Final Project

RBE/CS 549

Max Weissman

GENERAL NOTES:

1. This report directly corresponds to the scripts and files included in the zip file.
2. This project uses Google's MediaPipe library, if you do not have this library already installed please run the command: ***pip install -q mediapipe==0.10***
3. Depending on your environment configuration, there can be a conflict with MediaPipe and Tensorflow versions of protobuf. If you receive an error pertaining to a *builder.py* file, please follow the steps of the highest voted solution in this link: https://stackoverflow.com/questions/71759248/importerror-cannot-import-name-builder-from-google-protobuf-internal
4. If using a virtual machine to run these scripts, the video feed will likely be laggy due to the intensive background computation. I recommend running this script on a local machine. Due to these limitations, I had to develop and run this program on my base Windows machine.
5. There is another script, *utils.py*, which the main script, *reactions.py*, relies on. The *utils.py* function contains some basic helper functions for running the main script.

In this project, the goal was to recreate and improvise existing and new reactions in Apple's PhotoBooth app. There are eight existing reactions that are all triggered through hand gestures. Once a specific hand gesture is recognized, some type of visual effect will begin such as fireworks, rain, balloons, etc. The first part of this tasks involves implementing a Deep Learning technique for gesture recognition. The second part of this task involves creating the screen effects using different image processing techniques. These tasks can be further broken down into the following steps:

- Research and select a pre-trained model for gesture recognition and fine-tune model if necessary
- Run a webcam function that continuously reads frames from a webcam
- Pass each frame through the gesture recognition model and return the highest probability gesture
- Create visual effects (VFX) for each possible gesture that trigger when the respective gesture is recognized

For the first part of this assignment, I looked into available pre-trained models for gesture recognition. One of the most popular models for gesture recognition is Google's MediaPipe. They have a live model running on their servers that is accessible through the MediaPipe Studio website with a webcam feed. I was impressed by the model and the speed in which it was able to classify different gestures. The gesture classification model relies on hand landmarks to model a hand pose, which can update in real time. In its base form, the model has eight output classes:

- Closed Fist
- Open Palm
- Pointing Up
- Thumb Down
- Thumb Up
- Victory (Peace sign)
- I Love You
- None (No gesture recognized)

Many of these gestures are the same exact gestures that Apple is using in their PhotoBooth app. The model also allows for customization and fine-tuning with new data. Overall, this model seemed like a great fit for the application, and if it did not meet my expectations, there were alternative options (other pre-trained models or training a new model from scratch).

With the model selected, I proceeded to start developing the *reactions.py* script. As stated in the General Notes section, the MediaPipe library is required to run this script. The library makes it fairly simple to import models, set options, and classify images. The model was set up to run inference tasks on images, rather than videos or a live feed. By setting it up for images, each frame of the webcam can be passed in and altered using different VFX. The model was also set up to detect two hands at any given time, which is necessary for recreating certain PhotoBooth effects, such as rain when two thumbs down are detected. Once the model was loaded and configured, a *run_webcam* function was developed to continuously capture each webcam frame. For each frame, the frame is converted to a MediaPipe image and passed through the recognizer. The recognizer returns the top gestures (one for single hand, two for two hands) as well as the hand landmarks. To ensure the model was working properly, I developed a function based off of an example from the MediaPipe API to annotate the frame with the hand landmarks. I also added text for the top gesture of each hand. Figure 1 shows what this looks like, this can be replicated if the DEBUG variable in the script is changed to True. The model works by first mapping these hand landmarks onto the hands in the image and then predicts the output class based on the landmark configurations.

As I tested this model, I was impressed with the speed in which it was able to adjust the hand landmarks and classifications. The classifications were quite accurate as well, so I decided to continue using this model for the remainder of the assignment. With the model working properly, the next step was to structure the code to run the VFX.

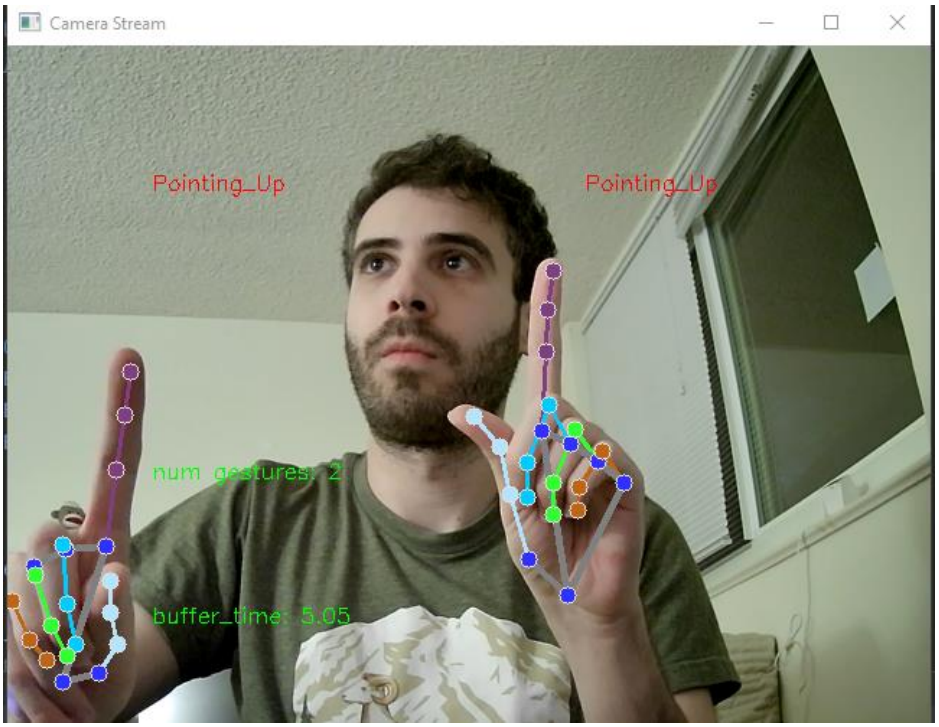

Figure 1: MediaPipe Model with Hand Landmarks

Architecture from previous PhotBooth scripts developed in this course were leveraged. Specifically, I used a transforms dictionary of booleans that had an entry for each possible gesture. Whenever a gesture was detected, the entry for that gesture was set to true. A *check_gesture* function was developed to check each gesture case and set the corresponding entry to True. At the end of the loop, these transform entries are checked again, and if they are set to True, the VFX are applied corresponding to the True gesture. One issue with this implementation that I noticed early on is that with this structure, the VFX can be applied on top of one another if the user quickly changed gestures. Without testing Apple's PhotoBooth app myself, I was not sure how they dealt with this issue. For the implementation in this assignment, I decided that I did not want the VFX to interfere with one another, and that only one effect should play at a given time.

In order to implement this one at a time VFX solution, I developed a buffer system. The time of each VFX activation (and current time) were saved for each frame loop. This allows for the calculation of the elapsed time of a given effect. If the elapsed time is greater than the buffer time, then all entries in the Boolean dictionary are reset and the model can inference again. I used a *can_inference* Boolean to control whether or not the gesture detection is included in the loop. This means that while an effect is playing, the script is not making any inferences, which also improves the speed and fluidity of the VFX.

With the structure of the code complete, all that was left to implement was the VFX themselves. For this assignment I decided to implement 5 different VFX:

- Thumbs Up – Bubble with thumb up emoji plays on screen
- Victory (Peace sign) – Balloons pop up on screen
- Two Victory (Victory with both hands) – Confetti shoots on screen
- Two Thumbs Down – The video darkens and rain falls on screen
- Open Palm – The background of the feed turns into space

For each of these VFX, I found a video online that well represented the effect. Most of these videos used green screens, or at least had a specific background color that would make it possible to filter out. The videos are all read in at the start of the script and a function was created for each effect. Since each function is called within the video loop, each function simply has to pull the current frame of each video.

To create the thumbs up effect, I took a video of a thumbs up emoji in a blue bubble. The video itself has a green background. First, the position on screen to place the bubble is set, and then a frame from the video is read. The frame is resized accordingly and the green parts are removed using thresholding of specific colors. Bitwise functions are used to create a mask and an region of interest (ROI) of the same size as the video. These video and the frame are then blended together and applied to the ROI in the frame. Figure 2 shows the thumbs up effect being applied on screen.

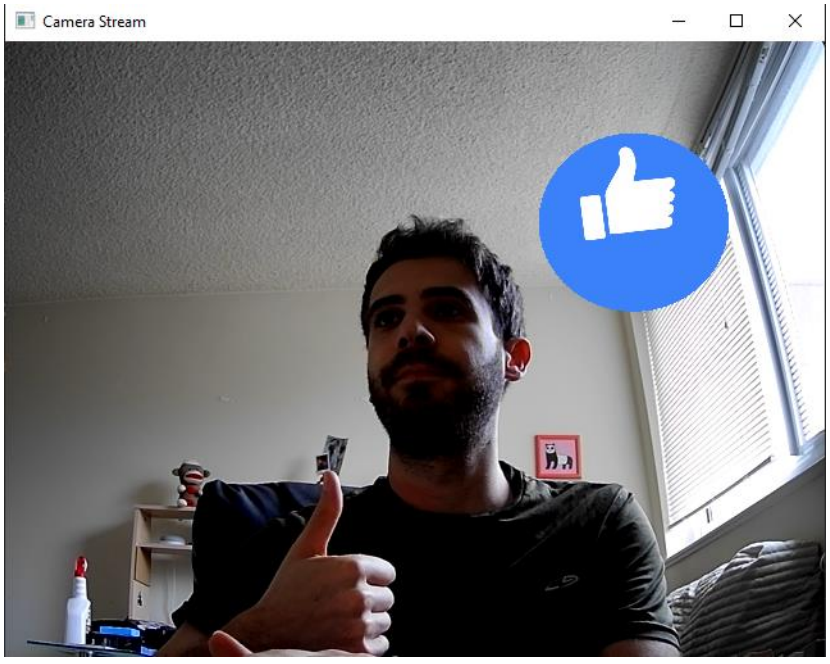

Figure 2: Thumbs Up Effect (Thumbs Up)

To create the balloons effect, I took a stock video of some balloons popping up on screen. The video itself has a green background. For this effect, I used the same functionality as the thumbs up effect. The video frame was read and thresholding was applied to remove the green. Some trial and error was necessary to get the threshold colors just right to ensure that the colors of the balloons were not removed with the background. A mask was created and used to overlay the video frame on top of the webcam frame. Figure 3 shows the balloons effect popping up when a victory sign is made.

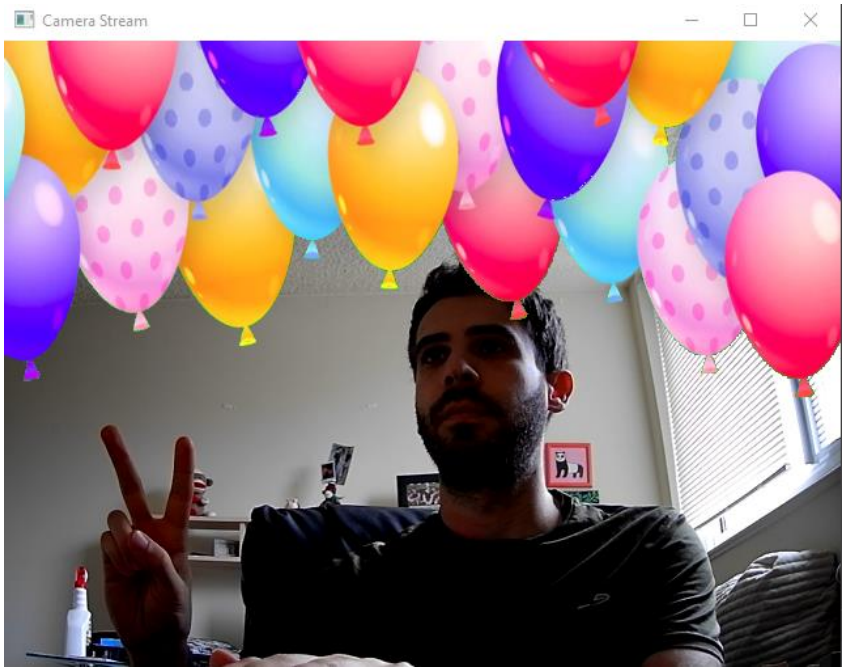

Figure 3: Balloons Effect (Victory)

To create the confetti effect, I took a stock video of some balloons popping up on screen. The video itself has a green background. For this effect, I used the same functionality as the thumbs up effect and balloons effect, however, this one was slightly more difficult to implement. Since the confetti is fairly small, it was difficult to extract the green background while keeping the full size of the confetti. Sometimes this would also create some unwanted artifacts on screen. I had to try a couple different confetti videos before getting one to work as intended. In the implementation, I cropped and resized the confetti video to essentially zoom in, thus enlarging the confetti. This allowed for the threshold to better filter out the green background. It's not a perfect implementation, but Figure 4 shows the confetti effect in action when two victory signs are used simultaneously.

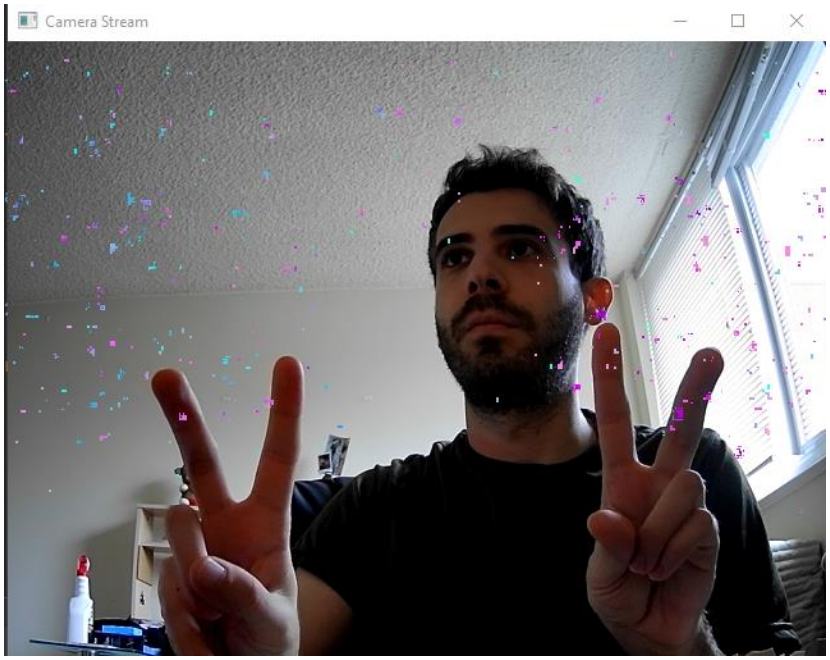

Figure 4: Confetti Effect (Two Victories)

To create the rain effect, I took a stock video of rain from online. This effect was actually the easiest of all to implement. The video of rain is on a black background, and with the rain drops being so small and fast, it would be difficult to use thresholding. Instead, since the intention of the effect is also to make the screen darker, I simply used the *addWeighted* function to combine the frame of the rain video with the frame of the webcam at an alpha of 0.5. Figure 5 shows the rain effect when two thumbs down are used.

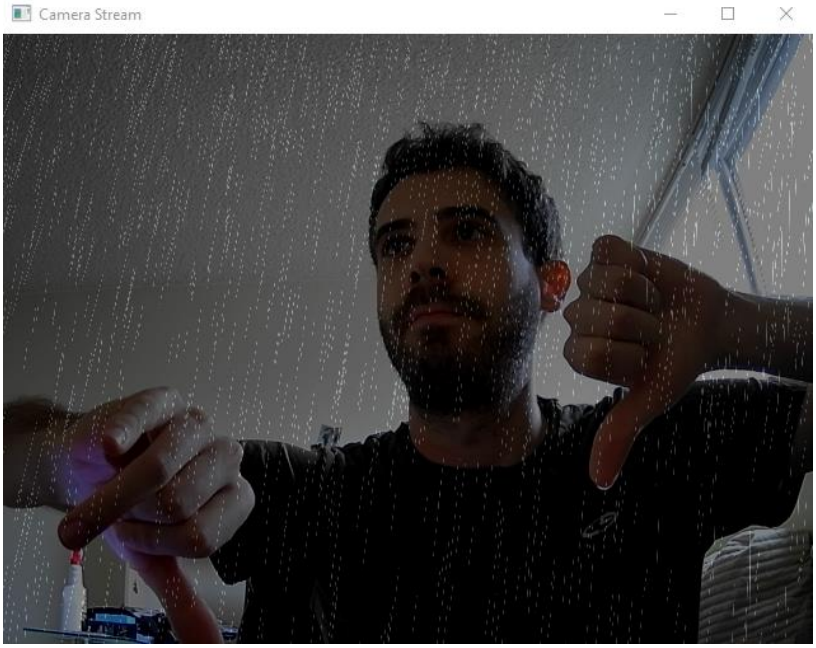

Figure 5: Rain Effect (Two Thumbs Down)
(Apologies for the weird hand sign, its difficult to capture the screenshot while doing two thumbs down)

The space effect was the most fun to create. When researching the MediaPipe model for gesture classification, I came across another model for image segmentation. The model is able to classify different segments in an image as different objects. For example, image segmentation is being employed for use with autonomous vehicles to classify signs, pedestrians, other vehicles, etc. This particular model is great at detecting selfies, meaning it should work fairly well for a webcam feed of myself. In the script, the segmentation model was imported similarly to the gesture recognition model. A couple settings were tuned and the model was ready to go. After some initial testing with the segmentation, it seemed to work fairly well at separating myself from the background. I found a stock video of space with some galaxies and stars and read it into the script. I developed an *apply_space* function to create this effect. First the webcam frame was turned into a MediaPipe image and passed through the image segmenter. The segmenter returns a category mask, which could be used similarly to alpha masks. A condition was created from this mask, where if true, the preserved image (myself) is used, and if false, the background image of space is used. The segmenter could definitely be improved, as it does not always perfectly distinguish the person from the background, but it works well enough for this implementation. Figure 6 shows the space effect, which can be activated when showing an open palm.

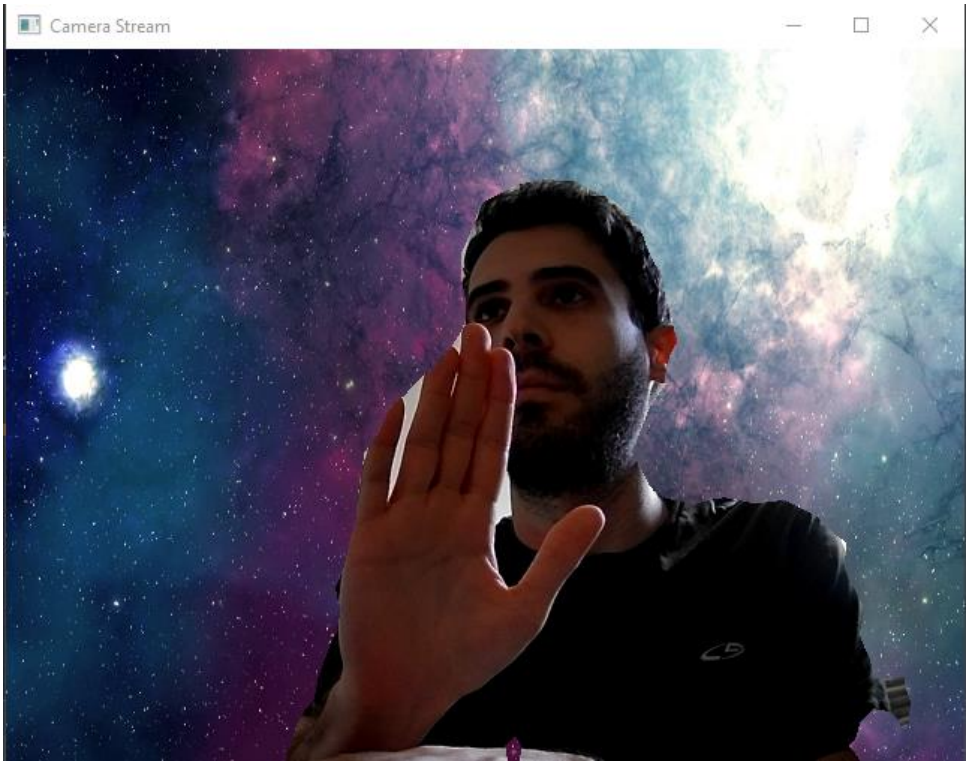

Figure 6: Space Effect (Open Palm)