Project Title: Point Cloud Interpolation Using Natural-Neighbors on the GPU

Project and Algorithm Description:

Data acquired in certain imaging modalities yield data that does not necessarily lie on a grid. This prevents us from using commonly used interpolation techniques, such as bilinear interpolation, when increasing image resolution, acquiring data that lies between the collected data points, or displaying the data on a screen. Natural-neighbors interpolation enables us to conform the data points to a grid, which enables us to perform the operations mentioned above.

Natural-neighbors interpolation (NNI) works by combining weighted portions of data from surrounding data points (neighbors) in order to create a new (interpolated) data point at the query location. One method of determining these weights is by forming a Voronoi diagram of the data, and then calculating how much area the new point steals from its neighbors. A major problem here is that calculating the Voronoi diagram is computationally expensive, and serial implementations make it impossible for NNI to be used in real-time.

We use a parallelized method for computing a discretized version of the Voronoi diagram, called the Jump-Flooding Algorithm (JFA), specifically, the 1+JFA variant of this algorithm. In JFA, the original data points are discretized (snapped to a grid) and used as seeds for flooding the grid (we can think of these seeds as holes in the bottom of a boat). It is important to note that, the finer the grid, the more accurate the results are, since the points will be snapped to locations that are closer to their original positions, however increasing the mesh resolution increases the computation time. Each square on the grid is mapped to a thread on the GPU. At every step, the thread looks in eight directions around it and a certain step size away from itself, in order to determine which seed it belongs to. This can be thought of as back-tracing the source of a water molecule on the bottom of my boat; the source of the molecule should be the hole that is closest to the molecule (assuming that water molecules from different sources do not mix). By halving the step size after each iteration, we compute the Voronoi diagram. The 1+JFA variant starts with a step size of 1, then sets the step size to half the width of the grid, and then proceeds with the steps described above. This yields a much more accurate Voronoi diagram, as shown by Rong et. al.

Now that we can compute the Voronoi diagram of our data, we can interpolate it. The interpolated points will form a grid. For each square in this grid, we can compute how much area this square is stealing from each seed, and divide by the area of the square in order to get the weights for the interpolation. Multiplying these weights by the value the corresponding seed and summing these numbers we get the interpolated value for the square, and we are done.

How to Run the Program:

The main program is *NNI.py*. There is a helper class dataIO.py that is used to read-in bitmap images and data from text files, and *JumpFlooding.py* that performs the actual NNI computation. These Python files can be found under the *./source* directory. There is a set of test data under the *./data/2D/* directory.

My program doesn't currently use command line arguments. In order to perform an NNI, the following command suffices:

```
>> python NNI.py
```

At the time of submission, the program was set to perform NNI on data stored in a text file '*../data/2D/testdata2.txt*'. The contents of this file is:

```
12.00670    17.00110    255
11.50040    17.50030    0
```

The first column holds the x-position, the second column holds the y-position, and the final column holds the data (in this case, the brightness value) of each point.

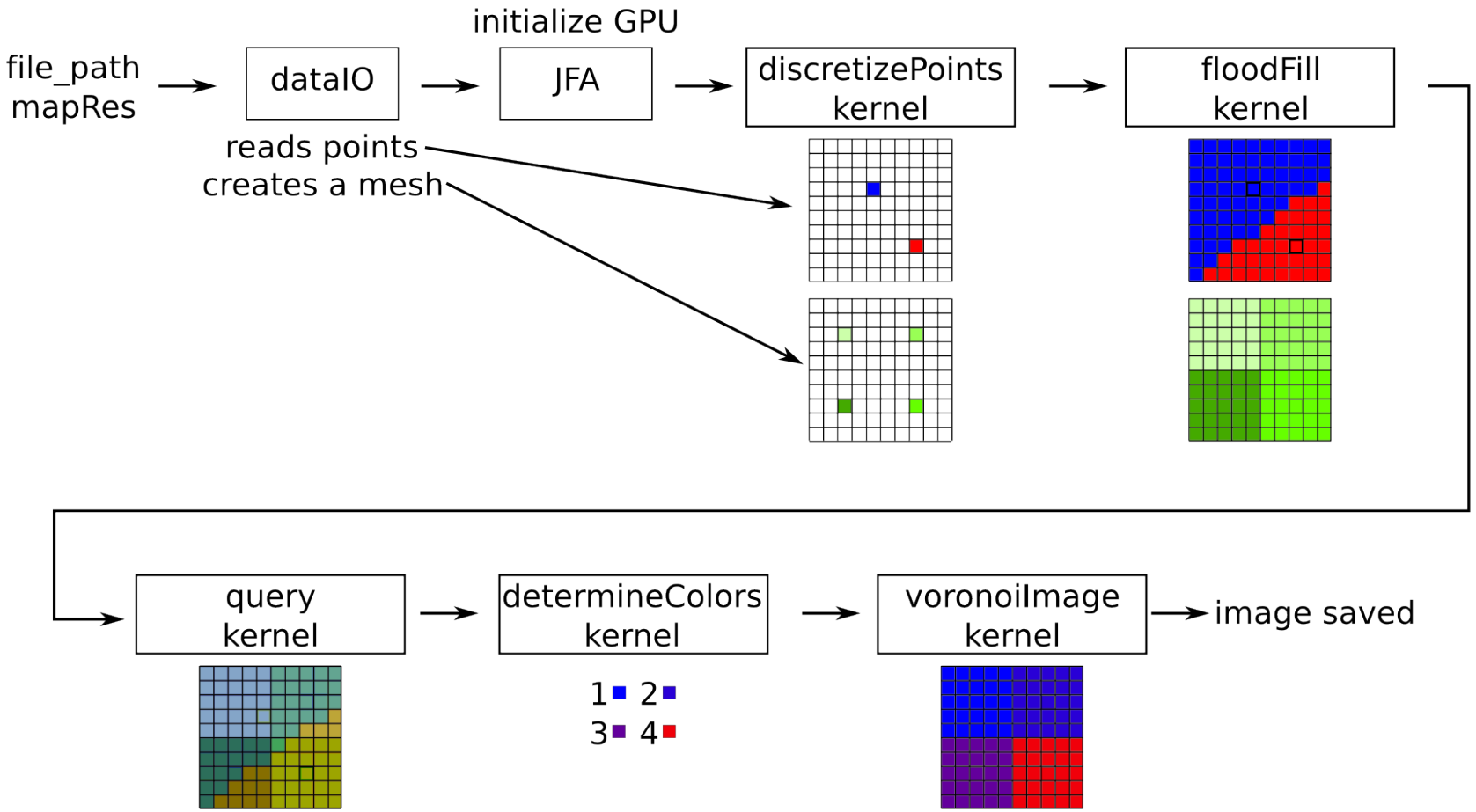
If we want to change the program parameters, we can edit *NNI.py*. *in_file_name* is the input file containing the data points. *out_file_name* is the output file where the result gets saved to, as a PNG image. *mapRes* is the resolution coefficient of the Voronoi diagram; the smaller this value is, the higher the resolution will be, and the interpolation will be more accurate. 0.05 is a good value for mapRes. A mapRes of 0.05 means that one unit length in the data coordinates corresponds to 20 unit lengths in the Voronoi image.

In order to use an image instead of a text file, do the following in *NNI.py*:

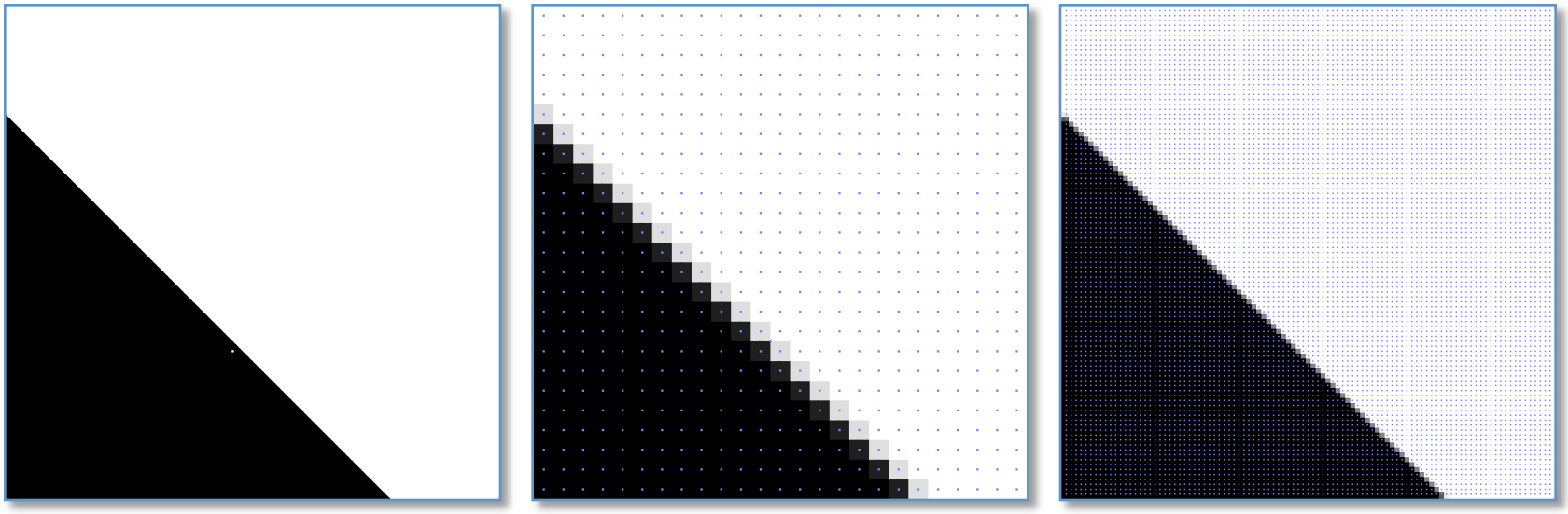
```
# Inputs
in_file_name = "../data/2D/testdata2.txt" comment out
#in_file_name = "../data/2D/testMRimage.raw" uncomment
out_file_name = "voronoi2.png"
mapRes = 0.05 #0.1 - 0.01 / 0.05 is good

# Import data using dataIO
data = dataIO.textDataIO(in_file_name, np.float32) comment out
#data = dataIO.imgDataIO(in_file_name, np.uint16, [64,64], 1) uncomment
```

The number 1 highlighted in red on the last line of code above controls a variable in *dataIO.imgDataIO*, and it can be changed to a 0. When it's set to 1, *imgDataIO* will remove a strip of data (row 28) of the *testMRimage.raw*. This functionality is used to demonstrate the program's ability to deal with non-gridded data / data corruptions.



The diagram above is a simplified representation of the path that data follows in the pipeline. This diagram corresponds to running NNI on *testdata2.txt*. The output of this can be seen on the next page.



(1) (2) (3)

(1) shows the Voronoi diagram of two points (given on page 2, and in test). (2) shows the NNI of these two points with a grid spacing of 1. (3) shows the NNI of these two points with a grid spacing of 0.25. Map resolution is 0.01 in all cases, resulting in a 2500 x 2500 image. The blue dots in (2) and (3) show the center of each element on the grid.

Future Work:

Currently, the text-based input has the query grid set at 25x25. I would like to determine this number automatically.

NNI only runs on square grids, however it is quite simple to make it run on rectangular grids.

I would like to extend to 3D/4D.

Currently, NNI computes the Voronoi diagram for both the input points, and the query points. When the query points are assumed to form a grid, computing the Voronoi diagram is unnecessary, and can be constructed using simpler means that will take much less time. This will reduce the computation time by almost a factor of 2. This is especially important if a stream of data is being interpolated, where the query grid structure remains the same.

Comments on Performance:

I'm quite happy with the performance of my implementation.

At the nominal settings (mapRes = 0.05, mesh reso = 0.25), *testdata2.txt* takes ~9 ms (500 x 500 image), and *testMRimage.raw* takes ~57ms (1320 x 1320 image) to run. The runtime seems to increase almost linearly with image size. This is probably mainly due to the atomicAdd function being used in the query_kernel. As we increase the image size, we also increase the number of iterations of the jump-flooding algorithm (# $iterations = \log_2(image\ width)$).

$$\frac{1320 * 1320}{500 * 500} = 6.97$$

$$\frac{57ms}{9ms} = 6.33$$

It looks like the algorithm satisfies the speed criteria (should run in real-time, meaning at least 24 fps, which is a maximum of 42 ms per frame).