Implementing Depth-Dependent Blurring and Gradient Depth Sweep using Intel RealSense D435

Sally Shin, Ysatis Tagle

Boston University
Department of Electrical and Computer Engineering
8 Saint Mary's Street
Boston, MA 02215
www.bu.edu/ece

April. 5, 2023

Technical Report No. ECE-2023-4

Summary

In this project, we are performing refocusing on a series of images obtained from the Intel RealSense D435. Intel RealSense is a RGB-D camera that uses stereo-based depth sensing in order to construct a depth map. From the depth map, we want to segment the accompanying color image into similar depths in order to blur regions such as the background. The purpose of this project is to build an Image Processing pipeline in MATLAB, with stages including image registration, hole-filling, segmentation, matting, and blurring.

The main target of this project is to refocus using a two-level field of depth, foreground and background. The additional objective is to implement a 'depth sweep', where starting from a threshold value in the depth map, the blurriness increases with the distance or difference from the threshold. To achieve this effect, the depth map segmentations will be used to inform a gradient effect through use of blur filter techniques, such as Gaussian distribution-based depth blurring.

Using our implementation pipeline, we successfully reach our target and additional objectives of implementing selective blurring and gradient depth sweep blurring on our original RGB image captured from Intel RealSense. Each step of the image changes in the pipeline is documented in this report. We also discuss main issues of the implementation process such as occlusions, image registration, and jagged edges.

Contents

List of Figures

1 Introduction

For our project, we are interested in recreation of commonly implemented ‘Portrait mode’ using depth map images acquired from Intel Realsense D435. ‘Portrait mode’ is popularly used across multiple devices, where the foreground of an image is captured while the background is blurred. The main focus of the project will be to build a pipeline that achieves consolidation of similarly-depth objects and separation of those object depth while minimizing distortions at the boundaries of the depth layers. This will be done via a depth solution using two images captured from Intel Realsense, one using its camera mode and the other using its depth mode, and then to separate and selectively blur objects in different fields of depth. These images will be processed in MATLAB with the result of one image with a focused foreground and blurred background.

2 Literature Review

In this literature review, we will discuss advancements of refocusing applications, as well as depth map reconstruction techniques such as hole-filling for a more informative depth map.

Relevant research includes using the depth maps from Microsoft Kinect, which is a Time of Flight IR based depth camera. In Castellanos [1] , artificial refocusing was done using the Kinect depth map and a separately captured color image by alpha matting. The process was to capture the image, fill holes in the depth map, segmenting, aligning, matting, and refocusing. While the results do show refocusing, they conclude that false blurring comes from improper segmentation and aligning issues. This paper inspired our image processing pipeline, and serves as an introduction to challenges that will come up in our project. [1]

3 Problem Statement

Currently, 3D imaging and computer vision have become more immersed in the public sphere, where it has been deployed for autonomous vehicles, virtual reality, and

camera features such as the iPhone “Portrait mode”. 3D imaging captures not only color, but distance (depth) from a single scene. This is done using stereo vision or time-of-flight imaging. When depth maps are captured, they may be prone to holes, where depth information wasn’t captured due to occlusion, material absorbance, or external illuminance. Computer vision comes into play in reconstructing these images as well as applying the depth map for further use. Computer vision is the domain of teaching computers how to interpret images, such as how depth affects what an object looks like in an image and how to use a depth map to change an image. A key application within these two fields, computer vision and 3D imaging, is refocusing, where depth information is used to blur or sharpen various parts of the image.

Our project seeks to perform refocusing using the Intel RealSense D435 with MATLAB. We will capture both the color and depth information of a scene, and use the depth map to compute what should be blurred within a photo. This will allow for selective blurring based on the depth of a scene.

Since we are using the Intel RealSense, we are constrained to what images it is capable of producing. This means that the image sizes are only: 1920 x 1080, 1280x720, 848x480, 640x480, 640x360, 480x270, and 424x240 depending on the image mode. [3] This also means that the image formats are restricted to: Z, Y8, UYVY, and YUY2. We were also constrained to scenes of depths between ~.11 meters to 10 meters. [4] We are also constrained to the computational resources available to us. We ran the Intel SDK on Windows 11. This meant that the depth map and color images were not able to be taken in the higher resolutions of 1280x720 and 1920x1080.

Due to this implementation, there are a few assumptions made. For the data coming into the system, it is assumed that the data set is stationary, has consistent features, similar viewpoints and the same resolutions. This allows for easier image alignment and direct image manipulation based on features found in the data set. For the depth map specifically, it is assumed that the foreground and background are known and that there are no holes due to reflective materials, and good lighting conditions. Refer to Appendix A for images that fail such assumptions. Lastly, because we are using the Intel RealSense, we assume that the images coming in are from such. This comes with additional assumptions that the user has calibrated the RealSense correctly, and that they have

consistent camera settings. Refer to Appendix B for our Intel SDK settings. A crucial SDK setting that we assume is that the depth map is created with the left priority. [3] This allows us to assume the behavior of holes in the depth map, specifically of those created by occlusion.

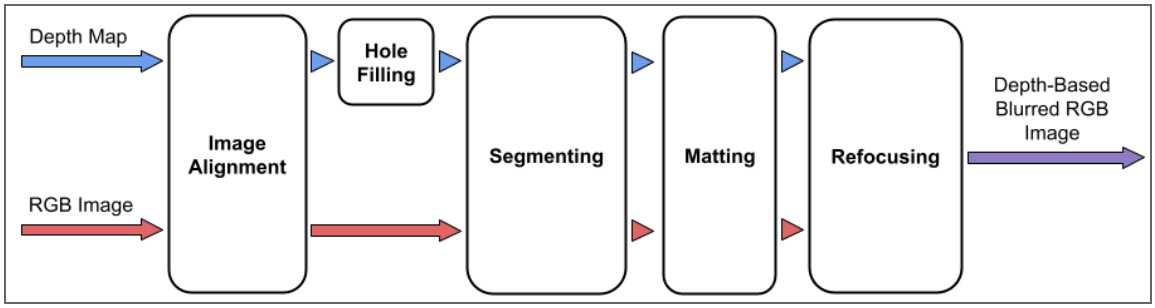

Figure 1. Simplified system diagram of Image Processing pipeline

In Figure 1, is the basic pipeline that we implemented for our project goals. The general concept of our solution is that there will be two inputs into the system, the depth map and the color image, and one output image, which is the depth-based blurred color image. There are a few processing steps that have to occur regardless of the final result of the project. The first is preprocessing. This includes image alignment and hole filling. These are done due to field of view differences between the two images and missing information in the depth image. Then, the project proceeds with segmentation and matting the depth map. This allows for depth segmented regions within the RGB image space. Then, when refocusing, the images will be processed together to blur the color pixels within each depth region. We also implemented an expansion of the refocusing algorithm, where, for multiple threshold values of the depth map (therefore more layers), the various areas in the image will be blurred with different radii.

4 Implementation

In Figure 1, we have defined the major components that will be implemented within our project. In this section, we will discuss each step of the implementation pipeline in detail.

4.1 Image Alignment Using Image Registration

This is the first step in our implementation. Due to the fact that the input images (RGB and Depth map) have different field of depth, the images must be aligned properly. This is done by using *cpselect* and *fitgeotform2d* from the MATLAB Image Processing toolbox via manual selection and automatic registration. First, *cpselect* is used to select points of similarity between the RGB and Depth map images. Then *fitgeotform2d* creates a geometric, linear transformation that uses the points of similarity as control point pairs. This is followed by *imref2d* function using the RGB image to set a point of reference to warp and then *imwarp* using the transformation points from *fitgeotform2d*, Depth map image, and reference point from RGB image.

While this image registration aligns images well generally, since fitgeotform2d performs a linear transformation, this methodology is not suitable for images that are not able to align well through a linear transformation only. For most images, this method will provide a suitable alignment for the blurring but images with planar surfaces would have better results with this image registration tool.

4.2 Hole-Filling Algorithm

The hole filling algorithm consists of two steps: the first is using *imfill*, the second is flooding any remaining holes with the left value.

The first step fills tiny holes in the depth map based on neighboring pixels. *imfill* by default assumes a first order neighborhood, and does a flood-fill algorithm, where the non-zero value of the pixel fills the neighboring zero values. [5]

Then, based on assumptions about the depth map and the Intel RealSense's settings, we can assume the remaining holes are either the invalid depth-band or from occlusion. [4] To fill these holes, first the 0 values in the image and their indexes are found using find. Then, we iterate through these values. If a pixel is in the leftmost part of the image (column 1), then we assign the value to 255. Otherwise, the pixel takes the color of the pixel to the left. In the case of a row of 0 values, the first pixel takes the color of the left, and this "floods" the row as the iterations grow. This is done since the

occlusion is based on the right as reference, therefore any occlusion holes must be the background therefore can take values from the left.

This step results in no zero values in an image, as they are all replaced with some existing value in the image or 255. The final step is to artificially add one hole at (0,0). This is done due to the fact that *imshow []* displays images by remapping the lowest and highest values in the image to 0 and 255. This was causing the depth image to display incorrectly through the code, and rather than change multiple lines, it is easier to add one. This also allows for better thresholding in later steps, as it guarantees at least one count of a pixel that is black in the depth map.

4.3 Segmentation

To segment the image into sections of interest, we threshold based on the values of the registered depth map. Because we know that the depth map is in grayscale, the pixel range is from 0 to 255. We also know that on the depth map, closer objects appear darker, with values closer to zero, while further objects are closer to 255. Based on this information, and also the distribution of the depth map values in a histogram, we can decide on a threshold value to divide the image into foreground and background. In our example results, we chose a middle value of 128 as our threshold.

The result after this step in the pipeline is a sharply divided foreground and background image, with clearly seen jagged edges where the image ends.

4.4 Matting

To smooth the edges of the segmentation, we perform matting. Matting allows for us to smooth out the boundaries without having the two sections appear clipped and unnatural. When we combine the image later in the pipeline after refocusing parts of the image, matting helps the boundary between the images appear smoother and more natural. For these reasons, matting is an essential step in our pipeline to achieve the result that is more natural and aesthetically nicer.

To perform matting, we first convert the segmented image into a binary image of 1s and 0s. At this step, we take the segmented image that contains parts of the image that

we are interested in blurring later. So if we are interested in blurring the background, the segmented image with the background is used. After converting the image into 1s and 0s, we blur this matting image with a Gaussian blur, making the edges appear less jagged. Before the Gaussian blur, the image will have sharp changes from 1s to 0s but after blurring, this edge is softened and more gradual. This step is performed with *imgaussfilt* function, with standard deviation of 4. We can then use the resulting blurred mask to define the softer boundaries between the foreground and background.

4.5 Refocusing Algorithm - Single Blur

For a simpler refocusing algorithm, we first implement a single blur effect on either the background or foreground using the matting image and the RGB image. To blur selective parts of the image, we create a Gaussian blur filter using a chosen blur radius of 4 and for each image channel, we blur the original RGB image with our filter and multiply the result by the matting image, which have values > 0 in the area of blur interest. We also multiply the original RGB image with the complement of the matting image, which zeroes out the area of the image that will be blurred. We then add the two images together to get our selectively blurred image.

To blur the background, we take the segmented image of the background and use it as our matting image and the opposite to blur the foreground.

4.6 Refocusing Algorithm - Gradient Depth Sweep

For the refocusing algorithm for gradient depth sweep, we utilize three different images, the matting image of the to-be-blurred area, the RGB image, and the registered depth map. We implement a depth map-dependent blurring technique by first defining a vector that ranges from threshold to 0 (descending order) or from threshold to 255 (ascending order) depending on the to-be-blurred area. For blurring the background, we use a vector in ascending order at an increasing step of 10. For example, if the threshold is 128, the resulting vector is [128, 138, 148, …, 238, 248, 255]. For either vector, the ending values, 0 and 255 are added so that the vectors can encompass the entire range from threshold to 0 or from threshold to 255.

For every consecutive range of values within the vector, (which would be from 128 to 138, then 138 to 148, so on until the entire vector is spanned) we initialize a matrix to be a blur mask using zeros function, using the same size as our RGB image. Using the range of depth map values in between each index of the vector, we can define our blur mask to be 1 only if at that index, the registered depth map is within the vector values of interest and the values of the matting mask from our previous step is higher than 0. With these restrictions, we define the blur areas to be constrained by both the depth map values and the matting mask, which then allows us to recompose the differently blurred images back together successively in a for loop. The mask is also blurred at the edges using *imgaussfilt*, with a sigma value of 10 so that the edges between the blurred images do not show in the final image.

To perform blurring, we make a convolutional Gaussian filter based on a set radius of how much the image should be blurred - this radius value is changed in a for loop so that areas farther from the threshold, ie. 0 or 255 depending on area to be blurred, are blurred with a higher radius, which implements a gradient-like blurring effect. Then, we blur the original RGB image with the blur filter and then multiply by the mask. This results in removal of any parts of the blurred image that are zeroed in the mask. Then, we multiply the original RGB image by the complement of the mask. This removes parts of the image that are zeroed out in the complement. Then we add these two images together to get our selectively blurred image. This process is performed successively in the for loop so that at each step, more of the image is included in the blurred area and at the final iteration, the entire area of blur interest is covered.

5 Results

In this section, we will demonstrate an example of the results obtained from each step in our implementation pipeline. For our results, we used a subjective testing methodology to evaluate the images based on their quality. In particular, we looked at image warping and distortion, smoothness/jaggedness of the image, and object edges. These three things are crucial for verifying image registration, segmentation, and our refocusing results. We looked at images in full size for an overall sense of its quality, as well as zoomed into "problem" regions, such as where matting takes place to evaluate the results. These subjective measures were crucial to verify implementation effectiveness and gave us the ability to test results via perceived realism, which is hard to quantify for refocusing images.

From the Intel Realsense D435 camera, we use stereo and RGB options to capture the images below using Intel's software on Windows 11. Both images in the example are of 480X640 resolution due to hardware limitations. For depth map image capturing, the IR sensor was turned on, and the included hole filling algorithm was also turned on in the software.

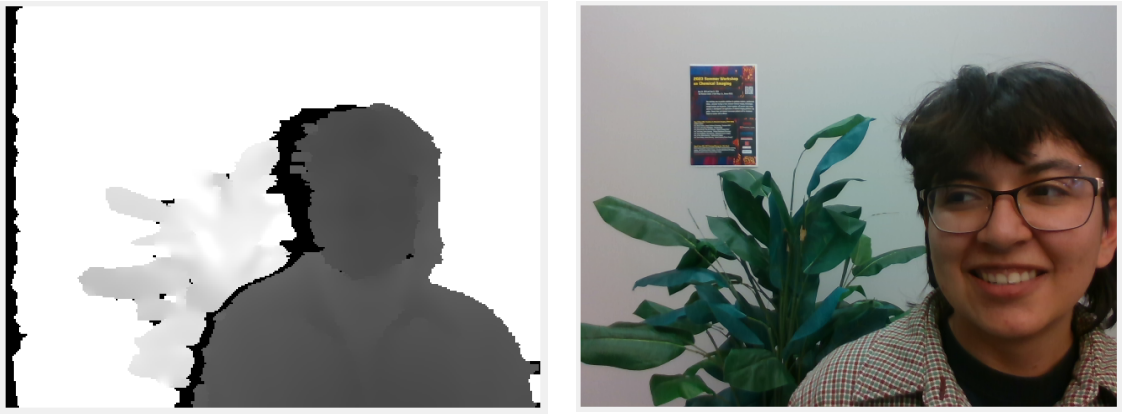

Figure 2. Original Depth Map and RGB Image

Next, we align the two images using cpselect to select for points of similarities between the two images. In the below example, 8 different points were used to create a linear transformation mapping between the images using *fitgeotform2d*. Followed by warping of the depth map image to the RGB image, the depth map is registered to the

RGB image. The resulting overlay between the registered depth map and RGB image is also shown below.

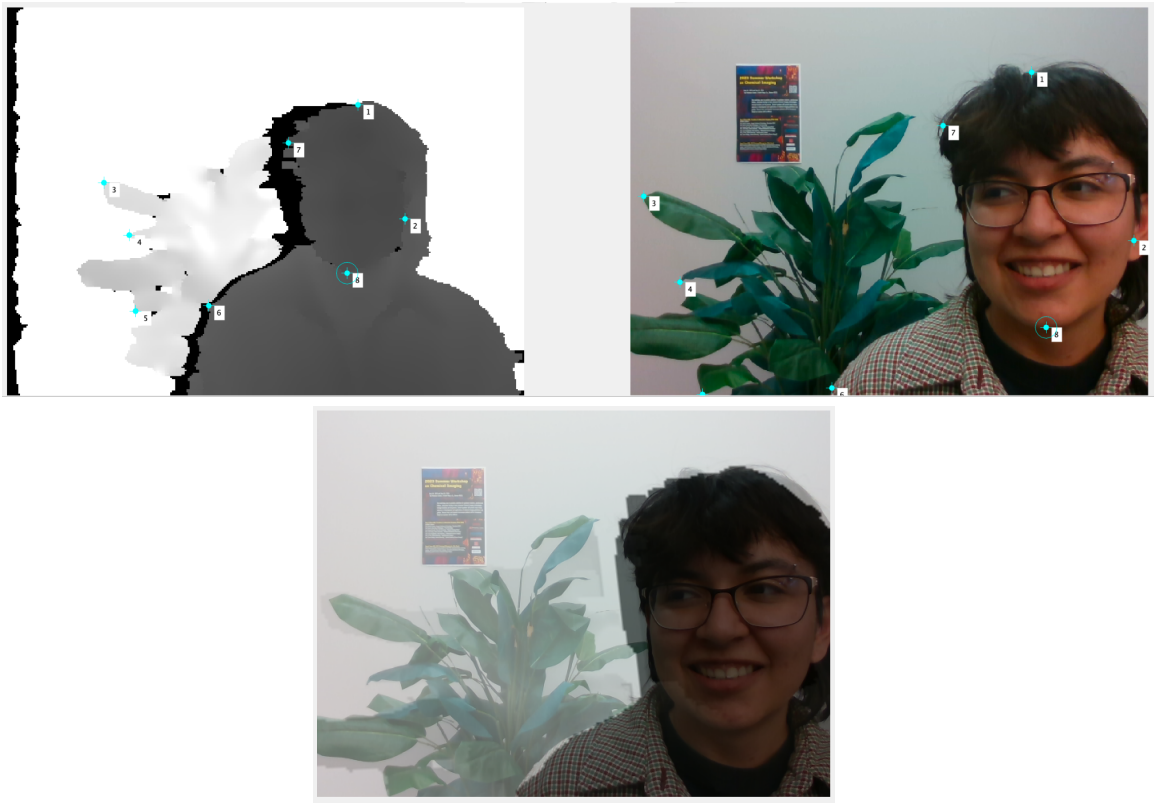

Figure 3. Selecting Points of Similarities and Resulting Image Registration Overlay

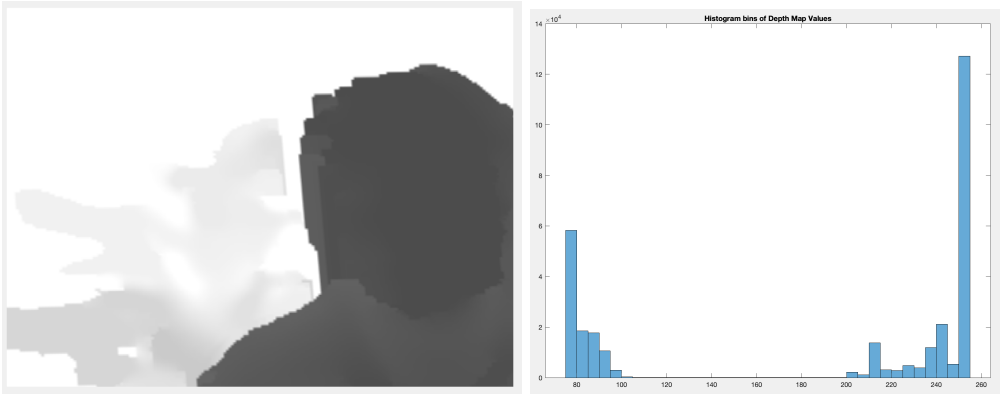

Figure 4. Hole-filled Depth Map and Corresponding Histogram

From the registered images, we then apply the hole filling algorithm to the depth map. We next segment the images based on the values of the depth map histogram and choose 128, a middle value in the depth map range of values as our threshold. The two segmented images here show jagged edges at the boundaries since segmenting by the threshold is a binary decision.

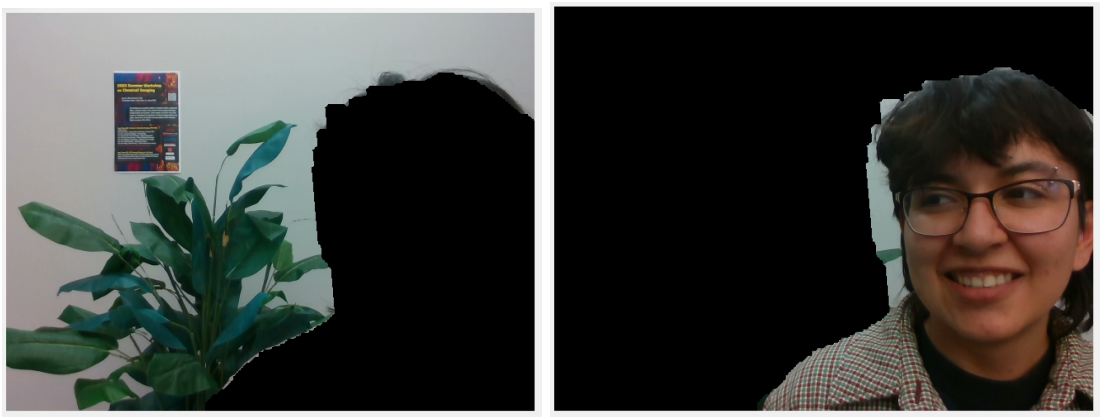

Figure 5. Segmented Images of Foreground and Background

Next, we take a segmented image of the area to be blurred and convert all its values to 1s and 0s to create a matting image. For black areas of the segmented image, it remains at 0 and for any other areas, the matting image has value of 1. We then blur the resulting matting image using a Gaussian filter of sigma value 4 to soften the edges. We use *imcomplement* or (1 - matting image) to find the complement of the matting image, which will have complementary values to the matting image.

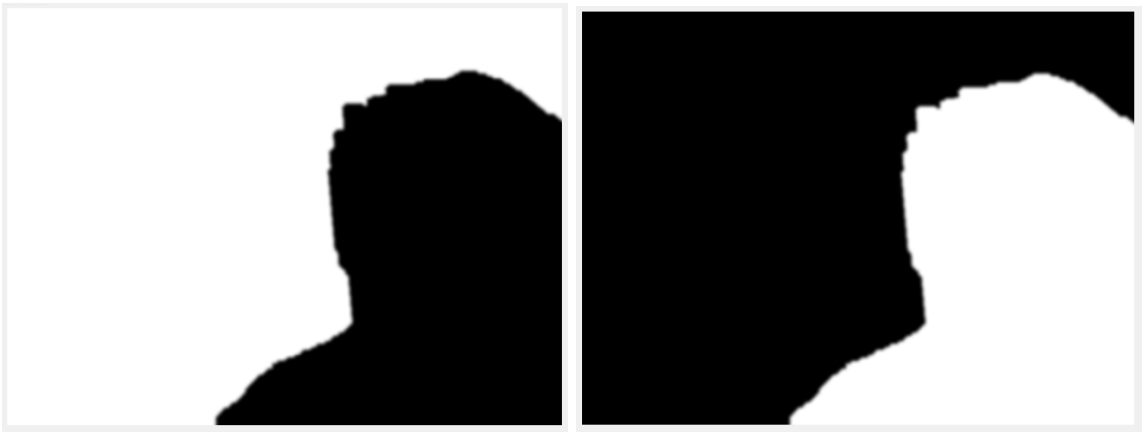

Figure 6. Matted Image Results from Segmented Background and its complement

Finally, to refocus parts of the image we are interested in blurring, for a single blurring effect, we blur our original RGB image by a Gaussian blur filter with chosen radius of 2 and multiply the resulting blurred image by our matting image. We then multiply the original RGB image with our complement of matting image and add the resulting 2 images together to form the final refocused image. Depending on which segmented image is used for matting, different areas of the image can be blurred.

Figure 7. Refocused Image Result of Blurred Background

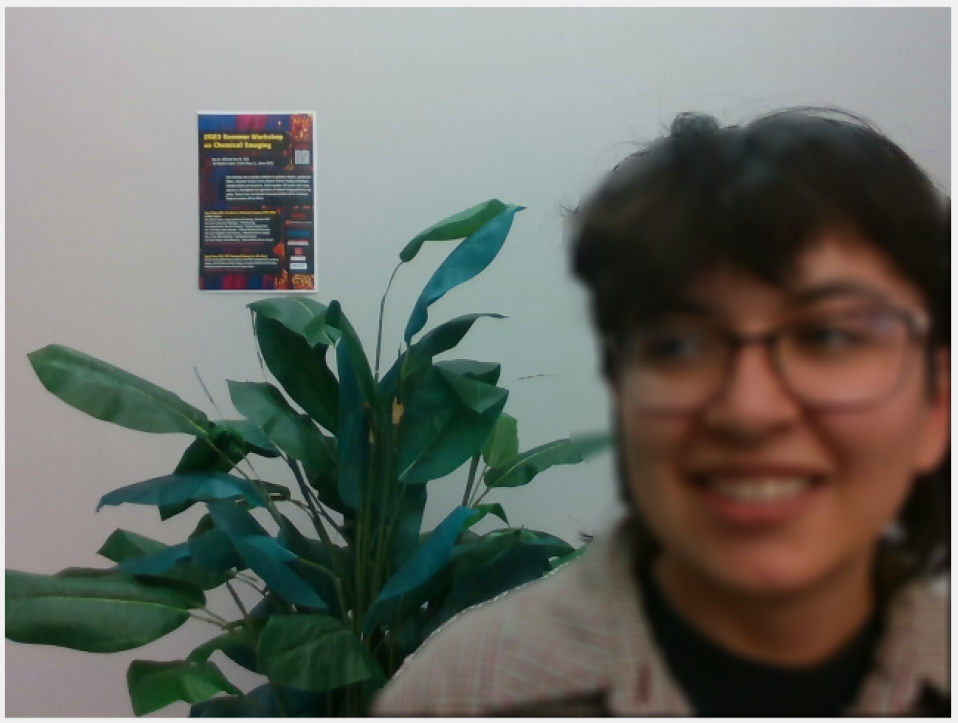

Figure 8. Refocused Image Result of Blurred Foreground

5.1 Gradient Depth Sweep

As an additional goal of our project, we implemented a gradient depth sweep effect in our blurring pipeline to blur objects at different depths differently. In our implementation, we chose specific threshold bins that contained different depth map areas of the image and in a for loop, we blurred only that part of the image and recomposed the image together every time. As we move further from the threshold (the middle value between foreground and background), we increase the blur radius, which blurs that area of the image more. The resulting series of blurring is below, where each step shows different areas of the image blurred until the entire background is blurred. At each step, our mask is also blurred at the edges so that the blurred edges blend together seamlessly.

Figure 9. Progressive Blur Demonstration of Gradient Depth Sweep

Figure 10. Final Result of Gradient Depth Sweep Implementation

5.2 Matting Implementation

Below we show an example of why matting is a necessary step in our implementation and how it contributes to more realistic results after selective blurring. In the image without matting, at the edges of the shoulder, it shows jagged edges between the blurred and clear areas, which makes the image less natural. When we implement matting in our pipeline, the boundary between the blurred and clear areas are much smoother, and results in both more natural and aesthetically pleasing results.

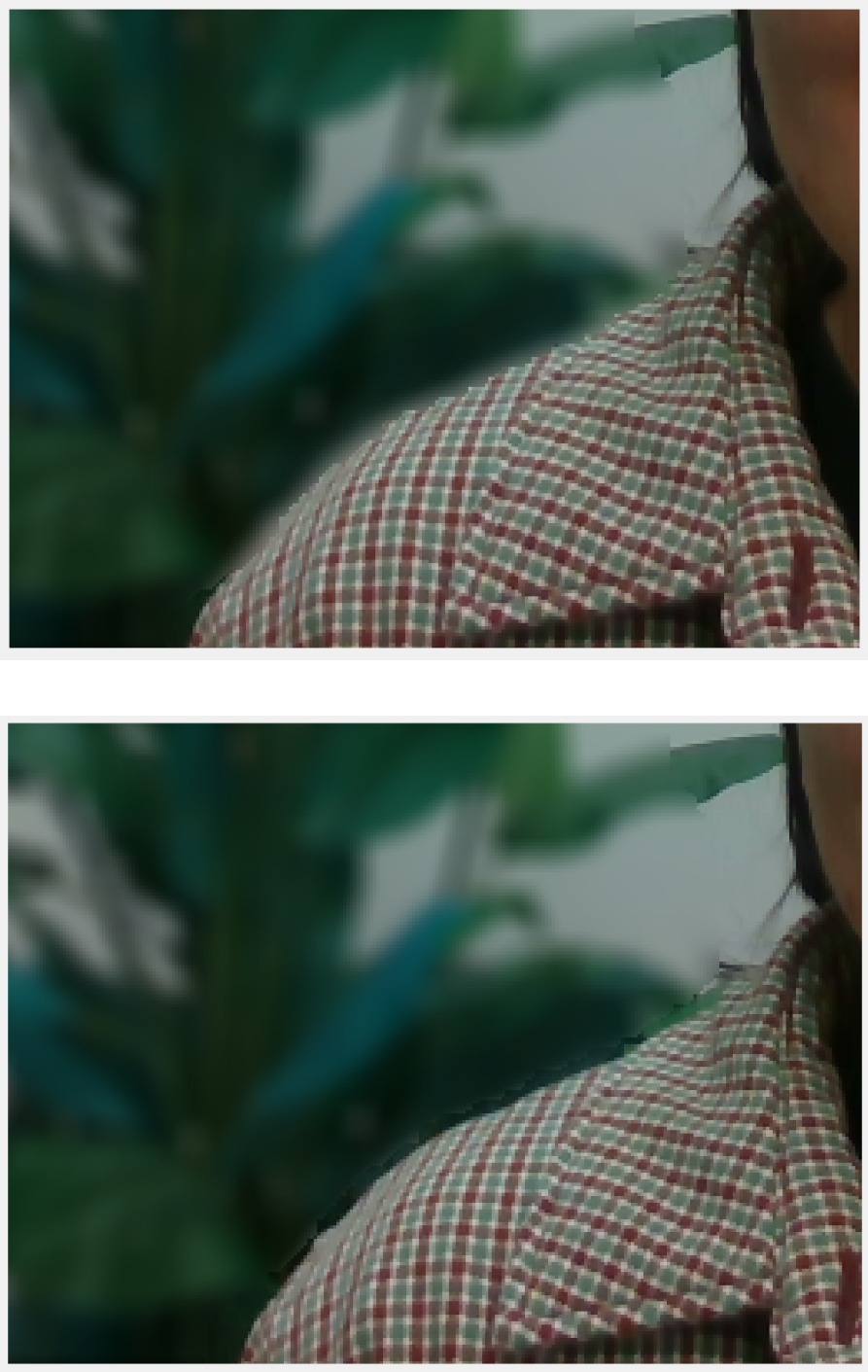

Figure 11. Refocusing Results without (Top) and with Matting (Bottom)

6 Conclusion

From our results, it can be concluded that image refocusing is possible using this pipeline of image registration, hole filling, segmentation, matting, and blurring within MATLAB from photos obtained by the Intel RealSense D435.

A few challenges we ran into are issues with alignment of the RGB and depth module. Since the captured images were of different depths of field, we had to register the images in reference to the RGB image before starting any implementations. Most standard methods of image registration did not work with our set of images since they differed greatly in their features (color, object shape, depth of field). We used manual selection of points of similarity to register the images and at different areas of the image, the alignment was still not perfect due to non-linear aspects of some objects. Since the image registration tool we used only performed linear transformations between the images, any parts that were non-linear were not matched as well as they could be. Because the images are not able to be perfectly aligned, the resulting blur areas are not as aligned to the foreground and background areas, which caused visible effects in our final result. Another major challenge shown in our results is issues due to the depth map imperfections, which resulted in the blur boundary issues. This is due to assumptions about the depth map coming in as well as the Intel RealSense limitations.

Further work for this project includes improving the hole filling algorithms from the neighbor-based *imfill* and occlusion-filling flood technique, to something more robust and based on the color image. For some ideas, refer to Appendix C.

Overall, in this project, we implemented a pipeline to use a depth map image and its RGB image to selectively blur specific parts of the image successfully. This included implementation of a hole-filling algorithm that can deal with occlusion-based holes. We've also additionally, achieved our target goal of implementing a gradient depth sweep that blurs areas differently depending on its depth map values. Through this project, we've learned to work with 2D signals. We also gained more confidence in manipulating images and using the MATLAB Image Processing Toolbox.

Appendix A

This appendix item is about images we took that justify our assumptions. This is to serve as an example of what we looked to avoid when implementing our solution.

In Figure 12, we see the depth map and color image taken with backlighting and windows. This resulted in a troublesome depth map due to the reflective surfaces and lighting, where the hole-filling algorithm failed to properly fix the depth map. In Figure 13, we see a color image, infrared, and depth map of a box. This resulted in image alignment issues since the depth map has an ‘unknown’ foreground, where the box is blended into the floor. The infrared, while more informative for depth, also caused alignment issues due to the dots not existing in the color image (non similar features).

Figure 12. Depth Map and Color Image from Improper Scene

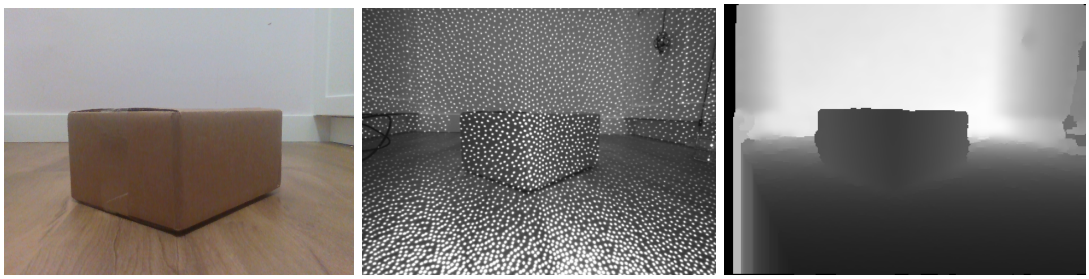

Figure 13. Color, Infrared, and Depth of Box

Appendix B

Here we have listed the settings that we used to capture images with the Intel RealSense D435 using Intel SDK.

- Default configuration with addition of:
    - Resolution of 480X640 for both images
    - Hole filling algorithm checked for depth map camera
    - Infrared laser checked for depth map camera

Appendix C

This appendix section is a brief literature review of topics that may be of interest for future work. This includes various hole-filling algorithms that we identified as possibly relevant for improved performance.

Hole-Filling Algorithms

One issue of the Kinect is that it generates noisy, error prone depth maps, which means that hole-filling is a crucial pre-processing step. [1] There are two major types of hole filling algorithms, one which refers to a color image or one that relies solely on the depth map. [8] For this project, we decided that, due to the amount of holes as well as the fact that the largest ones result from occlusion, a color-based hole filling algorithm is most relevant. This accuracy comes with the cost of algorithmic efficiency. [8] There are two types that we explored within our literature review, which is interpolating/filtering based and reconstruction based hole filling methods. They have their tradeoffs: filtering can cause ringing and artifacts but are easy to implement while reconstruction is more accurate with the cost of difficult implementation and higher complexity. [8]

In Yang et. all [6] , a Kinect-generated depth map is patched by interpolating with cross-bilateral filtering from the color image and neighboring depth information. The holes are first categorized within certain regions and their source. Within a region, the holes are dilated so that the depth information is spread. This is because it is assumed that the correct depth value is within the region. This then allows for the holes to be labeled with the depth information from neighboring pixels and thresholding. This produces the first iteration. Then, the cross-bilateral filtering is applied to refine the depth map so as to clean object edges. However, like earlier mentioned, this method is limited since it can't work on large regions and just blurs depth information in these locations. [6] This is meant for large holes caused by occlusion, thus may not be effective.

An alternative restoration-based hole filling method is proposed in [7]. This relies on a newer model of the Kinect that is RGB-D capable. Since RGB-D captures the same scene, there is a correlation between the color image and depth map. This means that a local linear regression model can be used to find the depth values from the color image

per pixel. From this, an energy function can be derived, where known depth values can inform the missing depth values. By minimizing the error energy, we can fill the hole values. This report also adds a total variation penalty, TV21, which both preserves edges of the color image as well as removing noise in newly filled regions. A benefit of this is that it is easy to implement for occlusion, but it is unknown if it can fill large holes. [7]

In [6], the Intel Realsense is used, which means it is more relevant as the source of holes is similar. Since a lot of the holes are due to occlusion, thus obstructing object/background boundaries, they use an edge map derived from the color image to inform how to fill the holes. First, the edge map is made, then the holes are classified on whether it is a background or an object hole using the hole border average and standard deviation. A background hole is filled with the average, while the object one is more complex. The object hole is made ‘solid’, removing any interior depth pixels or protrusions from the object edge first. The direction of filtering is then ‘alternated’ from the hole border towards the object edge. Then, the rest of it is filled with informed average values. [6] This seems the most promising and relevant of the papers we read, but the others may be easier to implement, so we seek to explore all three methods as possible hole-filling algorithms.

References

[1] C. Castellanos and A. Nguyen, “Artificial Refocusing of High Resolution SLR Images From Microsoft Kinect Depth Maps,” Stanford, 2017 [Online]. Available: https://stanford.edu/class/ee367/Winter2017/Nguyen_Castellanos_ee367_win17_report.pdf

[2] F. Moreno-Noguer, P. N. Belhumeur, and S. K. Nayar, “Active refocusing of images and videos,” ACM Trans. Graph., vol. 26, no. 3, p. 67, Jul. 2007, doi: 10.1145/1276377.1276461. [Online]. Available: https://dl.acm.org/doi/10.1145/1276377.1276461. [Accessed: Mar. 07, 2023]

[3] “Intel® RealSenseTM Product Family D400 Series Datasheet.” Intel, Nov. 2022 [Online]. Available: https://www.intelrealsense.com/wp-content/uploads/2022/11/Intel-RealSense-D400-Series-Datasheet-November-2022.pdf

[4] “Intel® RealSenseTM Depth Camera D435 - Product Specifications,” Intel. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/128255/intel-realsense-depth-camera-d435/specifications.html. [Accessed: May 05, 2023]

[5] “Fill image regions and holes - MATLAB imfill.” [Online]. Available: https://www.mathworks.com/help/images/ref/imfill.html. [Accessed: May 05, 2023]

[6] N.-E. Yang, Y.-G. Kim, and R.-H. Park, “Depth hole filling using the depth distribution of neighboring regions of depth holes in the Kinect sensor,” in 2012 IEEE International Conference on Signal Processing, Communication and Computing (ICSPCC 2012), Aug. 2012, pp. 658–661, doi: 10.1109/ICSPCC.2012.6335696.

[7] S. Liu, Y. Wang, J. Wang, H. Wang, J. Zhang, and C. Pan, “Kinect depth restoration via energy minimization with TV21 regularization,” in 2013 IEEE International Conference on Image Processing, Sep. 2013, pp. 724–724, doi: 10.1109/ICIP.2013.6738149.

[8] A. Atapour-Abarghouei and T. P. Breckon, “A comparative review of plausible hole filling strategies in the context of scene depth image completion,” Computers & Graphics, vol. 72, pp. 39–58, 2018, doi: 10.1016/j.cag.2018.02.001. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0097849318300219. [Accessed: Apr. 06, 2023]

[9] J.-M. Cho, S.-Y. Park, and S.-I. Chien, “Hole-Filling of RealSense Depth Images Using a Color Edge Map,” IEEE Access, vol. 8, pp. 53901–53914, 2020, doi: 10.1109/ACCESS.2020.2981378. [Online]. Available: https://ieeexplore.ieee.org/document/9039648/. [Accessed: Apr. 06, 2023]

[10] R. Liu et al., “Hole-filling Based on Disparity Map and Inpainting for Depth-Image-Based Rendering,” IJHIT, vol. 9, no. 5, pp. 145–164, May 2016, doi: 10.14257/ijhit.2016.9.5.12. [Online]. Available: http://gvpress.com/journals/IJHIT/vol9_no5/12.pdf. [Accessed: Apr. 06, 2023]

[11] Q. Chen, D. Li, and C.-K. Tang, “KNN Matting,” IEEE Trans. Pattern Anal. Mach. Intell., vol. 35, no. 9, pp. 2175–2188, 2013, doi: 10.1109/TPAMI.2013.18. [Online]. Available: http://ieeexplore.ieee.org/document/6409354/. [Accessed: Mar. 07, 2023]