ENPM 673 Project 1

Daniel Sahu
Brenda Scheufele

25 February 2020

1 Problem 1 - Detection

This section involved the process of detecting one or more given fiducials in a single frame. This is accomplished via contour matching and shape detection. The detailed procedure for each given frame is as follows:

1. Grayscale: convert the image from color to grayscale.
2. Filtering: The backgrounds of the given frames were generally uniform, so this step wasn't terrible important. We employed a median blur filter.
3. Thresholding: We converted the image to black and white via an adaptive threshold.
4. Contour Detection.
5. Contour Matching: Contours were compared to a reference fiducial template and all contours meeting a certain criteria (number of edges and shape matching error) were considered to be detected tags.
6. Orientation: The known non-symmetric properties of the reference tag were used to determine the correct "upright" orientation of the tag.
7. Identification: The ID of the tag was determined via the relative location of the innermost contour (if it existed). Note that, due to the small size of the innermost contour, this step had the most uncertainty.

In general the largest problem encountered during the detection process involved handling the variance / noise of the input frame. Detections of tags at far distances from the camera sensor are significantly worse than those that are close. The processes that relied on the most minute details of the detected tag (e.g. Identification) were naturally the most prone to error.

We tried to maximize the object oriented approach to solve this problem. To that end, the classes and selected methods are defined as follows:

```python
class ARDetector:
    """Object oriented approach to filtering and detecting AR
    tags in an individual image.
    """

    def detect(self):
        """Top level API for this class.
        Performs all operations (filtering, thresholding, etc.)
        Returns detected AR points and orientation."""
        ...

    def grayscale(frame):
        """Convert the given frame to grayscale."""
        ...

    def filter(frame):
        """Apply a median filter to the given frame."""
        ...

    def threshold(frame):
        """Threshold the given frame."""
        ...

    def _find_tags(self, frame):
        """Apply contour detection to determine all the tags in
        the given frame."""
        ...

    def _align_closest_point(self, ref, src, offset=2):
        """ Orient the given contours.
        This function returns a reordered src such that its first
        point is the closest point to ref[0] plus the given offset."""
        ...

    def _tag_id(self, tag_contour, child_contour):
        """ Extract the ID of the tag.
        Our assumption is that the inner contour represents
        the bitwise ID of the tag, and that only 1 bit will
        be encoded.
        Note that the given videos don't seem to follow this
        naming convention properly."""
        ...
    def _get_closest_points(self, c1, c2):
        """Returns the indices of the two closest points in each
        contour."""
        ...
```

2 Problem 2a - Tracking

The problem of tracking involves collecting and retaining useful information across a series of images (i.e. a video or in real time). In this case we desired to track a given fiducial across a range of images composed from a moving camera. A reference image is superimposed upon the fiducial, completely replacing it.

Given the location of the fiducial in image coordinates (i.e. the detected corners returned in part 1), we have the information necessary to calculate the mapping from the fiducial to an arbitary plane (in this case our reference image) via homography.

The general process followed for this part is as follows:

1. Homography: Calculate the homography matrix between the given fiducial location and the reference image.
2. Warp Image: Use the calculated homography matrix to "warp" the reference image. This transforms the selected image to appear to have the same orientation / position as the fiducial in the image frame.
3. Replace: Substitute the pixels of the fiducial (in the frame) with the warped reference image.

The class, its instances, and methods created for this task:

```
class ARTracker:
    """Class containing functions for AR tracking / manipulation.
    """

    def track(self, frame, ar_contour):
        """Core API; calls all other subfunctions."""
        ...

    def get_homography(template_corners, corners):
        """Compute homography between sets of corners."""
        ...
```

```
def warp_image(frame, H, output_shape):
    """Warp the given image with the given homography matrix."""
    ...

def replace_section(self, src, replacement, contour):
    """Replace the contour in src with the same contour in dst.
    Images must be the same size.
    """
    ...

def get_contour_x_bounds(self, y_val, X, Y):
    """Calculate the minimum and maximum X that are still
    within the given contour (X,Y) for the given Y value.
    """
    ...
```

3 Problem 2b - A virtual cube on the tag

Augmented reality applications generally place 3D objects onto the real world, which maintain the three dimensional properties such as rotation and scaling as you move around the "object" with your camera. In this part of the project you will implement a simple version of the same, by placing a 3D cube on the tag. This is the process of "projecting" a 3D shape onto a 2D image. The "cube" is a simple structure made up of 8 points and lines joining them. There is no need for a complex shape with planes and shading.

In this section, there are several functions required to perform the task. the list of corners (*detections*) for each frame along with the world coordinates of the AR reference tag, are used to calculate the homography matrix, H using the $classARTracker$ method, *get_homography*. This method takes in two lists of points and returns the homography matrix between them. The issues were making sure the two inputs were compatible, since one of the inputs, from $cv2.findContours()$ returns an array with extra dimensions, and ensuring you input them in the correct order. The resulting matrix H was then used to calculate, along with the given camera matrix, K, the projection matrix for the points on the cube using the function *projection_matrix*(*homography*), which is in main. This function takes as inputs the homography matrix H and outputs the projection matrix for each frame. Taking the transpose of K is the first step, and we found the numpy '$.T$' functionality did not seem to work, and so we used *np.transpose*(K). Also, it was important to RTFQ, and not multiply the cross product by λ in the algorithm.

The projection matrix, P, is then passed to the function that calculates the transformed 2D cube from 3D coordinates. $cube(proj\_mat, image)$ takes in P and an image, and outputs the image with the cube superimposed. This is done by constructing the homogeneous matrix form of the world coordinates from

$$W = \begin{bmatrix} 0 & 0 \\ 0 & 512 \\ 512 & 512 \\ 512 & 0 \end{bmatrix} \tag{1}$$

which becomes

$$WH = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 512 & 0 & 1 \\ 512 & 512 & 0 & 1 \\ 512 & 0 & 0 & 1 \\ 0 & 0 & -512 & 1 \\ 0 & 512 & -512 & 1 \\ 512 & 512 & -512 & 1 \\ 512 & 0 & -512 & 1 \end{bmatrix} \tag{2}$$

where negative values for z are pointed out of the plane of the image. Since we normalize the resulting matrix in the z direction, we then truncate the z-coordinate from the points list. We then get a list of 8 (row, column) points on the cube, the first four being the bottom of the cube and the last four being the top. Using a section of code straight out of the OpenCV tutorial, $draw(image, image\_pts)$, we then draw the lines connecting the points. This was useful code, since it illustrated the need to format the list of points as integers, and inputting the numbers as tuples. The pictures below are sample screen shots of our videos for the single and multiple input videos. The google drive where these videos are is: videos.

In a perfect world, the code we wrote would suffice, but there were problems. Sometimes the ID would not be made on the AR tag, and so no points would be available to project. This resulted in a very wobbly square. At first, when this happened it would end the program. Strategies were then implemented to reduce/eliminate this problem (such as the warp_image() function). Also, there were problems getting the cube sides to display. This was endlessly frustrating but ultimately was just a miscalculation in the projection matrix. The most significant issue was integrating the code. We both wrote to generate our own sections, then had to integrate the relevant parts. This required careful reading and understanding of variable data types and sizes. Lastly, debugging python in Spyder was a nightmare. Breakpoints did not work and variables would not display. Any suggestions for a good debugging IDE would be appreciated.

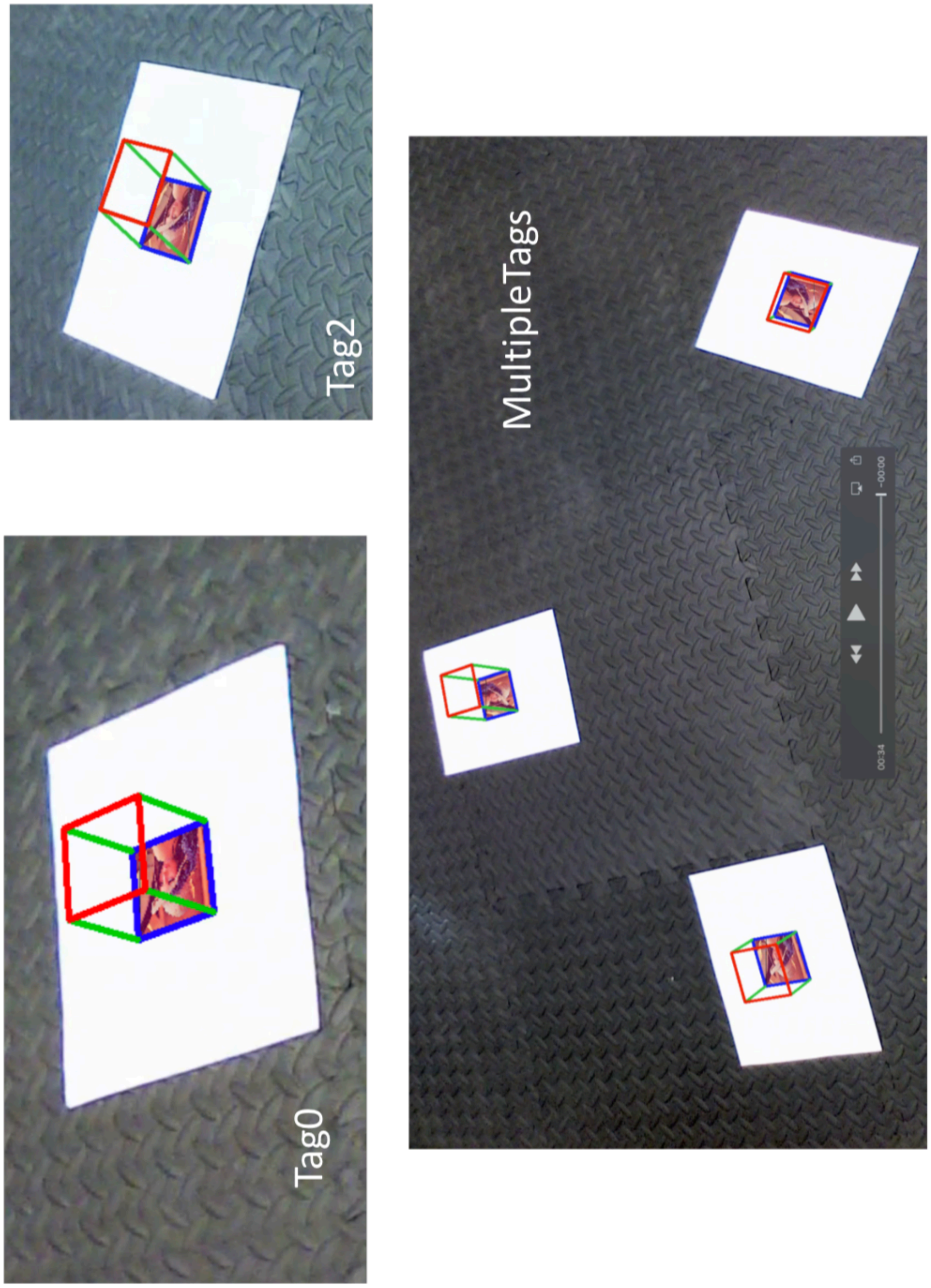

Figure 1: Screen shots of output videos for single and multiple tags