

Trabalho de Implementação 2

Discipline: CMP197 - Introdução à Visão Computacional

Professor:

- Cláudio Jung

Authors:

- Lucas Nedel Kirsten, 230262
- Diego Severo Jardim, 319862

Importing the necessary libraries

```
In [1]:  
from time import time  
import numpy as np  
  
import matplotlib.pyplot as plt  
from skimage.io import imread  
  
from utils.block_matching import compute_ssd, compute_aggregation  
from utils.plot_utils import plot_images
```

Setting global parameters

```
In [2]: max_disp_steps = 50 # maximum disparity to consider  
  
window_sizes = [1, 3, 5, 7, 11]  
ssd_window_sizes = [1, 3, 5, 7, 11]  
  
penalties = [10, 100, 1000]  
  
# teddy image path  
teddy_left_img_path = "../data/teddy/im2.png"  
teddy_right_img_path = "../data/teddy/im6.png"  
  
cones_left_img_path = "../data/cones/im2.png"  
cones_right_img_path = "../data/cones/im6.png"  
  
# open images  
teddy_left = imread(teddy_left_img_path)  
teddy_right = imread(teddy_right_img_path)  
  
cones_left = imread(cones_left_img_path)  
cones_right = imread(cones_right_img_path)  
  
images = [(teddy_left, teddy_right), (cones_left, cones_right)]
```

Task 1

Task 1A

Computing Block Matching using SSD

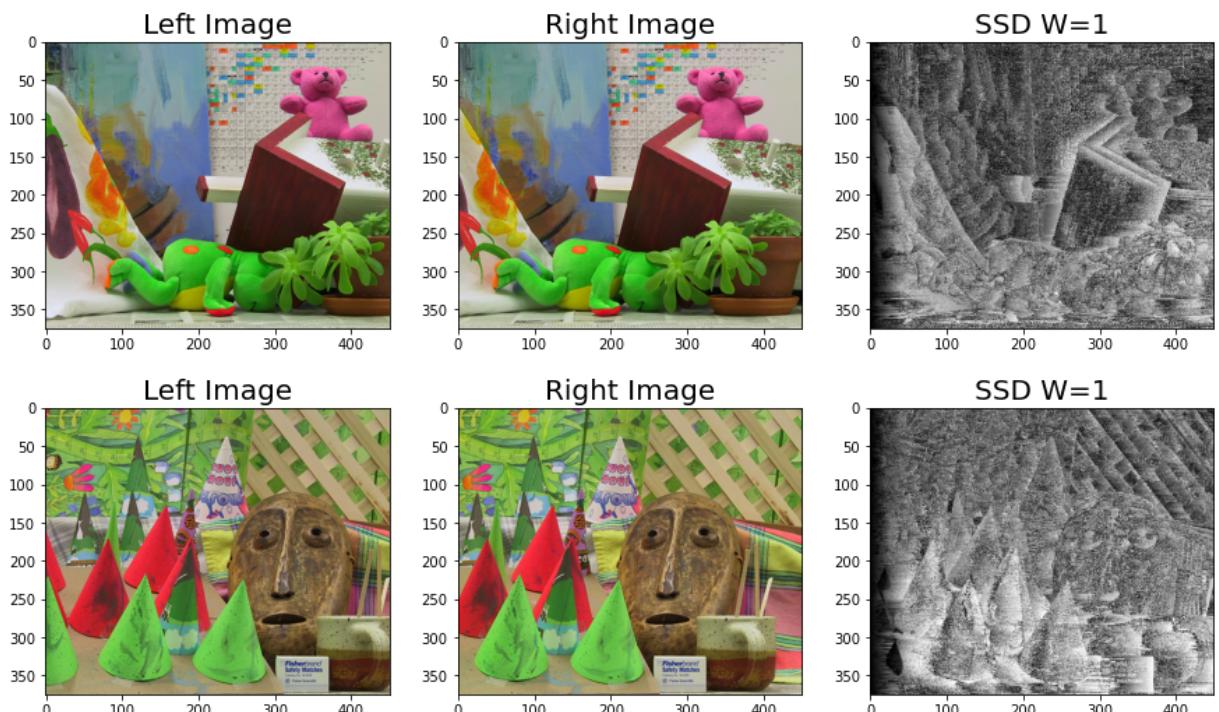
```
In [3]: # define window size
window_size = 1
```

```
In [4]: for (left_img, right_img) in images:
    init = time()
    costs0 = compute_ssd(left_image=left_img,
                          right_image=right_img,
                          disparities=max_disp_steps,
                          window_size=window_size,
                          apply_dist=False)
    min_cost = np.mean(np.argmin(costs0, axis=-1), axis=-1)

    print(f'SSD time: {(time() - init):.2f} s')

    plot_images(imageL=left_img,
                imageR=right_img,
                disp_map1=min_cost,
                disp_map1_title=f"SSD W={window_size}")
```

SSD time: 0.56 s
SSD time: 0.51 s



Task 1B

Computing Block Matching using SSD + Penalty value

```
In [5]: # define window size and penalty
window_size = 1
penalty = 100
```

```
In [6]: for (left_img, right_img) in images:
    init = time()
    costs0 = compute_ssd(left_image=left_img,
                          right_image=right_img,
                          disparities=max_disp_steps,
                          window_size=window_size,
                          apply_dist=True,
```

```

penalty=penalty)
min_cost = np.mean(np.argmin(costs0, axis=-1), axis=-1)

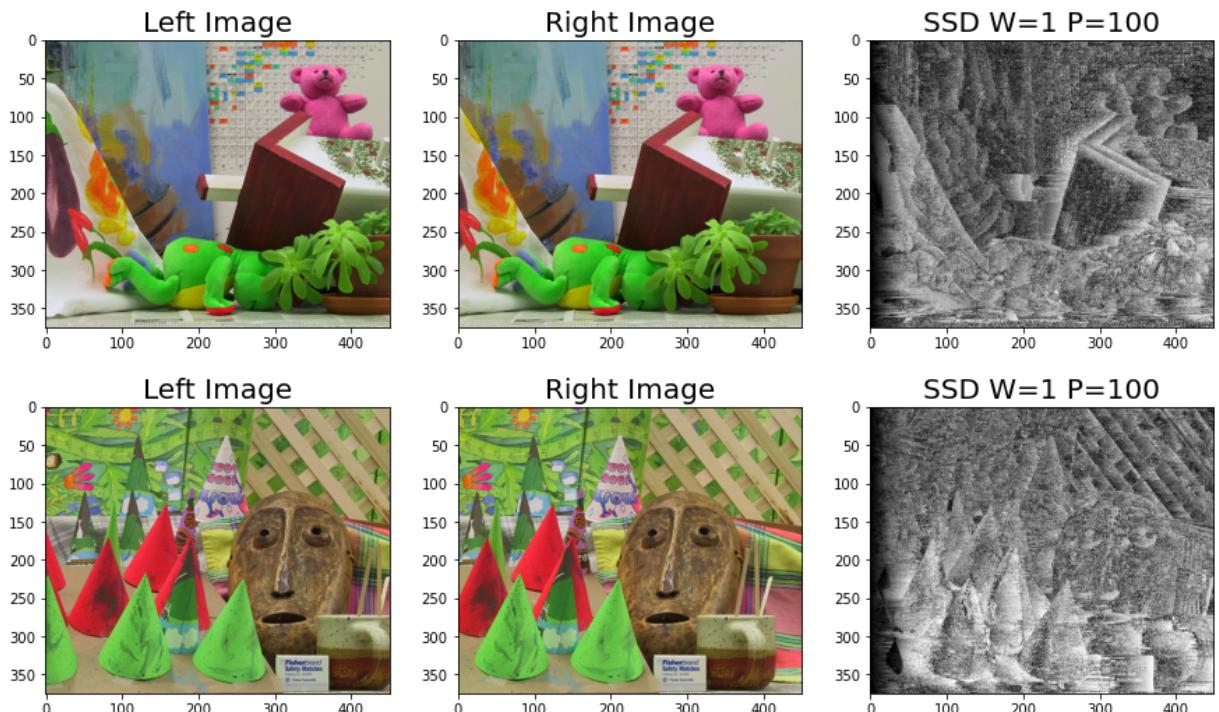
print(f'SSD time: {(time() - init):.2f} s')

plot_images(imageL=left_img,
            imageR=right_img,
            disp_mapl=min_cost,
            disp_mapl_title=f"SSD W={window_size} P={penalty}")

```

SSD time: 0.48 s

SSD time: 0.48 s



Looping over windows and penalties

(Be aware: this can take awhile and consume a lot of memory!)

In [7]:

```
# define if to use teddy or cones images
left_image = cones_left
right_image = cones_right
```

In [8]:

```
init = time()

fig = plt.figure(figsize=(15, 15))
ax1 = fig.add_subplot(1, 2, 1)
ax1.set_title("Left Image", size=20)
ax1.imshow(left_img)
ax2 = fig.add_subplot(1, 2, 2)
ax2.set_title("Right Image", size=20)
ax2.imshow(right_img)
plt.show()

cols = len(penalties)+1
for window_size in window_sizes:
    costs0 = compute_ssd(left_image=left_img,
                          right_image=right_img,
                          disparities=max_disp_steps,
                          window_size=window_size,
                          apply_dist=False)
```

```

min_cost = np.mean(np.argmin(costs0, axis=-1), axis=-1)

plt.figure(figsize=(25, 10))
plt.subplot(1,cols,1)
plt.title(f"SSD W={window_size}", size=20)
plt.imshow(min_cost, cmap='gray')

for i,penalty in enumerate(penalties):
    costs0_pen = compute_ssd(left_image=left_img,
                             right_image=right_img,
                             disparities=max_disp_steps,
                             window_size=window_size,
                             apply_dist=True,
                             penalty=penalty)
    min_cost_pen = np.mean(np.argmin(costs0_pen, axis=-1), axis=-1)

    plt.subplot(1,cols,i+2)
    plt.title(f"SSD W={window_size} P={penalty}", size=20)
    plt.imshow(min_cost_pen, cmap='gray')

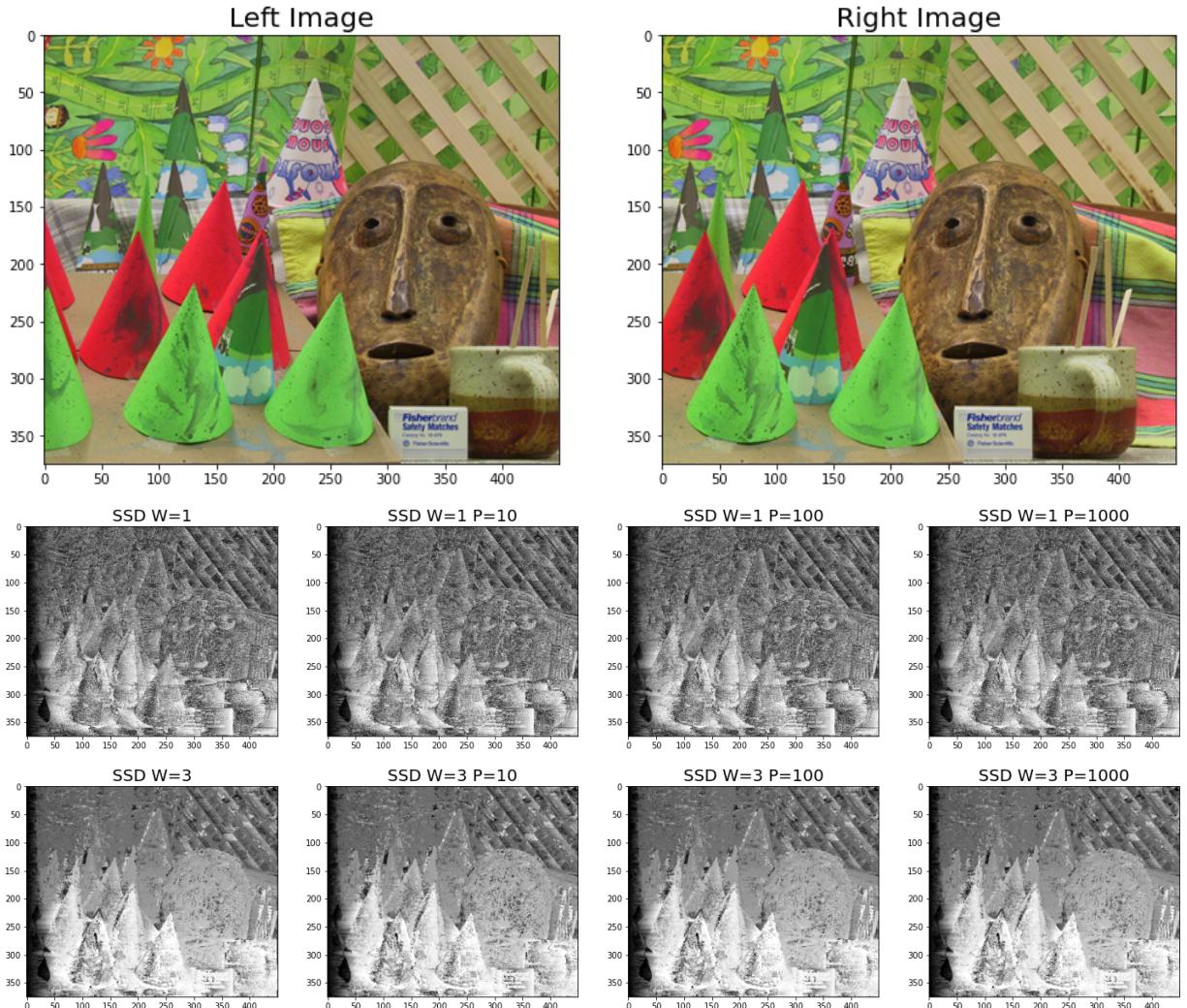
del costs0_pen, min_cost_pen

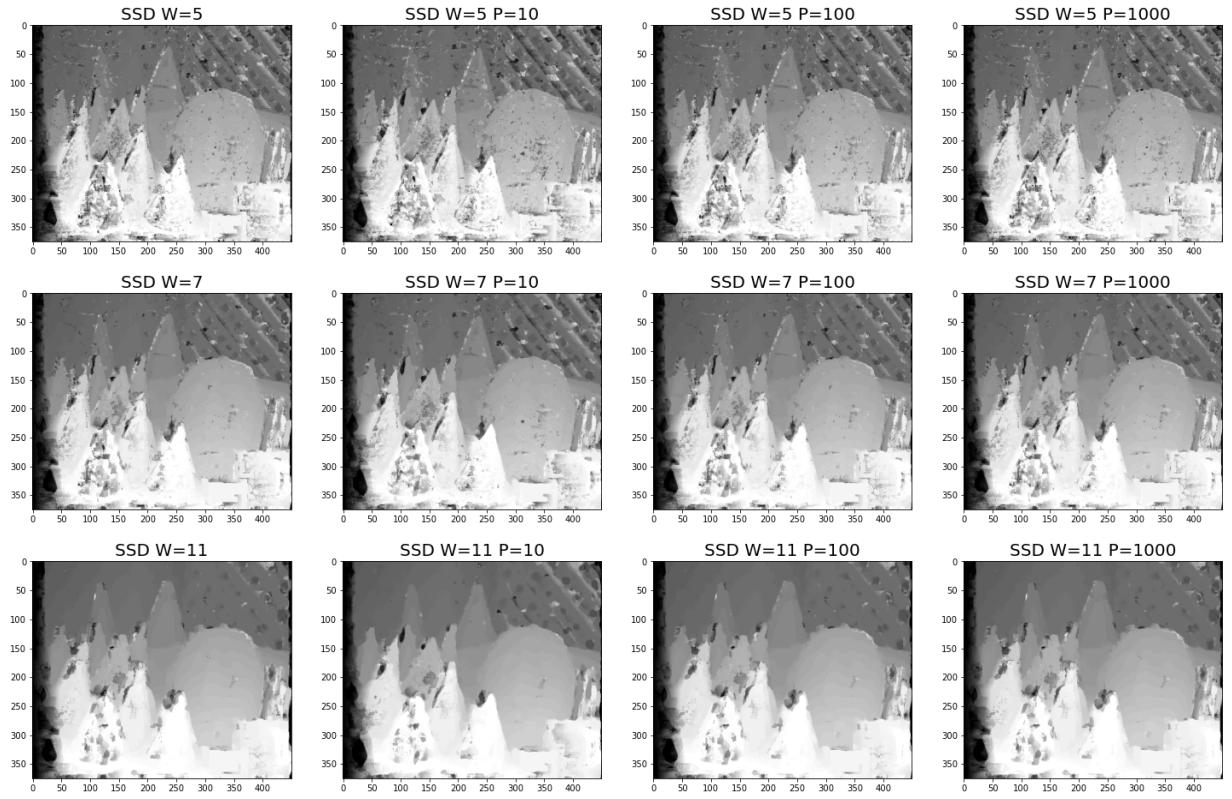
plt.show()

del costs0, min_cost

print(f'Elapsed time: {(time() - init):.2f} s')

```





Elapsed time: 133.03 s

Comments

When we run the algorithm considering only a pixel as reference (neighborhood of 1x1), there is a big room for ambiguity pixels on the block matching calculation. Analyzing the outcomes from tasks 1A and 1B, we can see visually this problem. Because of the ambiguity of pixels between both stereo images, the disparity map obtained is very noisy.

When we apply the match windows approach we get better results. So, as we can see in the previous images plots, as we increase the window sizes until a certain neighborhood size we get smoother disparity maps with fewer noise. However, as the window sizes increases we end up losing fine details, such as object boundaries.

Lastly, another topic to be consider is about the robust distance metric (also called penalty) which we applied in the SSD calculation of the Task 1B. We test different penalty values and we conclude that it seems to be difficult to get the right setup for it. So, in our results, as we increases the penalty value, we got fewer matching mistakes (black pixels in the previous images) on the disparity maps comparing the plots with smaller penalty values or even without penalty. But on the other hand we also end up losing fine details, such as object boundaries.

Task 2

Task 2A

Computing Block Matching using SSD + Mean Aggregation

In [9]:

```
# define windows values
ssd_window = 1 # for the vanilla ssd
agg_window = 7 # for the aggregation
```

In [10]:

```

for left_img,right_img in images:
    init = time()

    costs0 = compute_ssd(left_image=left_img,
                          right_image=right_img,
                          disparities=max_disp_steps,
                          window_size=ssd_window,
                          apply_dist=False)
    min_cost = np.mean(np.argmin(costs0, axis=-1), axis=-1)

    mean_agg = compute_aggregation(costs0=costs0, window_size=agg_window, mode='mean')
    min_cost_mean_agg = np.mean(np.argmin(mean_agg, axis=-1), axis=-1)

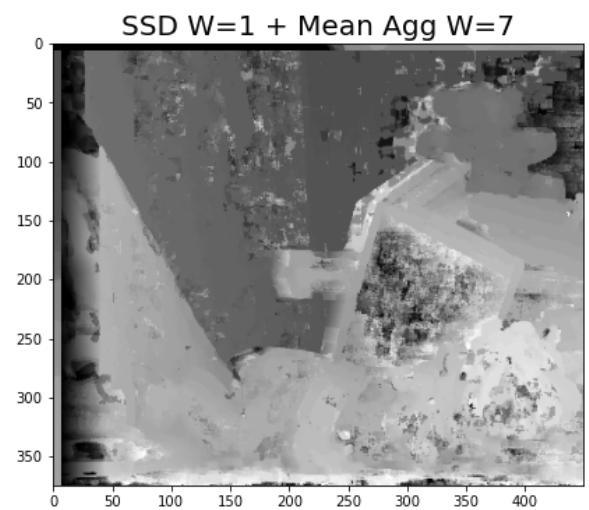
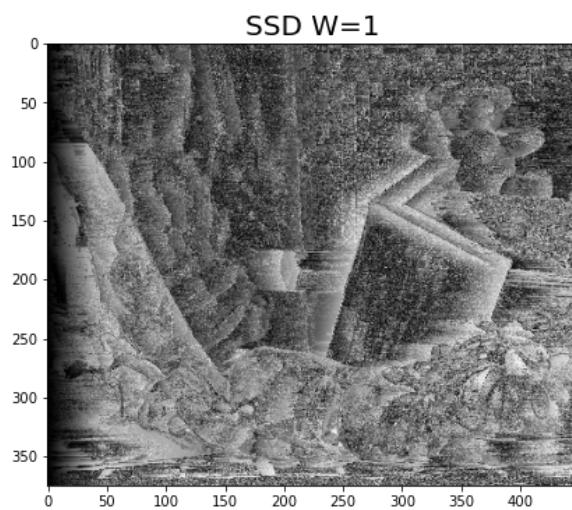
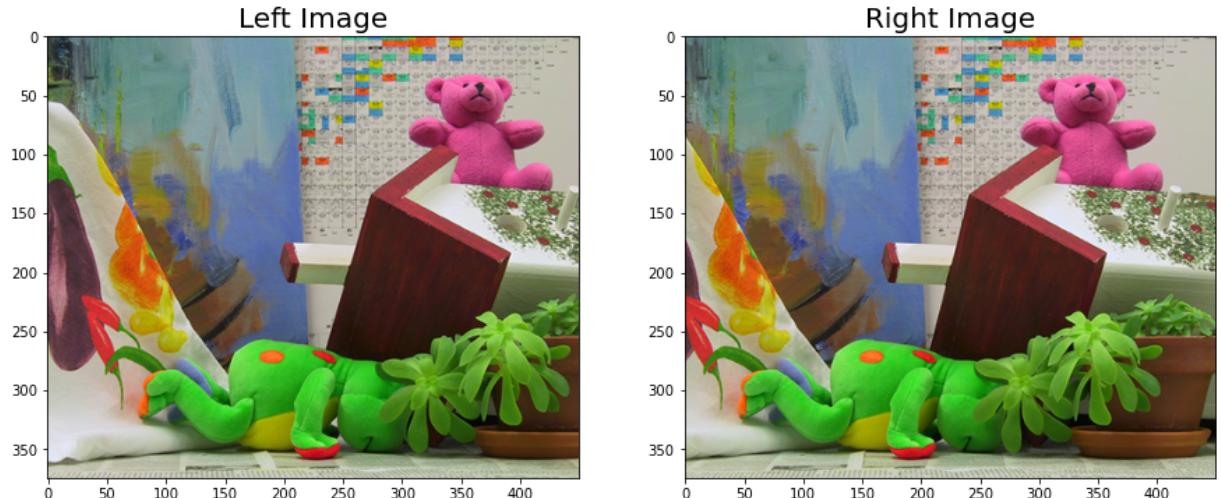
    print(f'Mean Aggregation time: {(time() - init):.2f} s')

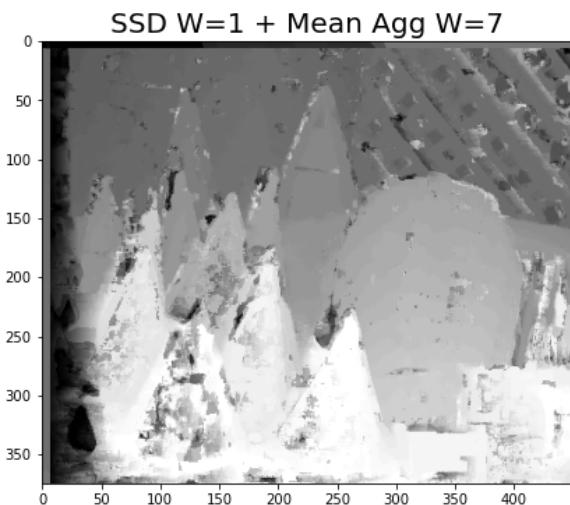
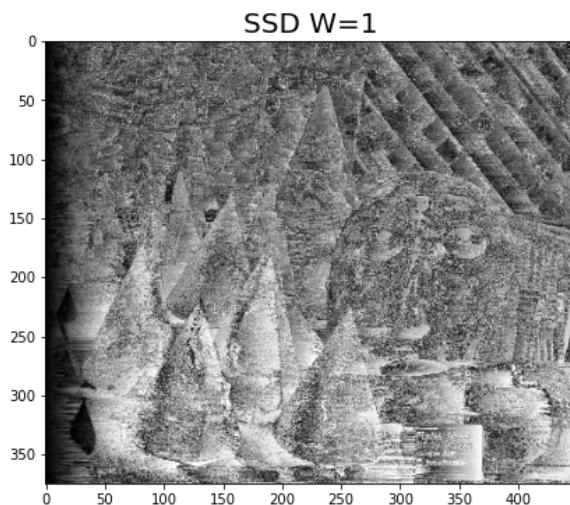
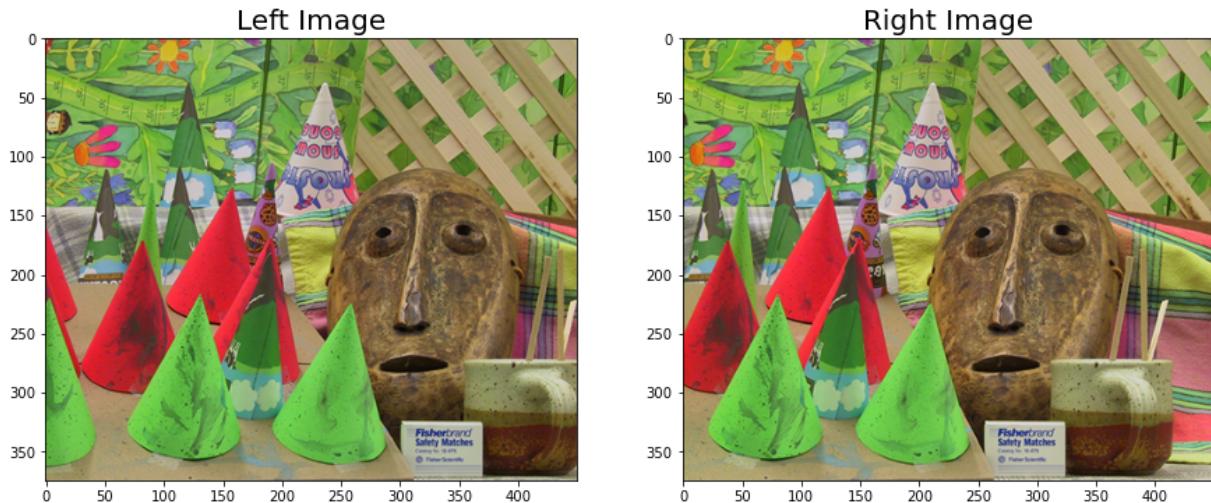
    plot_images(imageL=left_img,
                imageR=right_img,
                disp_map1=min_cost,
                disp_map2=min_cost_mean_agg,
                disp_map1_title=f"SSD W={ssd_window}",
                disp_map2_title=f"SSD W={ssd_window} + Mean Agg W={agg_window}")

```

Mean Aggregation time: 2.73 s

Mean Aggregation time: 2.46 s





Computing Block Matching using SSD + Penalty value + Mean Aggregation

In [11]:

```
# define values
ssd_window = 1 # window size for the vanilla ssd
agg_window = 7 # window size for the aggregation
penalty = 100 # penalty value
```

In [12]:

```
for left_img,right_img in images:
    init = time()

    costs0 = compute_ssd(left_image=left_img,
                          right_image=right_img,
                          disparities=max_disp_steps,
                          window_size=ssd_window,
                          apply_dist=True,
                          penalty=penalty)
    min_cost = np.mean(np.argmin(costs0, axis=-1), axis=-1)

    mean_agg = compute_aggregation(costs0=costs0, window_size=agg_window, mode='mean')
    min_cost_mean_agg = np.mean(np.argmin(mean_agg, axis=-1), axis=-1)

    print(f'Mean Aggregation time: {(time() - init):.2f} s')

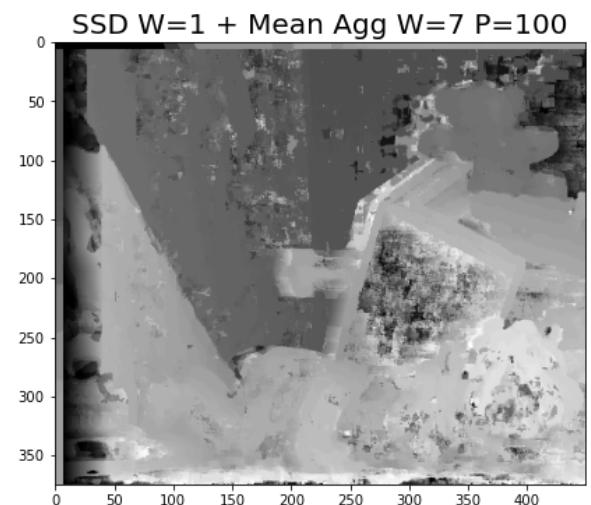
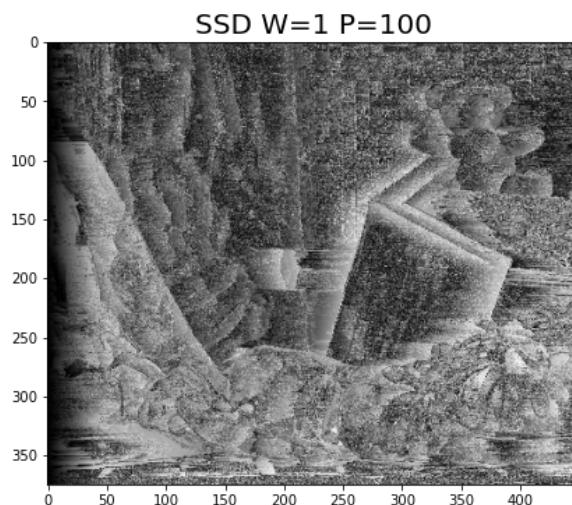
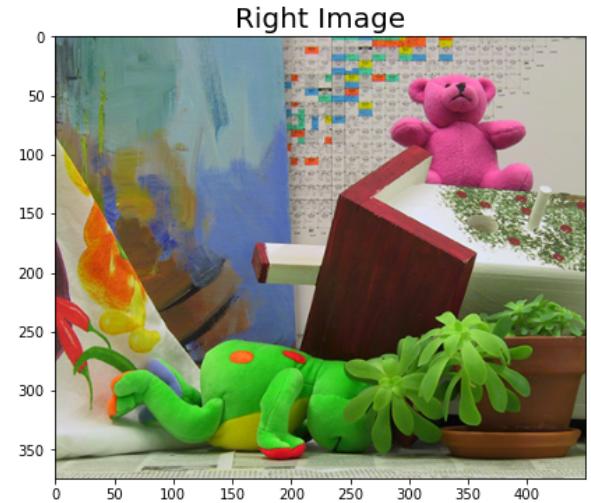
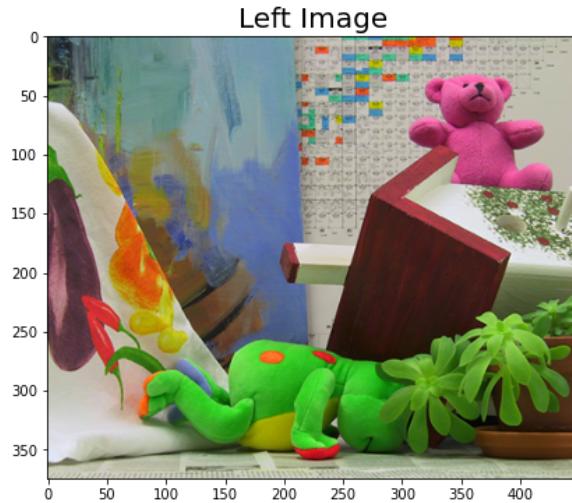
    plot_images(imageL=left_img,
                imageR=right_img,
                disp_mapL=min_cost,
```

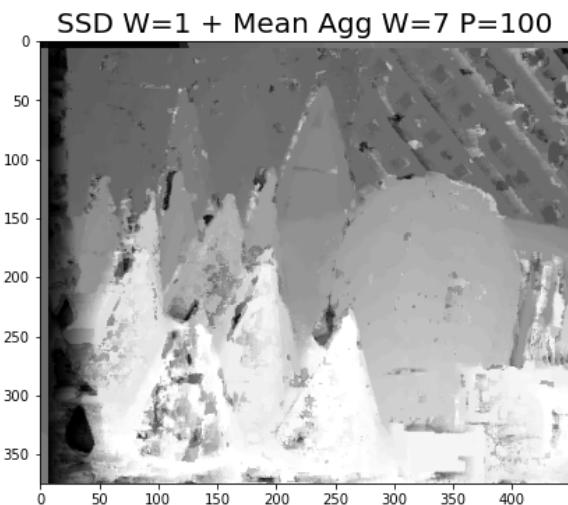
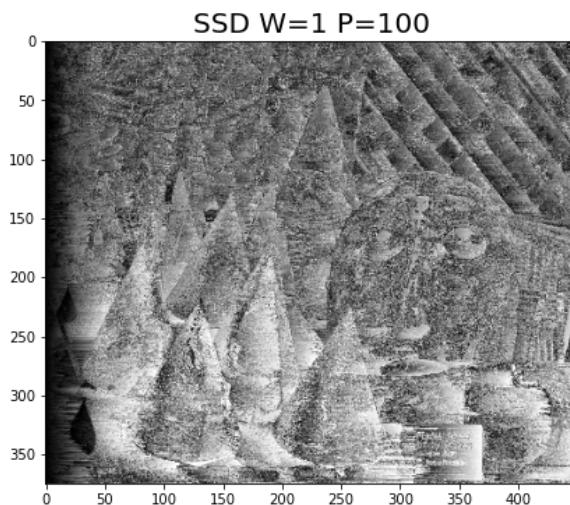
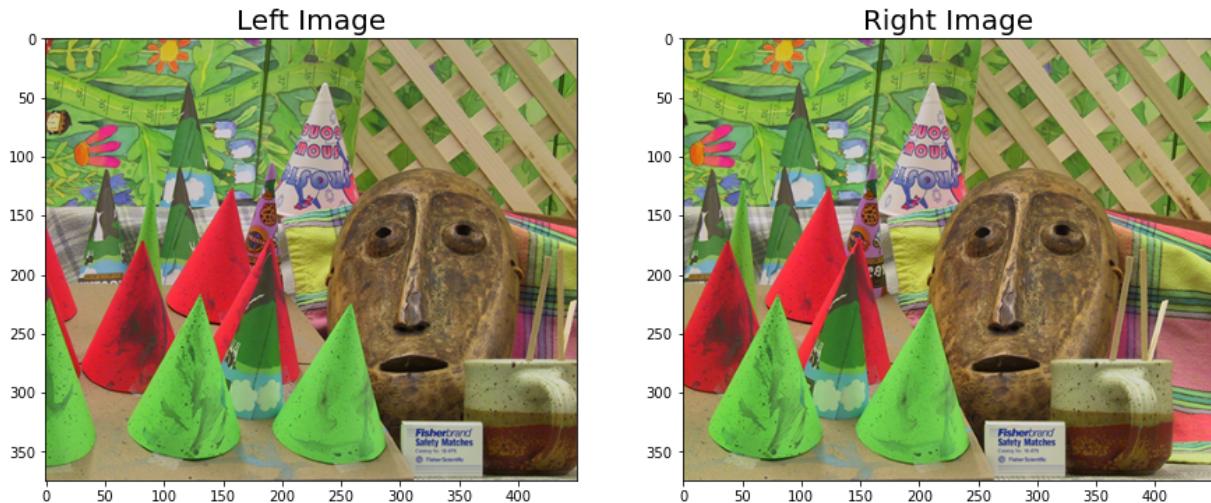
```

disp_map2=min_cost_mean_agg,
disp_map1_title=f"SSD W={ssd_window} P={penalty}",
disp_map2_title=f"SSD W={ssd_window} + Mean Agg W={agg_window}"

```

Mean Aggregation time: 2.57 s
 Mean Aggregation time: 2.65 s





Task 2B

Computing Block Matching using SSD + Median Aggregation

In [13]:

```
# define values
ssd_window = 1 # window size for the vanilla ssd
agg_window = 7 # window size for the aggregation
```

In [14]:

```
for left_img,right_img in images:
    init = time()

    costs0 = compute_ssd(left_image=left_img,
                          right_image=right_img,
                          disparities=max_disp_steps,
                          window_size=ssd_window,
                          apply_dist=False)
    min_cost = np.mean(np.argmin(costs0, axis=-1), axis=-1)

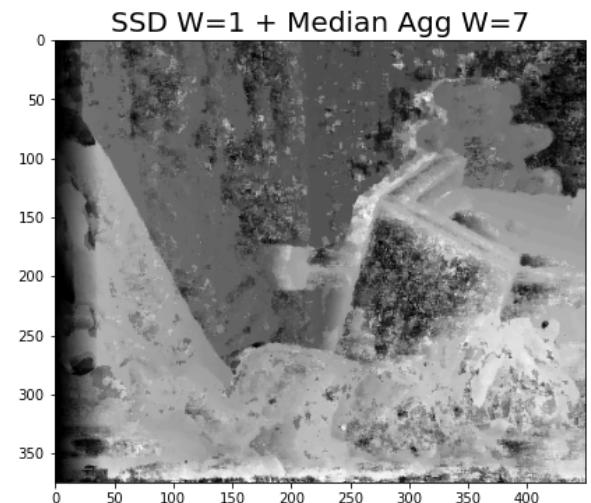
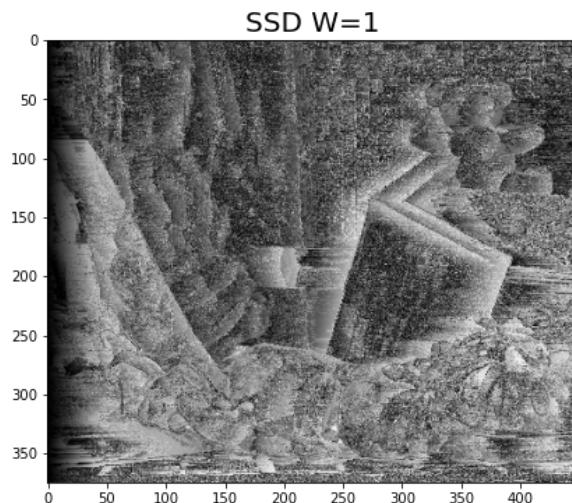
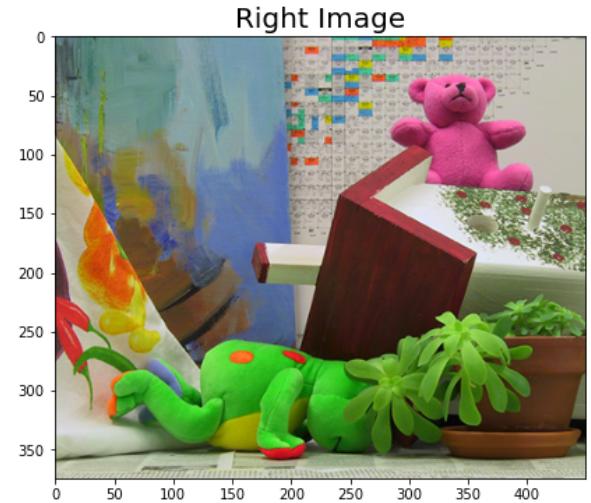
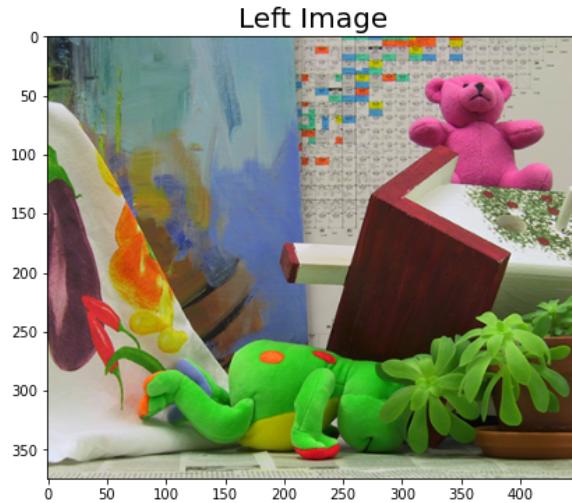
    median_agg = compute_aggregation(costs0=costs0, window_size=agg_window,
                                      min_cost_median_agg = np.mean(np.argmin(median_agg, axis=-1), axis=-1))

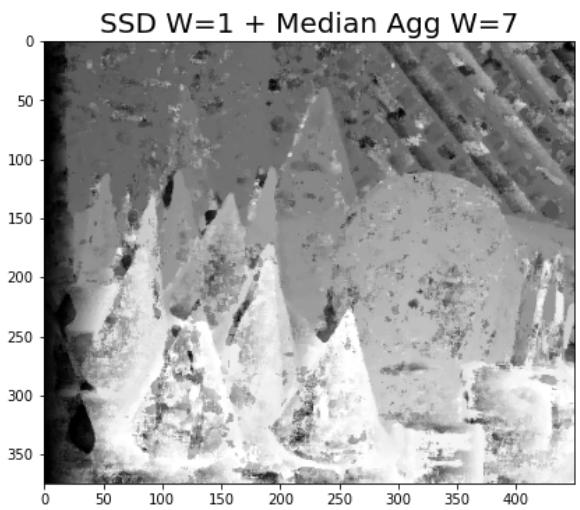
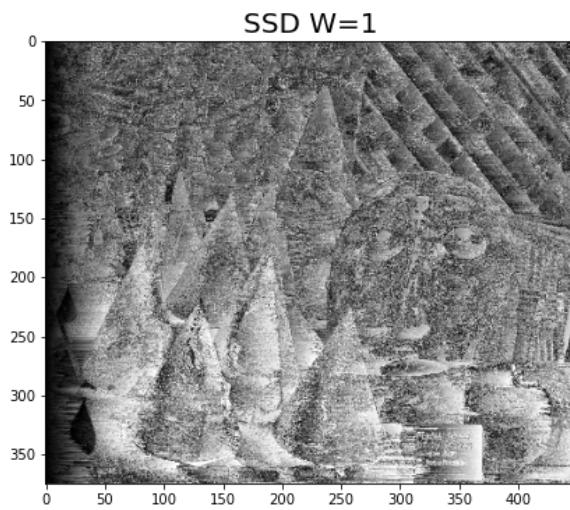
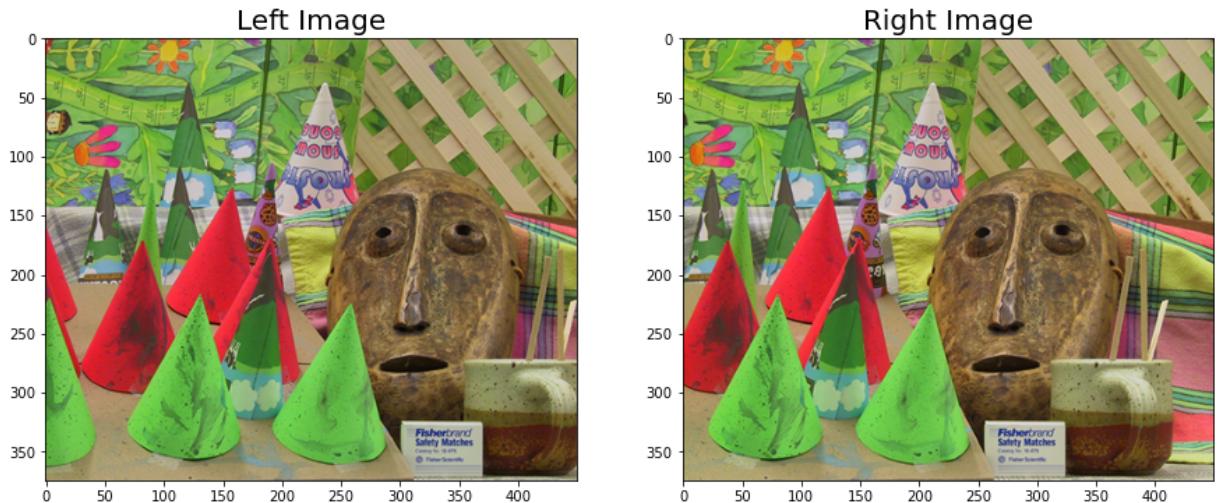
    print(f'Median Aggregation time: {(time() - init):.2f} s')

    plot_images(imageL=left_img,
                imageR=right_img,
                disp_map1=min_cost,
```

```
disp_map2=min_cost_median_agg,  
disp_map1_title=f"SSD W={ssd_window}",  
disp_map2_title=f"SSD W={ssd_window} + Median Agg W={agg_wind
```

Median Aggregation time: 21.94 s
Median Aggregation time: 23.26 s





Computing Block Matching using SSD + Penalty value + Median Aggregation

In [15]:

```
# define values
ssd_window = 1 # window size for the vanilla ssd
agg_window = 7 # window size for the aggregation
penalty = 100 # penalty value
```

In [16]:

```
for left_img,right_img in images:
    init = time()

    costs0 = compute_ssd(left_image=left_img,
                          right_image=right_img,
                          disparities=max_disp_steps,
                          window_size=ssd_window,
                          apply_dist=True,
                          penalty=penalty)
    min_cost = np.mean(np.argmin(costs0, axis=-1), axis=-1)

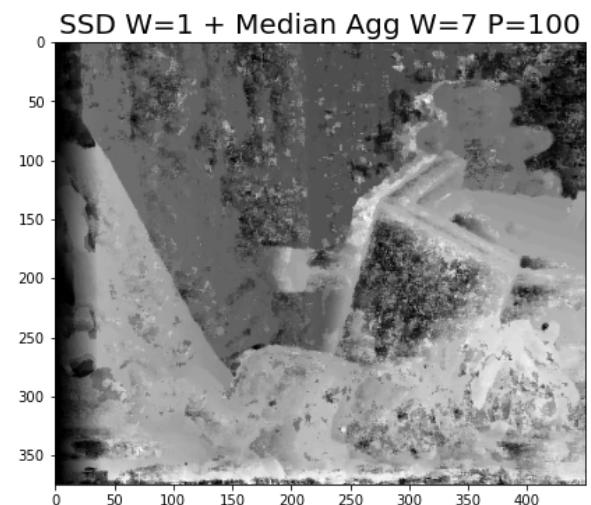
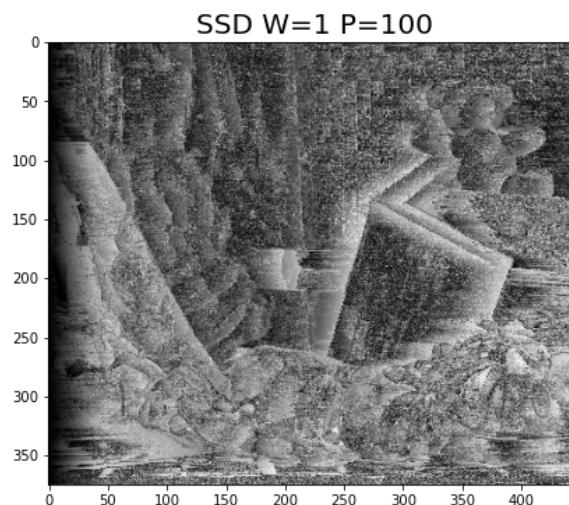
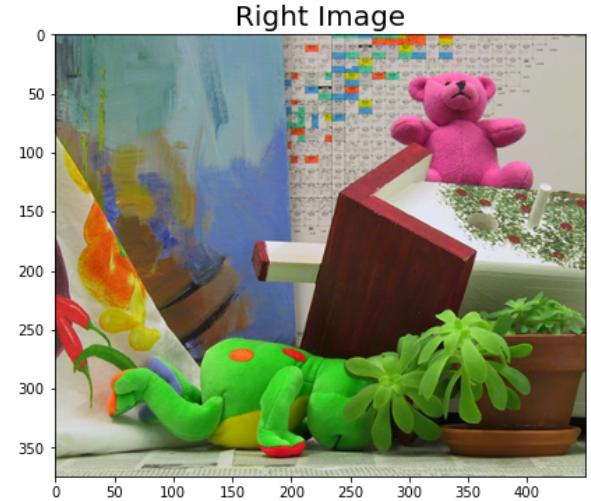
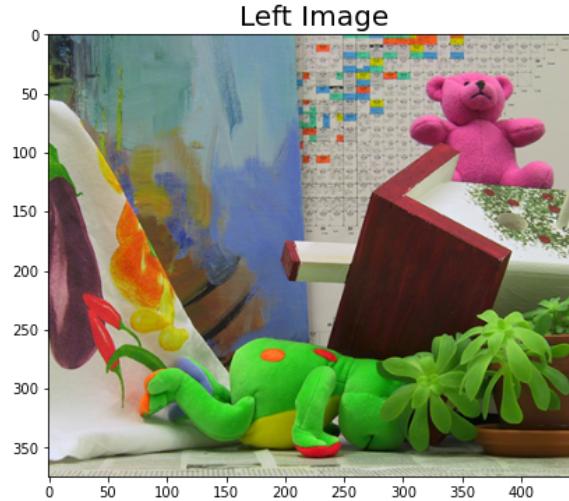
    median_agg = compute_aggregation(costs0=costs0, window_size=agg_window, n)
    min_cost_median_agg = np.mean(np.argmin(median_agg, axis=-1), axis=-1)

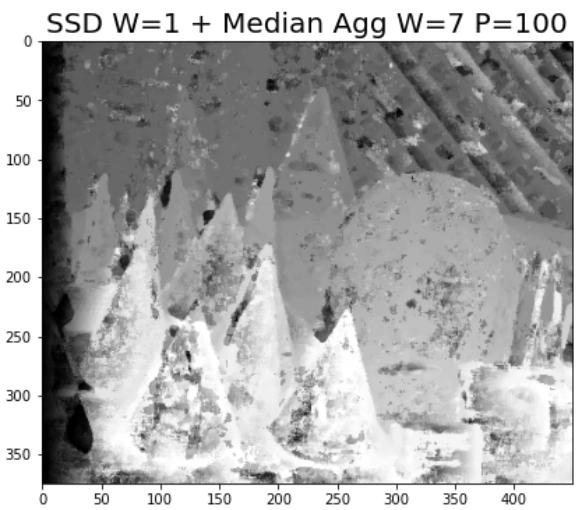
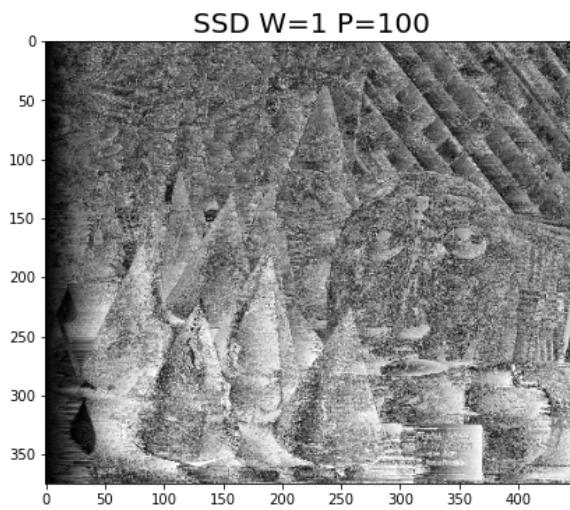
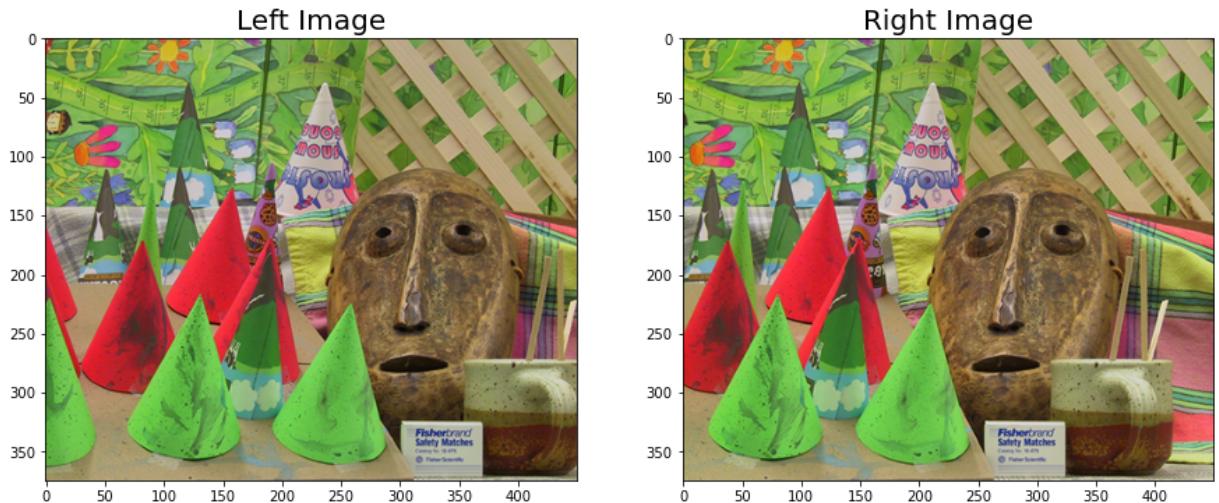
    print(f'Median Aggregation time: {(time() - init):.2f} s')

    plot_images(imageL=left_img,
                imageR=right_img,
                disp_mapL=min_cost,
```

```
disp_map2=min_cost_median_agg,  
disp_map1_title=f"SSD W={ssd_window} P={penalty}",  
disp_map2_title=f"SSD W={ssd_window} + Median Agg W={agg_winc}
```

Median Aggregation time: 23.64 s
Median Aggregation time: 23.78 s





Looping over windows for aggregation

(Be aware: this can take awhile and consume a lot of memory!)

In [17]:

```
# setup values to be used
ssd_window = 1 # window size for the vanilla ssd

# define penalty parameters (if to use)
apply_dist = False # if to use penalty
penalty = 100      # penalty value
```

In [18]:

```
init = time()

fig = plt.figure(figsize=(15, 10))
ax1 = fig.add_subplot(1, 2, 1)
ax1.set_title("Left Image", size=20)
ax1.imshow(left_img)
ax2 = fig.add_subplot(1, 2, 2)
ax2.set_title("Right Image", size=20)
ax2.imshow(right_img)
plt.show()

for ssd_window in ssd_window_sizes:
    for window_size in window_sizes[1:]:
        costs0 = compute_ssd(left_image=left_img,
```

```

        right_image=right_img,
        disparities=max_disp_steps,
        window_size=ssd_window,
        apply_dist=apply_dist,
        penalty=penalty)
min_cost = np.mean(np.argmin(costs0, axis=-1), axis=-1)
mean_agg = compute_aggregation(costs0=costs0, window_size=window_size)
min_cost_mean_agg = np.mean(np.argmin(mean_agg, axis=-1), axis=-1)

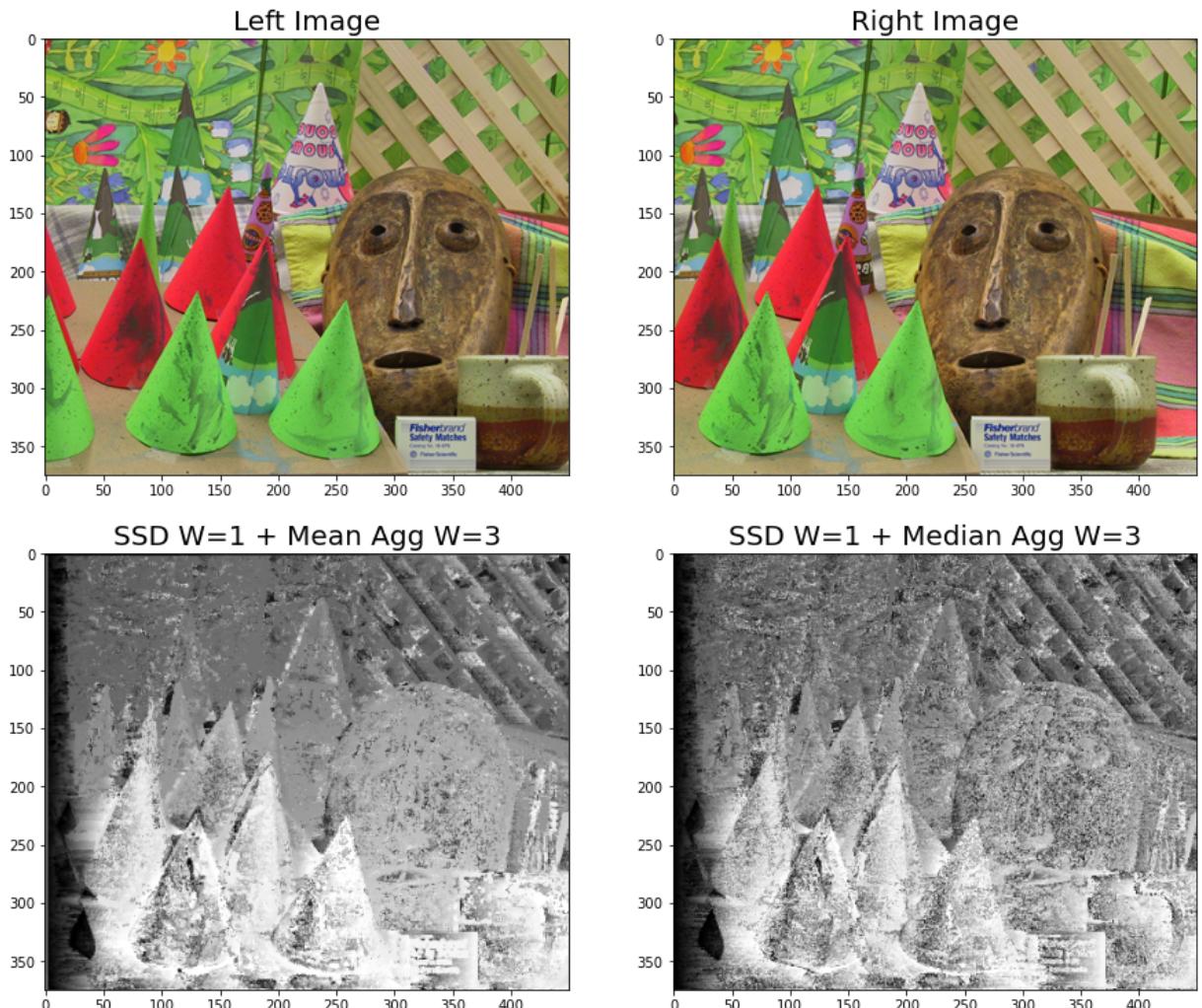
median_agg = compute_aggregation(costs0=costs0, window_size=window_size)
min_cost_median_agg = np.mean(np.argmin(median_agg, axis=-1), axis=-1)

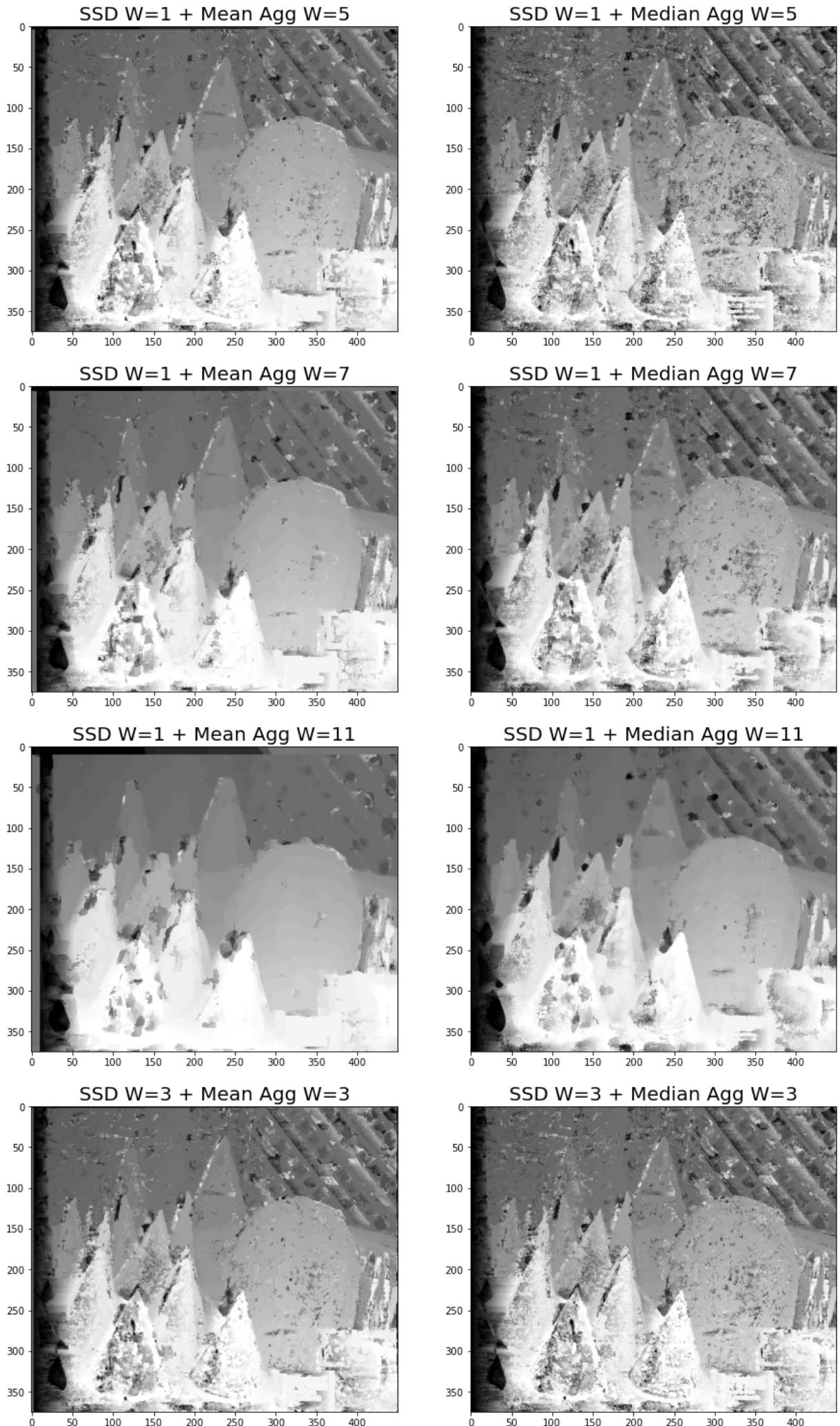
fig = plt.figure(figsize=(15, 10))
ax1 = fig.add_subplot(1, 2, 1)
ax1.set_title(f"SSD W={ssd_window} + Mean Agg W={window_size}", size=16)
ax1.imshow(min_cost_mean_agg, cmap='gray')
ax3 = fig.add_subplot(1, 2, 2)
ax3.set_title(f"SSD W={ssd_window} + Median Agg W={window_size}", size=16)
ax3.imshow(min_cost_median_agg, cmap='gray')
plt.show()

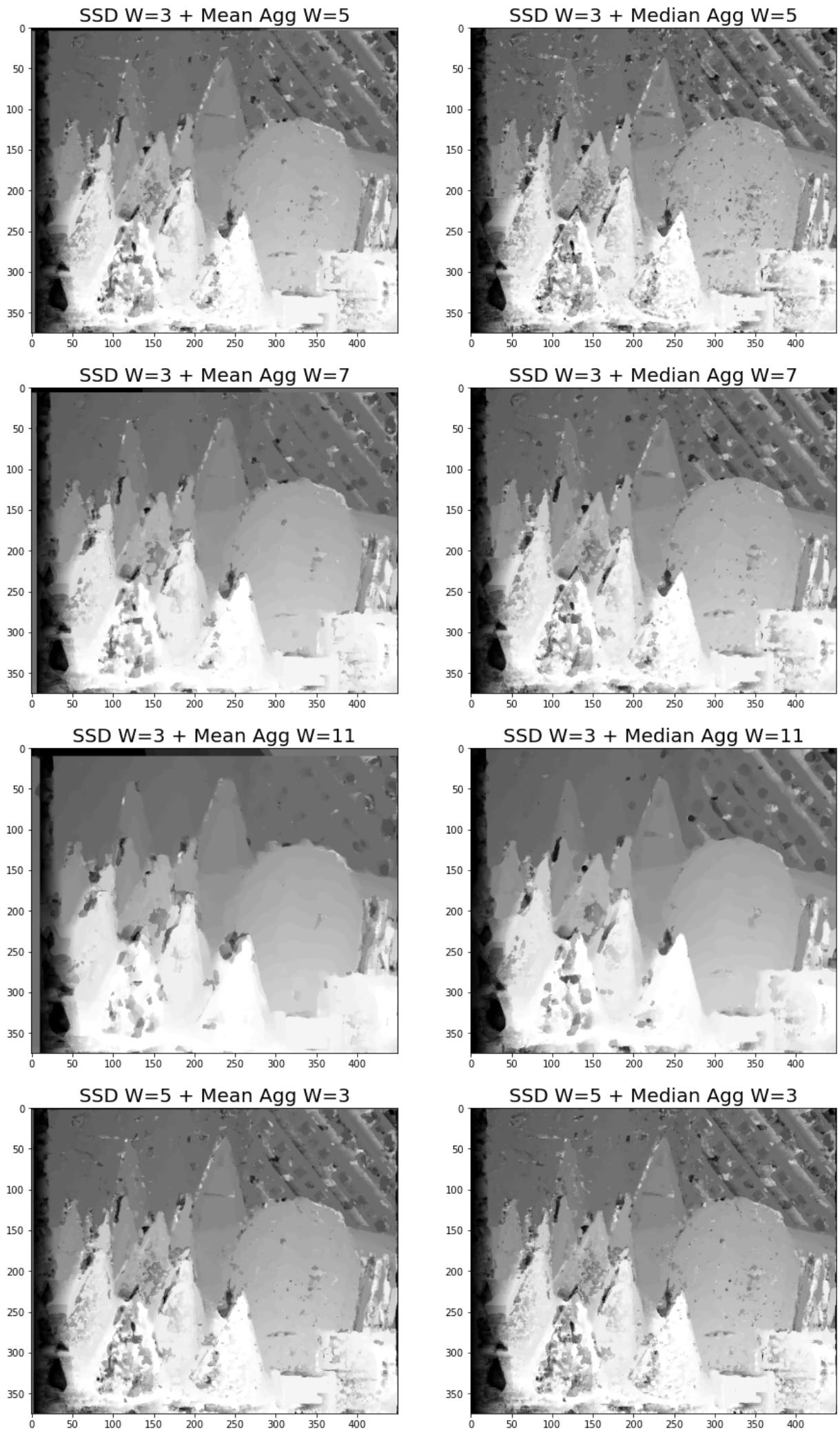
del costs0, min_cost, mean_agg, min_cost_mean_agg
del median_agg, min_cost_median_agg

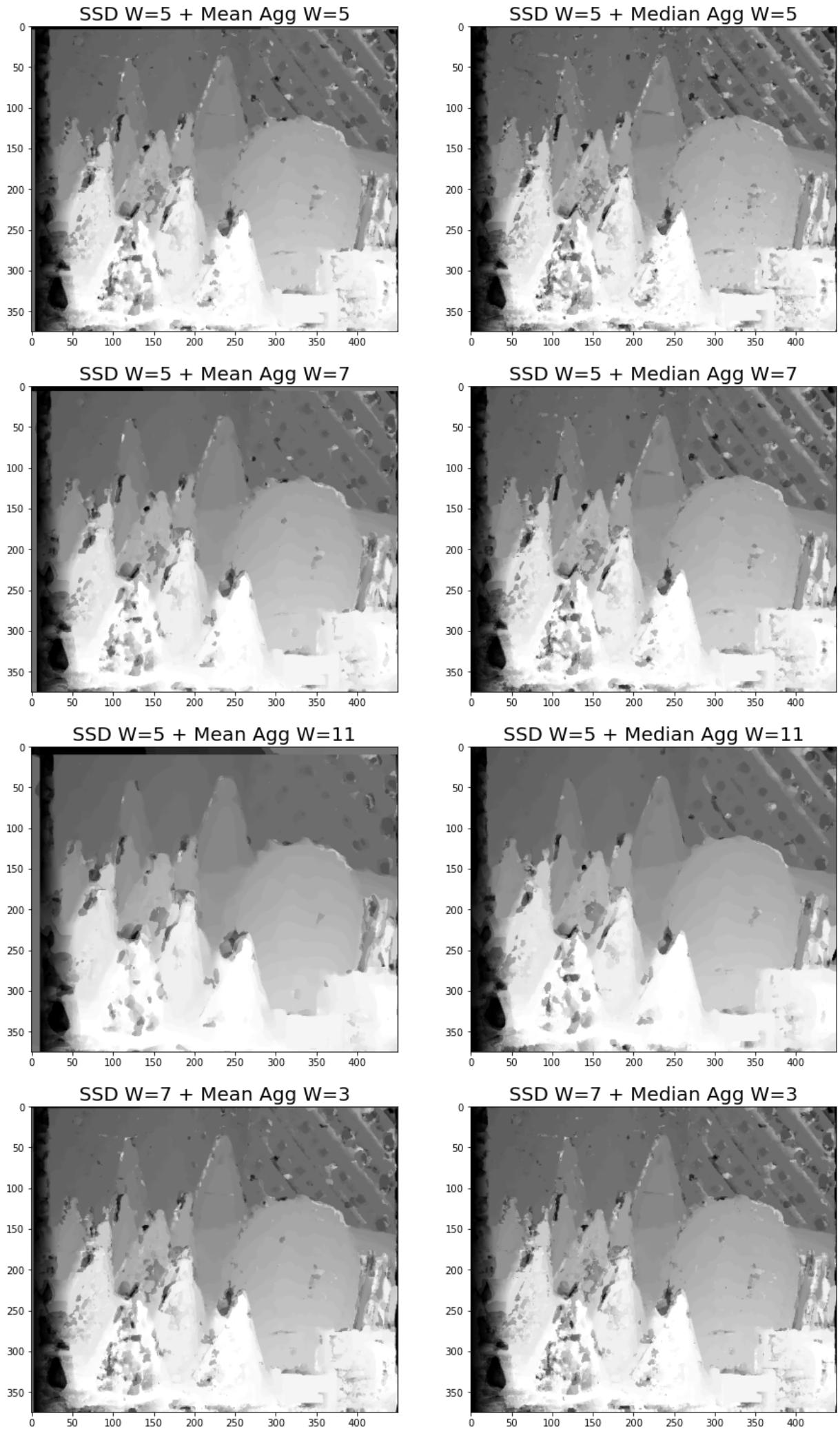
print(f'Elapsed time: {(time() - init):.2f} s')

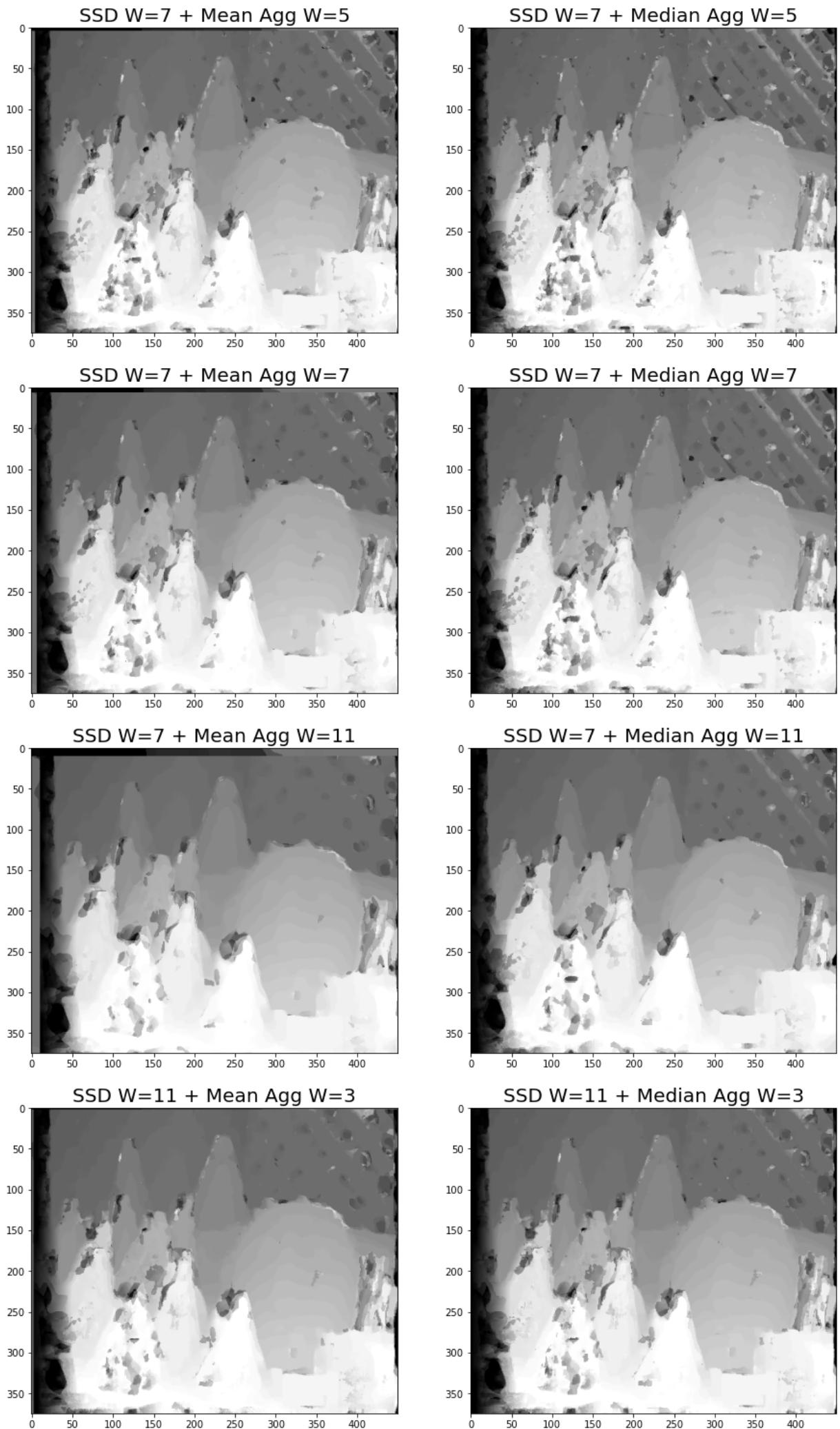
```

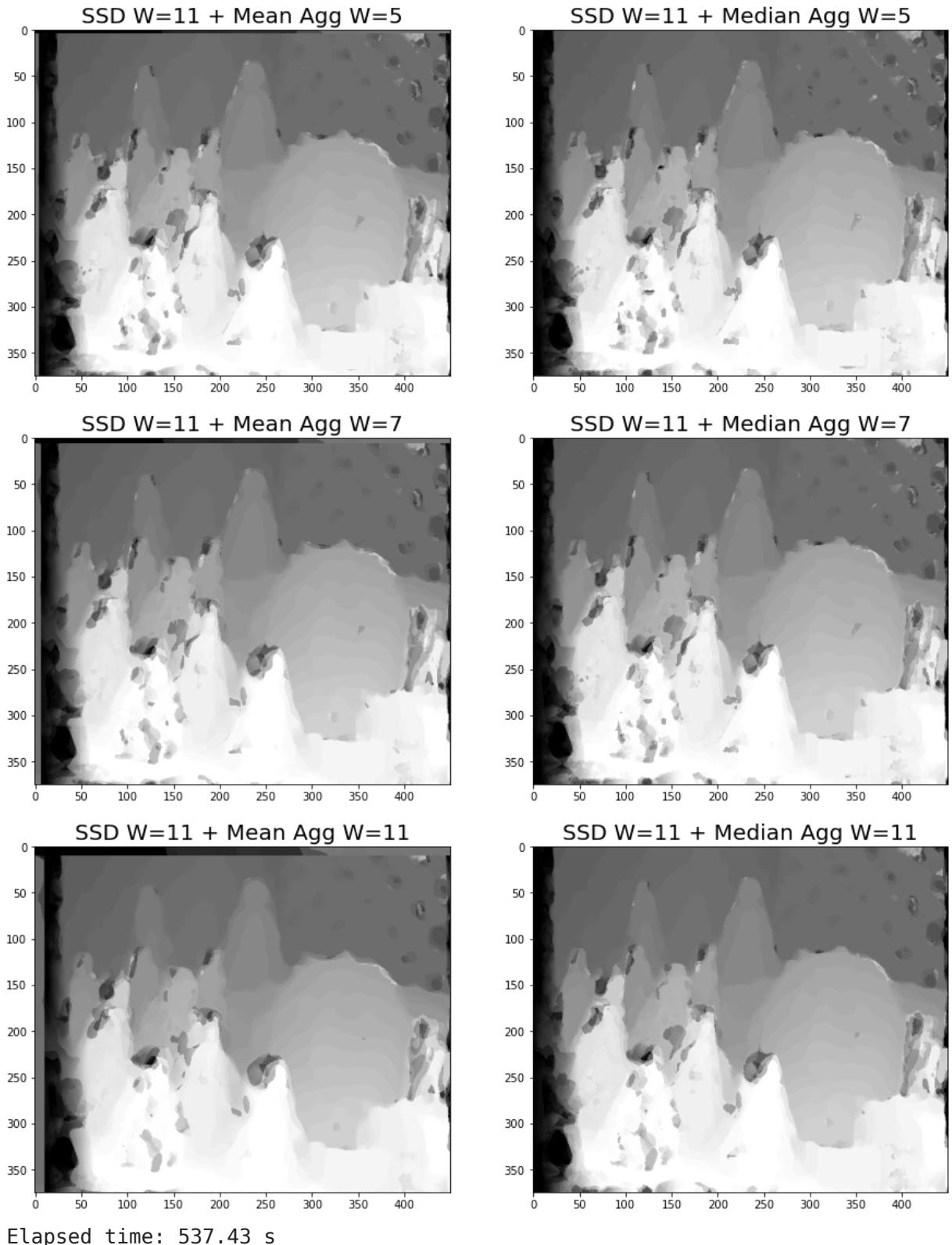












Comments

Putting both Mean and Median strategies in comparison, we have got better results comparing either one with the approaches we applied on the Task 1. The visual results showed smoother and more detailed images than the first task of this project.

Comparing both Mean and Median strategies against each other, we concluded that the results generated after applying mean filters are smoother than the Median approach. However, the last one apparently preserve more the fine lines of the objects on the scene.

We also tested different window sizes values, using some odd numbers starting in 1 until 11. Analyzing the visual outcomes presented on the previous plots, we concluded that the best results were obtained using window sizes and kernel sizes (for the mean and median filters) of 5x5 and 7x7. The window size of 11x11 showed good results as well, but from this neighborhood size we started to loose details, such as object boundaries and textures.