Brandon Carr
CSE 473
Homework 3 Report

Part I: Disparity Estimation Using Block Matching

Implementation:

In this version of stereo image disparity calculation, we are using a technique known as block matching. Block matching will allow us to match pixels in stereo image sets by examining their respective neighborhoods. The algorithm itself is quite simple. We cycle through one image's pixels, scanning the other image's corresponding row to find that same point in the "world." Upon each iteration, the SSD is taken for the window of pixels around the current pixel and compared to the current lowest SSD entry. The pixels with the lowest SSDs between them are matches, in most cases. The distance between the matched pixels is updated in their corresponding disparity map, where the intensity of every pixel of the map is that pixel's disparity. We must run this algorithm twice, one for each side, in order to obtain both maps. This is done in "Block_Matching.m".

There are some exceptions where pixels can be incorrectly matched due to occlusion. This occurs when a point in the world is visible in one image, but not in the other. Because of this, the visible pixel is matched to an unrelated pixel in the other image. We can remove these spurious matches via consistency checking. This is done by tracing each pixel in the disparity maps and trying to find their match in the other disparity map. For example, pixel (4,5) in the left map has a disparity value of 50. If we were to check pixel (4,55) in the right map, the disparity value should also be 50. This same thing can be done in reverse. This process is carried out in "Consistency.m".

Part II: Dynamic Programming

Implementation:

In this version of disparity calculation, we construct something called a “cost matrix” from our images. The formation of the cost matrix takes scanlines from each image in the stereo pair and forms a “layer” of a three dimensional matrix. These layers are made by starting with an empty matrix nxn with the (1,1) value set to zero. The matrix is formed by considering the “cost” of each pixel combination of the scanline and filling in each element with their difference between the images. Due to ordering constraint, the matrix will be constructed in such a way that the optimal “path” through it will return the disparities of the scanline values.

After the cost matrix is calculated, the algorithm traces each layer from the bottom right to the top left, and tries to find the path tracing over the smallest values. After this path is made, the algorithm backtracks through the path, calculating the disparity for each value on the path. This process is repeated as many times as there are layers in the cost matrix, and it has as many layers as how many pixels high the images are.
The benefits to this approach are great, as the accuracy and runtime are both improved over block matching.

Discussion

Matching Textureless Regions:

One observation made about the data is that the Books image set had the least amount of texture. This made the generated disparity maps quite incorrect, as they had many pixels removed during consistency checking. This is likely due to the large, textureless, white regions which cause our algorithms to incorrectly and/or prematurely match pixels between the image set. The only way we could get around this would be to use some form of filter in order to accentuate the features of the books. We could also potentially use edge/blob detection, as the books have many defined edges and corners.

Built-in MATLAB Functions:

In order to improve performance, a few built-in matlab functions were used to analyze and calculate data. Padarray() was used in block matching to avoid out of bounds errors. Size() was used as opposed to using loops to count numbers of row/column elements. Functions like zeros() and NaN() were used to quickly generate and initialize empty matrices. Nansum() adds up elements, skipping over NaN values.

Runtime Analysis:

As the size of the images increase, runtime for Block Matching increases exponentially. This is due to the double nested loops in the code. Increasing the window size also seems to have a minor effect. In a previous (failed) implementation, the window was populated using yet another for loop. This made runtime unbearably long. Fortunately, the current implementation uses so loops in populating the windows in each iteration, so it receives quite a speed boost in that regard.

A better-optimized algorithm for matching is RANSAC. This algorithm uses a somewhat random method of choosing two pixels and measuring the data between them. It has been found that generally, it is faster to choose two random points and going from there than it is to cycle through all possible points. This is especially the case when it comes to image processing, as image data can get huge very quickly.

Another algorithm was mentioned in a paper by Takeo Kanade from Carnegie Mellon University that worked similarly to Block-Matching, but used a dynamically shrinking/growing window matrix. This would greatly improve runtime, as the window would get smaller when it needed to with little overhead.

Part III: View Interpolation

Implementation:

This simple approach uses the provided ground-truth disparity maps in order to shift the images on a pixel-wise level. The trick here, however, is to multiply the disparity values by 0.5 so that they only shift halfway. This approach tends to make images with holes, so we just fill them up using the other view and replacing the NaN values with values shifted in the other direction.

Calculated MSE : 37.2135

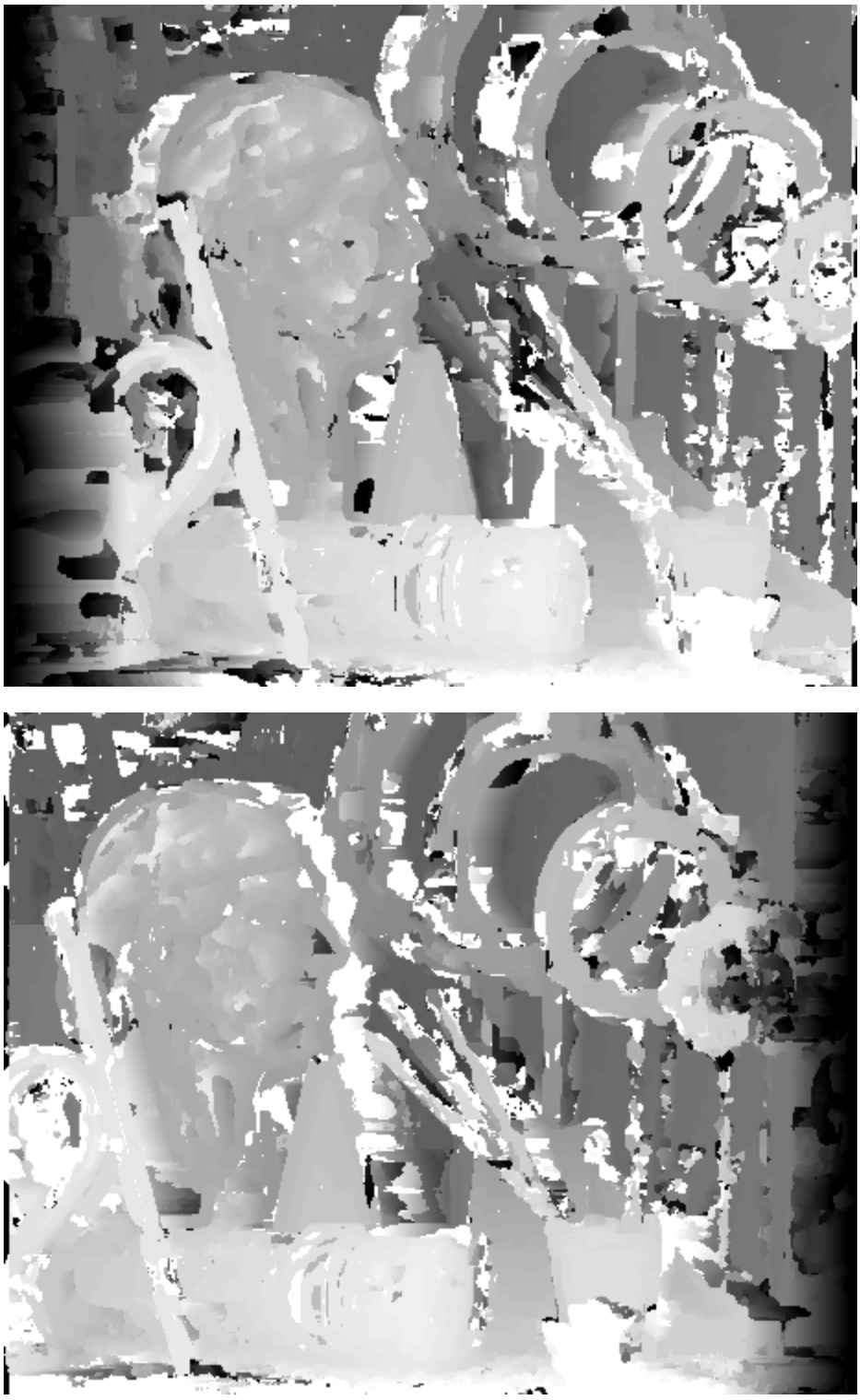
Here are the results for the Block-Matching portion of the assignment:

Original Data Image Pair:

3x3

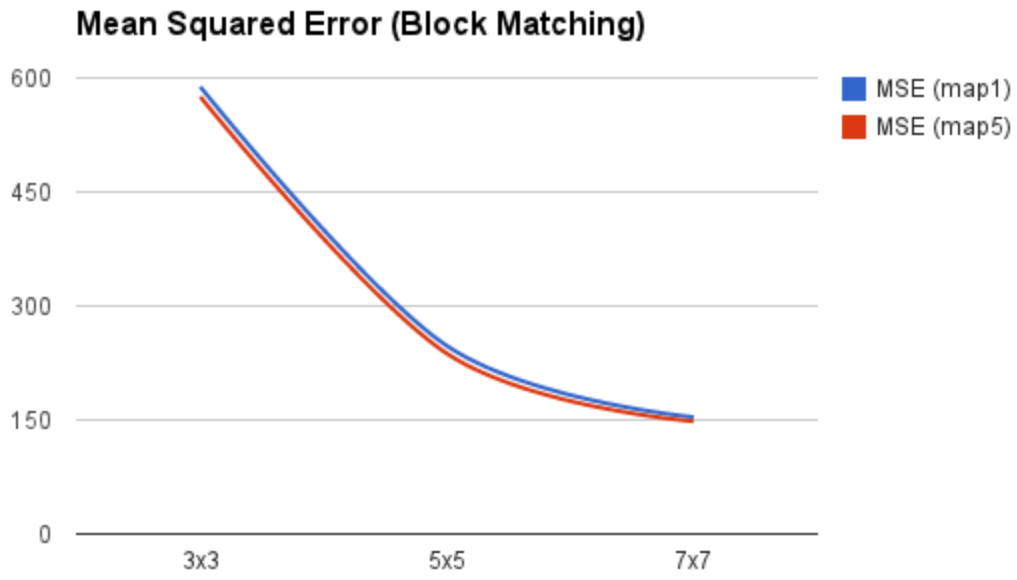
5x5

7x7

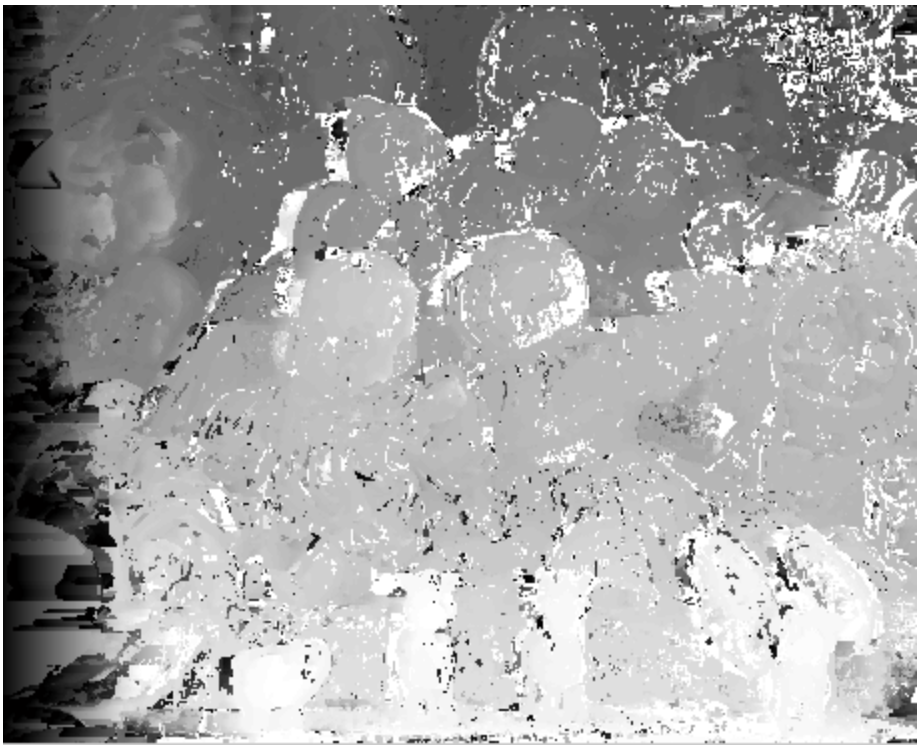


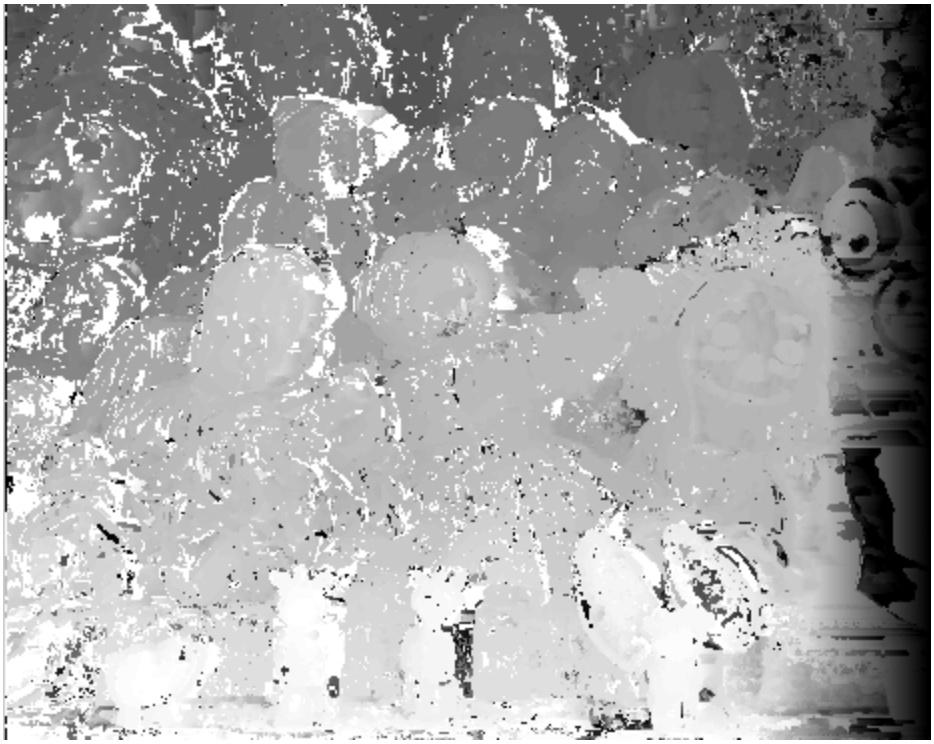
MSE Results for Original Image Pair

Window Size	3x3	5x5	7x7
MSE (map1)	588.4725	247.4985	153.712
MSE (map5)	575.2827	237.3034	148.2167



Dolls:
3x3

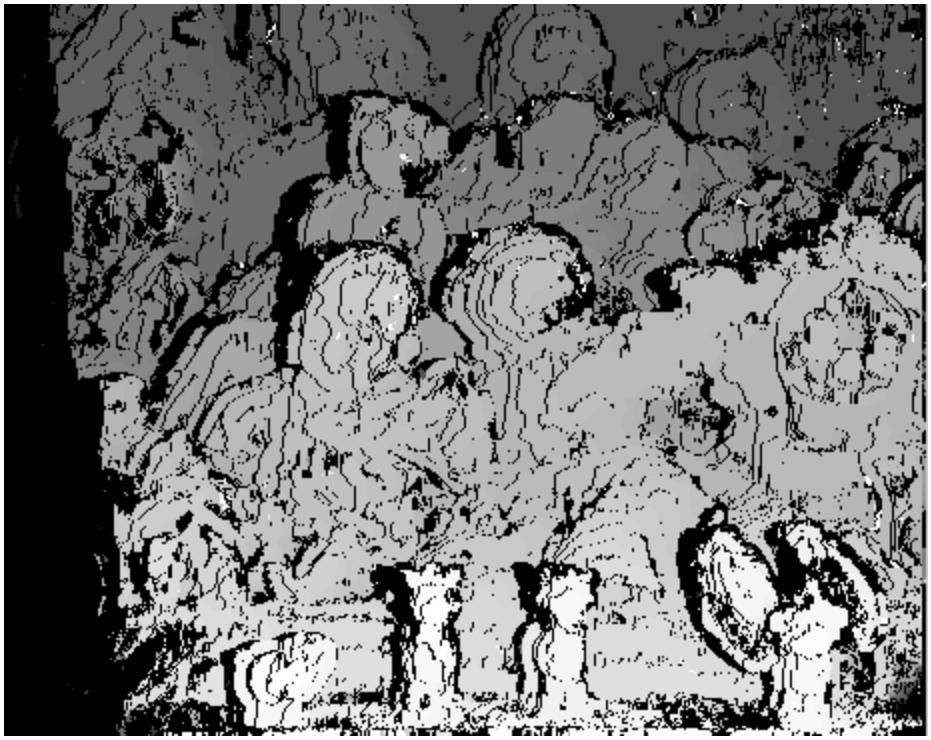


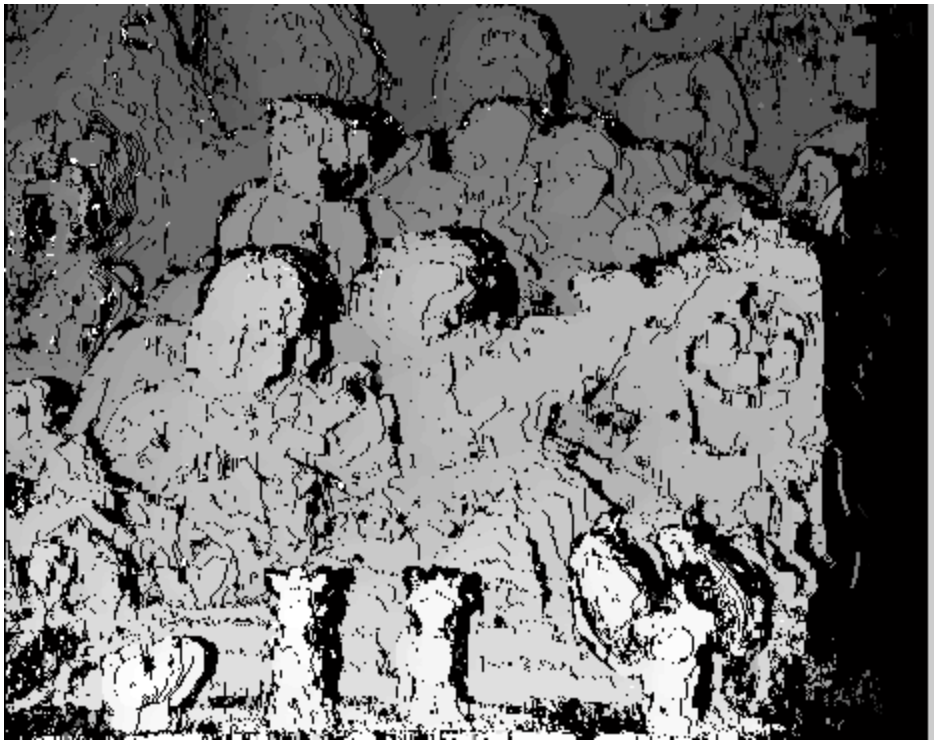


(Before consistency check, to save space I will only include maps after consistency checks from here)

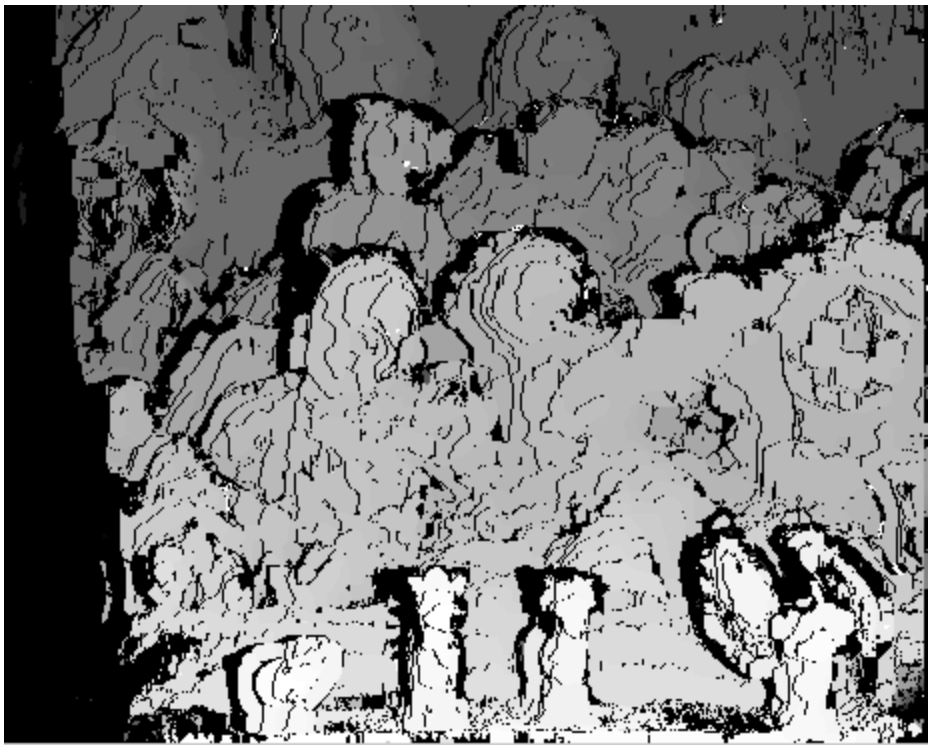
3x3 continued

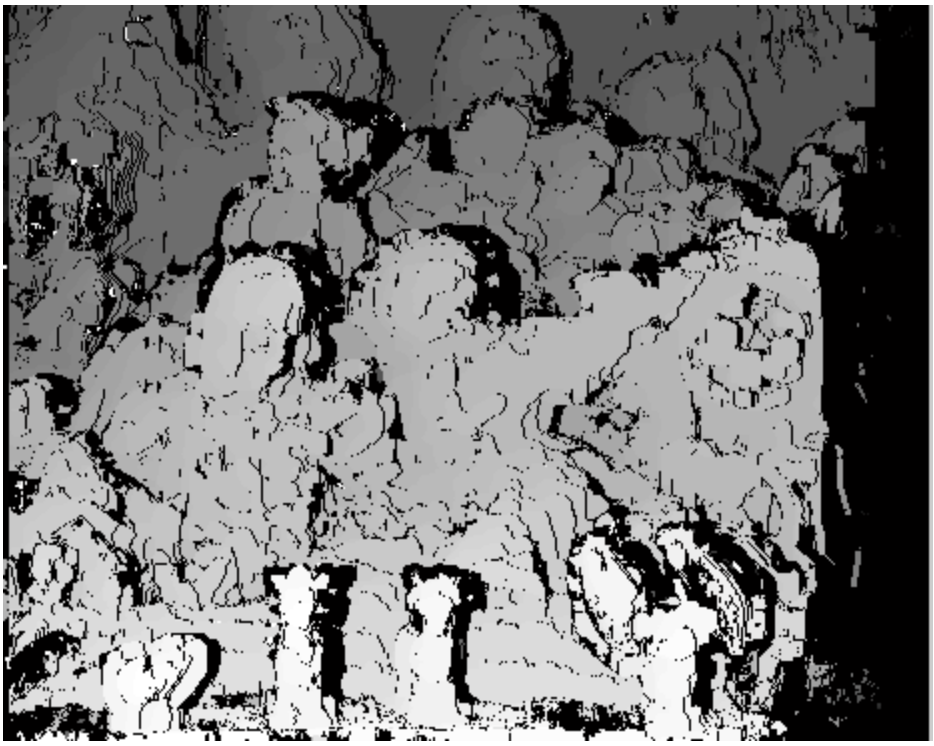
5x5





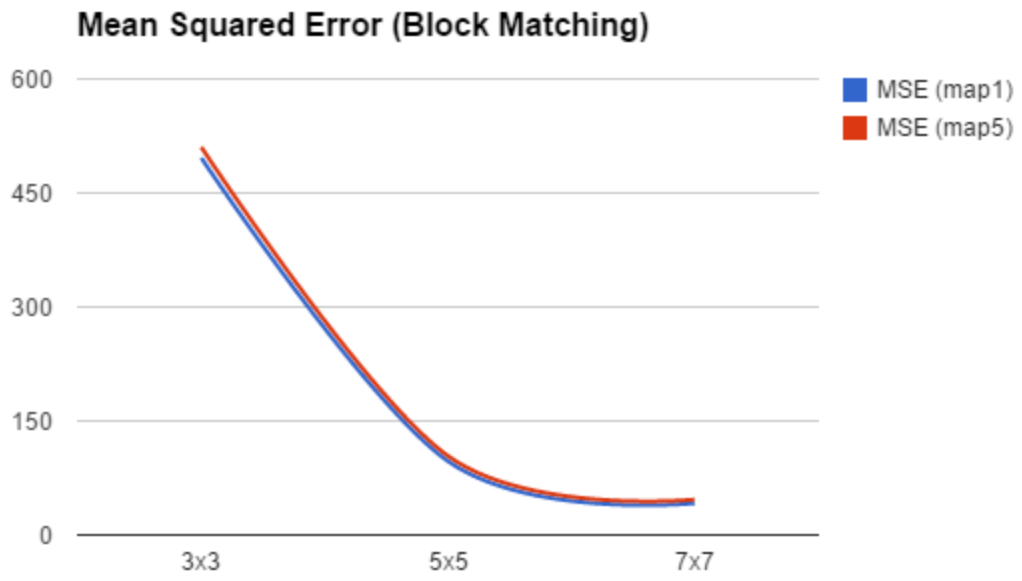
7x7



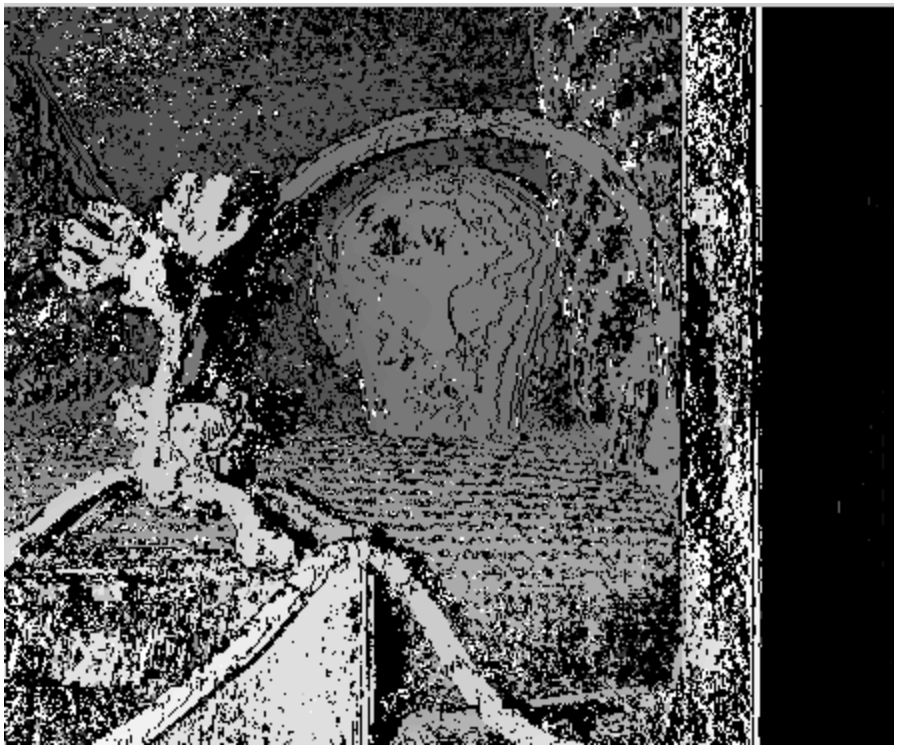


MSE Results for Doll Images

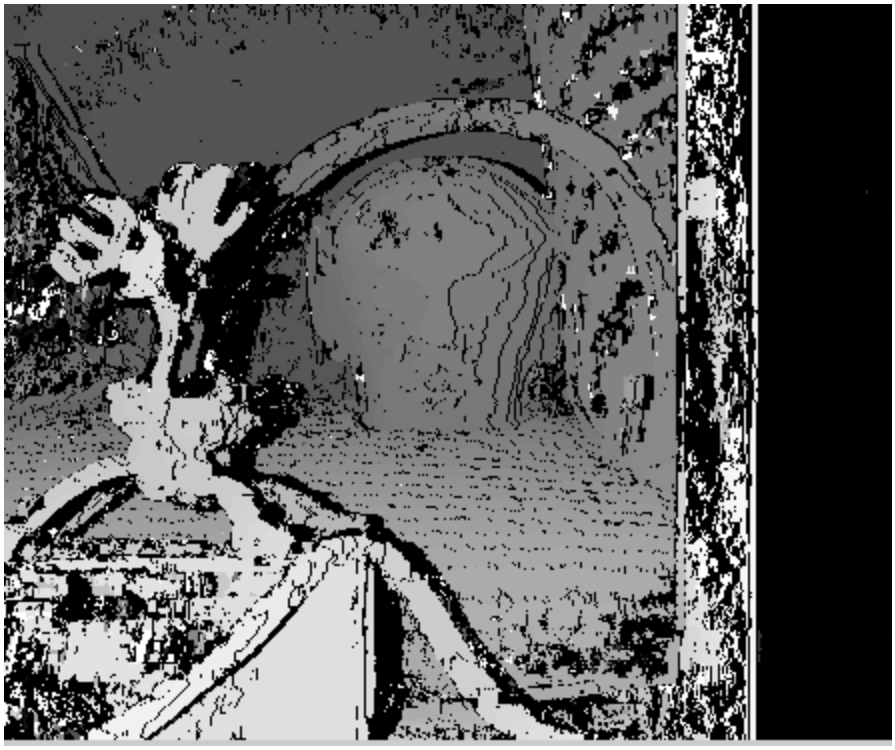
Window Size	3x3	5x5	7x7
MSE (map1)	496.0764	97.0512	41.8047
MSE (map5)	510.3954	103.9687	46.6172



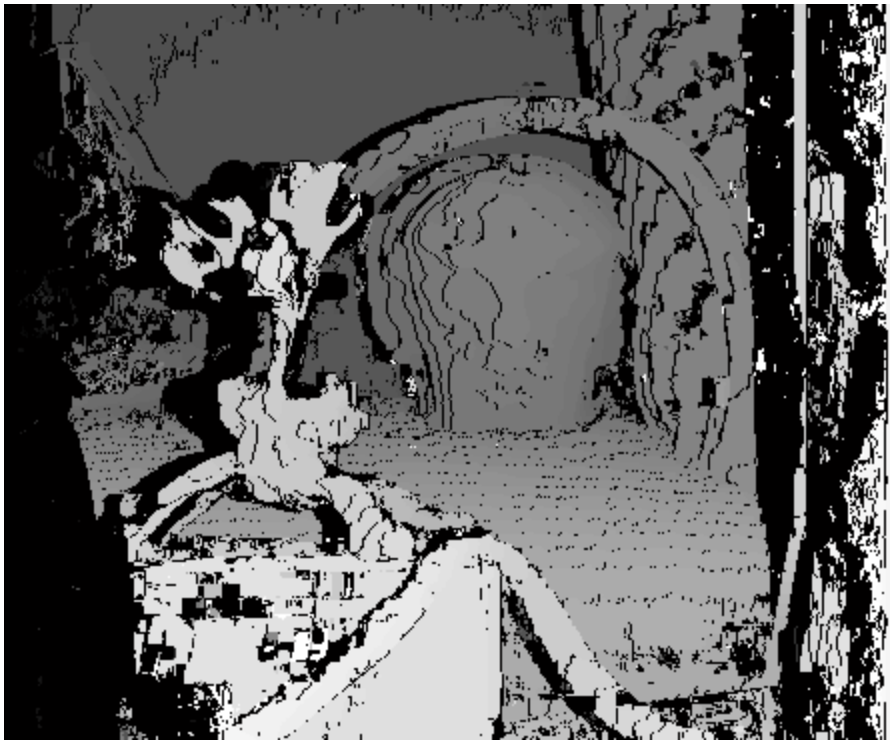
Reindeer Images
3x3

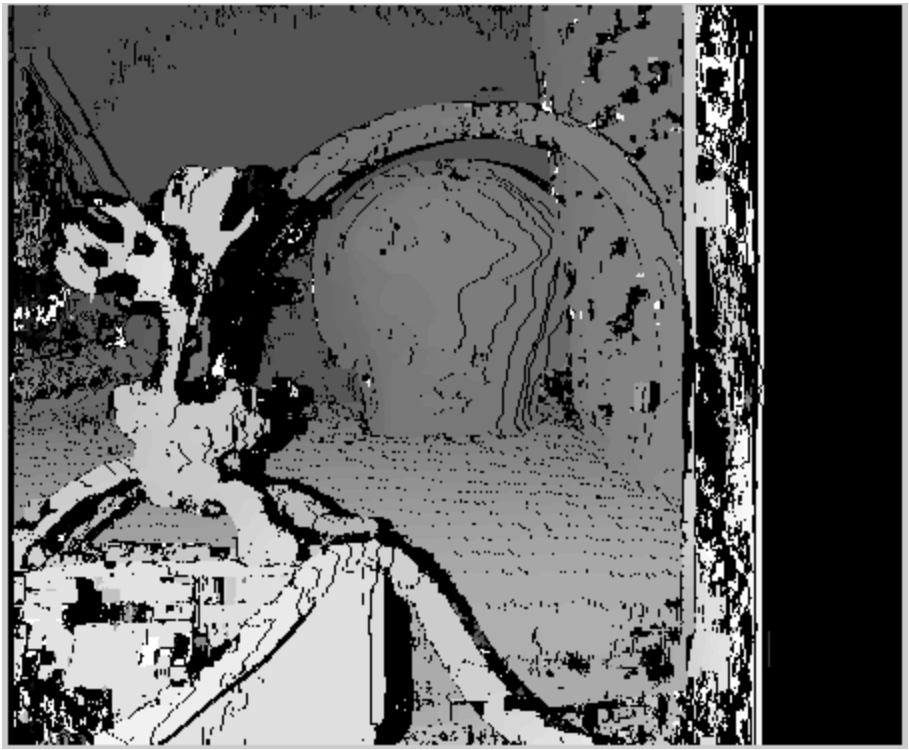


5x5



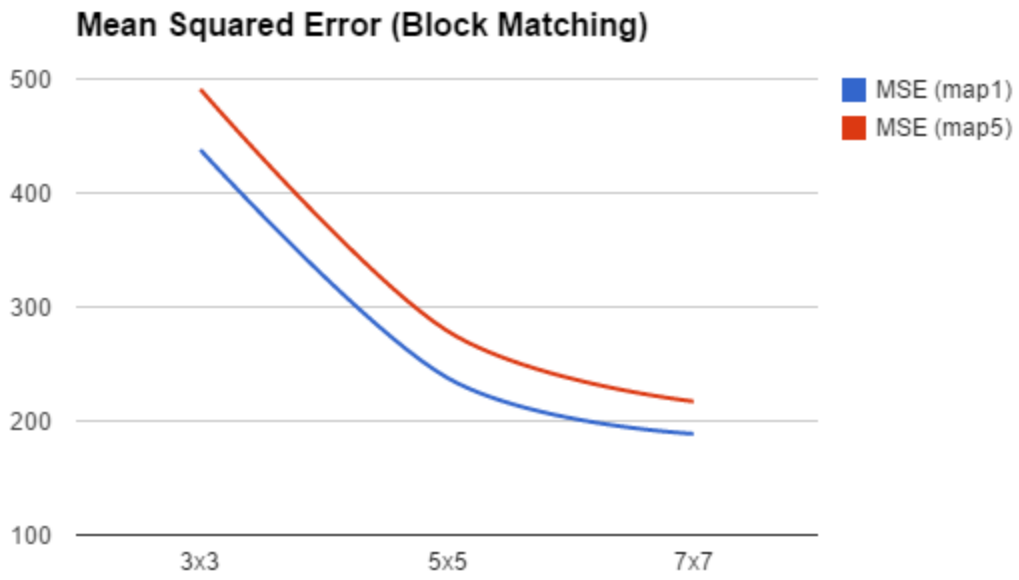
7x7



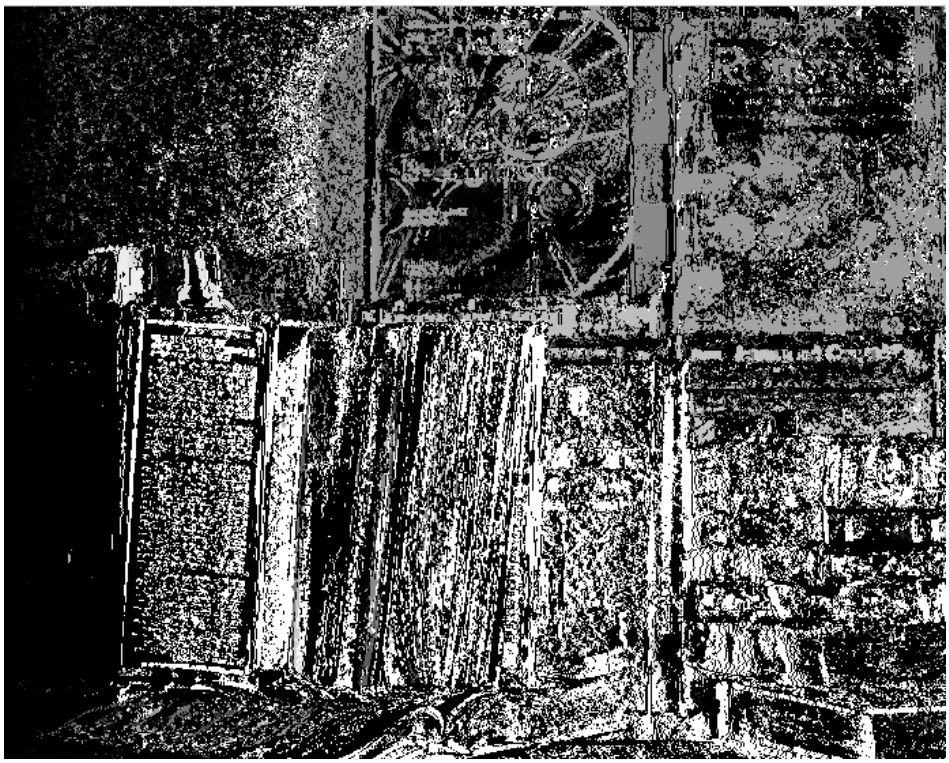


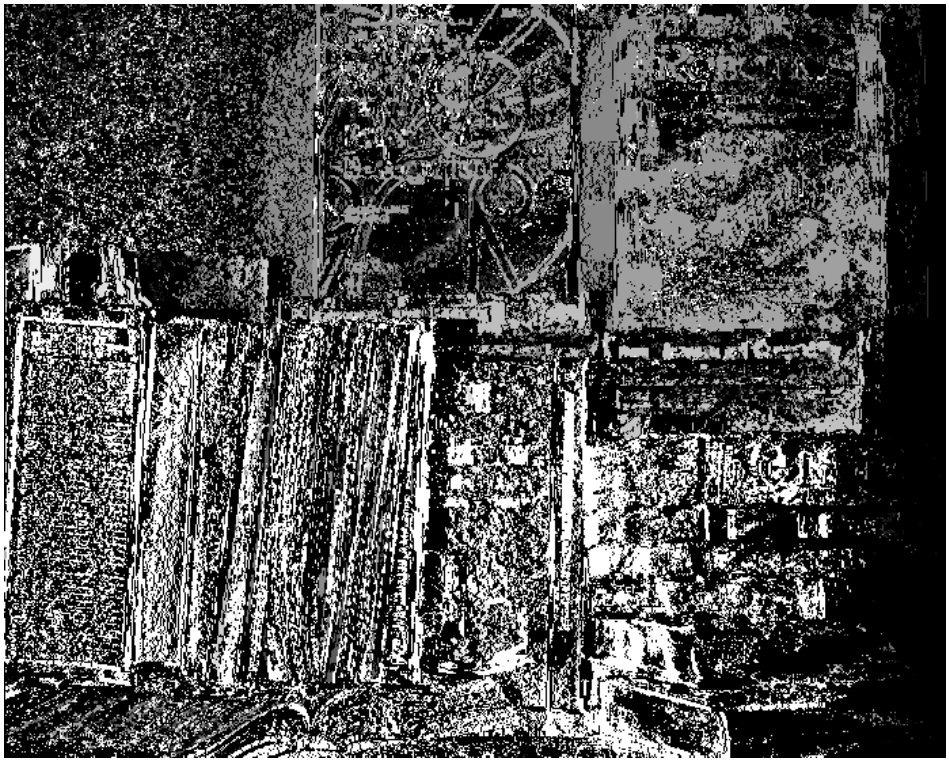
MSE Results for Reindeer Images

Window Size	3x3	5x5	7x7
MSE (map1)	437.9312	238.0566	188.7529
MSE (map5)	491.0369	279.1891	217.1534



Book Images
3x3





Credits:

Papers from Piazza Posts:

http://raycast.org/powerup/publications/stereo.cvpr.pdf

http://www.cs.umd.edu/~djacobs/CMSC426/Slides/stereo_algo.pdf