MP4: Oncrop Reflection

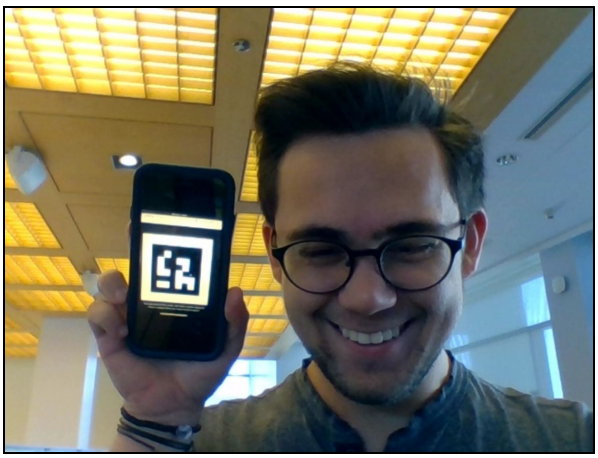
Elias Gabriel, Duncan Mazza

Project Overview

From the beginning of the project, our goal was to create a utility that anyone could use, would involve computer vision, and fit the form factor of a model-view-controller program. The use case we addressed was when friends/family want to take a group photo but are missing an individual and want to crop them into the photo. As an alternative to Photoshop, our utility crops the missing face in in real time, allowing the user to place the face using the position of their phone in the viewfinder. All of this functionality exists in an easy-to-use web application.

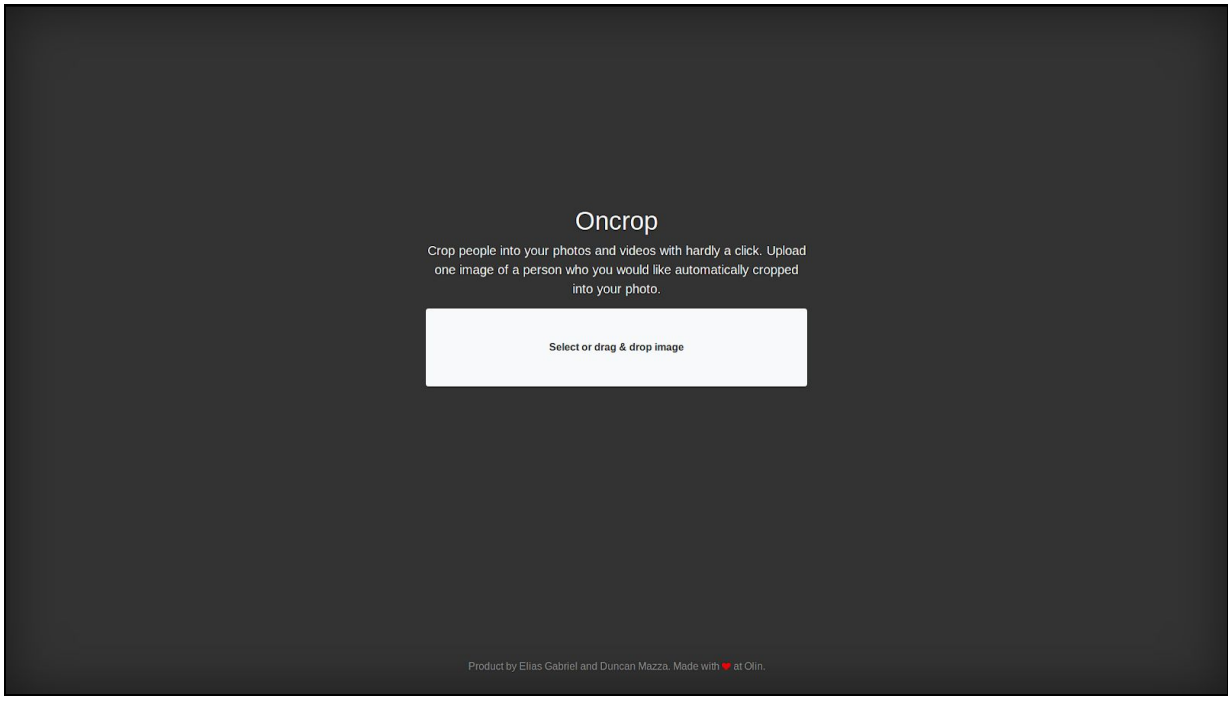
Results

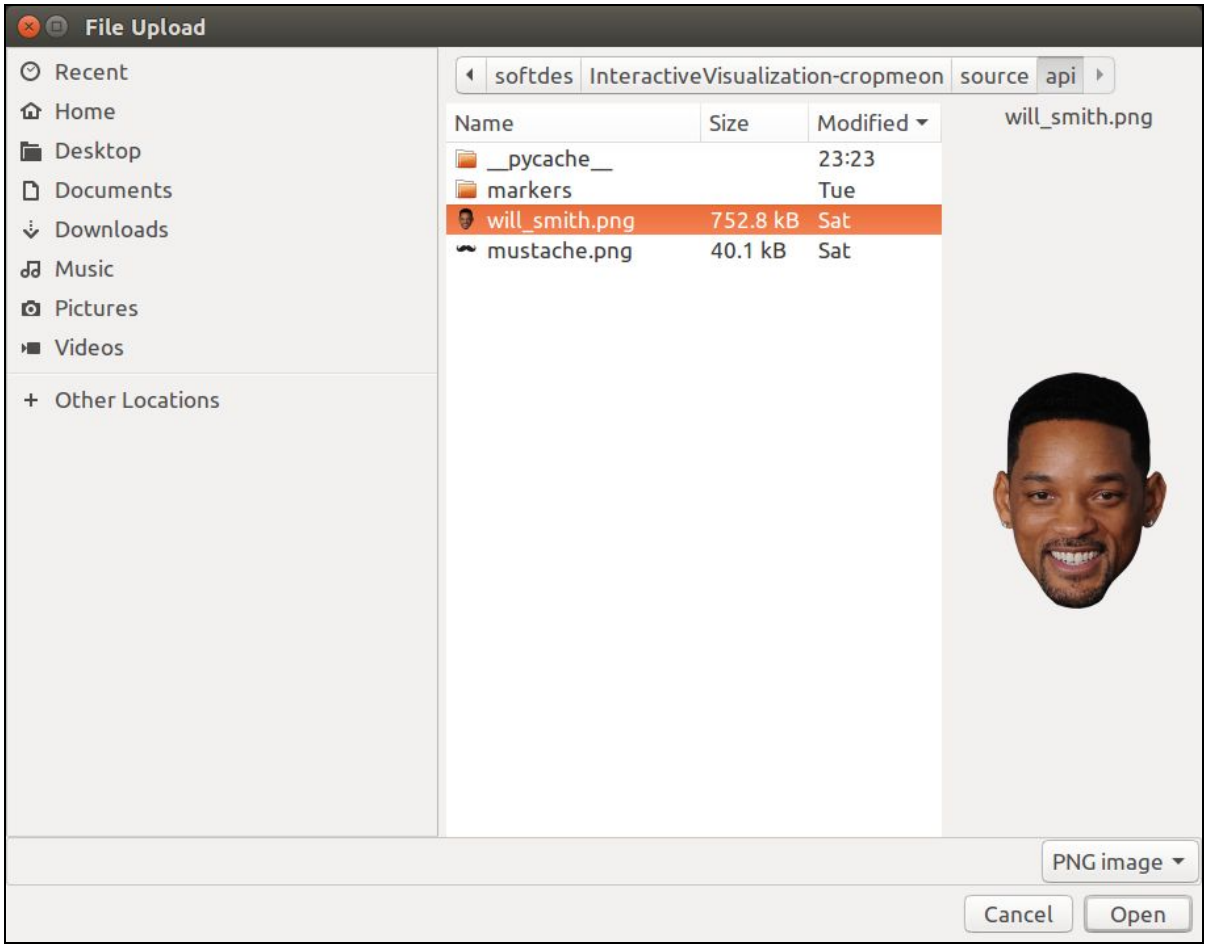
The final iteration yields a web application that allows users to, in real time, crop an uploaded face onto a picture. By tracking an AURCO and waiting until the user presses the spacebar, they can take their perfect photo.

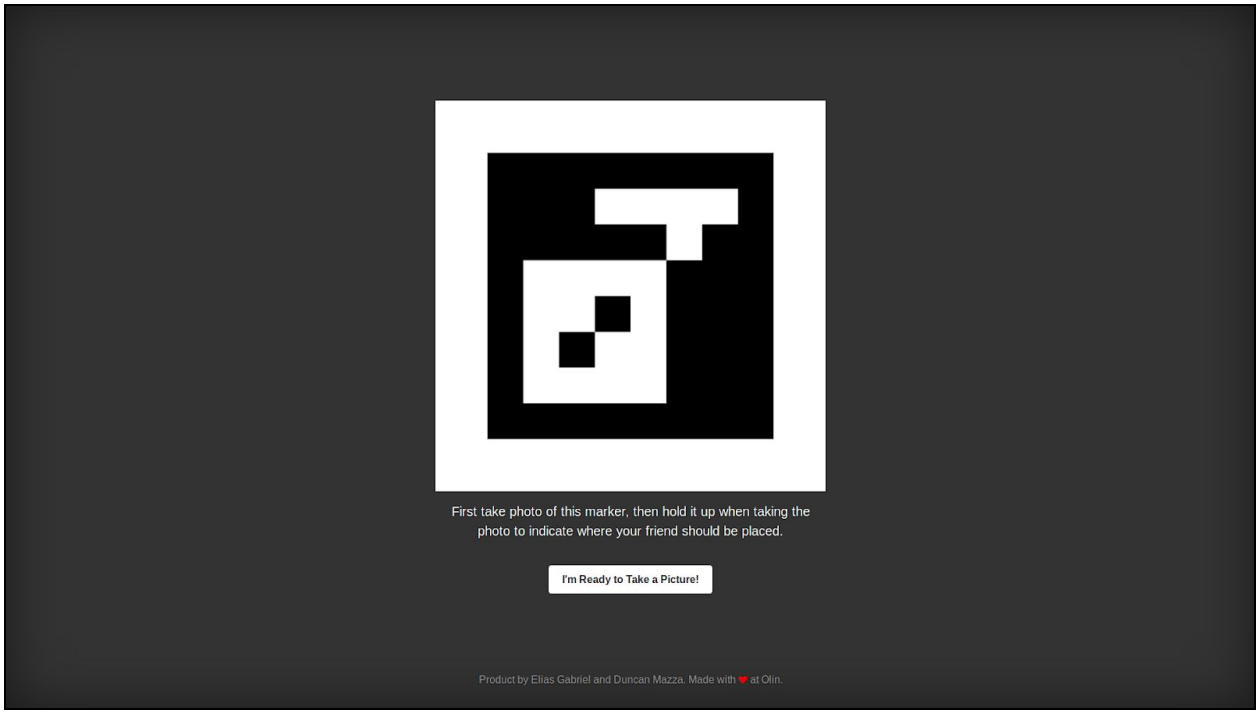


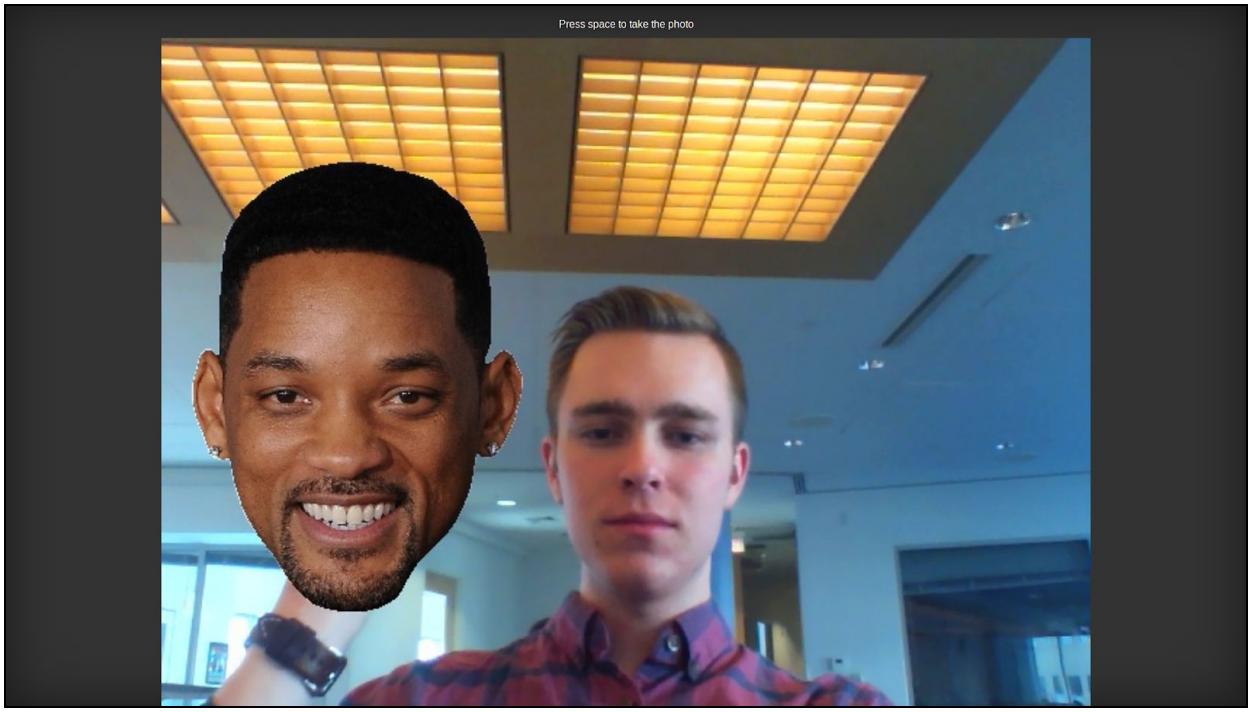

Flow of The Web App:

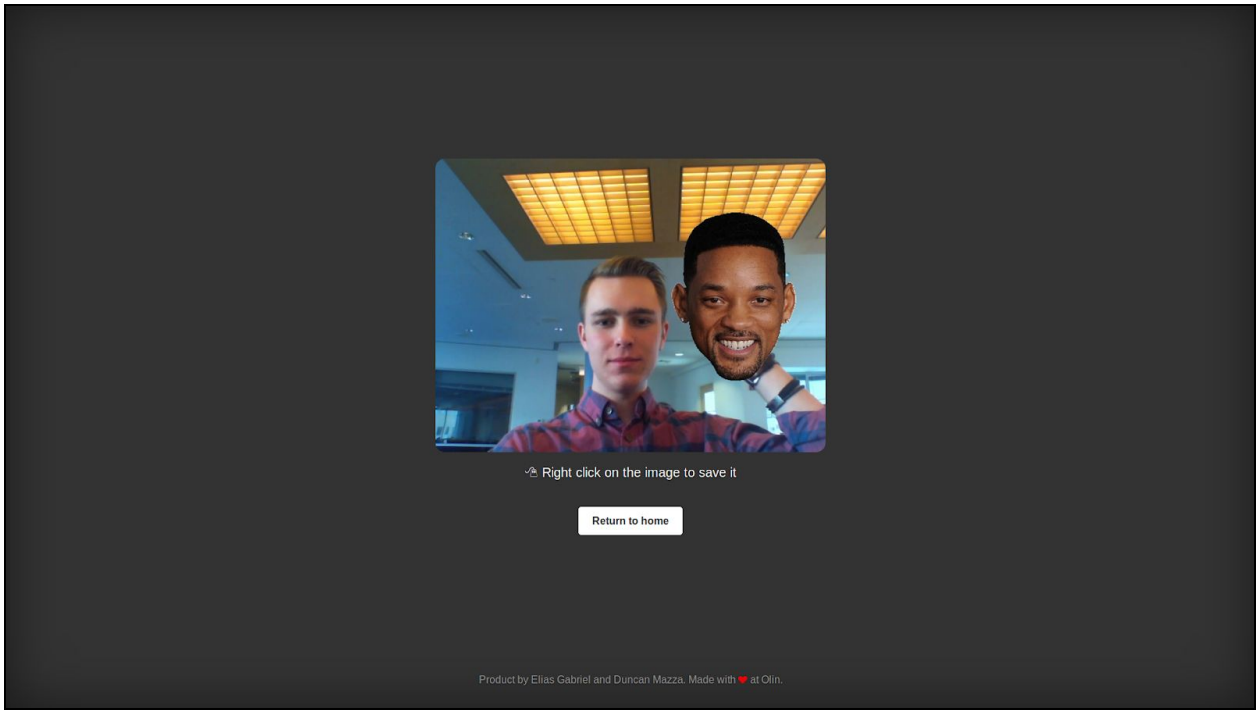
1. Index (landing page)
    a. File selection and upload
2. Marker (displaying the ARUCO code)
3. Snapshot (displays the viewfinder of the camera augmented with the overlaying of the image)
4. Show (displays captured image, prompts user to save photo and start the process over).


Oncrop
Crop people into your photos and videos with hardly a click. Upload one image of a person who you would like automatically cropped into your photo.
Select or drag & drop image
Product by Elias Gabriel and Duncan Mazza. Made with ♥ at Olin.


File Upload
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Other Locations
softdes
InteractiveVisualization-cropmeon
source
api
Name
Size
Modified
__pycache__
23:23
markers
Tue
will_smith.png
752.8 kB
Sat
mustache.png
40.1 kB
Sat
will_smith.png
PNG image
Cancel
Open


First take photo of this marker, then hold it up when taking the photo to indicate where your friend should be placed.
I'm Ready to Take a Picture!
Product by Elias Gabriel and Duncan Mazza. Made with ❤ at Olin.

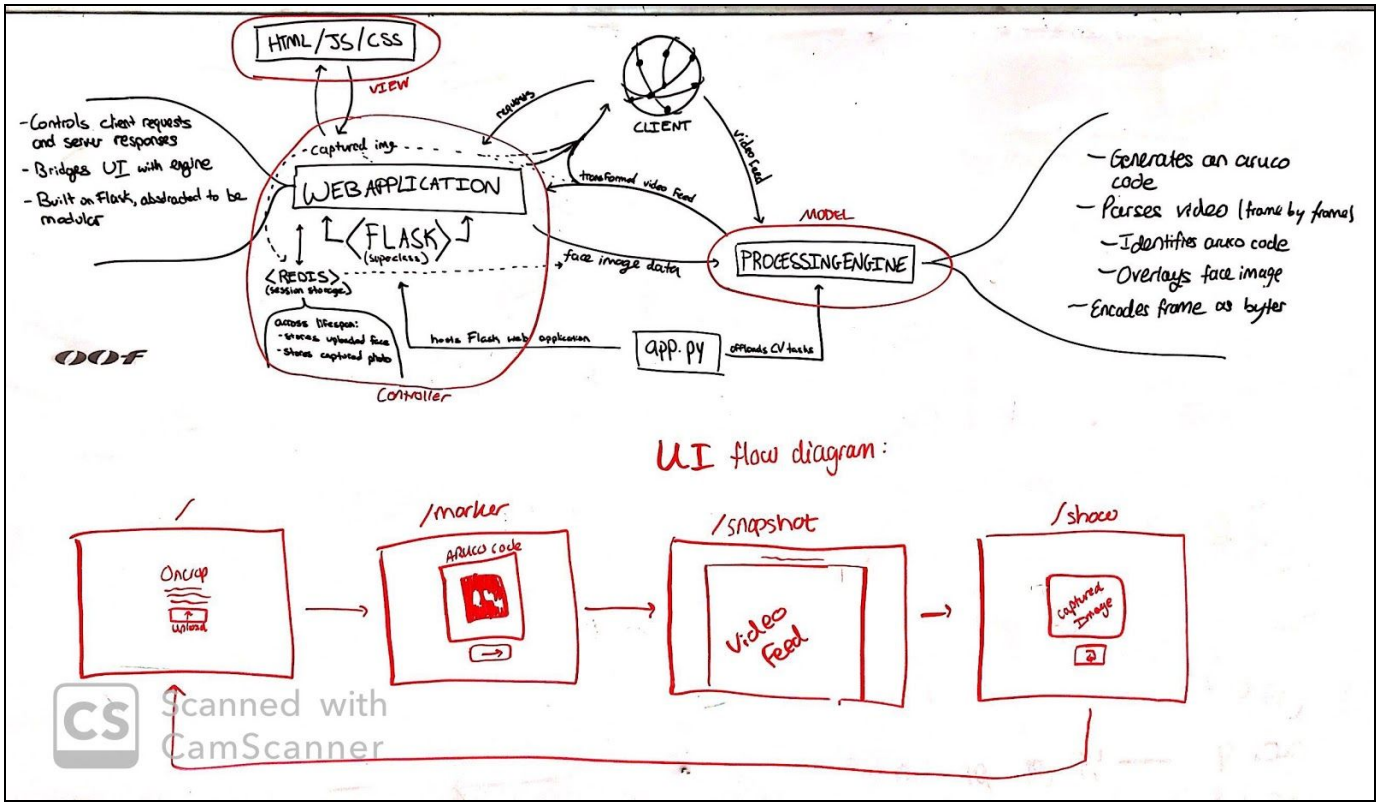

Press space to take the photo



The end result is an easy to use and aesthetic utility for anyone to use. The only limitation of the web app is that it must run locally, and therefore requires the user to download the files and run the app through their terminal. While we have made this process as simple as possible by encapsulating the setup code in to a single python script and making the web app just as easy to launch, ideally, this web app would be hosted by a remote server and accessible by anyone at any time. In a future version, this will be rectified. Nonetheless, for it being a download-and-use-yourself app, it is very easy to use, and successfully abstracts many of the complexities of data sharing between web pages and dynamically cropping a photo onto the frame.

Implementation

There are two principal components of our project, the web application and the OpenCV processing engine. Combined, they dictate how data flows between different web pages, and how the uploaded images are manipulated and transformed. Both of them are implemented as classes within importable subpackages. We structured the physical layout of the code in such a way as to make it more like an API, as in the code we wrote to make our specific app was not intertwined with the principal components.



ProcessingEngine is a custom class that builds off of OpenCV subpackages and algorithms. The algorithm used to detect AURCO markers is implemented in OpenCV, and likely uses an advanced form of blob and edge detection. Rather than execute all the video processing in a single function, we split it into several different “step” functions (which also removes the lag that comes from each function call in Python). The processing engine was written in such a way that the web app requests (through get_frame()) in a while loop, which makes displaying the image easy to implement in HTML. A simplified description of how this works:

1. The camera’s frame is captured, and the picture to crop is uploaded.
2. Corners of the aruco code are detected, and parsed into parameters for cropping the image onto the frame.
    a. Edge cases, such as the code being too close to the camera, are handled appropriately.
3. Depending on whether the image is a .jpg or .png, the code executes different methods for cropping the photo into the camera’s frame.
4. The resulting frame is converted into bytes and sent to the web app for rendering (the web app is not able to handle the numpy arrays that Python interprets the images as).

Rather than use Pygame for an interface, we opted to use the web. We wanted our project to be a simple and accessible as possible. By clearly separating the frontend and the backend, we could ensure that its potential users would not need to have advanced Python knowledge to use the app. In its final form, the application makes heavy use of HTML, CSS and JavaScript to create a web interface. While each page of our application is its own HTML file, there is a site-wide CSS file that dictates the aesthetics of every element on every page.

`WebApplication` is a class that extends Flask. Rather than implementing Flask directly, we decided to abstract away some of its core components into a simpler interface. In order to persist data across multiple different web pages, the app also makes use of session storage. The session is implemented using a package called Flask-Session, and configured to use a Redis server for temporary in-memory and fast data storage. As a client makes their way through the application, the server responds:

1. Attempt to route the user to the method with which the accessing web page is associated.
    a. If the page attempting to be access is recognized, execute the associated method.
    b. If the page is not recognized, Flask shows a “Error 404: Page Not Found” page.
2. Execute the method and send back the returned response.
    a. The controller responds with a HTML file or with JSON data, depending if the request was trying to retrieve data or submit data.
3. Upon face-image upload, the server creates a temporary file, saves the uploaded image, and stores the temporary filepath in a session variable to be used later
4. When the user gets to the “photobooth” stage, the web server creates an instance of `ProcessingEngine`, retrieves the uploaded image from the session and temporary storage, and sends it to the instance. Every frame, the `ProcessingEngine` instance runs the OpenCV code on the webcam feed, and returns the result to the web server to be sent back to the client and displayed.

Initially, we intended to make an application that simply accepted a taken picture, manipulated it, and returned the result. However, after some discussing and planning we decided to try and implement a live version instead. The live implementation proved to me much more complicated than the static version, but greatly (in our opinion) increased accessibility and usability.

Reflection

From a process point of view, we felt like we worked together really well. The division of work was successful because we adhered to our strengths while at the same time made sure that we were always in understanding of what the other was doing and also contributed in each other’s respective areas of strength. For Duncan, this meant taking a lead on the computer vision, and for Elias, this meant taking a lead on the web development. Furthermore, while we

both feel like the project was nearly over-scoped, we would still consider it scoped appropriately, and it's difficult nature challenged us to grow in our software development skills.

Our plan to do work was split between class time, time working together outside of class, and by ourselves in such a way that our class time was always optimized, we programmed together as much as our schedules allowed, and the rest would happen on our own time. This wasn't something we explicitly defined as we set out on the project, but it resulted out of a shared desired for the same work style. Additionally, there were no issues from a teaming standpoint.

On the computer vision side, we expected the ARUCO code tracking to be the most difficult and the cropping of the face photo onto the frame to be the easiest part, but in reality these were flipped. After initialization, the ARUCO code tracking worked within one line of code, and successfully cropping a dynamic photo (moving and of varying size) onto the frame without crashing Python because of incorrectly-sized matrices was much more difficult. On the web side of the development process, everything except styling the webpage proved to be more difficult than we had thought. As it turned out, it was surprisingly difficult to send data between different web pages. Thus, we wish we had known beforehand to not to look into blob detection and tracking as a backup to the ARUCO code tracking, and that we had instead diverted that time to improving the back-end of the web app. For example, we wish that the web app had a button on the final page that allowed the user to retake the photo without re-uploading the photo they want to crop, but we didn't have the time to figure out how to successfully access that data again.