**Assignment 2:**

**Visual Information Retrieval**

**Jason Mann**

**3/12/13**

**COMS 4735: Visual Interfaces for Computers**

Table of Contents

**Application OverView 3**

**Assignment Detail 4**

**Part 1: Gross Color Matching 4**

**Part 2: Gross Texture Matching 5**

**Part 3: Combine Similarities and cluster 6**

**Part 4: Creativity with D3 7**

**OutPut 8**

**Sequence 1 38**

**Sequence 2 39**

**Sequence 3 40**

**Sequence 4 41**

**Sequence 5 43**

**Sequence 6 44**

**Sequence 7 45**

**Sequence 8\*\*\* 46**

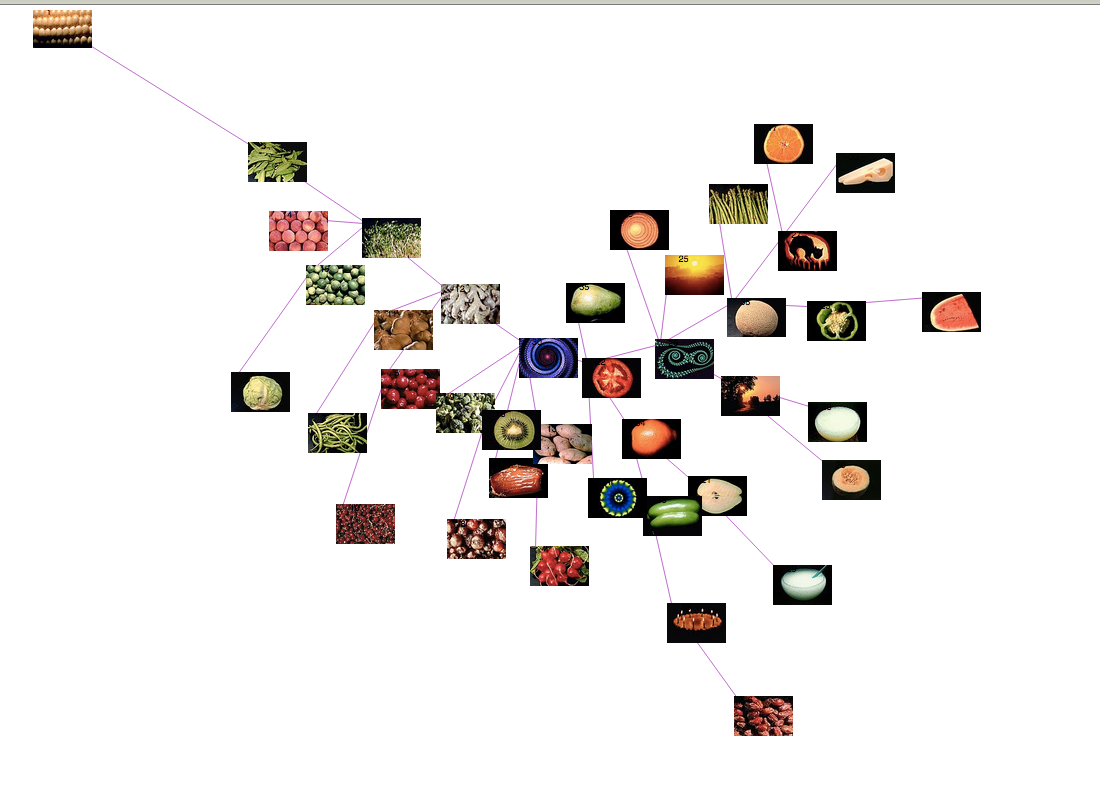
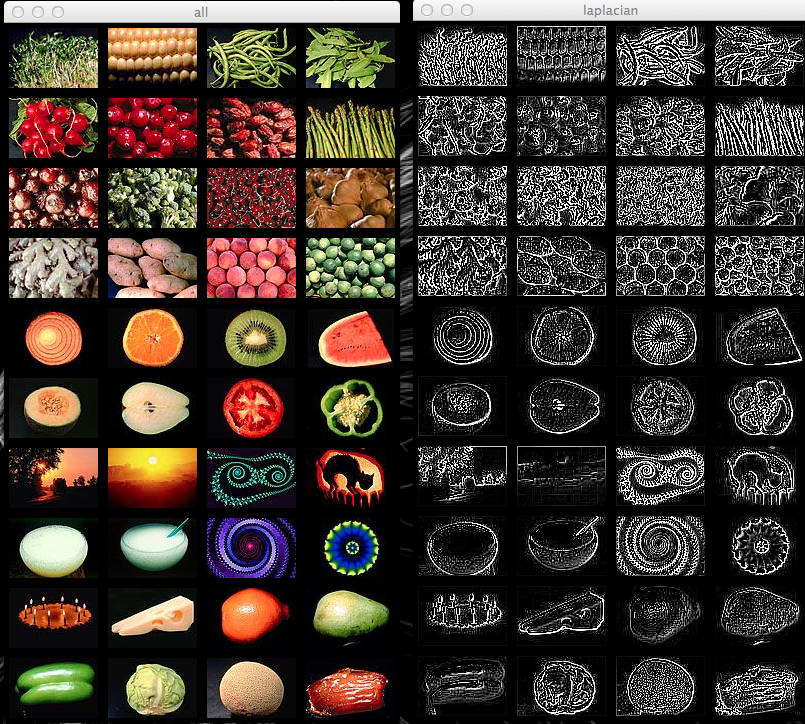
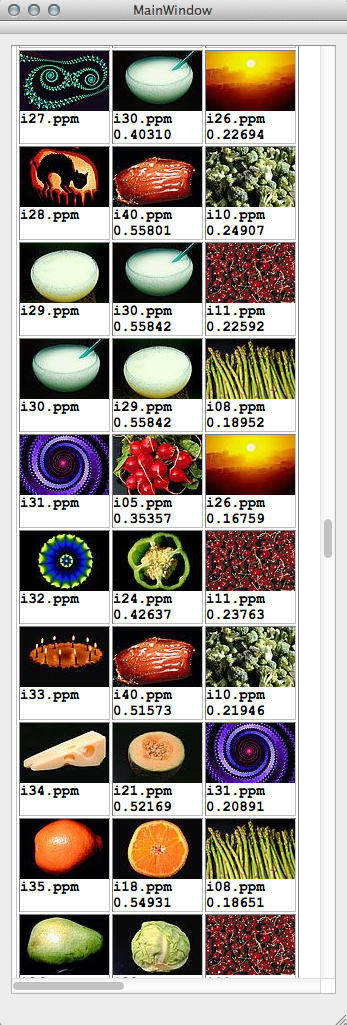
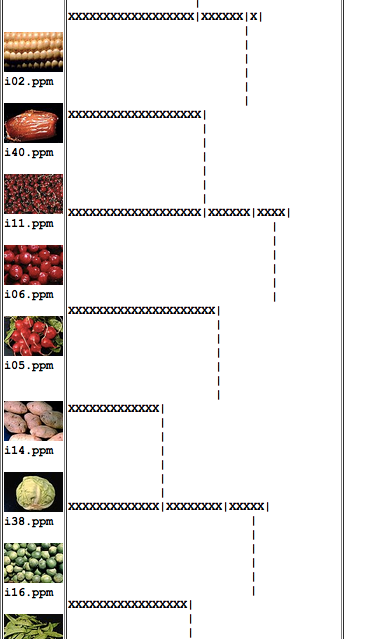
**Sequence 9\*\*\* 47**

**Sequence 10\*\*\* 48**

# Application Overview

**This project was designed for an assignment in COMS 4735 Visual Interfaces for Computers under the supervision of Prof. John Kender at Columbia University in Spring 2013.**

**The goal of this project was to filter out skin regions in imagery (static or motion), detect regions belonging to hands, and then interpret specific pre-defined gestures from the captures, as part of a password grammar.**

**The project is coded in C++, relying on OpenCV 2.4.3 for computer vision algorithms, and QT 4.8 (and QT Creator) for the design of the GUI forms. Some code and inspiration was found in the OpenCV2 recipes, but the majority of the algorithms are my own implementations, and therefore may not be the best performance wise. For the creativity step I used the D3 JavaScript library and examples to plot the images in real time interactively. For my own reference, I created a GUI with a text pane for outputting the images and dendrograms in HTML, as well as two OpenCV windows, one with a combined view of all the original images, and one of the combined laplacians.**

# Assignment Detail

Step One: Gross Color Matching

For step one, we were instructed to compare the 40 images against each other using a normalized L1 comparison of their RGB histograms on a scale of [0,1] (1 meaning total similarity). We were then to display the images in 3 columns showing the original picture, the best match, and the worst match for each image.

The first choice I made was to implement my own algorithm for creating histograms for the images, after I had library issues with the OpenCV includes early in my work. I finally settled on a bucket size of 10 after comparing the results after implementation with a range of bucket sizes. With such a small database of pictures, a bucket size larger than 16 drives any similarity among colors below .4 in my data. I chose to threshold not only the black pixels, but also provided an option to threshold the white values as well, for future use in the case there is white in the background. Here leaving the background out of consideration helped with the appearance of matching, when pixels with a total R + G + B value of under 75 were encountered, they were simply not totaled in the histogram. I also implemented my own L1 normalization function so that I would have better control over the minimum and maximum values for display purposes.

After collecting the L1 data The matching is obviously not perfect, while there are two more steps, but I am actually impressed with the results that I received. There were still quite a few pairs that had almost no histogram overlap, even with only 10 buckets, which shows that it is not a perfect metric, but the best matches align well with my first impressions of matching in the images. The spinning wheels were notoriously bad matches with other pictures, but that is probably why they are here.

See Addendum on Page < > for Images and Data

Step Two: Gross Texture Matching

For part two we were to perform the same comparisons on the 40 images but with the histograms of the laplacians of the images. We were to first create a greyscale image from the RGB (or BGR in OpenCV), and then measure the texture of the images by applying a kernel. **I converted the images to greyscale with my own algorithm, simply dividing the total R + G + B values by 3 and storing the value.** For this part I attempted to use OpenCV’s linearFilter2D function, but found the results were sub-par. However, their Laplacian function performed better than my own, but it uses a matrix that leaves out the corner values of the 3x3 matrix, which i did not think would be allowed. This allowed me to implement my own kernel after the one described in class. The kernel I used is as below:

**When implementing the Laplacian Histogram function I allowed for a threshold to be set to allow for some kind of “background” filter, but found that a threshold of any value made the matching considerably worse for my algorithm, so I decided to not threshold the laplacians. When you are measuring edginess, it makes sense to leave everything in so you get a better proportion of the whole frame of the image, unlike a flat color like black that is fairly uniform. For these histograms I chose a much larger bucket size, while the range of values was much larger and allowed for more refined comparison. I finally decided on a bucket size of 4000, while with the small amount of pictures we have here, I could afford to sacrifice a little performace for accuracy (anything under 3600 for my algorithm drastically reduced matching while my range of values was -1800 to 1800 from the laplacian).**

**I was also impressed with the texture matches (once you can trick your mind to not pay attention to the color). Again the spinning wheels of color cause issues here because of their strange patterns, but the majority mesh nicely.**

See Addendum on Page < > for Images and Data

Step Three: Combine Similarities and Cluster

**For part three, we were to take the measurements we had already collected from the L1 normalizations of the color and texture steps and find a ratio to plug into the linear sum comparing them. This is a problem that could be well suited to machine learning, but I did not have access to a larger database of already “verified” images to test against. Instead I did it the old fashioned way, and asked my girlfriend to help me tally up the ratio choices that had different best matches, giving a point to the ratio that had the best match. On ~90% of the images there was no difference between these ratios, but the fine tuning of the algorithm can drop the outliers out of the equation. I finally decided on a .5/.5 ratio for the color and texture, while the .25 case had a few outliers that I did not like. Hence these measurements from such a small dataset are highly subjective. Here are the results I found:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Ratio (r \* Color) ((1-r) \* Texture):** | 0.25 | 0.5 | 0.75 | 1 |
| **Total Differing Better Matches:** | **11** | **10** | **7** | **6** |

**For the clustering step, I again implemented my own algorithm allowing for one function to cluster on either complete or single link algorithms (while there is only a comparison step change between them). The dendogram was probably my least favorite part of this assignment, and I made a poor choice to try and represent it in ASCII like on the assignment sheet. With A LOT of tweaking I finally got something acceptable to output, but the conversion from the QTextEdit pane to HTML again messes all alignment up, so after a few hours I stopped messing with it. I realize now I could have used a drawing package and drawn the lines, but I have already put too much work into this step.**

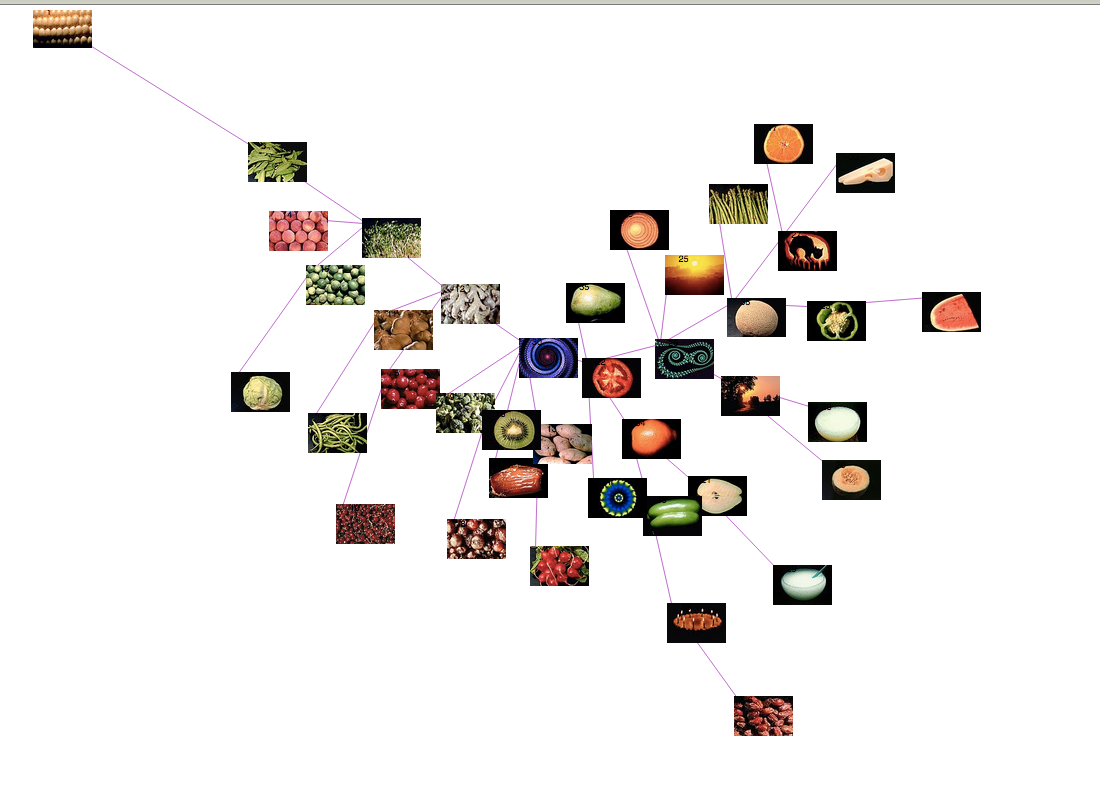
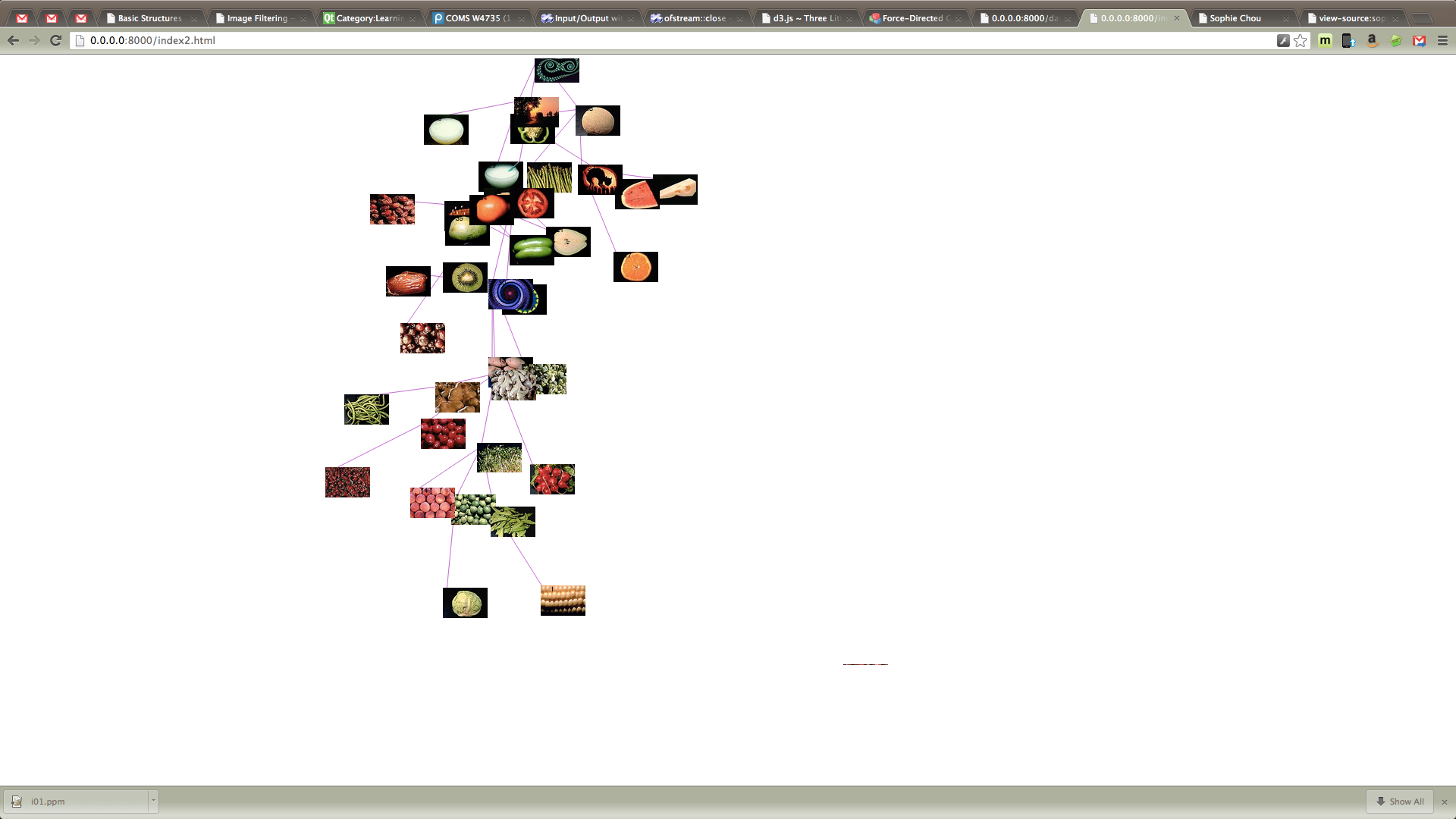
**I can however still notice that the single link algorithm is a much worse clustering, especially in the creativity step below. It seems that with single link simply adds the next nearest, and so therefore not really clustering, but as mentioned in the notes, is more like a minimum spanning tree.**

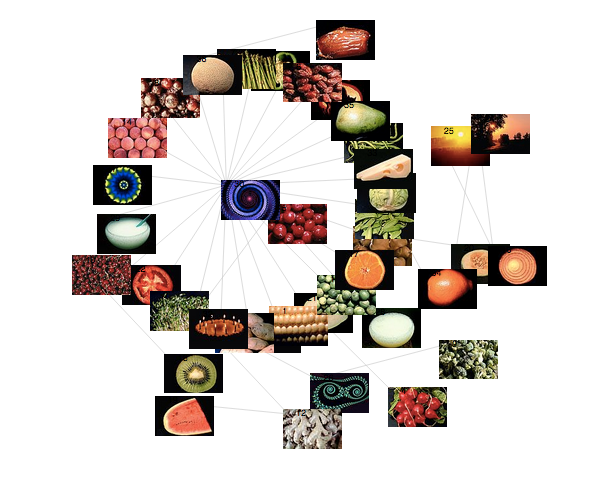
See Addendum on Page < > for Images and Data

Part 4: Creativity with D3 mapping

**For the creativity step of this assignment, I decided to use the D3 JavaScript library, that provides interactive force directed graphs straight from your web browser. I created a tree structure, and a output function that will successfully fill in a JSON file to match the requirements of a standard D3 force directed graph script. This allows for variable adjustment and tweaking. I have found that the most interesting comparisons this creates is between the two clustering methods above, where the complete link method is represented by a fairly diverse graph, with branches to the edge of the frame, while the single link method is the least similar photo in the middle surrounded by the majority at a closed length circle with a few on the periphery.**

**This method of graphing the photos is not only interesting because of its versatility (easily shifting a constant up or down creates different interactions) but it is also interactive, such that you can click on any of the photos, and drag them around to show the ties that photo has to others. I also plan on implementing a complete graph, but I have not yet had the time to get the values right. With enough tweaking, you could easily make this graph into a MDS style one.**

****



**The single link clustering method:**