Image Warping Report - Lab 2

What I implemented & How

we focus on the classes and methods I wrote instead of the algorithms that are explained in the homework documents.
I implemented the following three buttons under the "Warping" menu bar.

IDW (Inverse distance weighted)

- I implemented *local_weight_function* which is then used for *weight_function* here I chose the other weight function presented in the paper with $R = 2500$ and $\mu = 3$, then I implemented *local_appro_with_matrix* which is used for *local_appro*, it takes in a vector of matrices D_i, here D_i's are simply assumed to be zero.
- Then I combined the two and got a function *transform* which is then used in the main *warp* function. If D_i's are optimized then we can feed them back to *local_appro_with_matrix* and everything works out. I implemented an optimization process in *initialize()* but there are bugs I haven't fixed yet - it runs but somehow the gradient is too small, thus the optimization process is not started. I intended to use the *LevenbergMarquardt* algorithm.

RBF (Radial basis function)

First I implemented *Gaussian()* with sigma = 200, here I picked the transition function in the paper instead of Gaussian. I maintain an Eigen::MatrixXf private to keep track of the weights.
Then in *initialize()*, we obtain the weights in this function by solving the linear equation, which is then used in a private helper function *transform()*.

Gap filling (used with RBF)

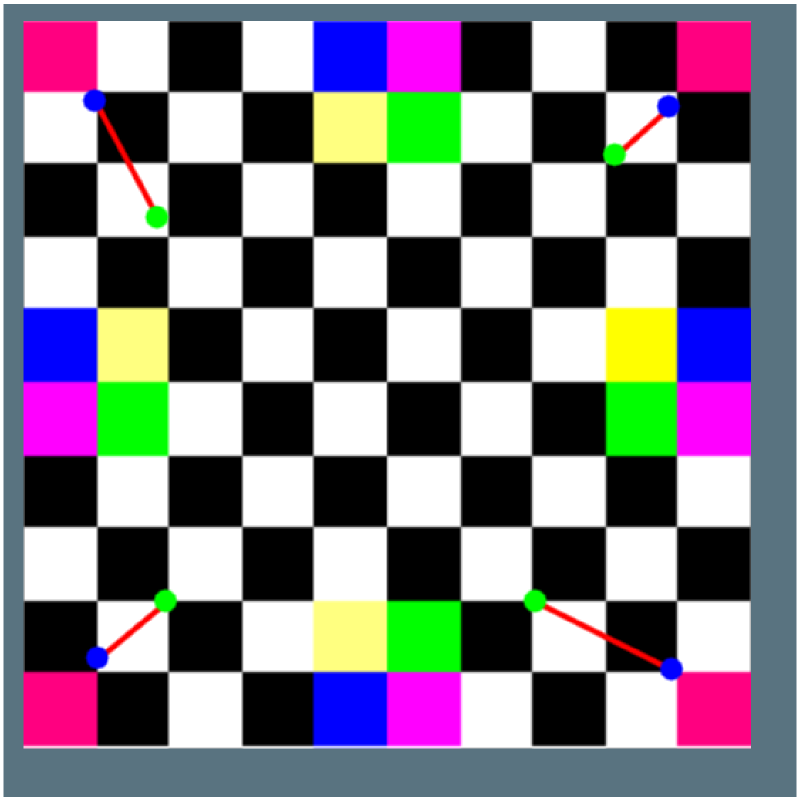
I defined a structure called *Pixel* with three fields. Then maintain two vectors of Pixels: mapped_to and unmapped_to.

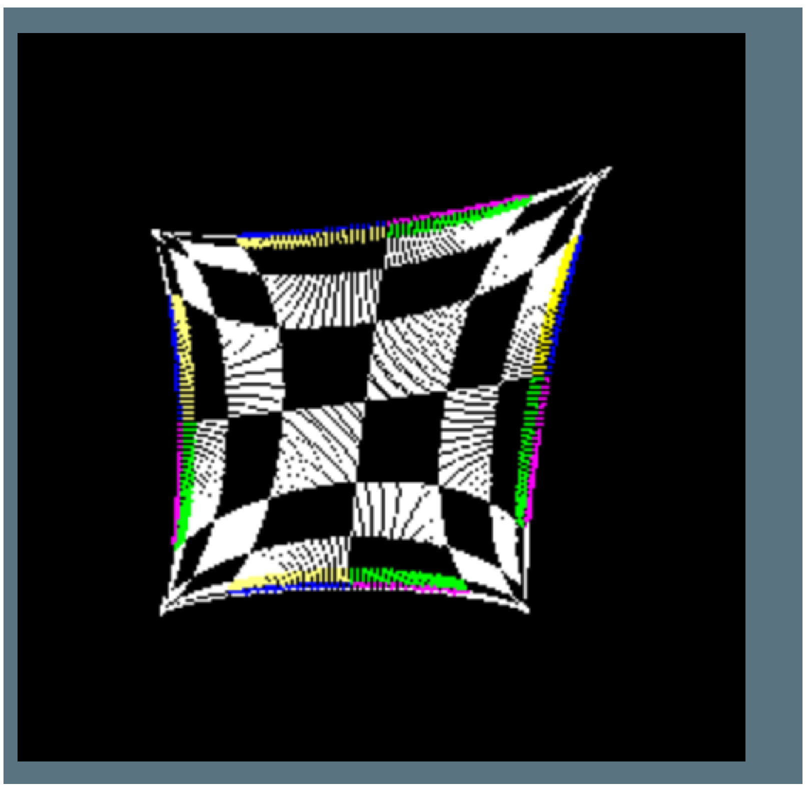
gap_filling()

- Our helper function *gap_filling* takes in these two vectors and returns another vector of Pixels - those Pixels that were not mapped to with their interpolated values.
- I build an AnnoyIndex<int, float, Euclidean, Kiss32Random, AnnoyIndexSingleThreadedBuildPolicy> in *gap_filling*, find its k nearest neighbors, and take the distance-weighted average of their values, k = 70 in our case.
- To discern the boundary, I count the number of uncolored pixels in its k neighbors, only when $k <= 1$ do we update its pixel value, this way we can detect the boundary when contracting the picture. One can tune these parameters to adjust how blurred-out the picture is: k, uncolored count, weights used for average_value.
- It takes roughly 30 seconds to execute one warping.

Result

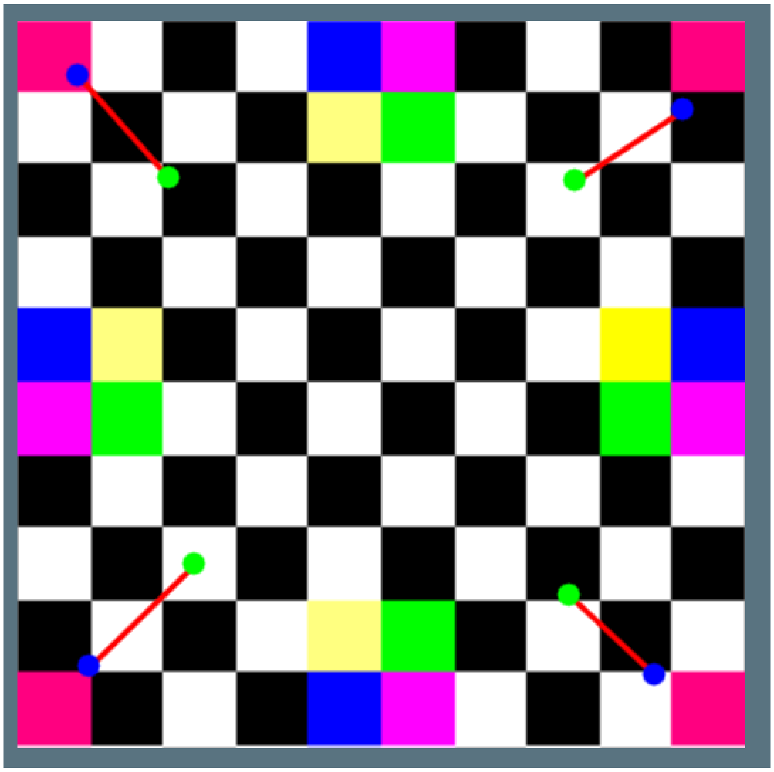
IDW

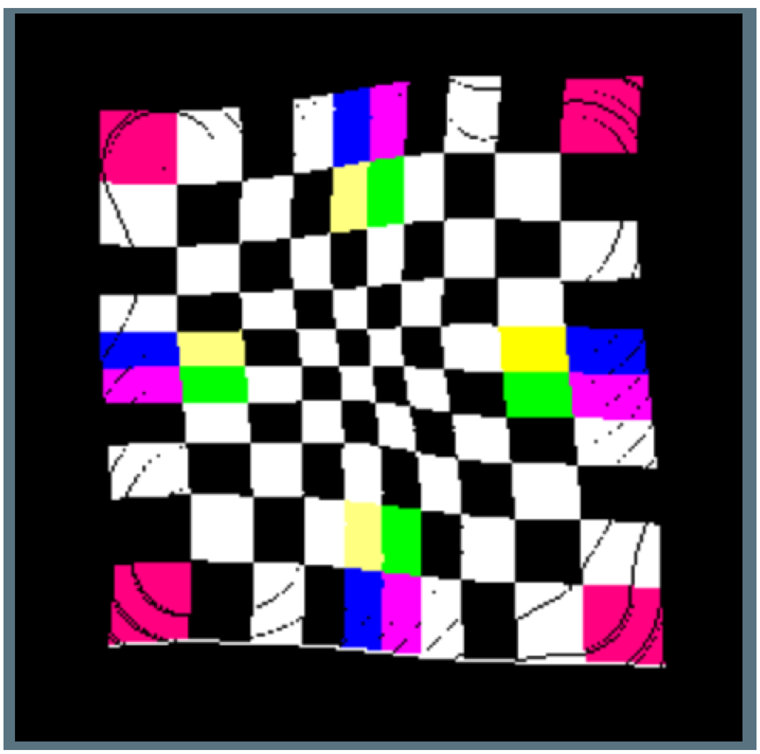


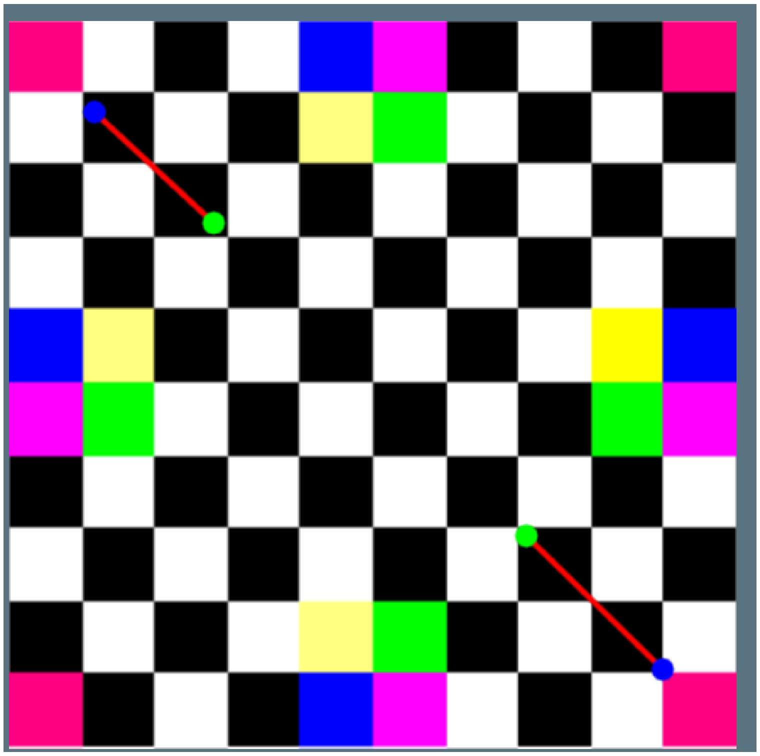


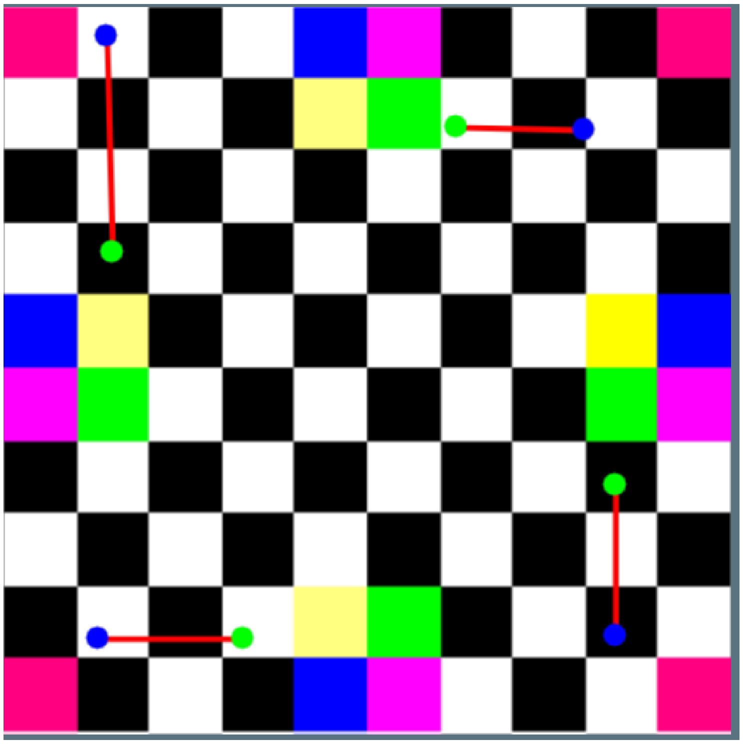
IDW behaves poorly when the number of points is small because D_i 's are not optimized.

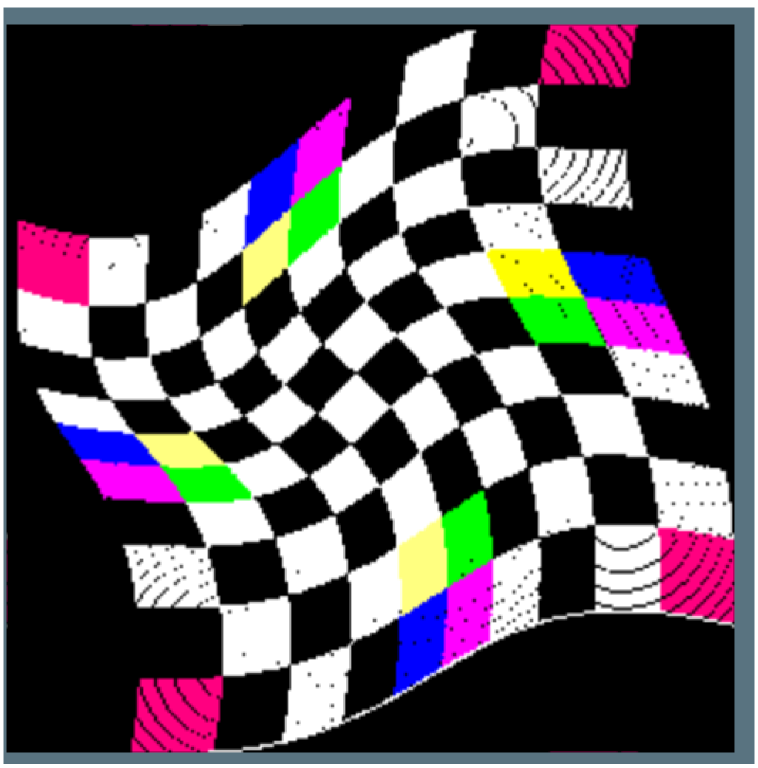
RBF

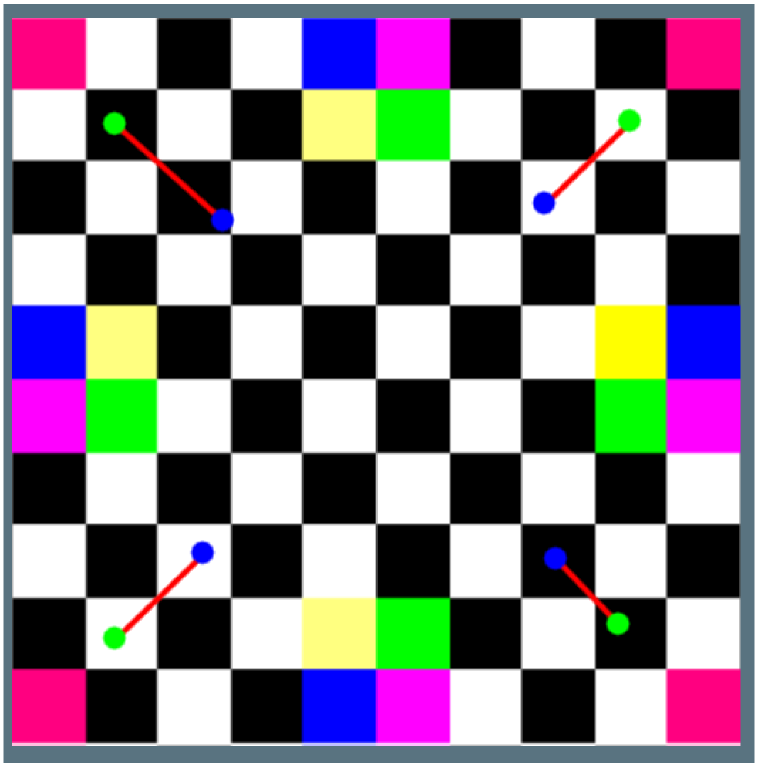


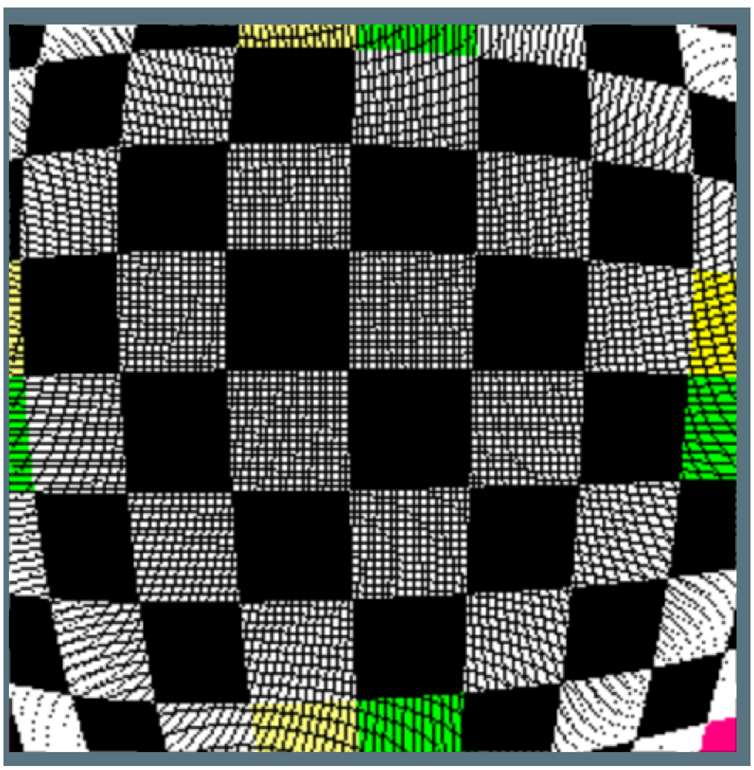




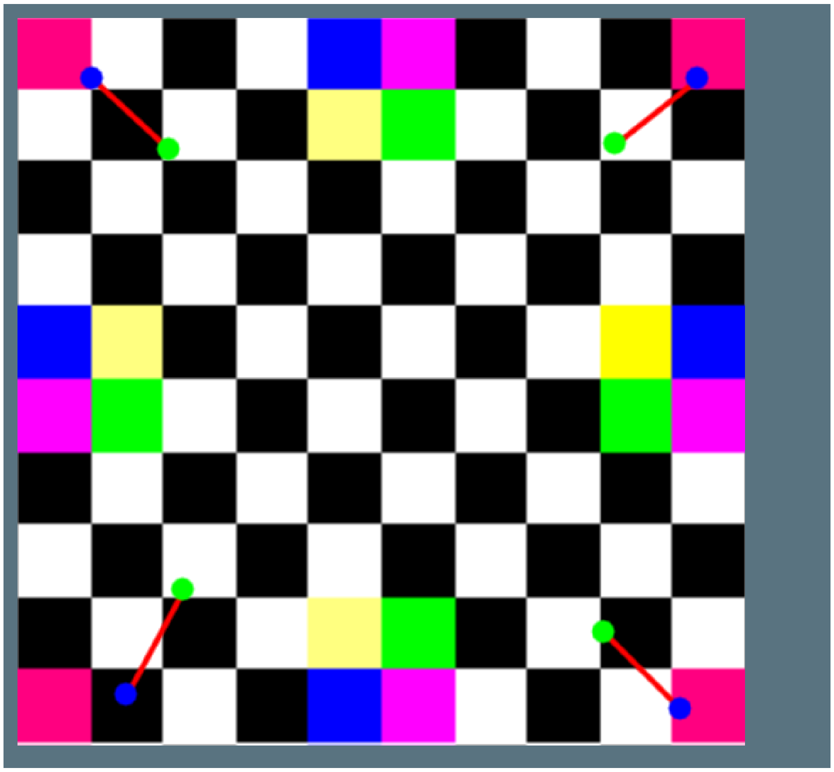



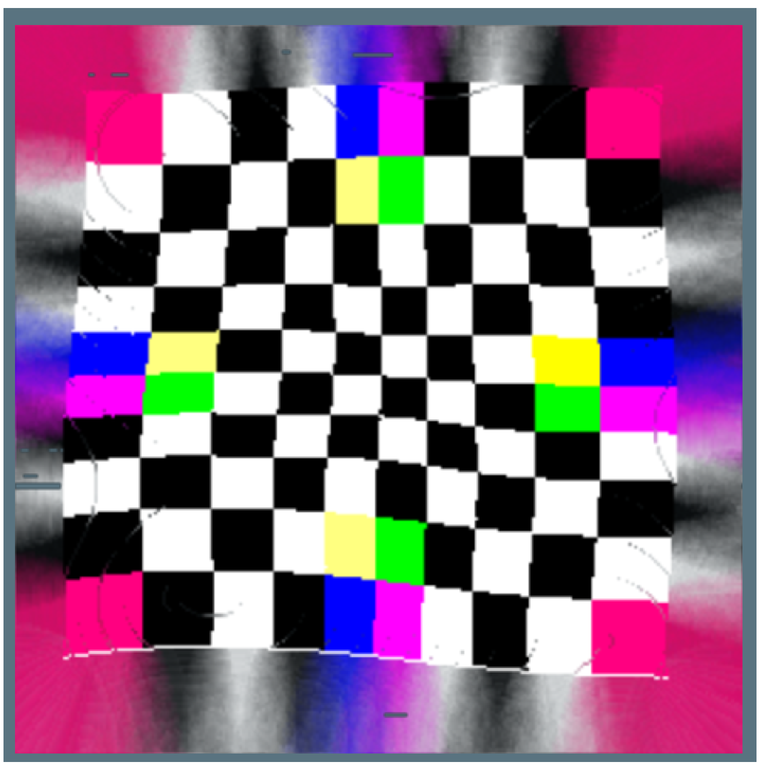


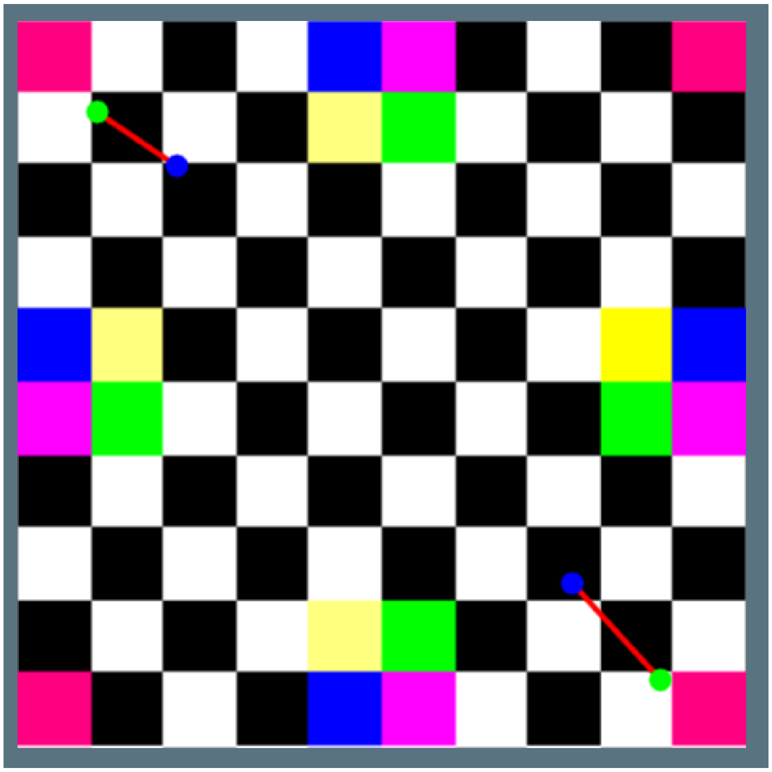


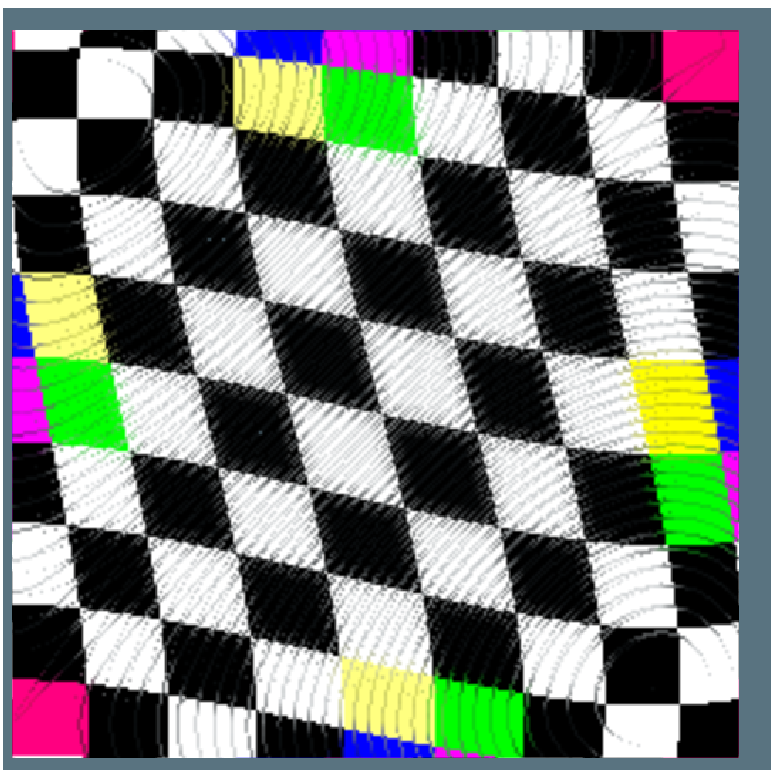


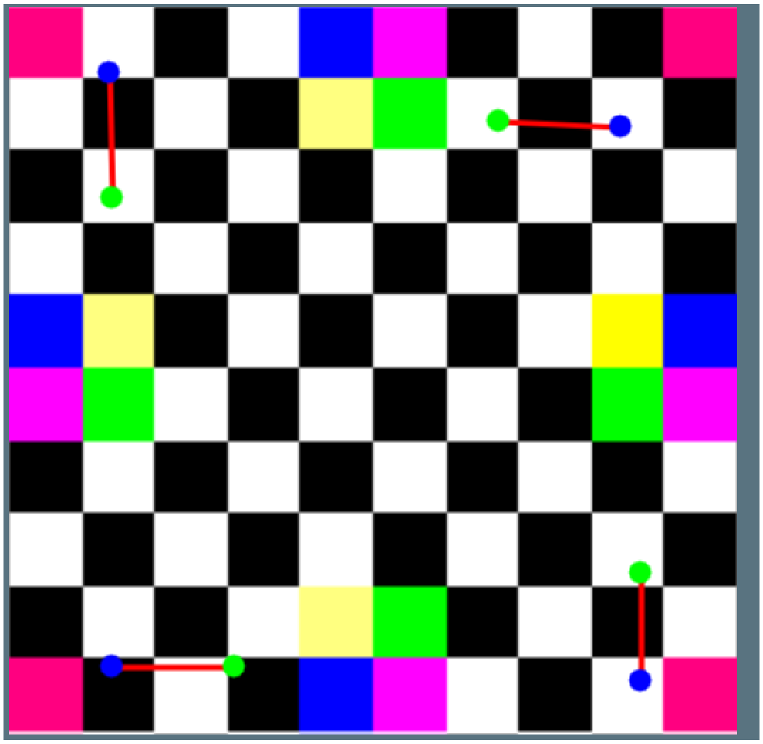
RBF with gap-filling











Future Work (Features I haven't implemented)

For gap-filling

- If we implement a boundary detect method in the *warp()* for gap filling, **I've tried that it can deal with the contraction cases perfectly, but not with expansion since it neglects too many uncolored pixels**. Thus I deleted it, but maybe we can make it work by tuning its parameters.
- tune parameters in *gap_filling()*.
- optimize the algorithm for better performance and shorter running time.

For IDW optimization

- there is something wrong with the setup of *struct IDW::ObjectiveFunction : Functor* < *float* >, the method runs but the optimization process isn't started due to zero gradient. We likely need to adjust how we set up the optimizer, such as its *df* field.
- maybe change the optimization algorithm, using TensorFlow instead.
- Tune the parameters of the weight function in IDW.cpp