TP6: Voronoi Map/Distance Transformation via Jump Flooding

In this TP, the idea is to implement a Voronoi map algorithm called *jump flooding algorithm* (JFA). This algorithm is not exact in the sense that it may contains errors and is not optimal in terms of computational cost. However, it has several properties which makes it useful in some applications.

Exercise 1 Jump Flooding Algorithm

Warm up Create a program which constructs an image with -1 values everywhere and a positive value at some random sites (one unique value per site, this value will be used to characterize each site). The generating function is thus parameterized by n, the image width, and N the number of sites.

The *JFA of step k* (denoted $JFA(k)$) is defined as follow. For each point $p = (x, y)$ in the image:

- Compute the distance between p and the eight sites stored at pixel $q = p + (i, j)$ with $i, j \in \{-k, 0, k\}$ (when q is outside of the image bounds or when no site has been stored at q, consider this distance to be $+\infty$).
- Store the site with closest distance at p in another image.

Questions:

- Implement the $JFA(k)$ method. Save the iterations of the algorithm using stb image (cf Fig. 1).
- What is the computational cost of one JFA step ?
- The overall JFA algorithm consist in JFA(k) steps with $k = \{\frac{n}{2}, \frac{n}{4}, \ldots, 1\}$ (see Fig. 1). Sometimes, we consider a $JFA+1$ in which the last step ($k = 1$) is performed twice. What is the overall complexity of such algorithm ?
- Implement the JFA algorithm.
- Implement the exact "brute force" Voronoi map algorithm (for each point, scan all sites and store the closest). Compare the result and return the average distance error.
- From the JFA Voronoi map, output the distance transformation.

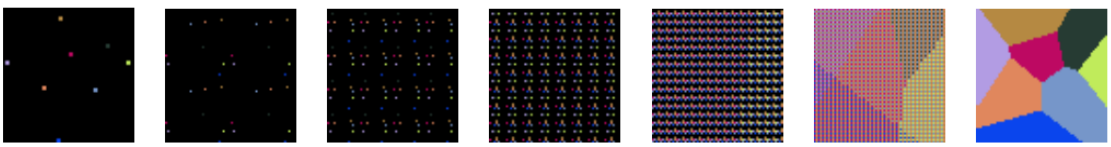

Figure 1: JFA illustration.

Discussion JFA (or JFA+1) is not optimal for the Voronoi map/Distance transformation problem. However, each jump can be implemented in a fine grain parallelism framework: during step k, each pixels can be processed in parallel (4 pixels are read and there is a final write). Hence, JFA is perfect for implementations on GPU where each parallel core runs on texture map pixels (roughly). Note that optimal algorithm discussed in the lecture requires 1D propagation which could be tricky to implement on such GPU model.

Exercise 2 Lloyd's Relaxation

In this exercise, we implement Lloyd's relaxation (also called $k-$means algorithm). In many computer graphics applications, Lloyd's relaxation is widely used in order to "uniformize" point distributions.

The brute-force process is quite simple:

1. Throw N random points in the domain
2. Compute the Voronoi map (here using JFA)
3. For each cell:
    - Compute its barycenter
    - Move the corresponding site to the barycenter
4. Goto step 2 until "stability"

Beside this naive description, Lloyd's relaxation is related to an explicit energy function with many links to several fields (geometry processing, data-mining, ...). The convergence in the continuous plane can be obtained and the limit point distribution has many interesting properties.

Question Implement the Lloyd's relaxation and at each step, output the site map.

You should observe that starting from uniform point distribution, the point set tends to a low discrepancy point distribution (points cover uniformly the space) and later toward hexagonal structures. At this point, the *stability* criterion could just be a number of steps in the iterative process.

An example of the Lloyd's approach can be seen here: `http://www.youtube.com/watch?v=S0sAnabdCLg`