Hand Tracking and Gesture Recognition for Automated VFX

Michelle Zhang (mz32@illinois.edu)
Noah Franceschini (nef3@illinois.edu)
Asher Mai (hanlinm2@illinois.edu)
Aditi Tiwari (aditit5@illinois.edu)

I. Introduction

Large scale movie productions typically spend millions of dollars every year on VFX alone. In addition to these extreme prices comes the amount of labor that goes into creating a high-production film. These two factors create a massive barrier to entry for any aspiring indie filmmaker who does not have the time nor budget to quality VFX for their films. We propose our gesture recognition and VFX pipeline to rectify this problem. This system also can be used in entertainment because our hands work great as game controllers. The detected gesture and hand location can be used to control different actions that can replace button presses and joystick movements of game controllers, creating a more interactive experience.

Our current approach utilizes a convex hull algorithm for gesture recognition, optical flow for motion effects and edge detection for our cartoon filter. Originally, our approach included various machine learning methods for gesture recognition and depth but we decided to use more classical computer vision methods due to hardware constraints and frame rate drop.

Real-time filters such as those found in SnapChat inspired us for this project as SnapChat can create effects in less than a second that are interactive with user input. We wanted to create similar effects but instead those that interact with specific gestures that the user makes and that interact with the motion of the user as well with the depth of the user's hand though the shortcomings of depth will be explained later in this report.

II. Approach

A. Overview

This and the following sections B - F detail the classical/non-learning implementation of our program. Before the main program runs, the video of the desired special effects (for our demo we used a fireball) is processed and turned into an array. This array is saved into a txt file for faster access when the main program runs. Once the main program starts, the user is first prompted for a sample of their hand color. The samples are used for color thresholding during the next part of the pipeline, which is gesture detection and recognition. The program utilizes contours and convex hull to determine if the user is holding up one, two, or more than two fingers. For the rest of this report, these gestures will be referred to as Gesture 1, 2, and 3 respectively. A combination of gesture recognition and optical flow is used to determine what effects to apply to the original frame. The effects include overlaying the special effects video, tilting the special effects applied, changing the color of the special effect, and applying a cartoon filter to the frame.

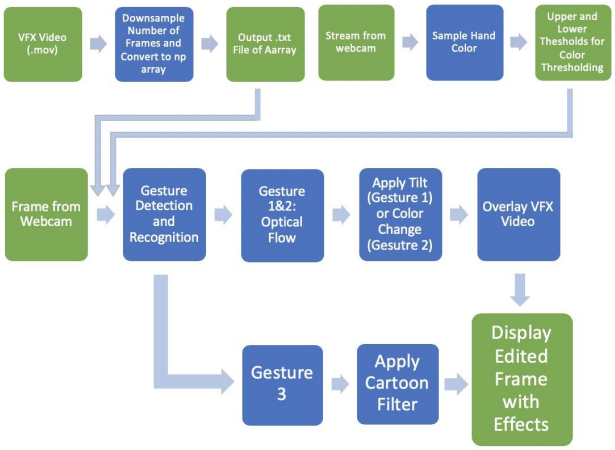

Figure 1. Visualization of program pipeline

B. Sample Hand Color

The program begins by sampling the color of the user's hand. To obtain a sample the user holds their hand over both sampling areas outlined by green rectangles, as shown in Figure 2, and presses 's'.
The pixels within the two areas are saved as arrays of HSV values. The average HSV value of each sample area is calculated and the min/max values of the averages are used to set the lower/upper HSV thresholds. These thresholds are used in the rest of the program to perform color thresholding.

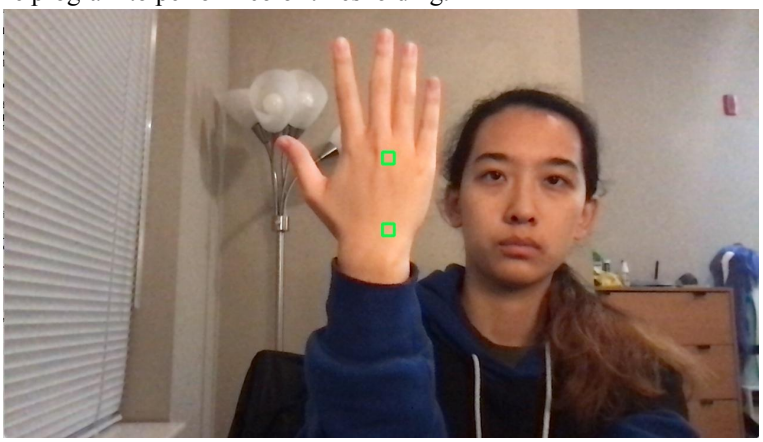

Figure 2. User obtaining sample of hand color

C. Gesture Recognition and Detection

We implemented a classical approach to gesture recognition and detection using OpenCV. The program begins by converting the frame obtained from the computer's web camera to the HSV color space. The HSV frame is then thresholded using the lower and upper thresholds obtained from sampling the user's hand. The output is a binary image such as the one in Figure 3.

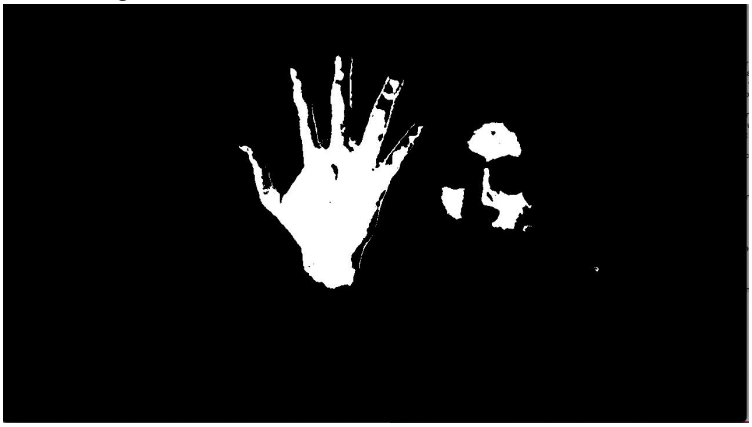

Figure 3. Example output of color thresholding

Using OpenCV's *findContours()* function, the program finds the external contours in the binary image. The contour with the largest area is taken as the hand's contour, because we make the assumption that either the user's face is not in frame or that the scene is lit from the front and that the user's face is more dimly lit. OpenCV's *convexHull()* and *approxPolyDP()* to fit a polygon to the hand's contour. The center of the polygon plus a small offset downwards is used as the center of the hand. The vertices of the polygon that are both above the center and outside a certain radius from the center are categorized as the user's finger tips. A single finger tip is recognized as Gesture 1, two finger tips is recognized as Gesture 2, and three or more is recognized as Gesture 3.

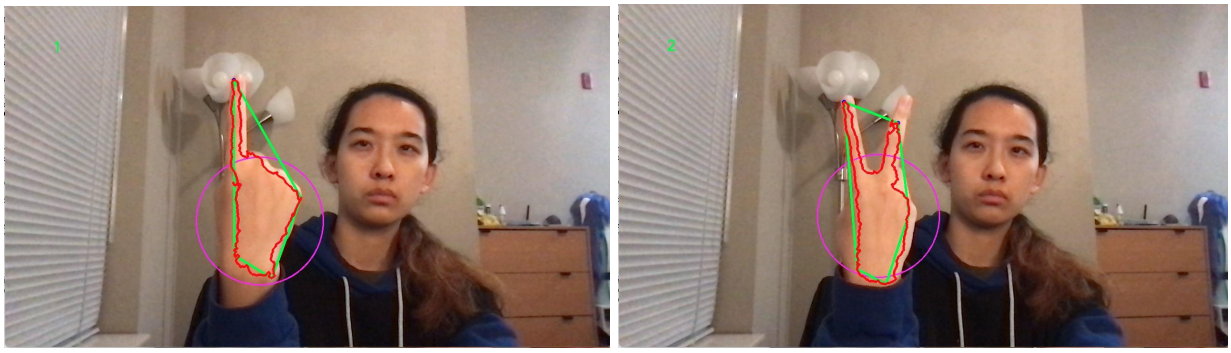

Figure 4. For each image the hand's contour is outlined in red and the convex hull approximation in green. Points that fall above and outside of the circle are classified as finger points. The number of fingers detected is written in green in the top left corner.

D. Overlaying VFX Video

The program requires that the special effects videos have four channels: RGB and A (alpha channel). When the special effect is added to a frame, alpha blending is used to weight the pixels of the webcam frame and the pixels of the special effect video. By doing so, transparency is added to the effect and creates a more realistic look.

E. Optical Flow

Optical flow (Lucas-Kanade) is calculated by the program when Gesture 1 or Gesture 2 is detected. The following calculations are performed on windows of 25x25 pixels to obtain the motion displacement (u,v):

$$\begin{bmatrix} Ix(x1,y1) & Iy(x1,y1) \\ Ix(xn,yn) & Iy(xn,yn) \end{bmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = -\begin{bmatrix} It(x1,y1) \\ It(xn,yn) \end{bmatrix}$$

A **b**

$$\begin{pmatrix} u \\ v \end{pmatrix} = (A^T A)^{-1} A^T b$$

$$Ix(x,y) = gradient\ with\ respect\ to\ x\ at\ pixel\ location\ (x,y)$$
$$Iy(x,y) = gradient\ with\ respect\ to\ y\ at\ pixel\ location\ (x,y)$$
$$It(x,y) = gradient\ with\ respect\ to\ time\ at\ pixel\ location\ (x,y)$$
$$(u,\ v) = displacement\ or\ movement$$

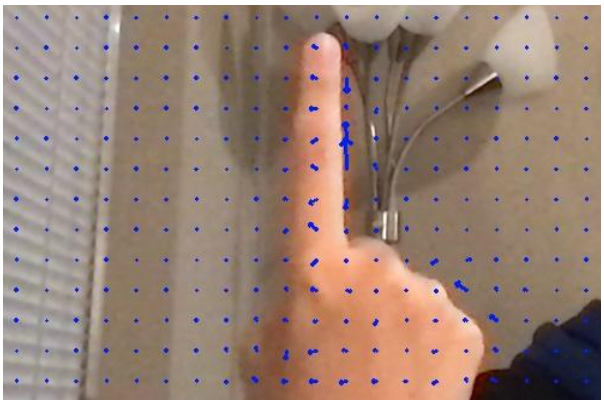

Figure 5. Optical flow (Lucas-Kanade) around hand slowly moving to the left

Once the (u,v) have been calculated for each window, the program sums all the components in the +x, -x, +y, and -y directions. It then uses these four sums to determine the dominant direction of motion (up, down, left, or right). If the maximum value of the four sums is below a certain threshold, then there is considered to be no movement. To further filter out noisy motions, the program stores the six most recent dominant directions of motion in an array. A

specific direction of motion must occur at least three consecutive times in order to be considered a deliberate motion. Deliberate motions are used to apply tilt and color change effects.

F. Applying Different Effects using Gesture and Flow

When Gesture 1 is detected, optical flow is used to detect deliberate left and right motions in a window surrounding the user's hand. If motion in one direction (left or right) is continuously detected, then a tilt effect is applied. The amount of tilt increases the longer the motion continues in the same direction. The tilting effect is achieved by shifting each row in the special effects video by different offsets.

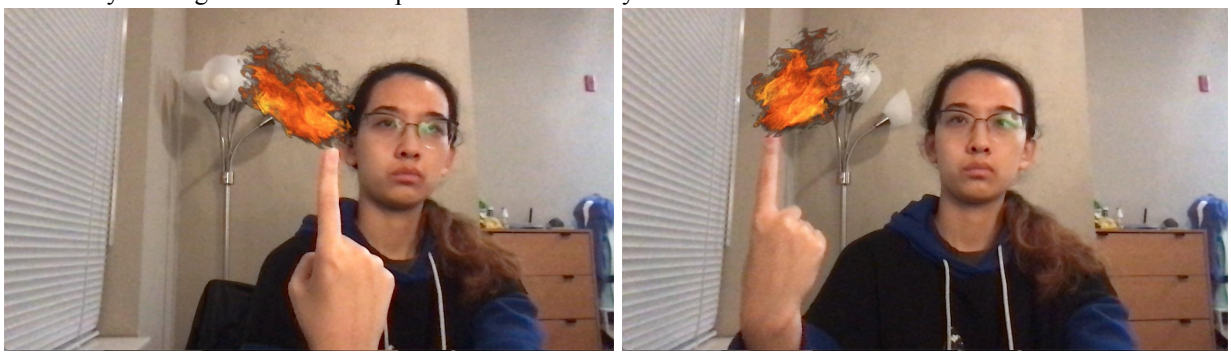

Figure 6. Tilt effect applied for hand moving to the right (left image) and moving to the left (right image)

When Gesture 2 is detected, optical flow is used to detect deliberate up and down motions in a window surrounding the user's hand. A variable color_effect is used to add or subtract an offset from the BGR values of the frame. Upward motion will increase the value of color_effect and down motion will decrease the value. In the case of the flame VFX increasing color_effect turns it blue and pink.

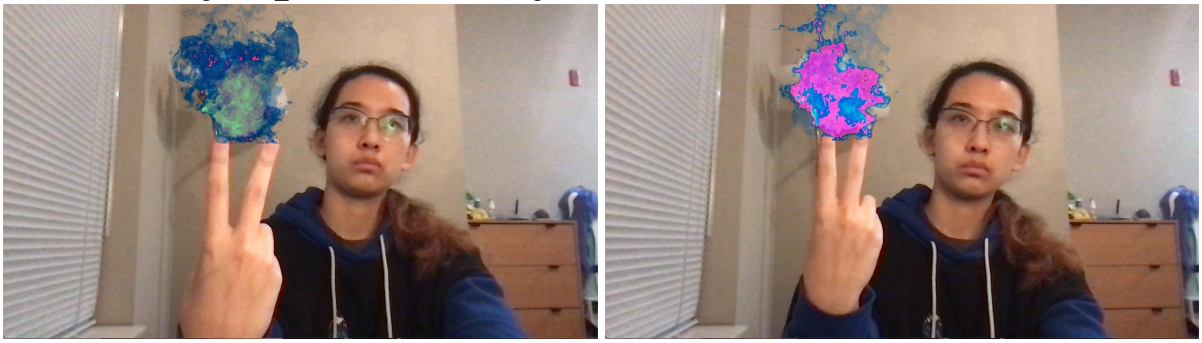

Figure 7. (Left image) color_effect is some nonzero value
(Right image) color_effect is at its maximum value

When Gesture 3 is detected, a cartoon filter is applied to the frame. The filter is implemented by using *cv2.adaptiveThreshold()* to detect the edges in the webcam frame. The edges are then added to a gaussian filtered version of the original frame. Cartoons typically have a discrete colorization look to them, so to achieve this effect the program bins each pixel's V value in the HSV color space.

Figure 8. Examples of applying the cartoon filter with different colored flame effects

G. Machine Learning Approach for gesture detection and classification

Our initial approach was to use a deep neural network to perform gesture detection and classification. We used the HAGRID (HAnd Gesture Recognition Image Dataset) to train a Resnet18 network to perform classification of 18 gestures in the dataset.

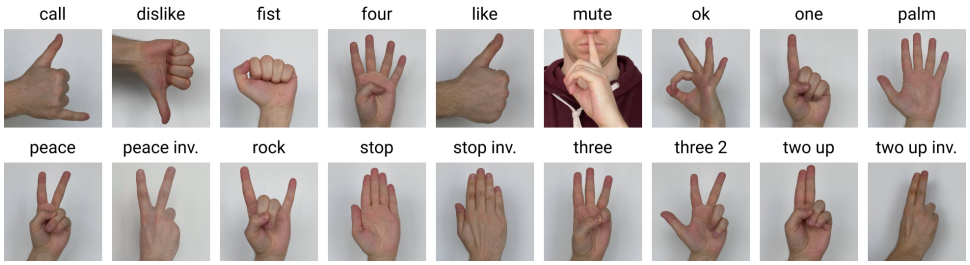

Figure 9. HAGRID gesture dataset with 18 gesture classes

The HAGRID dataset has 716 GB of data, containing 552,992 RGB images of size 1920 x 1080. Because we do not have the computational hardware to train all 716 GB of data, we used the included 2 GB subsample of the dataset, which contains only 100 images per gesture class, to train and test our Resnet18 network. We trained and tested the subsample on an Nvidia GPU for 100 epochs, which took 50 minutes. During real-time inference, we utilized HAGRID's pre-trained SSDLite MobileNetV3 for gesture detection and fed the detected hand into our trained Resnet18 network for classification.

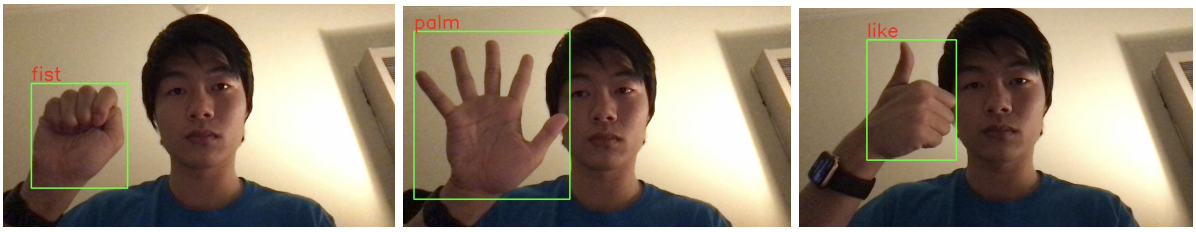

Figure 10. The real-time inferences of the MobileNetV3 detector and the Resnet18 classifier.

H. Depth Estimation

Our final approach for depth estimation is a neural network based on the DTS-Depth architecture which itself is based off of the MobileNetV2 architecture. This architecture is a lightweight architecture that is suitable for quick estimations at the minimal cost of accuracy which is required for performing real-time depth estimation. We trained our own neural network on the DIODE (Dense Indoor and Outdoor DEpth). As implied in the name, it is a depth dataset that we used for indoor depth estimation since with our model being trained on taking in RGB images and outputting depth estimations for those images. The indoor portion of the dataset included 8,574 RGB images at a 1024 x 768 resolution, each one with a corresponding 1024 x 768 depth map.

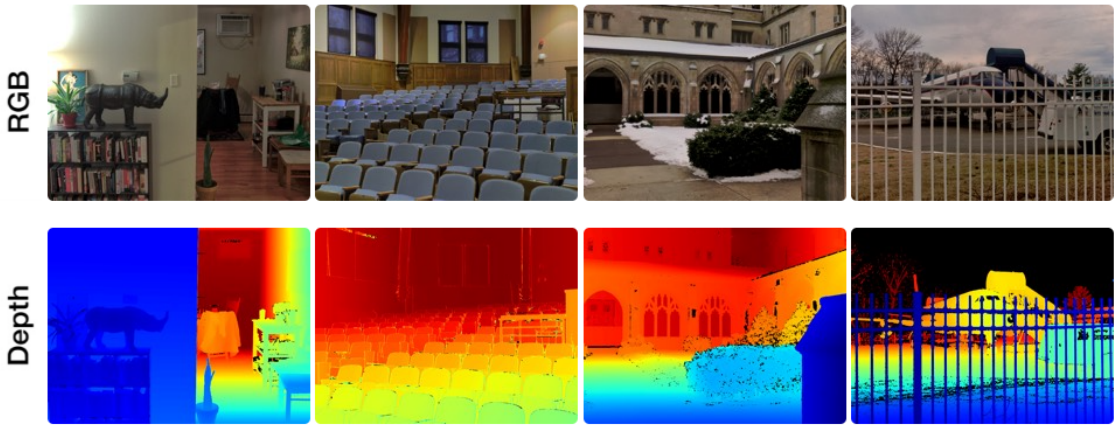

Figure 11. Examples from DIODE dataset

If we obtained better results from the trained network, the purpose of this real-time depth estimation was to allow various effects to interact and collide with camera-captured terrain that was on the same depth plane as the effects.

III. Results

A. Classical Implementation of VFX Pipeline

Link to Demo Video of non-learning VFX pipeline: https://youtu.be/c_B_kCH9-h4

B. Gesture using ML

The Resnet18 classifier we trained performs very well on the subsample testset, with an accuracy of 86.31%. However, during real time inference, the accuracy feels much lower qualitatively. This is likely due to overfitting on the small subsample of the dataset, and the classifier fails to generalize to inputs during real time inferencing.

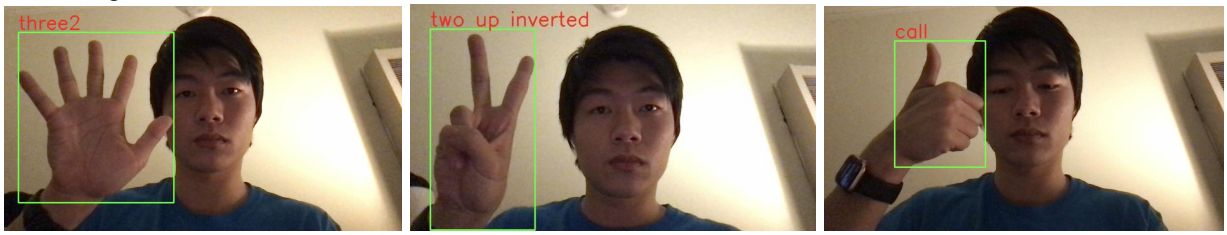

Figure 12. Instances during real-time inference where the classifier mis-classifies gestures "palm" as "three2", "peace" as "two up inverted", and "like" as "call"

The machine learning approach also results in lower frame rate during real-time performance. It has an average frame rate of 2.3 frames per second compared to 6.5 frames per second using the convex hull approach. In addition, the bounding box outputted by the detector does not contain segmentation of the shape of the hand and information of where the fingertips are. This results in less useful infomation that can be used to generate the VFX compared to the convex hull approach.

C. Depth Results

Classical approach:

Our classical approach utilized the disparity matching algorithm covered in class. For capturing the images, we used two identical $30 generic brand cameras from Amazon. For our parameters, we found that window size of 5 pixels and a disparity range of 60 would give us unusable results in most settings, most likely due to non-uniform lighting. These results also took a long time to compute, ~173.5 seconds per frame, and as such trying to improve the results with larger window sizes and disparity ranges proved fruitless. Another issue found in our experiementation was that the edges of the depth estimation were skewed. This effect is most likely caused by the poor camera quality which contained radial distortion around the edges of both cameras.

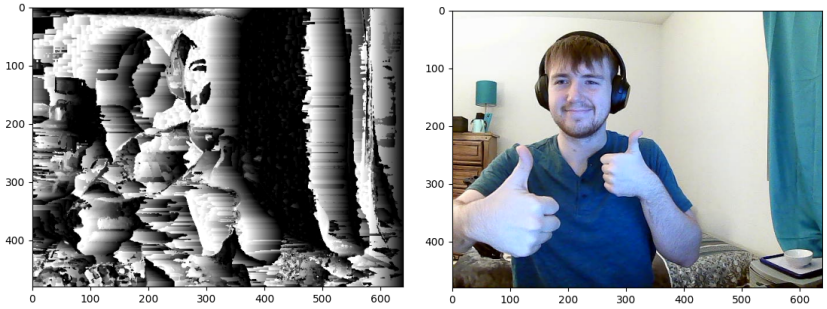

Figure 13. Depth from disparity (output vs input)

Machine learning approach:

The DTS-Depth architecture that we used always had trouble converging on the DIODE dataset and would often fluctuate between 6.5 and 13.7 mean squared loss each iteration with a batch size of 26 images per batch after 10 epochs of training. As for the qualitative output of each image, the neural network often contained "holes" of estimated zero depth which could be caused by the final ReLU layer on the output of the neural network as shown in

figure Z. The neural network was also extremely sensitive to bright light sources and would mislabeled their depths as being far away relative to their ground-truth depth as shown in figure ZX.

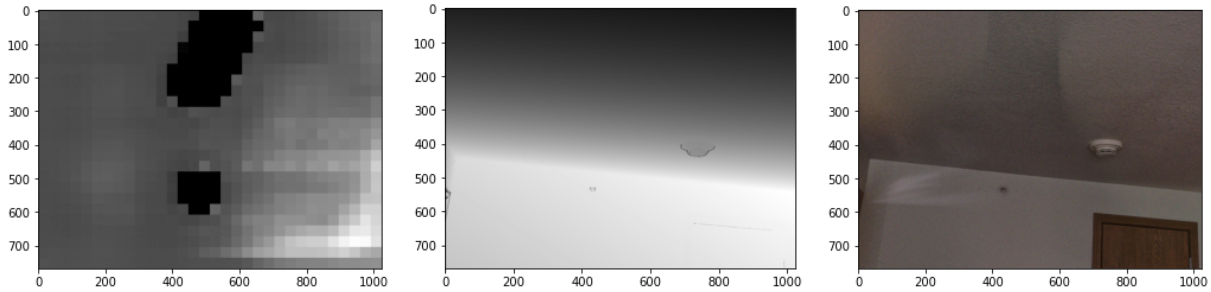

Figure 14. From left to right, trained network output, ground truth, RGB input, showing the "holes" in the network output

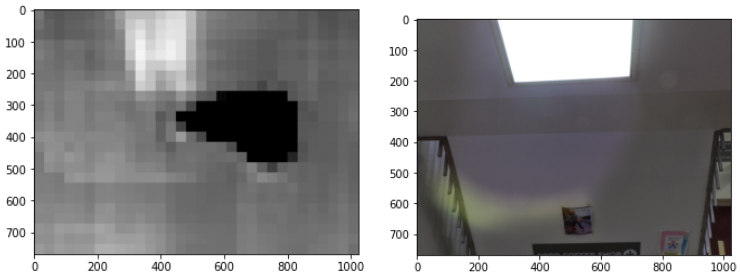

Figure 15. Network output showing sensitivity to bright light sources

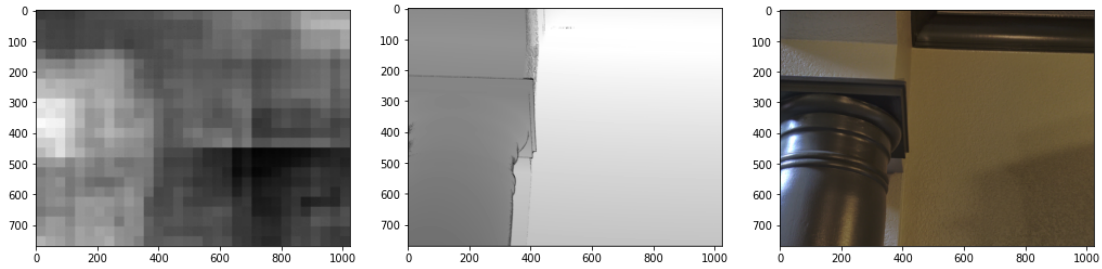

Figure 16. Network output

We believe that the inaccuracies in the neural network are caused by the insufficient size of the network. Since the DTS-Depth architecture is based on light-weight design meant to keep memory usage down as well as runtimes low, we believe that it was too light-weight to properly learn the complexities of monocular depth estimation and as such had these errors in its output.

IV. Discussion and Conclusion

A. Classical Implementation of VFX Pipeline

As seen from the demo video, the non-learning implementation yields reasonable results in real time (about 6.5 frames per second). However, it relies on multiple assumptions and has limited robustness. These assumptions include having the user's face out of frame or further away from the camera and more dimly lit. Through trial and error, we found that a front lit scene (light source from the front and behind the camera) yields good results; this is the setup used in the demo video. This implementation also assumes a static background during optical flow, that the scene does not contain objects of the same color as the user's hand, and that only the user's hand (not their whole arm) is exposed. In regards to gesture detection/recognition, the best results were obtained when the back of the user's hand was oriented toward the camera. This is due to the back of the hand providing a smooth surface; in contrast, with a forward facing hand putting down one or multiple fingers will create shadows on the hand. These shadows tend to disrupt the result of the color thresholding. In conclusion, while the classical implementation is usable and runs in realtime, its robustness to different environments and lighting conditions could be improved.

B. Gesture using ML

The Machine Learning approach for gesture recognition solves some of the problems of the classical implementation. It is able to be more robust in different lighting conditions, different skin colors, and both the front and back of the hand. In addition, it will not mistakenly use the face of the person as the detected hand due to similar skin color. However, the accuracy of the machine learning approach and the lower frame rate prompted us to use the classical implementation instead.

C. Depth Estimation

Our depth estimation implementation went through a few iterations to try and attempt depth estimation in real time. The first attempt was done using the disparity method found in homework 5. Initially we ran into a couple of issues. We had two substantial issues with this method. The first was that the identical cameras that we had ordered had severe radial distortion to them, warping and bending the edges of the camera image making pixel matching near impossible as pixels between the two cameras that should be on the same scan line are no longer there. The second issue that we had was speed. With the algorithm scanning small windows over each pixel in both images, the algorithm tended to be too slow for us to use in real time with each frame of depth taking about 20 seconds to compute. So we decided to explore other algorithms that could run more efficiently.

Our first other look at approaches came from a paper that attempted to utilize the GPUs of mobile devices to create real time depth estimation. Seeing that the paper utilized somewhat inefficient hardware and that the results were in real time we decided to dive deeper into this promising avenue. After some further research, however, the MobiDepth algorithm reaped no positive results due to being 5000+ lines of unavailable code without much suggestion of how to program it ourselves. Due to the lack of resources for the implementation of the MobiDepth algorithm and the lack of experience in GPU-based programming, we searched for other efficient implementations.

Our search led us into the realm of machine learning as two of our members have worked with neural networks and there were several papers that had utilized monocular depth estimation through them. We found a paper that created accurate depth estimations while also utilizing limited GPU resources which was perfect for our computer setups since the most powerful GPU amongst our group is an NVidia 1650 Ti. Unfortunately as shown in the results page, the architecture was too light-weight to capture the complexities of monocular depth estimation. We wanted to train a more powerful network however we surmised that a more powerful network would be too computationally expensive for our GPUs given that the current network already consumed 3.9 GB/4 GB of our GPU RAM and that a larger network would run too poorly on the CPU for it to be used in real-time depth estimation for our VFX. As such, we made the decision to no longer pursue depth estimation for our project and explore other avenues instead.

V. Individual Contributions

Asher: Trained and tested machine learning network for gesture classification, integrated it with the VFX pipeline, and compared the results with the convex hull approach done by Michelle to evaluate which approach is better.

Michelle: Implemented convex hull approach for gesture detection and recognition, alpha blending for overlaying VFX, optical flow, tilting VFX, color change VFX, and cartoon filter effect. Attempted stereo approach to depth using two different cameras, but did not obtain usable results.

Noah: Assisted in the stereo approach to depth using two cameras. Trained and tested machine learning network for monocular real time depth estimation, attempted to integrate it into the VFX pipeline but realized that results were too poor to continue down the depth estimation avenue.

Aditi: Contributed to project proposal and progress report

VI. References

1. HAGRID dataset: https://github.com/hukenovs/hagrid
2. OpenCV documentation: https://docs.opencv.org/4.x/d2/d75/namespacecv.html
3. OpenCV Docs Contours and Defects: https://docs.opencv.org/3.4/d5/d45/tutorial_py_contours_more_functions.html

4. Contours and Convex Hull in OpenCV Python:
   https://medium.com/analytics-vidhya/contours-and-convex-hull-in-opencv-python-d7503f6651bc
5. Implementation of Hand Gesture Recognition Technique for HCI Using Opencv:
   https://www.ijrdet.com/files/Volume2Issue5/IJRDET_0514_04.pdf
6. Handy, hand detection with OpenCV:
   https://pierfrancesco-soffritti.medium.com/handy-hands-detection-with-opencv-ac6e9fb3cec1
7. DTS- Depth
   https://www.mdpi.com/1424-8220/22/5/1914
8. MobiDepth
   https://www.microsoft.com/en-us/research/uploads/prod/2022/09/mobicom22-final138.pdf
9. DIODE Dataset
   https://diode-dataset.org/