Occlusion Handling in Augmented Reality

CSE 493V Final Project Report

DEVESH SARDA and NEEL JOG, University of Washington

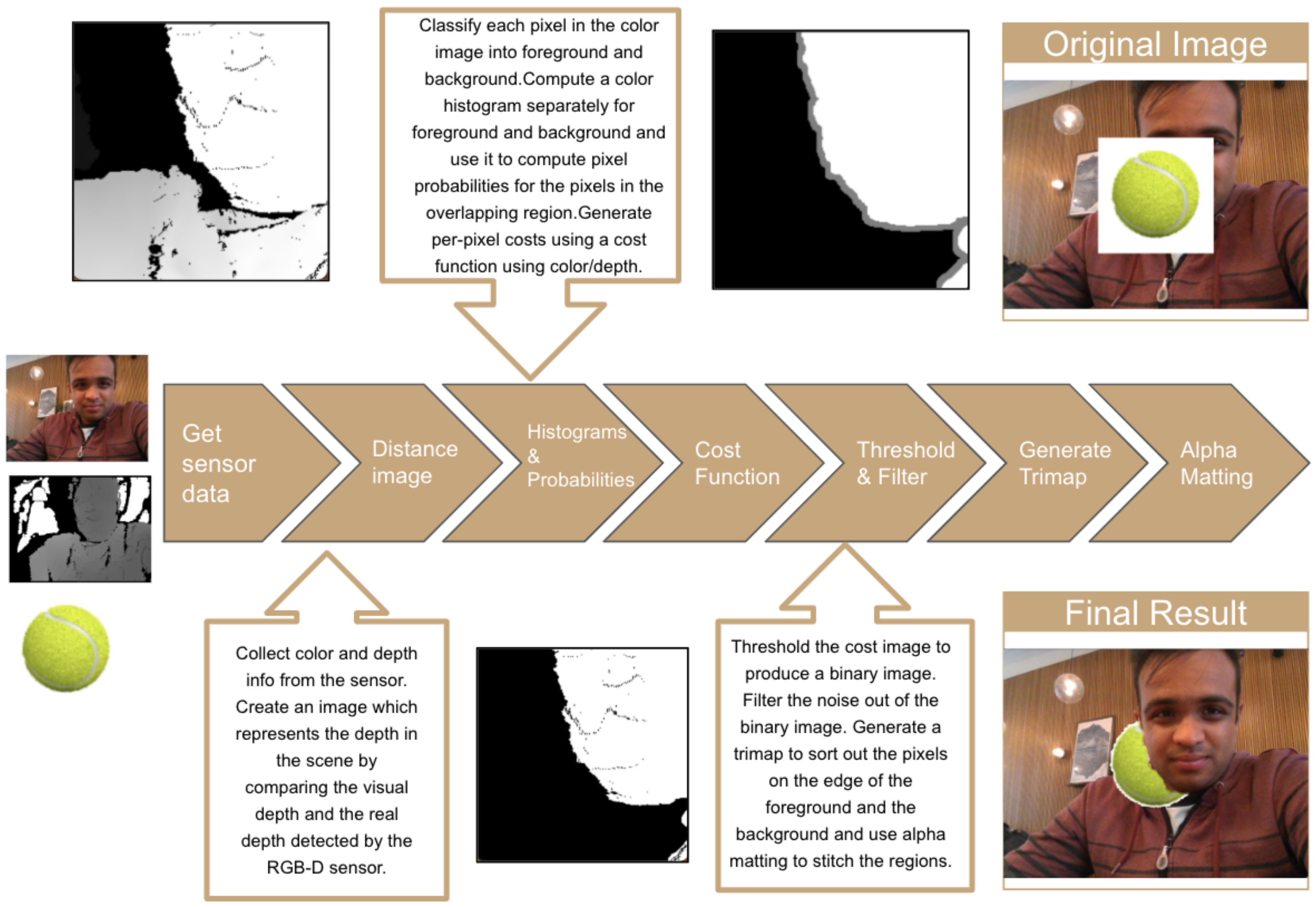

Classify each pixel in the color image into foreground and background.Compute a color histogram separately for foreground and background and use it to compute pixel probabilities for the pixels in the overlapping region.Generate per-pixel costs using a cost function using color/depth.
Original Image
Get sensor data
Distance image
Histograms & Probabilities
Cost Function
Threshold & Filter
Generate Trimap
Alpha Matting
Collect color and depth info from the sensor. Create an image which represents the depth in the scene by comparing the visual depth and the real depth detected by the RGB-D sensor.
Threshold the cost image to produce a binary image. Filter the noise out of the binary image. Generate a trimap to sort out the pixels on the edge of the foreground and the background and use alpha matting to stitch the regions.
Final Result

Fig. 1. The following is the pipeline that was followed to implement basic occlusion in the context of Augmented Reality. This pipeline is a snapshot of the entire rendering process, from getting the digital, depth, and virtual images to rendering the final blended image. Each stage of the pipeline was implemented primarily using Python and the final results of the occlusion were streamed to an Oculus Quest 2.

Abstract: One of the most important challenges to tackle in Augmented Reality is the interaction between real and virtual objects, especially in the context of occlusion. Depth perception is one of the most important factors affecting the realism of a scene, so it is very important to properly blend virtual and real objects that are occluding each other as inaccurate occlusion can dramatically impact a user's sense of depth, thus making the AR experience non-immersive. Generally, a depth sensor is used to tackle the problem of determining the depth of real objects in the scene, but simply relying on the depth sensor can lead to incorrect results because of the significant noise in the sensor's readings as well as the effect of other environmental factors like lighting, especially in outdoor conditions. The method that we implemented for our final project built on top of the typical depth-sensor approach by using the color distribution information of the real scene to calculate the probability of each pixel being in the foreground or the background and using those probabilities to determine what part of the virtual object to render. We also implemented smoothing methods such as trimap-generation and alpha matting to realistically blend the foreground and background and thus create a seamless render. Such a probabilistic pixel-based approach allowed us to effectively implement occlusion (albeit not at a very good frame rate) even in cases where multiple objects were occluding different parts of the virtual object. The code for our project can be found here.

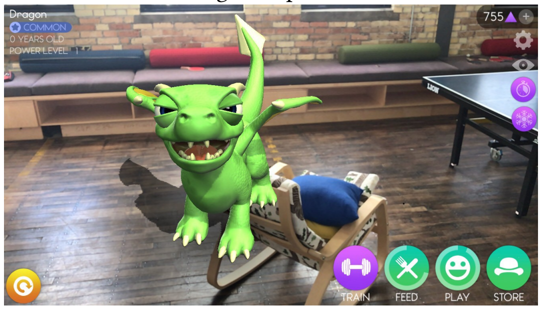
Authors' address: Devesh Sarda, devess@cs.washington.edu; Neel Jog, njog02@cs.washington.edu, University of Washington.

1 INTRODUCTION

The primary goal of AR is to blend the virtual object/image with the user's real world in a seamless, efficient, and realistic manner. While there are many scenarios that affect how realistic a scene appears to the users, one of the most prominent ones is depth perception. A user's depth perception not only allows the user to localize their own position in the scene but also helps the user gauge how to interact with the objects in the scene. A skewed depth perception can not only have aesthetic implications for the scene but can completely alter how the user engages with the virtual/real objects in the scene. One of the biggest indicators of depth in a scene is occlusion.

Occlusion happens when an object that is closer to the viewer partially blocks the viewer's sight of another object that is farther away in the scene. This is one of the ways that humans can tell which objects are closer to them than another object.
Consider the following example:

In this image, the primary way in which we know that the seated person is closer to us than the lake is because the person is partially occluding the lake. If it was the lake that was closer to us than the person, then it would be the person that was occluded by the lake and not the other way around. While this might seem like a fairly straightforward concept, it plays special prominence in the world of Augmented Reality when virtual objects are placed in the real world at a certain depth and those virtual objects have to interact with objects in the real world. It is easiest to understand the importance of occlusion by considering an example in which occlusion is not properly handled.
Consider the following example:



In this case, it appears that the dragon is located in front of the chair because the dragon occludes the chair; however, we can tell from the shadow that the dragon is intended to be behind the chair. Improper occlusion handling can thus lead to skewed spatial properties which in turn can spoil a user's AR experience.

While it is tempting to think that simply using a depth sensor can solve the problem of occlusion, the noise in sensor data – as well as the variation in the environments that AR is used in – makes using depth data from depth sensors an incomplete approach at best. For our final project, we decided to explore implementing a method that built on the values from the depth sensor and used statistics and probability to handle occlusion and create a realistic AR experience. Even though our method processed the data differently, we used only a singular RGB-D camera. The specific ways in which we processed the depth sensor data are discussed in the "Method" section, but overall, our method is unique from other methods in that it uses both values from the depth sensor as well as the RGB values to determine attributes that can help determine which pixels belong in the foreground and which pixels belong in the background. Unlike other commonly used methods for dealing with occlusion, the method we researched assigns per-pixel probabilities based not only on the color of the surrounding pixels but also based on the general colors that are a part of the foreground and background categories (along with depth values, of course).

There are a couple of things to note, however. In general, there are several kinds of occlusions that can occur in an AR setting: real objects can occlude other real objects, real objects can block virtual objects, virtual objects can block real objects, and virtual objects can block other virtual objects. Given the time and hardware restrictions we had for the project, we decided to focus on only one of the above cases: when a real object blocks a virtual object. Another caveat of our method was that our data processing was fairly slow compared to certain other AR applications. This limitation stemmed primarily from limited computing power; since we did not have a GPU, we had to rely on slow frame-by-frame integration of virtual and real objects. However, in spite of these caveats, we were able to successfully use RGB and Depth data to accurately implement occlusion in several different types of environments.

1.1 Contributions

At a high level, we view our project as a super basic tutorial for anyone interested in the occlusion to get themselves familar with the field but also a framework that others can easily adapt and build off as they see fit.

- We introduce an overall pipeline to read capture images and read depth data from a RGBD, occlude a virtual object with the capture image, and to stream the result to a virtual reality headset which can easily replicated using our open source software.
- A synthesis of various computer vision algorithms and techniques to create a robust and reliable approach for occlusion of virtual objects that can be easily customized and adapted if needed in the future.

2 RELATED WORK

In this section we are going to be providing a brief background on each step in the pipeline as well as justification for how we implemented each step of the pipeline based on existing literature.

2.1 Capturing Camera Data

We utilized Intel's RealSense D415 camera which is capable of capturing both depth data alongside capturing video data. Additionally, while the depth and camera sensors are located in different places on the device, their provided SDK automatically merges the two data streams so that they appear to come from a single POV.

2.2 Distance Image Generation

Previous works [5], and [6] have shown that most commercial depth sensors suffer from some common issues including random holes in the middle of objects and randomly depth values for nearby pixels around the edges of objects. These works have also show that this problems worsens the farther the object is from the actual camera. Furthermore, [7] shows that poor lighting conditions and the presence of shadows significantly magnifies the noise in the depth readings. All of these effects results in depth sensors generally giving larger depth readings for non foreground objects than they actually are. As we discuss in the next section, we remedy this by introducing a exponential dropoff for our estimate of the depth of the non foreground regions of an image.

2.3 Probability Generation

Researchers at Microsoft showed how foreground and background can be done using both depth and image data in [8]. They found that converting the image to the HSV color space from the RGB color space and then performing analysis on it produced better overall results. Additionally, while perform the above algorithm using all three channels they also found using just the Hue channel didn't significantly decrease the accuracy of the system. For each frame, they generate seperate color histograms for the estimated foreground and estimated background and use those histograms to generate a probability for each pixel in actually being its assigned region. Specifically, this probability is based on the number of other pixels in the estimated region having the same color as [9] showed that dominant colors in the estimated foreground/background end up being the dominant colors in the actual foreground/background. We use these probabilities with the distance image to get an images reflecting the probabilities of each pixel actually being in the region we initially determined. Specifically, do a weight sum to combine the probabilities from the depth data and the image data and [10] showed that for most such sensors assigning equal weight generally produces the best results. Finally, we apply a cost function like the one used in [2] to get a singular probability measure of each pixel in the foreground.

2.4 Filtering

Now that we have a cost value associated with each image we need to determine which value represents the threshold between the foreground and background. To do so, we utilize the dynamic threshold calculation technique proposed by Ridlers and Calvard in [3]. However, the result produced by this adaptive threshold is quite noisy around the edges and we resolve this issue by using a guided filter approach proposed in [11]. The algorithm in [11] uses the original image captured from the camera as a "guide" in order to smooth the noise around the edges while still preserving the edge countours. For the sake of the project we utilize [12] which builds upon [11] but is faster than [11] using subsampling techniques based on the property of neighboring pixels in images having relatively similar colors.

2.5 Image Matting

Before we perform image matting, we need to generate a trimap that can be used by the alpha matting algorithm. We utilize the trimap generation algorithm proposed in [13] as it optimizes its trimap generation algorithm to get the most accurate image matting. The primary challenge in this step is to convert our RGB image to a RGBA where the A or alpha channel can be used to blend the virtual and real images as done in [14]. A wide variety of techniques exist to perform blending including the ones shown in [15][16][17][18]. We will dive deep into the approach we choose and why in the later sections.

3 METHOD

The method that we researched and implemented can be broken down into a pipeline similar to the one described in the teaser image on the first page. In general, the core idea of our method revolves around using the RGB data on top of the depth data received from the depth sensor to improve the initial/background segmentation. A trimap is then used to calculate the alpha value to be used for alpha blending. This step is followed by detecting pixels on the edge of the foreground and the background to ensure a smooth transition, and then using the alpha value to create an image in which the virtual world overlaps with the real world. Each step is described in more detail below. This entire methodology is a direct implementation of the work of Simona Gugliermo [1] whose Master's dissertation was the inspiration for our entire methodology.

3.1 Initialization

The first step in the pipeline was to simply collect the RGB-D data from the sensor and pre-process that data for the next steps in the pipeline. Because the RGB and D sensors are two different sensors located on the same device, they collect data about the same scene but from slightly different views. We thus had to first align the depth and the color data that we were receiving from the sensor to ensure that the features we needed would line up well. Fortunately, we did not have to do a lot of computation to calibrate the sensors in such a way, because the device already had pre-programmed functionality that enabled us to correct the depth map and the color map. After completing this calibration, our next step in this stage was to create an initial render of the virtual object (in our case, a tennis ball) as well as to create a binary mask for that virtual object. In the binary mask, a pixel was given the color black if the pixel was a part of the actual tennis ball, and the pixel was given the color white otherwise. The sole purpose of making such a binary mask, as seen in Figure 2, was to use it in future steps as a filter to get all the pixels that were relevant for the occlusion and thus save some computation time. The colored version of the virtual image was made directly using the color values that were a part of the tennis ball.

3.2 Distance Image

After gathering all the needed pre-processed RGB-D data and the virtual object and its binary mask, the second step in the pipeline was to create a distance image, i.e. a greyscale image representing the depth in the scene such that pixels that are in the foreground of

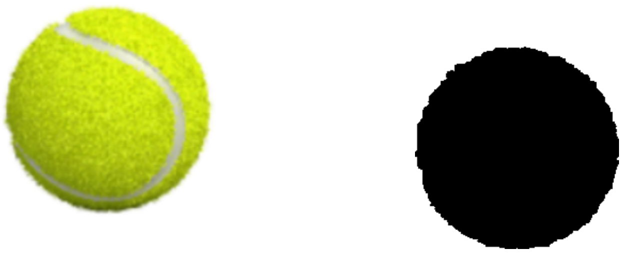


Fig. 2. The virtual object that we want to blend with our incoming video stream and the associated binary mask

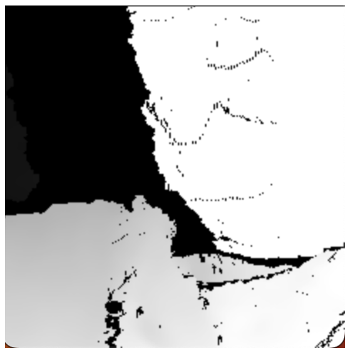


Fig. 3. Distance Image generated using depth data

the virtual object are colored white, and the pixels that are a part of the background are colored with different shades of grey depending on their depth (as measured using the depth sensor) and how that depth related to the pre-assigned depth of the virtual object. This portion of the pipeline relied completely on the values we received from the depth sensor. If a pixel had a depth value smaller than the pre-set depth of the virtual object, it was immediately classified as foreground and assigned a color of white. For the other pixels, we first calculated how far away the depth of that pixel was from the depth of the virtual object; the farther away a pixel was, the more black its color was. Let's look at how this calculation was made. For a generic pixel p, the depth measured in the real scene $D_{RS}(p)$, and the depth of the virtual scene $D_{VO}(p)$ we calculated a value x such that

$$x(p) = \frac{D_{VO}(p)}{D_{RS}(p) + \epsilon}$$

where ϵ was just a small arbitrary value to prevent division by zero. To get a value between [0,1) for that pixel, we performed the following calculation:

$$d(p) = \frac{a^{x(p)} - 1}{a - 1}$$

where a was an arbitrary constant greater than 1 to provide division by zero. Finally, to get a color value for that pixel (to determine how grey the pixel should be) between 0 and 255, we just made the simple mathematical calculation:

$$D(p) = 255 \cdot d(p)$$

After this was done for each pixel, we got the distance image showing in Figure 3.

3.3 Histograms and Probabilities

The distance image calculated in the previous step gave us a good approximation of which pixels were generally in the background and which pixels were generally in the foreground. Because the distance image relied solely on data gathered directly from the depth sensor, the segmentation between foreground and background was subject to quite a bit of sensor noise (especially at the edge of the foreground and the background) but we were still able to get a rough approximation of the pixels in the foreground and background. This third step in the pipeline was where the method we researched got very different from other methods used to handle occlusion. We used the segmentation provided by the distance image to calculate a color histogram for the background and the foreground separately. Then, for each pixel, we used the foreground color histogram to calculate the probability that – given the color of the current pixel – the pixel was a part of the foreground. A similar calculation was made for the background. One quick thing to not is that for the calculations above, because it was a little inefficient to build three histograms (one each for R, G, and B), we decided to have only one histogram containing as much information about the image as possible by converting the image from RGB to HSV and creating a histogram for the hue channel. After getting per-pixel probabilities for the foreground $p_{fore}(p)$ and the background $p_{back}(p)$, we had to consider another issue: because this segmentation was based only on the color properties of the image, we could get a lot of incorrect classifications if the colors of the background and the foreground were similar. We thus decided to base our final foreground image $I_{Foreground}(p)$ and our final background image $I_{Background}(p)$ by using a weighted sum of the color and the depth of each pixel. This was done by tuning a weight for how important the color is w_{color} and a weight for how important the depth is w_{depth} and integrating them as follows:

$$I_{Foreground}(p) = w_{color} \cdot p_{fore}(p) + w_{depth} \cdot d(p)$$

$$I_{Background}(p) = w_{color} \cdot p_{back}(p) + w_{depth} \cdot d(p)_{inverse}$$

In the equations above,

$$d(p)_{inverse} = 1 - d(p)$$

3.4 Cost Function

After the previous step, we had per-pixel probabilities for the foreground and background that also took the depth information into account. In this fourth step of the pipeline, we created a cost function where the higher the cost value, the less likely the pixel is in the background [2]. The computation for the cost pixel was fairly straightforward:

$$c_s(p) = \frac{I_{Foreground}(p)}{I_{Foreground}(p) + I_{Background}(p)}$$

Thus, the closer the cost value is to 0, the less likely the pixel is in the foreground and the closer the cost value is to 1, the more likely the pixel is in the foreground.

3.5 Thresholding/Filtering

After getting a grayscale image from the previous step, the fifth step in the pipeline is to threshold the cost image and get a binary image which can then be filtered to remove the noise.

For the thresholding aspect, the method we chose to implement used an algorithm that automatically computed a dynamic thresholding value between 0 and 255 such that all pixels below the thresholding value are classified as foreground and all pixels above the thresholding value are classified as background[3]. The algorithm can be described as follows:

```
Algorithm:
   // Calculate initial cost average and set threshold limit
    mu = mean(all pixels in cost image)
    l = some limit value that we have decided // constant

     // Classify pixels into foreground/background
     for each pixel in virtual image:
         if cost(pixel) < mu:
             pixel is part of background
         else:
             pixel is part of foreground

    // Calculate mean cost for foreground and background
     mu_f = mean(all pixels from foreground)
     mu_b = mean(all pixels from background)

   // new threshold = average(foreground & background mean)
     old_thresh = mu
     new_thresh = (mu_f + mu_b) / 2

    // Repeat process until we find threshold below limit
     while square(old thresh - new thresh) > square(l):
         // Classify pixels into foreground/background
       // Calculate mean cost for foreground and background
       // new threshold = average(foreground & background mean)
```

After we used the above algorithm to find a threshold value, we had to consider how we were going to filter out misclassified pixels (i.e. pixels that were actually foreground but misclassified as background and vice versa). To handle the first case (foreground misclassified as background) we first found any tiny holes inside the foreground cluster and then used a simple interpolation algorithm relying on a closing operation) to fill those holes with the same value as the pixels around the hole. This resulted in yet another binary image where outliers in the foreground had been filtered out. For the other case (background misclassified as foreground), we used a simple smoothing algorithm that took in that binary image as a filter and the original RGB color image as a guidance image to smooth out the background.

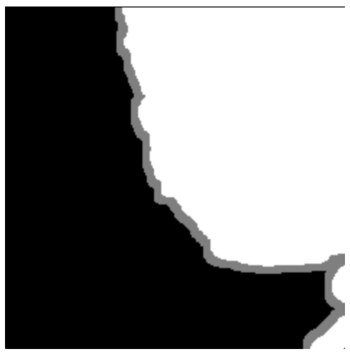


Fig. 4. Trimap for the example scene

3.6 Generating a Trimap

While the previous step was successful in smoothening out the classification for the foreground and the background, the classification near the edges of the foreground and background still had a few issues. We thus decided to use a trimap in this sixth step of the pipeline to specify which pixels along the edges were surely in the foreground and the background. First, we performed edge detection on the binary image computed at the end of the previous step to find the boundary between the background and the foreground. To also capture the pixels that were *around* the edges and not *on* the edges, we dilated the edge so we could cover a wider range of pixels on the background. The amount to dilate was a parameter that we pre-specified. After all of this edge-detection and border dilation, we got a trimap that looked like the one shown in Figure 4.

3.7 Alpha Matting

The seventh and penultimate step in our pipeline was to use the trimap that we made in the previous step and the original image to find the alpha matte. This process had three main steps[4].

The first step was to try to reduce the unknown region of the trimap by expanding the foreground and the background. To do this, we defined a spatial window w and applied it to each pixel p in the unknown region to check whether there was a neighboring pixel x that was similar in color and classification. Essentially, a pixel p was classified as background if there was a pixel x that was 1) in the window around p and 2) had the same color as p and 3) was also classified as the background. Similarly, a pixel p was classified as foreground if there was a pixel x that was 1) in the window around p and 2) had the same color as p and 3) was also classified as the foreground. All the pixels on the border that fit neither of the two criteria were still classified as unknown.

The second step was to sort out the classification for those unknown pixels by collecting background and foreground samples and choosing the best for the unknown pixels by minimizing a cost function. How to define the cost function was pretty open-ended, but we decided on a cost function that combined photometric affinity, probabilistic information, as well as spatial affinity. This meant that we compared the unknown pixels to the background/foreground samples and compared how similar an unknown pixel's color was to the samples, how close the unknown pixel was to the samples,

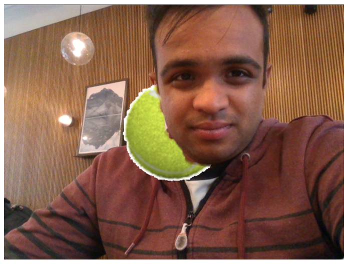


Fig. 5. Scene with the virtual object occluded

and how probable that unknown pixel was to be a part of either set. The best foreground sample we found in this process F_p and the best background samples we found in this process F_b were used along with the RGB color of a pixel I_p to compute an alpha matte. For each unknown pixel, the alpha value was computed as follows:

$$\alpha_p = \frac{(I_p - B_p)(F_p - B_p)}{||F_p - B_p||^2}$$

Finally, in the third step, we smoothened the matte by computing a weighted average of the closes m neighbors of each unknown pixel p.

3.8 Compositing

Finally, we had all the information we needed to make our final rendering. The eighth and final step was to use the alpha matte from the previous step and apply it to the real image I_r and the virtual image I_v to create an augmented image I_a as follows:

$$I_a = \alpha \cdot I_r + (1 - \alpha) \cdot I_v$$

What this conceptually means that the color of a pixel is:
1) the same as the corresponding pixel in the real image IF i) the corresponding pixel in the binary mask is black OR ii) the trimap value of the pixel is the foreground.
2) the same as the corresponding pixel in the virtual image IF i) the trimap value of the pixel is the background AND ii) the corresponding pixel in the binary mask is NOT black
3) a blend of the real and virtual image (as defined in the equation above) IF i) the trimap value of the pixel is unknown AND ii) the corresponding pixel in the binary mask is NOT black

The final result can be found in Figure 5.

4 IMPLEMENTATION DETAILS

In this section, we are going to be discussing the major steps in each step of the pipeline. The first step in the process is to read the image and depth data from the Intellisense RGBD camera. In order to interface the camera with the code, we used the PyRealSense2 library. Note that the above library is a prebuilt version of the official realsense library that is configured to work for the macOS and macOSx architectures. Once is configured the software to read from the sensors, we recorded a scale factor from the depth sensor. Now every time we read the depth data from the camera, we multiply the value in each pixel of the image by the scale factor. This is done in order to convert the distance from a virtual scale to a distance in meters.

Now that we can read the data from the sensor, we generate the distance image using the numpy library. The mathematical details of how the distance image is generated can be found in Section 3.2. Next, we implemented the algorithm discussed in Section 3.3 using the calcHist method of the OpenCV library. We then generate the cost image, as discussed in Section 3.4, using the numpy library.

As stated in Section 3.5, there are multiple steps involved in this part of the pipeline. We first implement the algorithm described in the aforementioned section to perform the thresholding using the numpy library and some of the matrix functions it provides us. We then implement the Type 1 filtering, using a morphological close through the morphologicalEx function of the OpenCV library. Next, we implement Type 2 filtering using the following implementation of a guided filter which ends up relying on the numpy and scipy libraries.

Next, we implement the trimap generation algorithm in Section 3.6. We are going to be using the following Trimap generation library. However before we utilize the library, we first detect the edges in the image using the Canny Edge Detection method provided by OpenCV. Then, we dilate the edges in the image using the dilate functionality of OpenCV. Finally, we then generate the trimap using the library we shared above.

Next, we perform Alpha Matting on the Trimap Image using the Pymatting library. The library has multiple different methods to perform alpha matting. We are going to be using a technique called KNN technique, as introduced in [19] , for matting, the reason for which we will discuss in Section 5.2. Finally, we perform the composition as described in Section 3.8 using the numpy library.

Finally, once we had generated the image we needed to show it onto the VR headset. Initially, we were planning to create our own Unity application and then use WebRTC to stream the data from the computer to the headset. However, during our research we discussed a free Oculus application that does PC-to-headset mirroring called Immersed. Through testing, we found that the application had rather little lag and thus decided to utilize that application rather than create our own system.

5 EVALUATION OF RESULTS

5.1 Timing

As hinted at in the previous sections, our occlusion was rather slow so in this section we are going to be diving into that in more detail in this section. A graph of the average time required for each step of the pipeline can be found in Figure. 6. The high-level takeaway is that on average it takes 0.52 seconds to evaluate a single image meaning that our system achieves an average FPS of $\frac{1}{0.52} = 1.92$. The first takeaway is that even though the pipeline has many steps, there are three primary steps that end up really slowing down the process. Thus, we are going to be examining each of these steps in more detail and seeing if we can determine the reason for this behavior.

The first step is the trimap generation which on average takes 0.08 seconds. Since we used an external library for this part of the project, we really don't have detailed insight into what causes it to

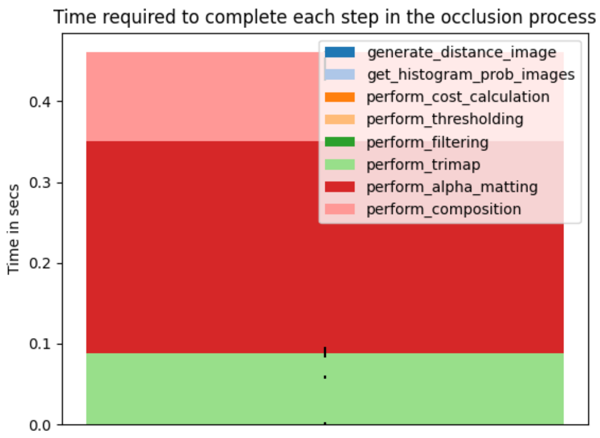


Fig. 6. Breakdown of time required for each step of the pipeline

take so much time. Neither the library nor the papers it indicates its implementation is based on containing detailed benchmarking information to allow us to dive into this number in more detail. We discuss some possible alternatives in Section 7.

The next computationally expensive step is alpha matting, which on average takes 0.34 seconds per frame. Unfortunately, the library that we used, Pymatting, only has benchmarking information on the accuracy of our alpha matting rather than the time required by the algorithm. Unfortunately, the paper ([19]) that the Pymatting library is based on also doesn't contain detailed performance information. We dive into some potential remedies to the problem in Section 7.

Finally, the final time-consuming step in the pipeline is the image composition step, which on average takes 0.1 seconds. We implemented this step using the algorithm described in Section 3.8 which calculates the color at every single pixel based on comparing and combining the value at that pixel in some of the other images we generated in the pipeline. This results in us being unable to vectorize the process, thus making it inefficient.

5.2 Hyperparameter Tuning

It is relatively hard for us to get an overall metric on how well our occlusion algorithm behaves because it is a rather subjective measure. Traditionally, researchers have used substitute metrics like the percentage of the image that was classified as unknown in the trimap, which our system averaged to around 4.4%, these may not actually capture what we are looking for. Even during our limited testing, we were able to find images where the percentage unknown was higher but subjectively we agreed that the occlusion was infact higher. However, there were quite a few "hyperparameters" associated with each step in the process that we had to "tune" in order to get what we perceived as better occlusion. In these sections, we are going to be discussing some of these hyperparameters and how their values influenced the algorithm

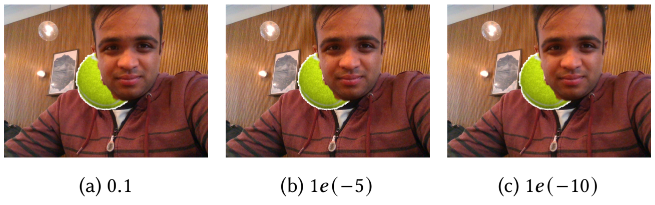


Fig. 7. Final occlusion images for three different threshold values.

5.2.1 Alpha Matting Algorithm. As we stated in Section 4, there were multiple different algorithms we could have used for the Alpha Matting stage. Initially, we tried to follow Pymatting's implementation of [20] which generates a quadratic cost function for alpha which then solves using gradient descent. Whether it is the technique or the library's implementation of its, we would get an error on about 5% of the images that the system failed to converge to a reasonable minimum. Thus, we decided to switch to Pymatting implementation of KNN method ([19]) which according to the Pymatting library's benchmarking has a slightly Mean Squared Error, 6.64 vs 6.09. However, this method worked rather reliably for us without ever throwing errors which is why we choose it over the Closed Form approach.

5.2.2 Thresholding constant. If we take a look at the algorithm discussed in Section 3.5, it relies on a constant called l which represents the acceptable difference between the previously determined threshold and the updated threshold is below a certain limit. Thus a small limit value will ensure that we actually choose a value close to that the algorithm eventually converges as it generally takes more iterations to meet the convergence threshold. This however indicates that the lower threshold we use, the more time it will take for us to determine the threshold value. We tried using threshold limits of 0.1, $1e(-5)$, $1e(-10)$ which on average took $3.93e(-04)$, $9.96e(-04)$, $9.97e(-04)$ seconds respectively. Furthermore, you can find the final occlusion images generated using each of these threshold values can be found in Figure 7. Ignoring minute differences, the final occlusion image is relatively similar whether we use a value of 0.1 to $1e(-10)$ but the larger threshold value takes about $\frac{2}{5}$ of the associated time. Thus, we decided to use a threshold value of 0.1.

5.2.3 Depth and Color Weights. Finally, note that in Section 3.3 we do a weighted sum of the color and depth probabilities to generate an initial estimate for the foreground and background of the image. We used a similar approach to what we discussed in Section 5.2.2 where we tuned the weights and then looked at both the time taken as well as the final image that was generated. During our experimentation, we found that increasing the relative weight of either the camera or the depth data didn't significantly improve the runtime or the subjective goodness of the final image and thus we decided to give equal weight, 0.5 each, to both the camera and depth data.

6 DISCUSSION OF BENEFITS AND LIMITATIONS

At a high level, I think that the behavior of our system can be summarized as "slow but works". As we discussed, in the previous section our system is only able to run at 1.92 FPS, a detailed discussion of which can be found in Section 5.1. However, our system was rather reliable and worked rather well in the wide range of scenarios that we tested in. Specifically, we visited a bunch of different locations, connected the camera to the laptop, and then ran our system and objectively evaluated if it behaved as expected. Some of the places where we tested our code included:

- A busy cafe in University Village
- An empty room inside CSE 2
- The atrium inside the Allen building
- The gallery of the Allen building

We found that our occlusion handling behaved as expected in this scenario, even when events like a random person walking by or us covering the camera with our hand changed the scene dynamics rather rapidly.

Additionally, note that we only tested our code by occluding a single two-dimensional, single-colored ball with the rest of the scene. Thus, we are completely unsure about how our system would behave if we had a single three-dimensional multi-colored object or what occlusion handling looks like we have multiple overlapping virtual objects, whether that overlap is in their color or in their depth. We believe that our project illustrates the high-level steps needed to occlude a simple object across a wide variety of scenes in the physical world. However, our system still needs to be improved so that it can occlude more complicated in the physical world as well as deal with the scenarios where we might want to occlude multiple overlapping virtual objects.

7 FUTURE WORK

We believe that there are many directions we can take to come closer to achieving the occlusion that is implemented by some of the state-of-the-art VR systems today. No matter what approach we choose, the first thing we would definitely need to do is attach the camera to the headset in a stable manner.

The first direction that comes to mind is using all of the work that has been done in deep learning when it pertains to computer vision. Researchers have trained models to perform almost all of the steps in our pipeline, such as Google training a convolution a neural network to perform alpha matting to take images in Portrait Mode on the newer pixel. Thus, we can host a server with access to dedicated GPUs to efficiently run the models that we need, and use WebRTC or similar frameworks to stream both the video and the sensor data from the camera to the server and then stream the result back to the headset. While this would definitely stream up the image processing pipeline and also produce more accurate results there would be added latency associated with the streaming. We want our occlusion technology to be rather real-time in that smoothly updates the scene as the user's POV changes, which makes us feel that anything involving streaming would be quite a bottleneck. Unless machine learning technology advances to the point that we can reliably and fastly run ML models on headsets we are likely looking at improving our existing algorithms in the near future.

In Section 5.1, we discovered that the three bottlenecks to our pipeline were trimap generation, alpha matting, and pixel-by-pixel image blending. One key point that we realized about our system is that we treating each frame in the video as completely independent from the previous or the next frame even though in real video streams, neighboring frames are often rather similar and share a lot of similarities. This property is leveraged by [21] who reported achieving 30 Hz on mobile devices. In a future iteration of the project, we would utilize the technique they propose and see if that gives us a performance improvement. Furthermore, [22] showed how you can smartly sample only a subset of the pixel in the trimap and use that to generate a rather accurate alpha matting. In the future, we can continue exploring some of these non ML based techniques and see if they can improve the performance of both our alpha matting and our trimap generation steps. In a similar vein, rather than implementing our custom approach to alpha blending the virtual and real image, we can utilize the technique proposed by Intel in [23].

8 CONCLUSION

This projects illustrates that even rudimentary combination of the virtual and real worlds can create truly immersive experience that smartly overlay the physical world with the virtual world. As the usage of mixed reality and augmented reality products continues to increase, smartly combining the virtual contents of the users screen with their physical surrounding in a realistic manner becomes especially relevant when it comes to user experience. This project reveals that the core pieces of technology to solve this problem have already been developed by the computer vision community. The challenge lies in however optimizing these solutions to be able to perform the necessary computation in real time and can display the final result to the users with very limited lag. This either requires improving hardware and software capabilities of such headsets so that they can run large models in real time or to develop further optimizing to existing algorithms geared at improving perfomance.

9 ACKNOWLEDGMENTS

We would like to thank the entire staff of CSE 493V for their technical and conceptual support and guidance throughout the development life-cycle of our final project. We would also like to explicitly thank John Akers for providing the Intel RealSense sensor and the Oculus Quest 2 needed to complete this project.

References:

[1] Gugliermo, S. (2019). Occlusion handling in Augmented Reality context.

[2] Walton DR, Steed A (2017) Accurate real-time occlusion for mixed reality. In: ACM Symposium on Virtual Reality Software and Technology, pp 1–10

[3] T. W. Ridler and S. Calvard. "Picture thresholding using an iterative selection method". In: IEEE Trans. Syst. Man Cybern. SMC-8 2010, 31 (1978), pp. 630–632.

[4] E. S. L Gastal and M. M. Oliveira. "Shared sampling for real-time alpha matting". In: Computer Graphics Forum 2010, 31 (4), pp. 575–584.

[5] B. Huhle, T. Schairer, P. Jenke and W. Strasser, "Robust non-local denoising of colored depth data," 2008 IEEE Computer Society

Conference on Computer Vision and Pattern Recognition Workshops, Anchorage, AK, USA, 2008, pp. 1-7, doi: 10.1109/CVPRW.2008.4563156.

[6] K. Essmaeel, L. Gallo, E. Damiani, G. De Pietro and A. Dipandà, "Temporal Denoising of Kinect Depth Data," 2012 Eighth International Conference on Signal Image Technology and Internet Based Systems, Sorrento, Italy, 2012, pp. 47-52, doi: 10.1109/SITIS.2012.18.

[7] J. Shen and S. -C. S. Cheung, "Layer Depth Denoising and Completion for Structured-Light RGB-D Cameras," 2013 IEEE Conference on Computer Vision and Pattern Recognition, Portland, OR, USA, 2013, pp. 1187-1194, doi: 10.1109/CVPR.2013.157.

[8] Wang, Liang Zhang, Chenxi Yang, Ruigang Zhang, Cha. (2010). TofCut: Towards Robust Real-time Foreground Extraction Using a Time-of-Flight Camera.

[9] A. Criminisi, G. Cross, A. Blake and V. Kolmogorov, "Bilayer Segmentation of Live Video," 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), New York, NY, USA, 2006, pp. 53-60, doi: 10.1109/CVPR.2006.69.

[10] A. Criminisi, G. Cross, A. Blake and V. Kolmogorov, "Bilayer Segmentation of Live Video," 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), New York, NY, USA, 2006, pp. 53-60, doi: 10.1109/CVPR.2006.69.

[11] K. He, J. Sun and X. Tang, "Guided Image Filtering," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 35, no. 6, pp. 1397-1409, June 2013, doi: 10.1109/TPAMI.2012.213.

[12] He, Kaiming Sun, Jian. (2015). Fast Guided Filter.

[13] Vikas Gupta and Shanmuganathan Raman. (2017). "Automatic Trimap Generation for Image Matting". Indian Institute of Technology, Gandhinagar, IND

[14] Thomas Porter and Tom Duff. 1984. Compositing digital images. SIGGRAPH Comput. Graph. 18, 3 (July 1984), 253–259. https://doi.org/10.1145/964965.808606

[15] T. Lu and S. Li. "Image matting with color and depth information". In: Proceedings of the 21st International Conference on Pattern Recogni- tion (ICPR2012) (2012), pp. 3787–3790.

[16] E. S. L Gastal and M. M. Oliveira. "Shared sampling for real-time alpha matting". In: Computer Graphics Forum 2010, 31 (4), pp. 575–584.

[17] M. Sarim, A. Hilton, and J. Y. Guillemaut. "Alpha matte estimation of natural images using local and global template correspondence". In: 2009 International Conference on Emerging Technologies 49(3) (2009), pp. 6011–659.

[18] J. Zhu et al. "Joint depth and alpha matte optimization via fusion of stereo and time-of-flight sensor". In: 2009 IEEE Computer Society Con- ference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2009 (2009), pp. 453–460.

[19] Qifeng Chen, Dingzeyu Li, Chi-Keung Tang (2013). KNN matting. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 35(9), 2175-2188.

[20] A. Levin, D. Lischinski and Y. Weiss, "A Closed-Form Solution to Natural Image Matting," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 30, no. 2, pp. 228-242, Feb. 2008, doi: 10.1109/TPAMI.2007.1177.

[21] Beato Nà, Pillat Rà and Eà Hughes Càà. REAL-TIME VIDEO MATTING FOR MIXED REALITY USING DEPTH GENERATED TRIMAPS.

[22] Shared Sampling for Real-Time Alpha Matting Eduardo S. L. Gastal and Manuel M. Oliveira Computer Graphics Forum. Volume 29 (2010), Number 2. Proceedings of Eurographics 2010, pp. 575-584.

[23] Marco Salvi and Karthik Vaidyanathan. 2014. Multi-layer alpha blending. In Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '14). Association for Computing Machinery, New York, NY, USA, 151–158. https://doi.org/10.1145/2556700.2556705