Remon Roshdy

Rr3531

Depth maps based on Normalized Cross Correlation

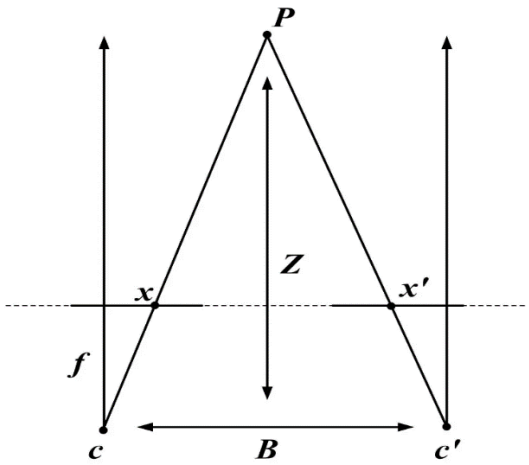
Our problem is to try do the distance estimation on a camera image by leveraging a stereo camera system. In stereo vision, we use two stereo images from two cameras to perform disparity estimation and compute depth. From there, it is possible to estimate the distance from the camera to a vehicle or any object in the image taken. Accurate depth estimation from cameras is a significant computer vision problem, yet to be solved entirely with a lot of ongoing research.

To get the depth, first, we need to compute the disparity but how? The same scene can be captured by two cameras mounted in a stereo system, but how are the images different?

The corresponding pixel points of the image pair will have different coordinates. Here disparity comes into play. Disparity is the distance between these two sets of coordinates for the same scene point. In other words, disparity measures the displacement of the image points between two frames, a reference image, and a target image, with a square window centered around the pixel of interest. The choice of window size is crucial, impacting both the disparity range and the desired level of accuracy. If we compute the disparity for each corresponding pixel on the image, the output will be the disparity map.

Computing the disparity map allows us to retrieve the depth information and output the depth map of the image, which can be helpful to know how far a vehicle is or for other applications such as 3D reconstruction.

Below is the stereo camera model.



The depth Z is the distance between a point P in the real world and the camera. This diagram presents a stereo vision system with two parallel cameras, C and C'. The distance B between the

cameras is called the baseline, f is the focal length, and x and x' are the image planes of the cameras C and C'.

By triangulation, we can compute the depth Z with the following formula, where (x - x') is the disparity:

$$Z = \frac{f \cdot B}{(x - x')}$$

From the above equation, it is essential to note that depth and disparity are inversely proportional. In other words, the greater the depth, the lesser the disparity, and the lesser the depth, the greater the disparity.

Now, we know how to get the depth granted that we know the disparity. It is a crucial step as it opens the doors to 3D reconstruction and 3D computer vision. But wait! How do we compute the disparity?

In my specific implementation, I employed the normalized cross-correlation (NCC) method for matching. Unlike other similarity measures like Sum of Absolute Differences (SAD) and Sum of Squared Differences (SSD), NCC normalizes within the window, compensating for differences in gain and bias. This normalization proves advantageous, especially when dealing with Gaussian noise, making NCC an optimal choice in stereo matching.

The matching process involves calculating the NCC between the intensity patterns of a pixel in the left image and all possible candidates in the right image. The formula for NCC is expressed as

NCC = (sum((p_l - mean_l) * (p_r - mean_r))) / sqrt((sum((p_l - mean_l)^2) * sum((p_r - mean_r)^2))).

In my implementation, this step plays an important role in finding corresponding pixels and establishing depth information.

It's worth noting that stereo matching can be approached with either left-to-right or right-to-left matching. Throughout the process, computational efficiency is a key consideration, and my implementation takes advantage of NCC's statistical robustness to deliver accurate depth maps efficiently."

Code Overview:

The provided code consists of functions to calculate the normalized cross-correlation (NCC) between corresponding patches in the left and right images. The disparity map is then computed by finding the disparity value that maximizes the NCC for each pixel. The final depth map is obtained by converting the disparity values to depth using a simple conversion formula.

Key Components:

Normalized Cross-Correlation:

The Normalized Cross Correlation function computes the NCC between two image patches. It uses the mean-subtracted pixel-wise multiplication and normalization to measure the similarity between the patches.

Disparity Map Computation:

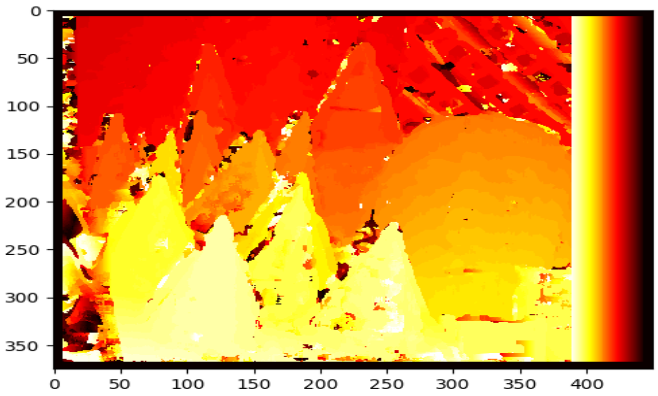
The compute_disparity_map function iterates through the image pixels and calculates the disparity by maximizing the NCC. The disparity range and window size are parameters that influence the algorithm's behavior.

Depth Map Conversion:

The convert_disparity_to_depth function converts the disparity map to a depth map. It assumes a baseline and focal length of 1, but it also addresses potential division by zero issues by setting a small non-zero value for zero disparities.

Normalization and Visualization:

The code includes functions to normalize the disparity values for better visualization and display the resulting disparity and depth maps using various colormaps.

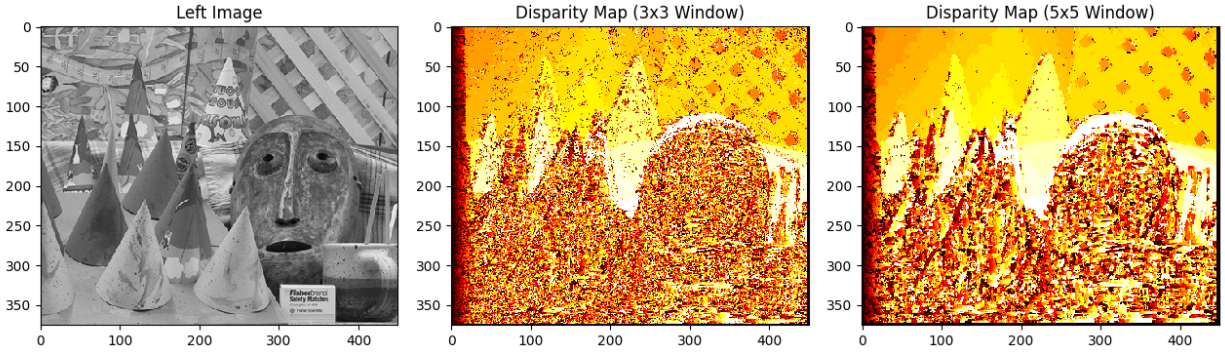


Results:

As we can see from the above image with windos size =5 and disparity 60, the depth map looks fine. Light colors are for the shapes with shortest distance and close to the camera , and the more the darker the color of the object the further the object from the camera.

Then I changed the window size to be 3x3 and 5x5 with disparity 30 and here is the results below.

We can see there are a lot of noise in both of the images with different windows size but noise is more with in the disparity map with smaller windows 3x3.



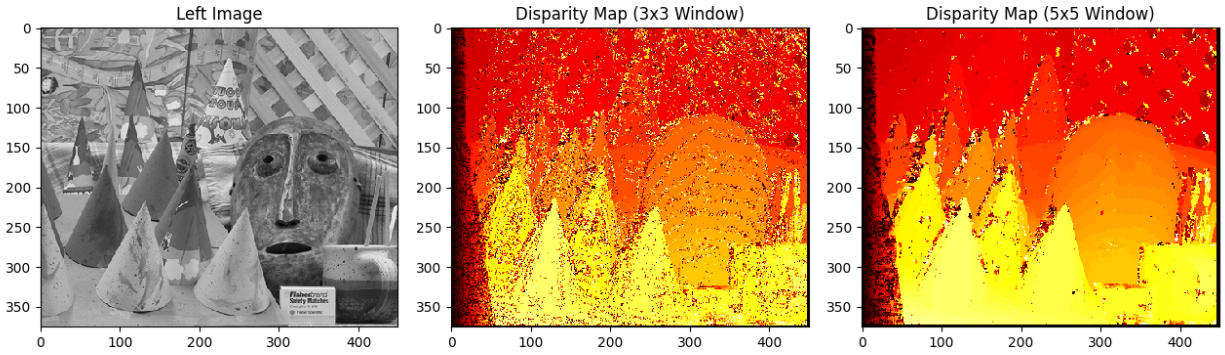
The disparity range is the range of possible disparity values that the depth estimation algorithm considers. A narrow disparity range means that the algorithm is only looking for disparities within a limited distance, and potentially excluding objects that are either very close or very far away.

Based on the image above with disparity 30 small.

As the disparity range is too narrow, objects that are close to the stereo camera may all have similar or even identical disparity values. We can se the result in these close objects being lumped together in the depth map above , as the algorithm is not able to differentiate between them based on the limited range of disparities.

A narrow disparity range limits the sensitivity of your depth estimation to capture fine depth variations. If the true disparities of objects in the scene span a wider range, a narrow disparity range might not capture the nuances of depth, leading to loss of details in the depth map

Then I changed the disparity value to be 60 again and run the program and here is the results for both windows.



Yes, there are a lot of noise in the 3x3 window.

2. Smaller Window Sizes (ex. Like 3x3):

Advantages:

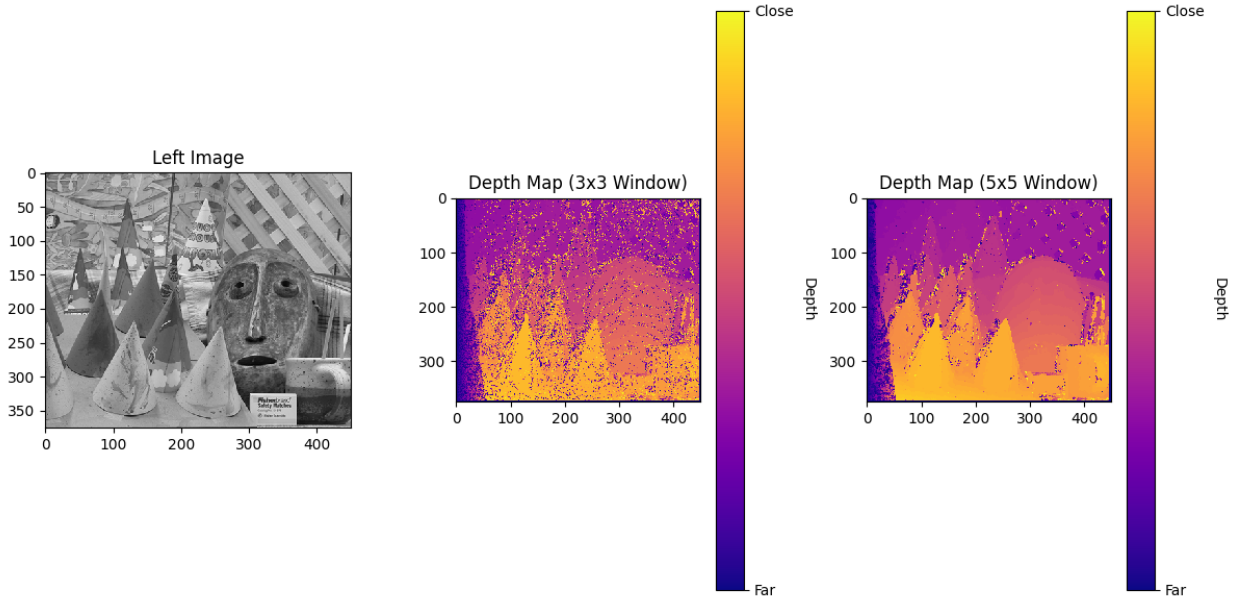
Smaller window sizes allow for the capture of finer details in the scene. This is because smaller windows focus on local information, making it possible to discern small features and depth variations. We can see in the image of the mask in the 3x3 there is some lines represents the texture and various depth in the mask while on the 5x5 image the mask represented by very smooth shape, lacking any fine details or textures. Also computationally is faster and less expensive.

Also, Smaller windows can better preserve edges and contours in the scene, as they are less likely to smooth out rapid changes in intensity.

Challenges:

Noise Sensitivity: Smaller windows are more sensitive to noise in the images. If there's noise present, it can influence the matching process, leading to less robust depth estimates.

Less Robust to Texture less Regions: In areas with low texture or repetitive patterns, smaller windows might struggle to find reliable correspondence.



Larger Window Sizes (ex. 5x5):

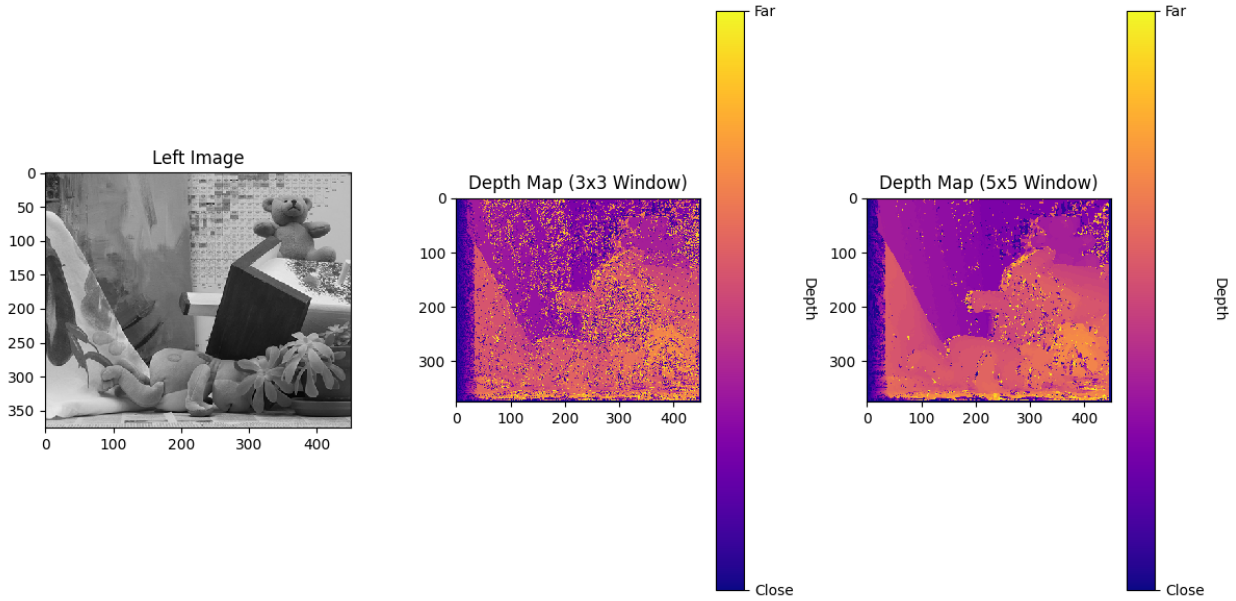
Advantages:

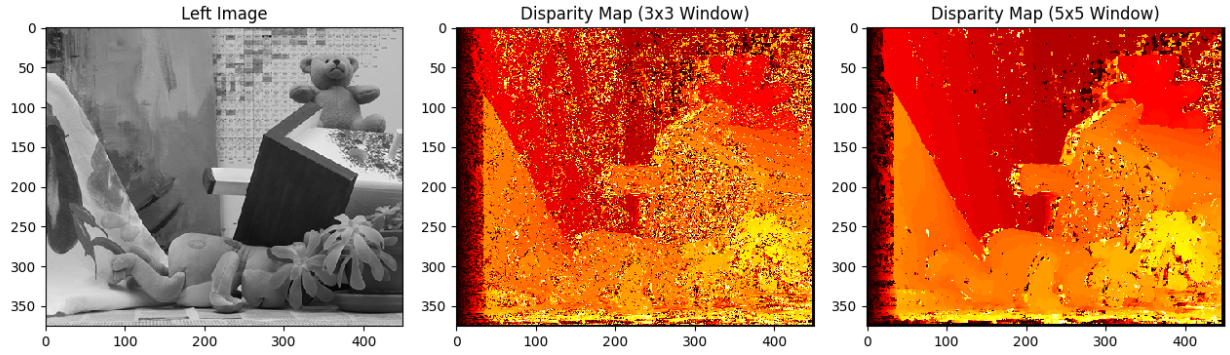
Smoothing Effect: Larger window sizes have a smoothing effect on the depth map. They incorporate more global information, which can help reduce the impact of noise and produce more stable depth estimates. As it is obvious at the depth map 5x5 window , cones and the mask are very smooth. Also Robust to Textureless Regions. In regions with low texture or repetitive patterns, larger windows can better average out noise and provide more reliable correspondences as we can see above. Also its computationally very expensive and it takes longer time to produce the depth map.

Challenges:

Loss of Fine Details: Larger windows tend to provide smoother results but might lose finer details in the scene. Rapid changes in depth or object boundaries may be less accurately captured.

Blurry Edges: The smoothing effect can result in less defined edges, leading to a blurrier appearance in the depth map.





As we can see from the photo with windows size 5x5 is better and less noisy.

Then I investigated the issue to know why we have the noise in the images with smaller windows size.

The appearance of noise and colored dots in the disparity map, especially when using a smaller window size like 3x3, is a common issue in stereo vision and disparity map computation. This phenomenon is often referred to as "stereo correspondence errors" or "stereo mismatches."

A 3x3 window might introduce more noise due to its smaller size, making it sensitive to local variations in pixel intensity. The 3x3 window may capture finer details but could be prone to errors in textureless or repetitive regions.

A 5x5 window is expected to provide smoother results and potentially higher accuracy by considering a larger neighborhood. The 5x5 window sacrifices some spatial resolution but may produce more reliable results in textured and complex areas.

Several factors can contribute to this problem:

Lack of Texture:

In regions with low texture or repetitive patterns, it can be challenging for the algorithm to find accurate correspondences. This results in noisy disparity estimates.

Ambiguity in Matching:

Smaller window sizes may lead to ambiguity in finding the best match, especially in areas with similar intensity patterns. The algorithm may incorrectly identify correspondences, leading to noise in the disparity map.

Limited Search Range:

The disparity range parameter limits the search for correspondences. If the range is too small, it may miss valid matches, and if it's too large, it may include incorrect matches.

Occlusions and Discontinuities:

Occluded regions or areas with depth discontinuities can also cause stereo correspondence errors. The algorithm may struggle to handle such cases correctly.

Parameter Tuning:

The performance of stereo matching algorithms is sensitive to parameter values, such as the window size, search range, and similarity measure. Suboptimal parameter values can lead to noisy results.

To address these problem , you may consider the following:

Increase Window Size:

A larger window size can provide more context for matching and improve the accuracy of disparity estimates. However, this may come at the cost of reduced spatial resolution.

Adjust Disparity Range:

Experiment with different disparity ranges to ensure that it covers the expected disparities in the scene. A wider range may help capture a broader range of disparities.

Post-Processing:

Apply post-processing techniques, such as median filtering or morphological operations, to reduce noise in the disparity map.

Use More Robust Matching Techniques:

Explore more advanced matching techniques, such as Semi-Global Matching (SGM) or Graph Cuts, which are designed to handle challenging scenarios.

Consider Calibration:

Ensure that the stereo camera system is properly calibrated, as calibration errors can significantly impact the accuracy of disparity estimation.

Experimenting with these suggestions and fine-tuning the parameters may help improve the quality of the disparity map and reduce the presence of noise and artifacts which can do in as a future improvements.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def NormalisedCrossCorrelation(patch_left, patch_right):

    mean_left = np.mean(patch_left)
    mean_right = np.mean(patch_right)

    numerator = np.sum((patch_left - mean_left) * (patch_right - mean_right))
    denominator = np.sqrt(np.sum((patch_left - mean_left)**2) *
np.sum((patch_right - mean_right)**2))

    #ncc_value = numerator / denominator
    ncc_value = numerator / denominator if denominator != 0 else 0.0
    """
    print("Numerator:", numerator)
    print("Denominator:", denominator)
    print("NCC Value:", ncc_value)
    """

    return ncc_value

def compute_disparity_map(left_image_array, right_image_array, window_size,
disparity_range):
    height, width = left_image_array.shape
    half_window = window_size // 2
    disparity_map = np.zeros(left_image_array.shape)

    for y in range(half_window, height):
        print("Row:", y)

        #for x in range(0, width):
        for x in range(half_window, width):
            #print("Column:", x)

            patch_left = left_image_array[max(y - half_window, 0):y +
half_window + 1,
                          max(x - half_window, 0):x + half_window + 1]

            best_disparity = 0
            max_ncc = -1

            for d in range(1, disparity_range + 1):
                x_disp = x - d
                if x_disp >= half_window:
```

```
                    # Extract the right patch with the same size as the left
patch
                    patch_right = right_image_array[max(y - half_window, 0):
y + half_window + 1,
                                  max(x_disp - half_window, 0): x_disp +
half_window + 1]

                    if patch_left.shape == patch_right.shape:  # Ensure both
patches have the same shape
                        ncc = NormalisedCrossCorrelation(patch_left,
patch_right)
                        #print("Disparity:", d, "NCC:", ncc)
                        if ncc > max_ncc:
                            max_ncc = ncc
                            best_disparity = d

            disparity_map[y - half_window, x - half_window] = best_disparity

    return disparity_map


def normalize_disparity(disparity_map):
    # Determine the min and max values
    min_disparity = np.min(disparity_map)
    max_disparity = np.max(disparity_map)

    # Normalize values to the range [0, 255]
    normalized_disparity = 255 * (disparity_map - min_disparity) /
(max_disparity - min_disparity)

    return normalized_disparity

def convert_disparity_to_depth(disparity_map):
    # Simple conversion without calibration information
    # Assuming a baseline of 1 (unitless) and focal length of 1 (unitless)

    # Avoid divide by zero by setting zero disparities to a small non-zero
value
    epsilon = 1e-8
    disparity_map = np.where(disparity_map == 0, epsilon, disparity_map)

    depth_map = 1 / disparity_map
    return depth_map




def visualize_3d_mesh(depth_map):
    # Create meshgrid for X, Y coordinates
    y_coords, x_coords = np.mgrid[:depth_map.shape[0], :depth_map.shape[1]]

    # Create a 3D plot
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    # Flatten the arrays for plotting
    x_coords = x_coords.flatten()
```

```
    y_coords = y_coords.flatten()
    depth_map = depth_map.flatten()

    # Plot the 3D mesh
    ax.scatter(x_coords, y_coords, depth_map, c='r', marker='o')

    # Set labels
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Depth')

    # Show the plot
    plt.show()




if __name__ == '__main__':
    left_image_array = cv2.imread("im2.png", 0)
    right_image_array = cv2.imread("im6.png", 0)

    # Test with a 3x3 window
    window_size_3x3 = 3
    disparity_range =60

    disparity_map_3x3 = compute_disparity_map(left_image_array,
right_image_array, window_size_3x3, disparity_range)

    # Test with a 5x5 window
    window_size_5x5 = 5
    disparity_map_5x5 = compute_disparity_map(left_image_array,
right_image_array, window_size_5x5, disparity_range)

    depth_map3 = convert_disparity_to_depth(disparity_map_3x3)
    depth_map5 = convert_disparity_to_depth(disparity_map_5x5)

    # Display the results for comparison with colorbars
    # Display the results for comparison with colorbars
    plt.figure(figsize=(12, 6))

    # Left Image
    plt.subplot(1, 3, 1)
    plt.imshow(left_image_array, cmap='gray', interpolation='nearest')
    plt.title('Left Image')

    # Depth Map (3x3 Window) with Reversed Colorbar and Custom Labels
    # Display the results for comparison with colorbars
    plt.figure(figsize=(12, 6))

    # Left Image
    plt.subplot(1, 3, 1)
    plt.imshow(left_image_array, cmap='gray', interpolation='nearest')
    plt.title('Left Image')

    # Depth Map (3x3 Window) with Original Colorbar and Custom Labels
    plt.subplot(1, 3, 2)
    depth_map3 = convert_disparity_to_depth(depth_map3)
```

```
    img2 = plt.imshow(depth_map3, cmap='plasma', interpolation='nearest')
    plt.title('Depth Map (3x3 Window)')
    cbar2 = plt.colorbar(img2, ax=plt.gca(), label='Depth')
    cbar2.set_label('Depth', rotation=270, labelpad=15)
    cbar2.set_ticks([cbar2.vmin, cbar2.vmax])
    cbar2.set_ticklabels(['Far', 'Close'])

    # Depth Map (5x5 Window) with Original Colorbar and Custom Labels
    plt.subplot(1, 3, 3)
    depth_map5 = convert_disparity_to_depth(depth_map5)
    img3 = plt.imshow(depth_map5, cmap='plasma', interpolation='nearest')
    plt.title('Depth Map (5x5 Window)')
    cbar3 = plt.colorbar(img3, ax=plt.gca(), label='Depth')
    cbar3.set_label('Depth', rotation=270, labelpad=15)
    cbar3.set_ticks([cbar3.vmin, cbar3.vmax])
    cbar3.set_ticklabels(['Far', 'Close'])

    plt.tight_layout()
    plt.savefig('disparity_comparison_with_custom_colorbars.png')
    plt.show()
    #visualize_3d_mesh(disparity_map_3x3)
```