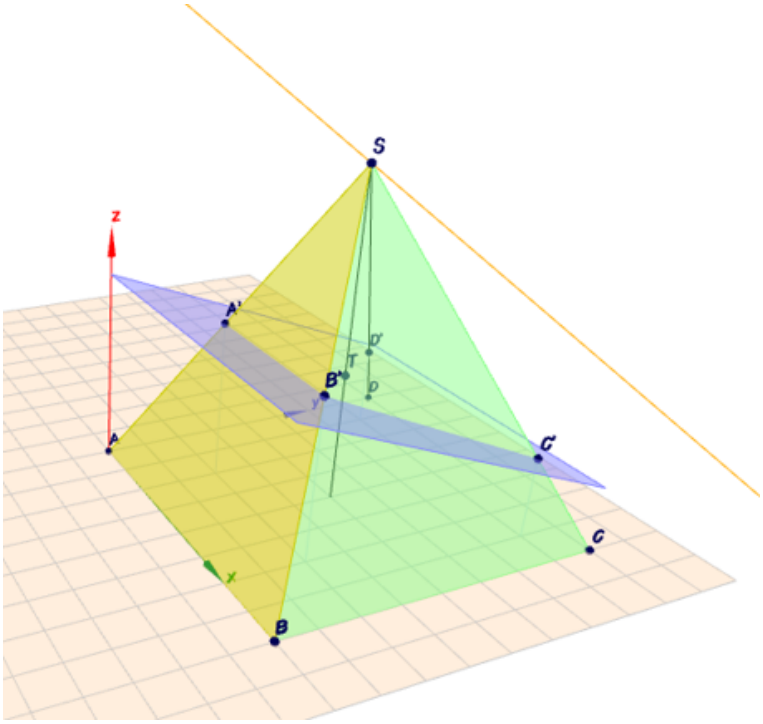

Figure 6: Sketch of regression plane and pyramid

on a side of the pyramid and if yes which side this is. This process is exactly the same for square and hexagonal tiles. For hexagonal tile, we just have to check for more edges. A more detailed description of this checking process look at the fragment shaders: *square-tiles-fs* and *hexagon-tiles-fs*.

If it is in the intersection part, we simply use the regression plane normal to compute the color using phong shading. If the fragment lies on a pyramid side, we first compute the normal of this pyramid side using the two intersection points of this side and one of the corner points of the tile with height 0. We then use this normal again to compute the color using phong shading. To get rid of the aliasing effect produced by the resulting edges we then check for the fragment if it is in the vicinity of an edge and blend the colors accordingly (see Figure 7). For a more detailed description of the colouring process for fragments on the pyramid sides look the the function *calculateBorderColor()* in the shader file *globals.glsl*.

In the course of this process we also save the lighting normal and the z-value into the texture *normalsAndDepthTexture*, which can be used for debugging and implementing post-processing effect in future work.

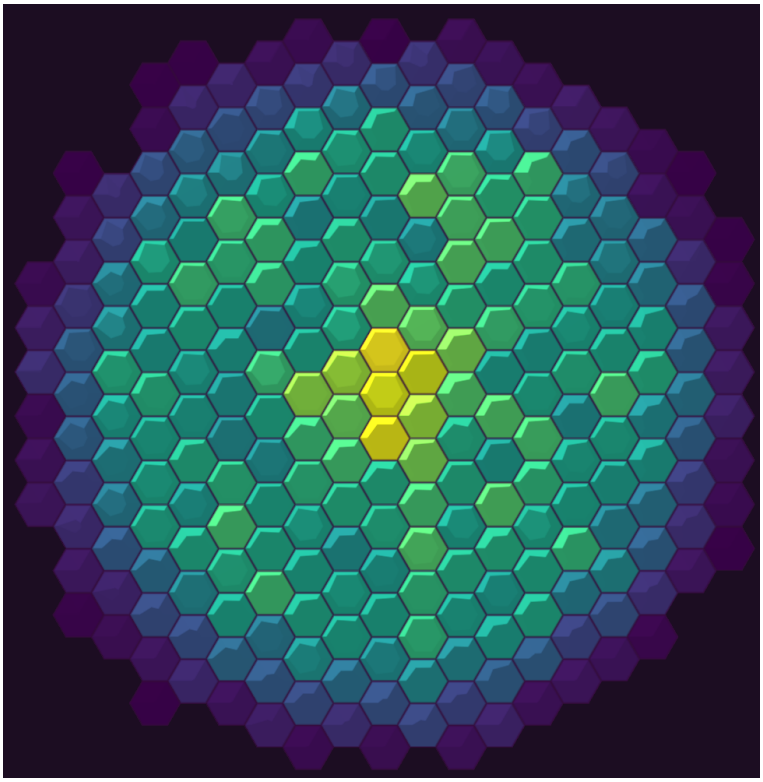

Figure 7: Regression plane visualisation

2.5 Discrepancy

The tile discrepancy is computed on the CPU in the function *calculateDiscrepancy2D()* in the file *TileRenderer.cpp* in four distinct steps. First, we count how many data-points belong to each tile using the corresponding tiling algorithm. Second, we create a prefix sum array according to the count. In the third step, we sort the data-points into the arrays *sortedX* and *sortedY* according to the tile they belong to using the precomputed prefix sum for correct indexing.

In the fourth step we compute the actual discrepancy. We iterate over each tile and within each tile over each point p_i within the tile. For each point p_i we calculate its point discrepancy d_i, by first creating a bounding box from the smallest point inside the tile (= lowest x, y) to p_i. Then we compare the number of points p_j which are inside this bounding box with the number of points inside the tile and the are of the bounding box. In the end the tile gets assigned the discrepancy $max(d_i)$. Since this algorithm has a running time of $O(t * n^2)$, with t being the number of tiles and n being the total number of data-points, it can be very slow. To reduce the running time we use **OpenMP** to parallelize the iteration over the tiles. Because of this it is important to compile the program with OpenMP support (add it to CMakeList).

Once computed the discrepancy is saved in the VAO *vaoTiles* and transferred to the texture *tilesDiscrepanciesTexture* using the shader program *tiles-disc*. The value of this texture is then used in the final render pass to set the opacity of the tiles as seen in Figure 8.

3 Rendering Pipeline

In this Section, I will describe how the rendering pipeline is set up and how the different stages depend on each other. We will also discuss how everything is blended together in the end.

3.1 Tile Classes

Since square and hexagonal tiles need different shader programs, uniforms and pre-computing steps but on the other hand share their basic structure, I implemented an abstract class *Tile* from which the two classes *SquareTile* and *HexagonTile* inherit. The *Tile* class provides and interface for the *TileRenderer* to call the corresponding function or shader program of the specific tile type. It also defines some properties which are shared by square and hexagonal tiles. The classes *SquareTile* and *HexagonTile* provide implementations for fetching shader programs, setting tile type specific uniforms and computing tile type specific tile size measurements.

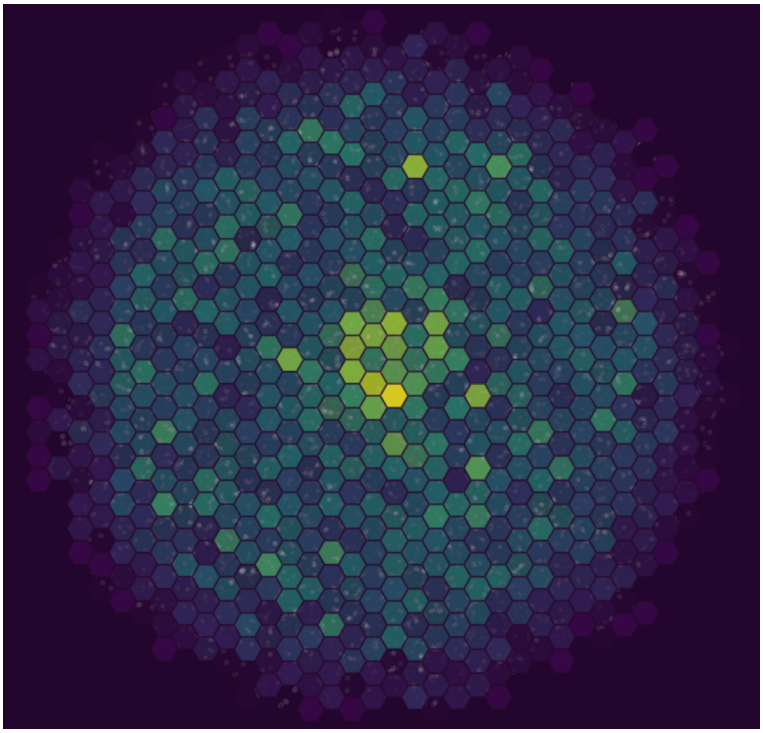

Figure 8: Discrepancy

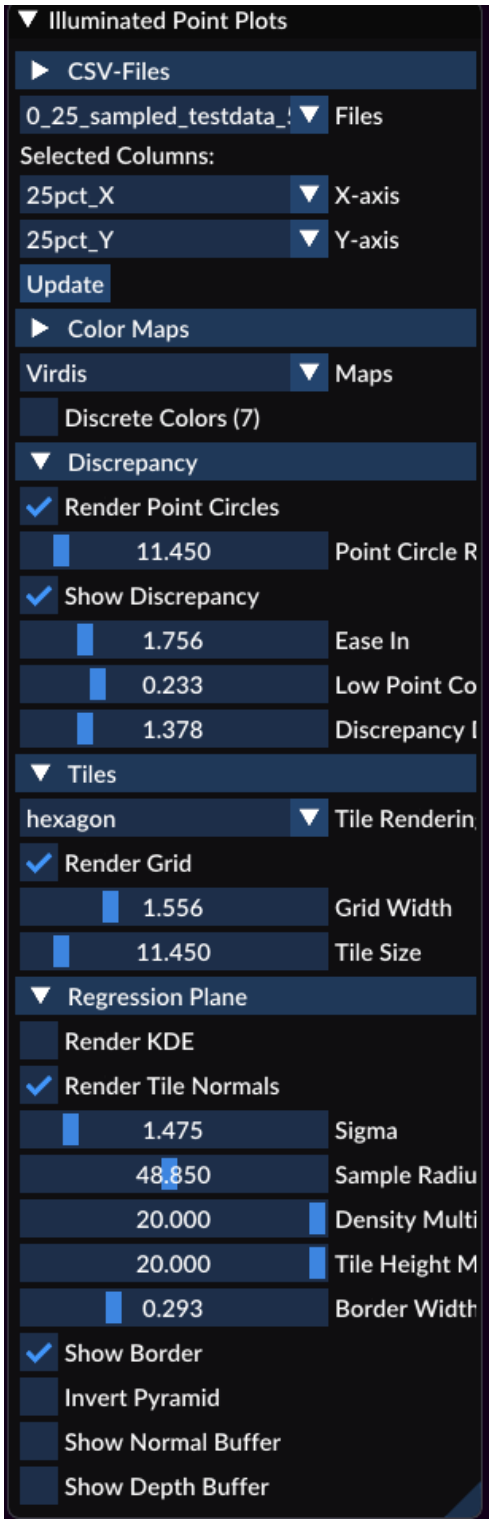

Figure 9: User Interface

When starting the program instances of both *SquareTile* and *HexagonTile* are created and saved in a dictionary with key(*string*) and value(*Tile*). When the user changes tile type we set a reference to the correct instance to the variable *tile* in *TileRenderer* and can simply call the correct function over this interface.

3.2 Dependencies

The different render passes in this program cannot be called in any arbitrary order, because some are heavily dependent on each other. The dependencies from top to bottom are shown in Figure 10 and I explain them in the following paragraphs.

The render pass called *shade* is used to blend everything together and therefore need to be last.

The *discrepancy* path is independent from all other passes and can be done anywhere before *shade*.

The passes *point circle & KDE* and *tile accumulation* do not depend on others and therefore should be the first ones in the pipeline.

The pass *max value* computes the maximum accumulation value created from the *tile accumulation* pass and the maximum alpha value from the *point circle* pass and therefore needs to be done after those two.

To compute the *tile normals*, we need the *KDE* to compute the pixel normals and the *tile accumulation* to check if actually need to compute the normal for this tile, or if it is empty anyways.

To compute the *grid*, we need the *tile accumulation* to check if actually need to compute the grid for this tile, or if it is empty . We also need it to check if the neighbouring tiles are empty, so we can draw the grid on the outside accordingly.

To perform correct *tile colouring* we need the maximum accumulation value from the *max value* pass. For the regression plane and pyramid visualisation, we need the computed *tile normals*.

3.3 Final Blending

The final render pass *shade*, is a fullscreen image render pass. As input it gets all texture that were used to store the intermediate results of the previous render passes. The output is the final colour *col* that will be shown on the screen. To blend on color $colA$ over another $colB$ we use the over operator:

$$col = \frac{colA.a * colA + (1 - colA.a) * colB.a * colB}{colA.a + (1 - colA.a) * colB.a} \tag{1}$$

First, we set *col* to the value of the *pointCircleTexture*. Next, we get the *tile color* of the current fragment from the *tilesTexture*. If the rendering

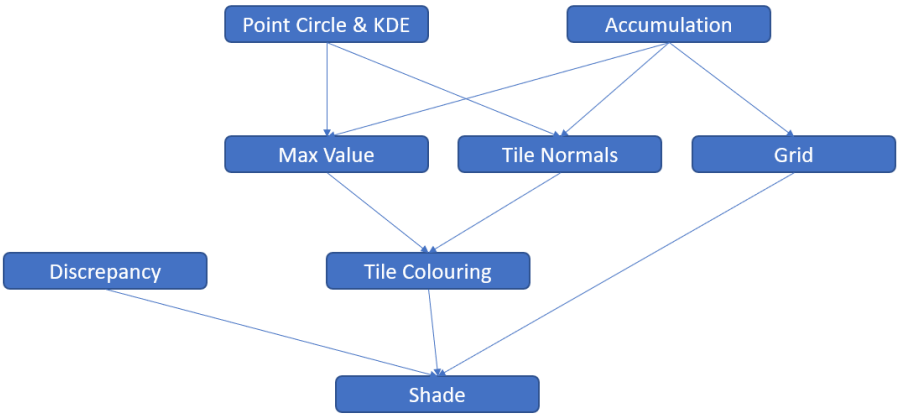

Figure 10: Shader Dependencies

of discrepancy is enabled, we use the discrepancy as alpha value to blend the *tile color* over the *point circle color*. If not we blend the *point circle color* over *the tile color*. If no points are rendered at all, we simply set *col* to the *tile color*. In the next step we add the background color to our image by blending it with the current color using the *1-alpha* method: $col = col + vec4(backgroundColor, 1) * (1.0f - col.a)$ In the end we blend the *grid color* over *col*;