AI6121 Computer Vision

Chen Yongquan (G2002341D)
Nanyang Technological University

Assignment 2

Contents

Introduction

Stereo Vision

Stereo vision is a field in computer vision concerned with 3D model reconstruction of scene from 2D images. In order to recover 3D information from 2D images, we would require more than one view of the same scene from different viewpoints. Because objects closer to the camera translates more along the image plane than far objects, by matching corresponding pixels across two images and finding the translation along the image plane, we can compute the 3D depth of an image point using the formula:

$$Z = \frac{fT}{x_l - x_r}$$

Z: Depth
f: Focal length
T: Stereo baseline
x_l: Left pixel x-coordinate
x_r: Right pixel x-coordinate

The stereo baseline is the distance between the two cameras, while the difference between the left and right pixel coordinates is also known as disparity. The process of computing 3D information from multiple images is referred to as triangulation. A key step in triangulation of stereo image pairs is to first find the matching left and right pixels, which is also called the correspondence problem or stereo matching.

Epipolar Geometry

To perform stereo matching, we require that the image pairs be undistorted and fit the projections from a pinhole camera, or that pre-processing be done on them to ensure that they meet the conditions. Next, if the stereo cameras are not aligned parallel to each other or to the baseline, not at the same height or not having the same focal length, then the corresponding pixels may not fall along the same horizontal scanline.

Trying to solve the correspondence problem in such a scenario would result in a large search space across the entire image and is computationally expensive and inefficient. Thus, we would also need to perform image rectification, which re-projects the image planes to be parallel to the stereo baseline such that corresponding pixels are aligned along the same horizontal plane. Image rectification normally consists of scaling, rotation and skewing. The mathematics behind image rectification is epipolar geometry.

In epipolar geometry, the points at which the stereo baseline intersects each image plane are called epipoles. Any plane which contains the stereo baseline is called an epipolar plane and any lines resulting from epipolar planes intersecting with the image planes are called epipolar lines. Image rectification then aims to realign the epipolar lines to be parallel to the horizontal axis of the image planes. The stereo correspondence prerequisite that corresponding pixels be found on the same epipolar lines is then referred to as the epipolar constraint.

Stereo Matching Algorithm

Satisfying the conditions above, we can then simplify the search space to a single scanline for every pixel. The easiest way of finding stereo correspondence is the naïve local block matching algorithm which involves sliding a window over scanlines to find the pixel with the best similarity measure score. The naïve block matching method can encounter problems with finding an optimal window size as we will see in the later sections.

Another method is to use adaptive windows instead which modifies the window used for cost evaluation based on surrounding pixels. For instance, we can use nine asymmetrically aligned windows around each pixel and use the window with the lowest cost for evaluation [1]. This has the effect of using a window which does not contain a disparity discontinuity when available and can help with the foreground fattening problem when using large window sizes in the naïve algorithm. Another variant of adaptive windows is to divide the window into 9 sub-windows and only use the best 5 sub-windows with least color dissimilarity for cost computation [2]. We can also use adaptive similarity measures such as weighing the cost function with a likelihood function based on the probability that the pixel in the matching window lies on the same disparity as the reference pixel [3]. Apart from such local matching algorithms, there are also global and semi-global algorithms which formulate the correspondence problem as an energy minimization problem [4].

We would also require a similarity measure for comparing the region around each pair of pixels. Valid cost functions include sum of absolute differences (SAD) [5], sum of squared differences (SSD), zero-mean SAD, locally scaled SAD, and normalized cross correlation. Our implementation uses SSD which is given by the formula:

$$\text{S.S.D} = \sum_{u=-N}^{N} \sum_{v=-N}^{N} [I(x+u, y+v) - g(u,v)]^2$$

g is the image patch centered around the reference pixel in the first image while I is the second image being matched. Our goal is then to find the best x and y offset coordinates that gives us the lowest SSD and consequently the matching pixel coordinate in the second image. The SSD function can be expanded into the following form:

$$\text{S.S.D} = \sum_{u=-N}^{N} \sum_{v=-N}^{N} [I(x+u, y+v)^2 - 2I(x+u, y+v)g(u,v) + g(u,v)^2]$$

Because $g(u,v)$ is constant while finding a corresponding pixel in the second image, we can exclude the term without affecting the computation for the best match. The formula then becomes two times the cross-correlation between the image patch in the first image with the corresponding patch in the second, summed with the second patch squared. We can see the latter term as a penalty term, which penalizes pixel differences more. Alternatively, we can exclude it too and just use the middle term which is the cross-correlation.

After finding corresponding pixel pairs, we can then obtain the disparity which is the distance between the pixels' x-coordinates. To visualize small disparities properly, we would then normalize the final disparity range to the 8-bit range.

Implementation

The disparity computation algorithm for this report is programmed in Python in the attached Jupyter notebook. *computeDisparityRight* is the reverse of *computeDisparity* using the right view as the reference image. We need to compute the disparity map for both views so that we can compare both to check for occlusion later. To reduce computational complexity, we also reduce our search space to the left of the left reference pixel within a range specified by the maximum disparity value, *numDisparities*, provided by the user. This implicitly assumes that there are no negative disparities in our camera views, though it should be noted that negative disparity is possible when using uncalibrated cameras and estimation is done for image rectification or when using converging stereo cameras. This constraint is reversed for *computeDisparityRight* where we search right of the right reference pixel instead.

```
def computeDisparity(l, r, numDisparities, blockSize):

  N = int((blockSize / 2))
  lPadded = np.pad(l, N).astype(np.int)
  rPadded = np.pad(r, N).astype(np.int)
  lOut = np.zeros(l.shape, np.uint8)
  rOut = np.zeros(r.shape, np.uint8)
  lCrpd = np.zeros(l.shape, np.uint8)
  for y in range(l.shape[0]):
    if y % 10 == 0:
      clear_output(True)
      print('Progress: %d%%' % int(y/l.shape[0] * 100))

    for lx in range(numDisparities, l.shape[1]):
      # Extract block from left image
      lBlock = np.array(lPadded[y:y+blockSize,lx:lx+blockSize])

      # Extract blocks from right image for image correlation
      # Assume no negative disparity and only search within range left of left image pixel specified by numDisparities(maximum disparity)
      rBlocks = np.array([rPadded[np.arange(y,y+blockSize), rx:rx+blockSize] for rx in range(lx-numDisparities, lx+1)])

      # Batch apply sum of squared differences for all right image pixels with left image pixel
      similarity = np.array(np.sum((rBlocks-lBlock)**2, (1,2)))

      # Compute disparity and matching right pixel coordinate
      similarity = np.flip(similarity)
      disparity = similarity.argmin()
      #disparity = numDisparities - similarity.argmin()
      rx = lx - disparity

      # Normalize disparity to 8-bit range
      nDisparity = np.uint8(disparity / numDisparities * 255)

      lOut[y, lx] = nDisparity
      rOut[y, rx] = nDisparity
      lCrpd[y, lx] = rx

  clear_output(True)
  print('Progress: 100%')
  return [lOut, rOut], lCrpd


def computeDisparityRight(l, r, numDisparities, blockSize):
  N = int((blockSize / 2))
  lPadded = np.pad(l, N).astype(np.int)
  rPadded = np.pad(r, N).astype(np.int)
  lOut = np.zeros(l.shape, np.uint8)
  rOut = np.zeros(r.shape, np.uint8)
  rCrpd = np.zeros(r.shape, np.uint8)
  for y in range(l.shape[0]):
    if y % 10 == 0:
      clear_output(True)
      print('Progress: %d%%' % int(y/r.shape[0] * 100))

    for rx in range(0, r.shape[1] - numDisparities):
      # Extract block from right image
      rBlock = np.array(rPadded[y:y+blockSize,rx:rx+blockSize])

      # Extract blocks from left image for image correlation
      # Assume no negative disparity and only search within range right of right image pixel specified by numDisparities(maximum disparity)
      lBlocks = np.array([lPadded[np.arange(y,y+blockSize), lx:lx+blockSize] for lx in range(rx, rx+numDisparities+1)])

      # Batch apply sum of squared differences for all left image pixels with right image pixel
      similarity = np.array(np.sum((lBlocks-rBlock)**2, (1,2)))

      # Compute disparity and matching right pixel coordinate
      disparity = similarity.argmin()
      lx = rx + disparity

      # Normalize disparity to 8-bit range
      nDisparity = np.uint8(disparity / numDisparities * 255)

      lOut[y, lx] = nDisparity
      rOut[y, rx] = nDisparity
      rCrpd[y, rx] = lx

  clear_output(True)
  print('Progress: 100%')
  return [lOut, rOut], rCrpd
```

Results

Corridor

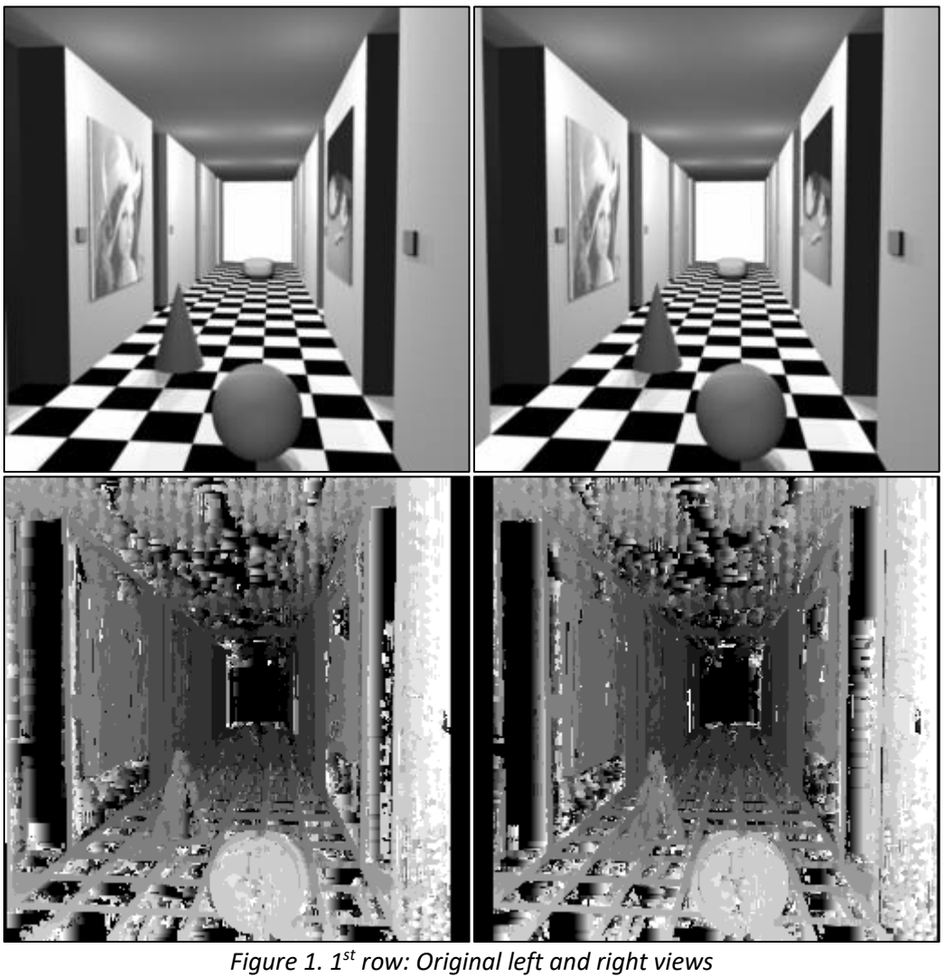

Figure 1. 1st row: Original left and right views
2nd row: Left and right disparity maps
(numDisparities = 10, blockSize = 3)

Triclopsi2

Figure 2. 1st row: Original left and right views
2nd row: Left and right disparity maps
(numDisparities = 10, blockSize = 12)

Comparison on sliding window size

Corridor

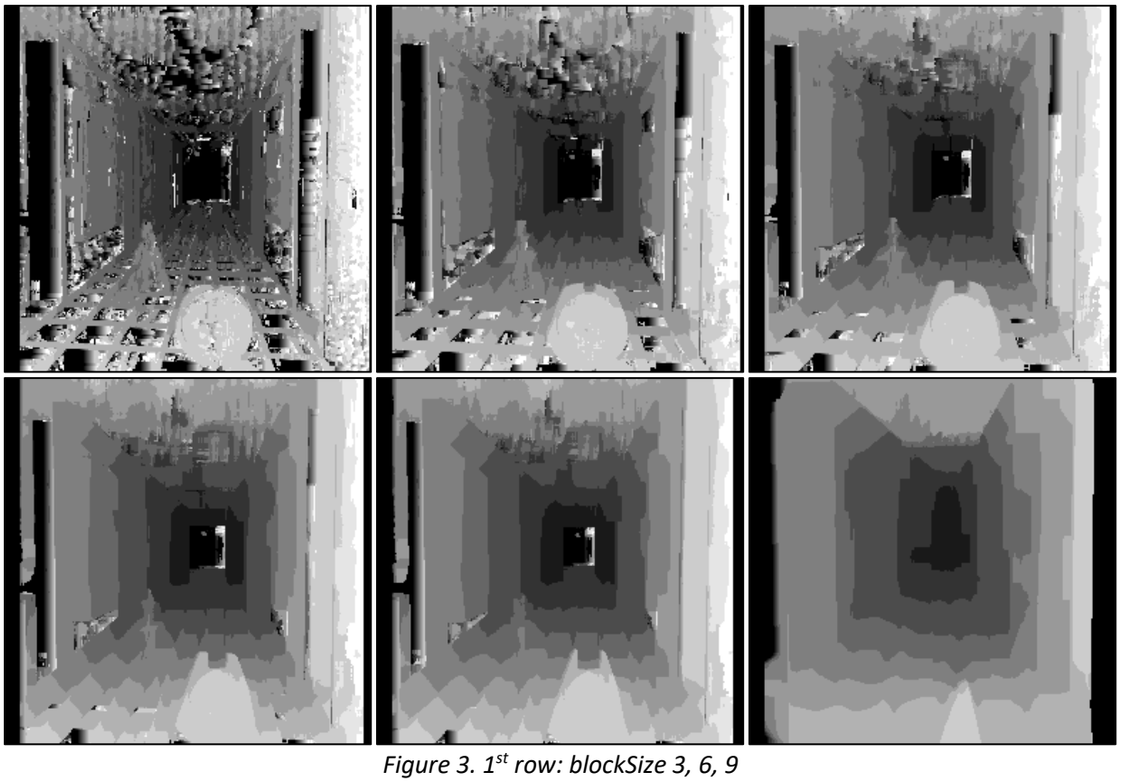

Figure 3. 1st row: blockSize 3, 6, 9
2nd row: blockSize 12, 15, 40

We can see that there are more details in the generated disparity map with smaller window sizes but also more erroneously computed disparities presenting as speckle noise. Larger window sizes result in less details with the ball and cone disappearing completely at window size 40 but has less wrongly mapped disparities due to homogeneous regions, repetitive patterns or when color gradients are in the vertical direction only. We do however see some foreground objects becoming larger than in the original image with larger window sizes such as in window sizes of 3 versus 9 where the ball in the foreground is larger in the latter disparity map.

Triclopsi2

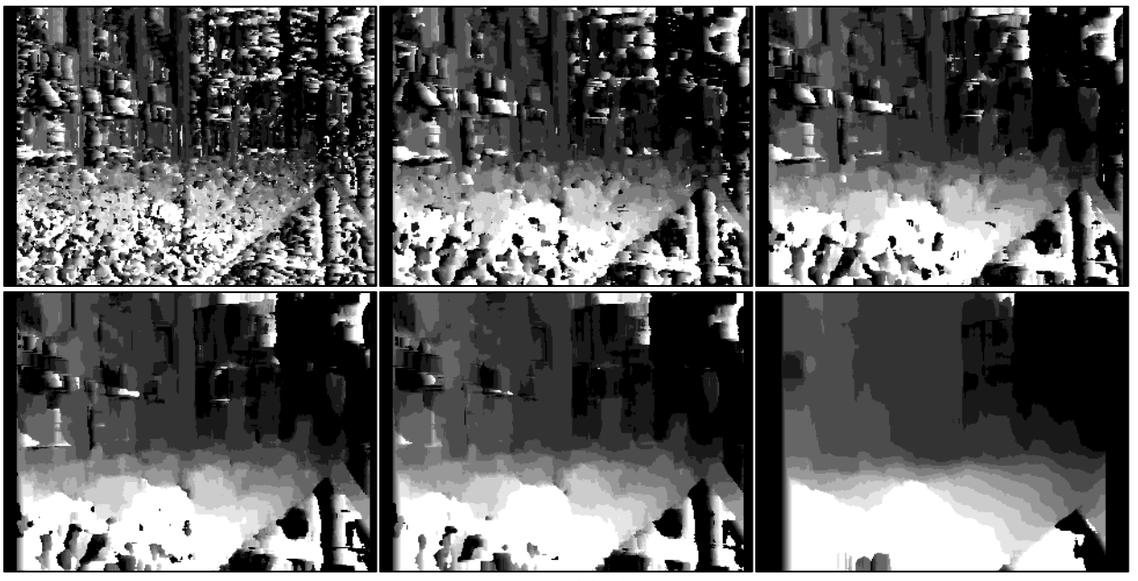

Figure 4. 1st row: blockSize 3, 6, 9
2nd row: blockSize 12, 15, 40

From the triclopsi2 samples, we can see that what is a good window size for one set of images may not be good for another, as the window size of 3 resulted in an extremely noisy disparity map as compared to in the corridor set where we could easily make out the cone, ball and corridor. Here the repetitive patterns in the bushes resulted in lots of speckle noise as the window size is around the same width as the leaves in the bushes. Thus, even though the actual pixel is further away, another pixel with a similar pattern of leaves around it may yield the best match, resulting in the black noise speckles i.e. smaller than actual disparity. Likewise, for window sizes smaller than the width of untextured regions, we see black blobs of wrongly computed disparities such as the sidewalk in triclopsi2 and floor tiles in corridor.

Comparison on maximum disparity

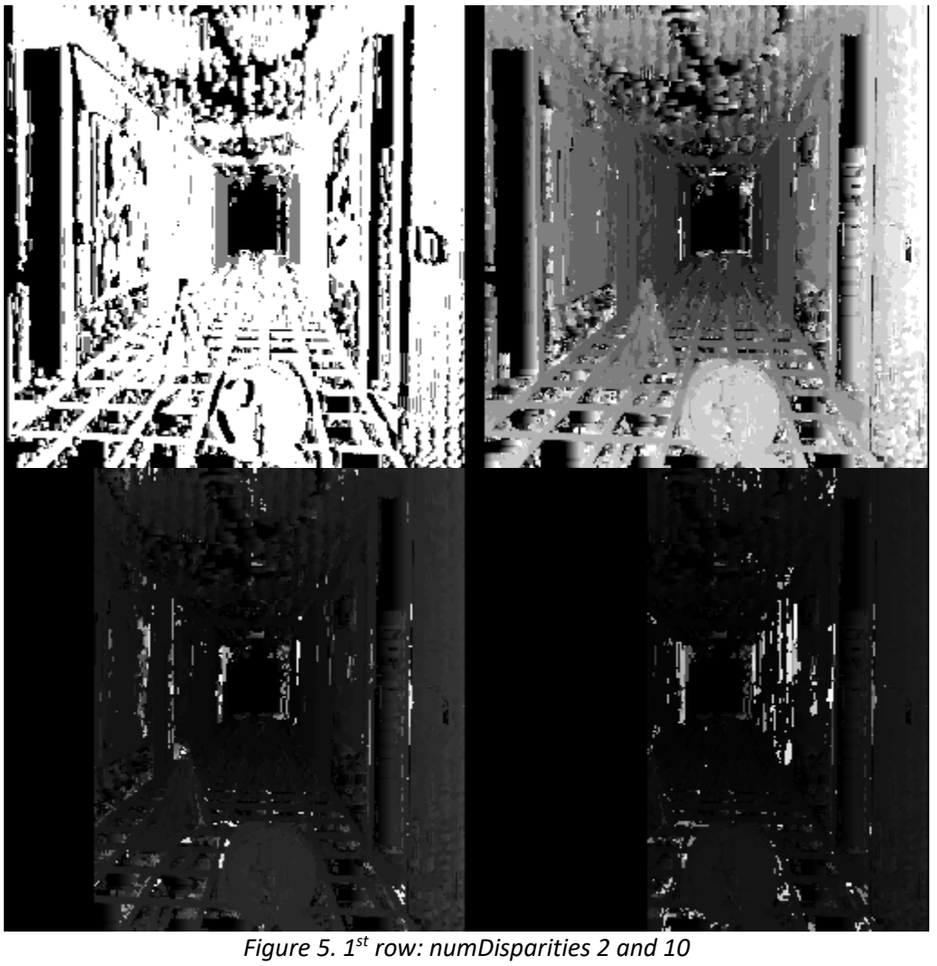

Figure 5. 1st row: numDisparities 2 and 10
2nd row: numDisparities 50 and 100

The large black bars on the left is because our implementation skips computation for the first N pixels equal to the maximum disparity. This can be avoided if we do not hard code the search range and account for the edge cases around the left and right boundaries i.e. use a smaller search range if there are less pixels than the maximum disparity. However, even accounting for such edge cases, there will still be some black areas left as there are indeed areas in the reference image that are not captured in the corresponding view so the above modification will only improve results for when the maximum disparity provided is more than the actual maximum disparity in the stereo images.

Increasing the maximum disparity will increase our search range, which will in turn result in some pixels erroneously being matched to pixels far away with similar surrounding patterns than in their immediate vicinity. Thus, the computed disparity map will contain outliers with large disparities that cause 8-bit normalization to not be as effective for the other correct smaller disparity values. This effect can be seen easier when we zoom in to the images in Figure 6.

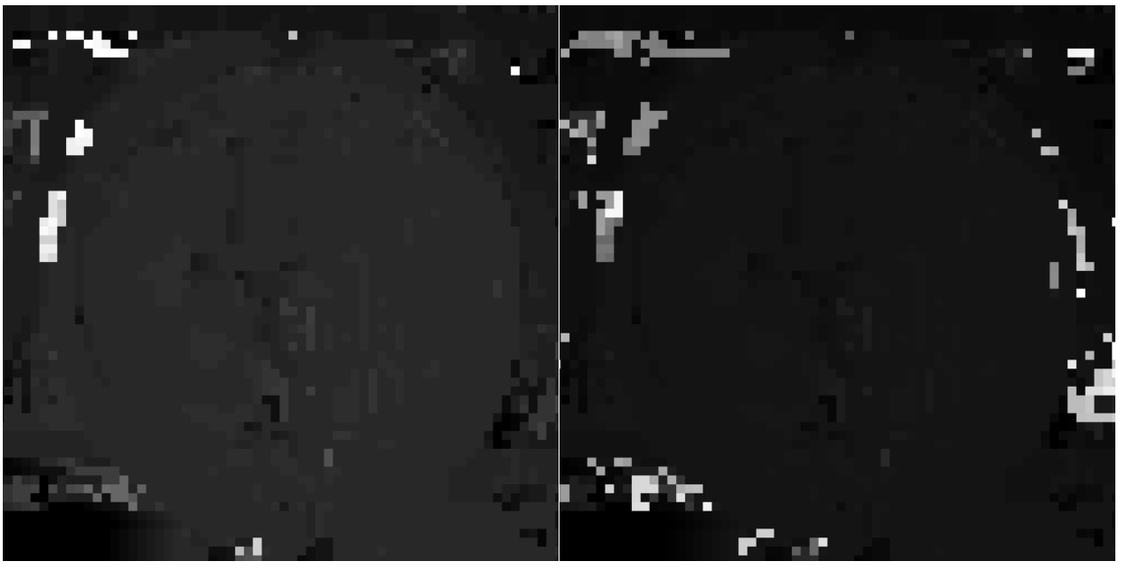

Figure 6. Zoomed crop of ball in foreground for numDisparities 50 and 100

Discussion

Based on our observations of our results, we can point out a few limitations of the naïve stereo matching algorithm and disparity computation that uses it for the correspondence problem.

Homogeneous Textures

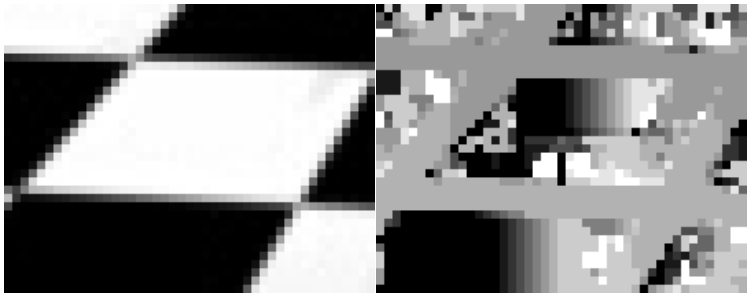

Figure 7. Effect of homogeneous textures: Floor tiles along corridor

Firstly, when there are large regions of untextured or homogeneous areas wider than the sliding window size, all the pixels in the search range have equal sum of squared differences for their surrounding patches and the algorithm cannot differentiate the actual corresponding pixel from the others. Thus, it wrongly picks the nearest one. From Figure 7, we see that the algorithm works correctly at the boundaries of each floor tile as there is enough color gradients for each surrounding patch to be distinctive from one another. However, disparity is wrongly computed for the center of each tile when window size is smaller than the tile size.

Occlusion

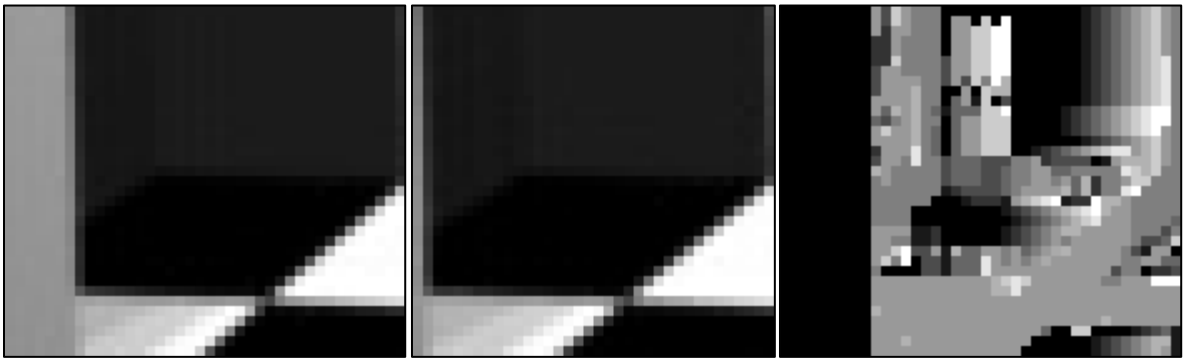

Figure 8. Effect of occlusion: Left view, right view, disparity map of left side of corridor

Next, occlusion in the left and right view of the input images will result in pixels that do not have actual corresponding pixels in the matching image. The naïve algorithm does not account for this and still tries to match them and erroneously fills in the disparity from the foreground into them. In Figure 8, we see that the pillar on the left side of the corridor occludes some parts of the floor tile and wall in the left view. This results in the algorithm wrongly matching the occluded region and we see a vertical bar of greyish disparities in the occluded area. The fact that there are pixels in each view that do not have a correspondence in the other view also means that there may be multiple pixels in the reference image that match to the same pixel in the right image, and also pixels in the right image that do not have a corresponding pixel in the reference image when disparity computation is done for only one view.

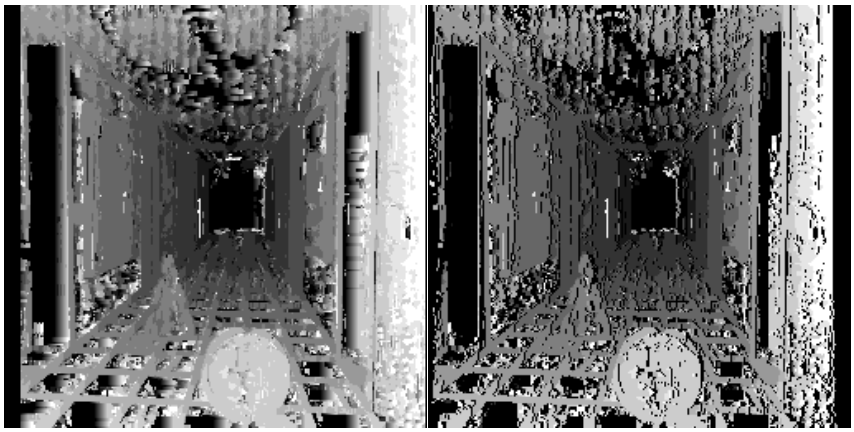

Figure 9. Left disparity map and corresponding right image filled with the same disparity

We can see this in Figure 9 which shows the corresponding right image where the matching right pixels are filled with the same disparity value when computation is done using the left view as the reference. The right image is filled with many 'holes' as there are no reference pixels that matched to them.

Aperture Problem

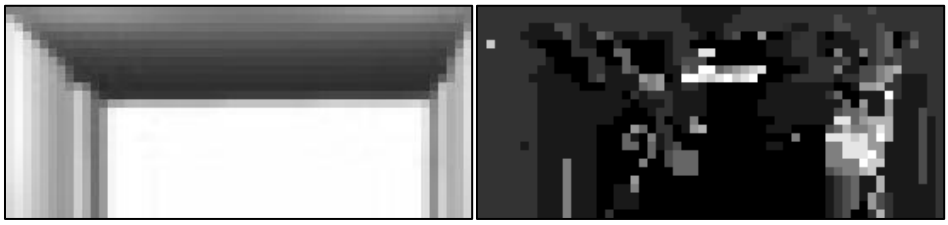

Figure 10. Effect of only having vertical gradients: Ceiling at the end of corridor

Another problem with the sliding window approach is that when color gradients is orthogonal to the direction of the sliding window, the algorithm cannot detect the actual displacement of the corresponding pixels. As seen in Figure 10, because the sliding window can only perceive patches in the horizontal direction, even though there is some form of texture and gradient at the end of the corridor, the algorithm still cannot identify horizontal displacement correctly and computes the disparity wrongly at the ceiling.

Other Limitations

Additionally, when there are repetitive textures or uniform distributions of repeating colors, the SSD of individual patches may also not be extremely distinctive from one another and can result in wrong disparities such as in the bush of triclopsi2. Triclopsi2 also has the problem of untextured regions in the sky in the top right corner and pathway along the right.

Finally, a limitation that isn't shown in both samples is that of specular reflection where specular highlights in reflective objects can cause captured pixels to have vastly different values from different view angles and thus completely different squared sums even though they correspond to the same point in the 3D scene. Thus, for point-based matching like as in the naïve stereo matching algorithm to work properly, we need to ensure that all objects in the image pairs exhibit Lambertian reflectance.

Improvements

One possible improvement that can be made for the naïve stereo matching algorithm is to account for occlusion by using disparity maps for left and right views that are computed separately.

First, we compute the left disparity map by using the left view as the reference point and matching to the right view. Then we separately compute a right disparity map by using the right view as the reference image. In both steps, we store the corresponding pixel coordinates in the other view for comparison.

Finally, by cross validating the corresponding pixels from both computation process, we can invalidate pixel disparities that do not have the same corresponding pixels. For instance, if for row 5 pixel 10 in the left image, we computed a corresponding pixel at column 5 in the right image, then the corresponding pixel for the right disparity computation for pixel 5 in the right image must be column 10 in the left image, else we invalidate the calculated disparity.

This process implicitly enforces the uniqueness assumption, which is that for any pixel in each image, it would have at most 1 corresponding pixel in the other image, and all other mappings are invalidated.

Finally, after scanning through the entire disparity map once for invalid disparities, we perform occlusion filling by filling in the invalid pixels with the nearest disparity value from the left or right side, depending on which side is smaller. By using the smaller disparity, we are in fact filling in the occluded region with the disparity from the background, which is also what occluded regions should have as their disparity.

Occlusion filling can result in horizontal streaking effect in the disparity map which can be lessened by smoothing the result with a median filter.

Implementation

This is the algorithm for simple occlusion checking for the left disparity map using the left and right correspondence mapping computed. Median filtering is performed separately on the output using OpenCV.

```
def occlusion_check(lcrpd, rcrpd, dp, numDisparities):
  tmp_dp = dp.astype(np.uint16)
  # If right corresponding pixel in left disparity map does not match
  # left corresponding pixel in right disparity map, invalidate computed
  # disparity
  for y, lrow in enumerate(lcrpd):
    for lx in range(numDisparities, len(lrow)):
      if rcrpd[y, lrow[lx]] != lx:
        tmp_dp[y, lx] = 256

  # For all invalidated disparities, replace with the smallest
  # neighbouring disparity i.e. disparity of background
  for y, lrow in enumerate(tmp_dp):
    for lx in range(numDisparities, len(lrow)):
      if lrow[lx] == 256:
        valid_dp = np.array([256, 256], np.int16)

        idx = lx-1
        while valid_dp[0] == 256 and idx >= 0:
          if lrow[idx] != 256:
            valid_dp[0] = lrow[idx]
          idx -= 1

        idx = lx+1
        while valid_dp[1] == 256 and idx < len(lrow):
          if lrow[idx] != 256:
            valid_dp[1] = lrow[idx]
          idx += 1

        lrow[lx] = valid_dp.min()

  return tmp_dp.astype(np.uint8)
```

Results

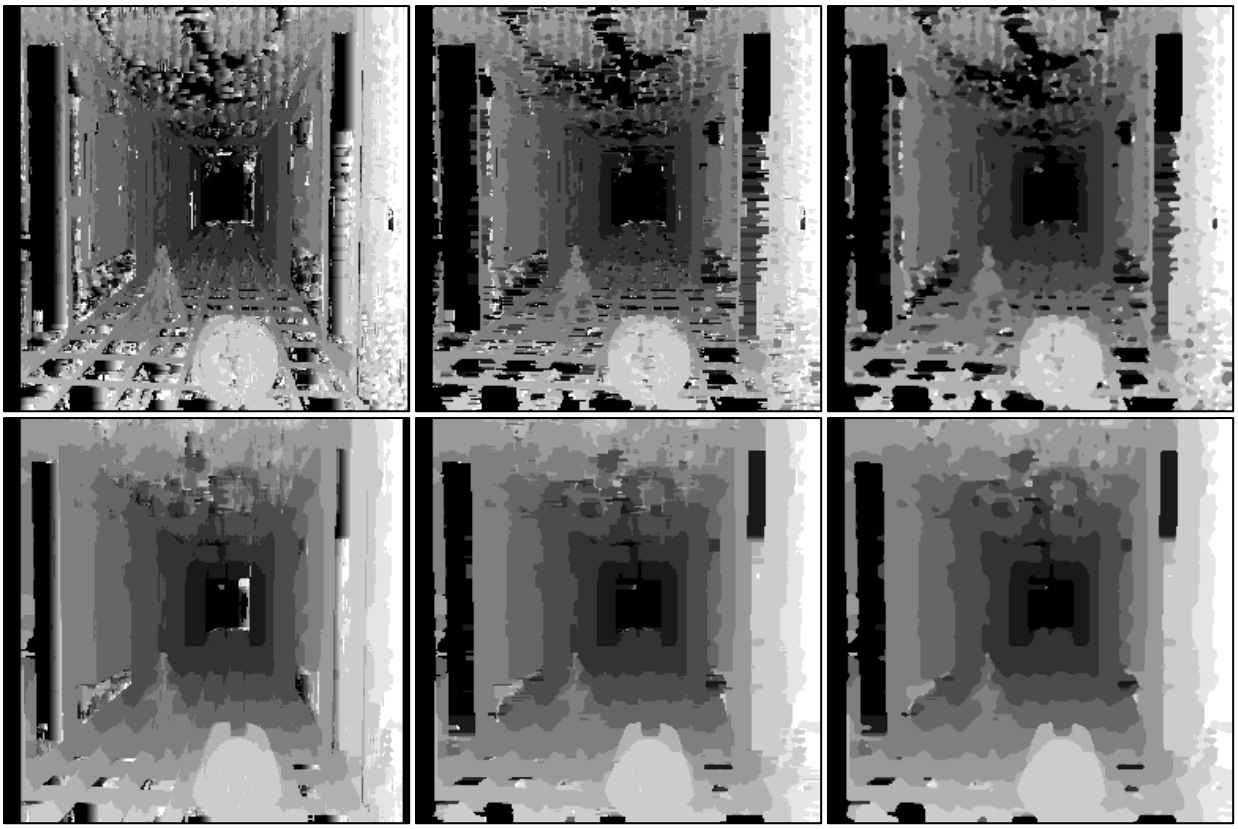

Figure 11. 1st row blockSize 3: Original disparity map, occlusion filling, occlusion filling + median filtering
2nd row blockSize 9: Original disparity map, occlusion filling, occlusion filling + median filtering

From the results above, we can see that occlusion filling can help with object fattening due to occlusion. For instance, in the center image in the 1st row, we can see the left side of the cone becomes smaller than in the original after performing disparity invalidation and occlusion filling. Median filtering done on the processed image further helps to reduce speckle noise and horizontal streaks, though streaks are not completely eliminated. For the larger window size of 9, we see that occlusion filling helped to correct some large disparities (full white patches) along the right wall so that they match the disparity at that depth. However, on the left wall, occlusion filling also caused the invalidated pixels to take on the wrong disparities from the homogeneous wall region. Finally, the wall at the end of the corridor had some wrong white disparities in the original disparity map which was corrected by occlusion checking.

Thus, we can see that occlusion checking and filling can help with some erroneous disparities, but it is not perfect and can provide worse results in some cases like in the left wall. There are other methods of stereo matching that can better deal with these limitations as described in the first section of this report like semi-global block matching, adaptive algorithms and featured based matching. However, because they are of a different paradigm than the naïve block matching algorithm we implement, we did not opt for them as a possible improved modification that can be done for our implementation.

References

[1] A. Fusiello, V. Roberto and E. Trucco, "Efficient Stereo with Multiple Windowing," in *Conference on Computer Vision and Pattern Recognition*, 1997.

[2] H. Hirschmuller, P. R. Innocent and J. Garibald, "Real-Time Correlation-Based Stereo Vision with Reduced Border Errors," in *International Journal of Computer Vision*, 2002.

[3] K.-J. Yoon and I. S. Kweon, "Adaptive Support-Weight Approachfor Correspondence Search," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2006.

[4] H. Hirschmuller, "Stereo Processing by Semi-Global Matching and Mutual Information," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2007.

[5] K. Konolige, "Small vision systems: Hardware and implementation," in *Proceedings of the Internationall Symposium of Robotics Research*, 1997.