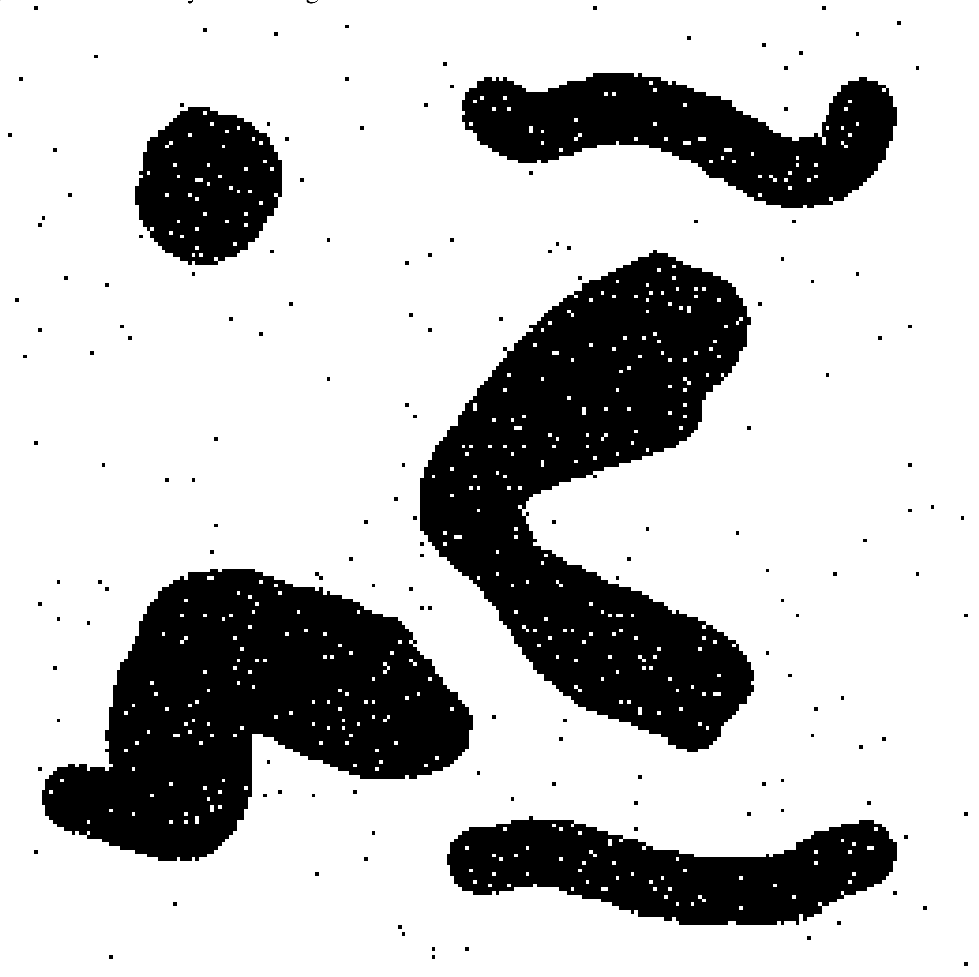
ECE3090 **Fall Semester, 2012**

Project 4 – Image Noise Filtering

Assigned: Feb 21, 2012 *Due: Mar 5, 2012, 11:59pm*

Introduction. High frequency signal noise is often a problem in real-world applications. For example, this noise manifests itself as "static" on phone lines or a fuzzy television reception, or "snow" in video images. While improving transmission line quality is always an important goal, there is a good bit of filtering that can be done on the digital side in order to improve the final quality. In this project, you will be given a noisy black (intensity value of 0) and white (intensity value of 255) image and asked to filter it. The image will contain large regions of black and white with small specs (several pixels of the opposite color) in it. These specs are to be removed from the image. To accomplish this, we will traverse through the image and eliminate any regions which are smaller than a given size (specified at run-time). This will, in effect, filter out the noise in the image and leave the original large signal elements. Small regions are filtered by converting them from white to black or vice-versa.



The Graphical API To visualize your algorithm, we have provided a simplified interface to the *Qt* graphics library. This library is common across Linux and Windows platforms, so the same interface should work on both systems. Our interface uses a `QDisplay` object to interface your code to the *Qt* code, simplifying many of the details of creating and modifying graphics images. The object class definition is provided in file `qdisplay.h`. Details of the API are below.

1. All *Qt* applications require a single instance of an object of class `QApplication`. This class requires the `argc` and `argv` parameters from the main program as arguments to the constructor. The simplest thing to do is create this object as the first line of your `main` program. This is provided for you in the skeleton `filter.cc` program.
2. A new display window can be created by creating an object of class `QDisplay`. The constructor requires a reference to the single `QApplication` object as a parameter. The window is initially empty, and does not display. You can load images or create a blank image with the member functions discussed below.
3. Member function `Load` is used to load an existing image into the display window.
4. Member function `BlankImage` creates an all white gray-scale image of the specified width and height. For later projects, we will modify this to create color images.
5. Member function `Save` will save your gray-scale image to a file.
6. Member functions `Width`, `Height`, and `Depth` return the width, height, and depth of the image. The depth value is the number of bits per pixel. For this assignment, it is always 8.
7. Member function `ImageData` returns a one dimensional array of bytes that represent the pixel data. The size of the array is `Width() * Height()`. Each byte is the gray-scale value of the pixel. For this assignment, we only use 0x00 (black) and 0xff (white) for pixel values. During your filtering, you may choose to put some gray pixels (0x80) in the image for debugging. The data in the array starts with pixel (0,0) in the upper left corner of the display at array index zero, and advances the x–coordinate by one as the array index advances until the right side of the display is reached, and the wraps to the next row, left side (ie. pixel(0,1)).
8. Member `UpdateRate` specifies the maximum allowable update rate for the window, specified in frames per second. Specifying zero indicates infinite updating which is what we want for this assignment, and is the default. For later assignments when we do animation, we will want 25 frames per second for realistic animation.
9. Member function `Update()` re-draws the image with the updated pixel values. Presumably, your program has modified the image data (obtained using the `ImageData` method) in some way and we want the new values displayed. This methods updates the entire image, and can be slow.
10. Member function `Update(int x, int y, int w, int h)` updates only a portion of the display, starting at the specified x and y coordinates and extending for the specified width and height. This is more efficient than simply updating the entire screen, and should be used for this assignment whenever possible.
11. Member function `Run` will simply process *Qt* events until the window is closed. This should be called when your filtering is complete so you can visually inspect your results on the display.

Filtering Algorithm. The algorithm works by finding contiguous sets of pixels that are the same color by visiting all neighbors recursively. If the total number of contiguous pixels is less than a predetermined threshold (we will use a 10 pixel threshold), it is noise and this region should be removed. Otherwise, it is a valid region and is left alone.

1. Define a boolean array `Visited` with one entry for each pixel in the image, initially `false` for each entry. Since we know the exact number of pixels at the start of the algorithm, we can use a `vector` for this, using the constructor that initializes a specified number of elements to a specified value:

   ```
   std::vector<bool> visited(nPixels, false); // nPixels is number of pixels
   ```

2. Find the next pixel that has not been visited. Obviously, this will be the zero'th pixel at (0,0) on the first iteration.
3. Mark the pixel as visited, note the color of this pixel, and push this on a queue of pending pixels. Use the STL *Double Ended Queue* (`deque`) for this, since a `deque` can have elements pushed on the back and removed from the front.

   ```
   std::deque<int> pending;
   ```

4. Create a vector of pixel numbers in this region, initially empty.

   ```
   std::vector<int> region;
   ```

5. While the `pending` queue is not empty:

   (a) Obtain the pixel number from the front of the pending `deque`, and remove it.

   (b) Add this pixel to the current region vector.

   (c) For each of the four neighbors of this pixel (left, right, up, down, *NOT diagonally*), if the color of the neighbor pixel matches the color of this region and it has not been visited, mark the neighbor as visited and append it to the back of the pending `deque`. If it has been visited, ignore it. Be sure to be careful here when taking the left neighbor of a pixel on the left edge of the image, and pixels at the other boundaries. You may want to color it gray (pixel value 0x80) for debugging, but this is not necessary for the assignment.

6. At this point the region starting at the pixel found in step 2 is in the `region` vector. If this size of this vector is less than the threshold (10 pixels), it is noise and each pixel in the `region` vector should be toggled from black to white, or white to black. Otherwise, leave it along.

7. When all pixels have been visited, the image is complete.

Copying the Project Skeletons

1. Log into `jinx-login.cc` using `ssh` and your prism log-in name.

2. Copy the files from the ECE3090 user account using the following command:

   ```
   /usr/bin/rsync -avu /nethome/ECE3090/ImageFiltering .
   ```

   Be sure to notice the period at the end of the above command.

3. Change your working directory to `ImageFiltering`

   ```
   cd ImageFiltering
   ```

4. Copy the provided `matrix-calc-skeleton.cc` to `matrix-calc.cc` as follows:

   ```
   cp filter-skeleton.cc filter.cc
   ```

Testing your program. Two sample images are provided, `test_in_256.pnm` and `test_in_512.pnm`. The images are identical, except that the 512 one is double size. You should debug your program with the 256 image, but final testing and demonstration should use the 512 one.

Turning in your Project. The system administrator for the jinx cluster has created a script that you are to use to turn in your project. The script is called `riley-turnin` and is found in `/usr/local/bin`, which should be in the search path for everyone. From your **home directory** (not the ImageFiltering subdirectory), enter:

```
riley-turnin ImageFiltering.
```

This automatically copies everything in your `ImageFiltering` directory to a place that I can access (and grade) it.