CS 378: 3D Reconstruction with Computer Vision
Motion Tweening and 3D Augmented Reality
Thursday, December 4, 2014

Group Members: Kevin Yeh, Matt Broussard, Kaelin Hooper, and Conner Collins

I. Overview

This report presents a motion tweening and augmented reality system designed to utilize 3D information recovered from a single 2D image to insert painted structures and mesh models automatically into the image and into subsequent frames of video. The goals of this project were twofold:

1. To replicate the parallel tracking behavior demonstrated in the 2007 Klein-Murray PTAM paper using analysis of a single two-dimensional image for multiple-surface detection.
2. To replicate the nonplanar motion tracking behavior demonstrated in Disney's Paperman short, which utilizes dense optical flow methods as part of the Meander system pipeline for motion tweening.

In particular, our overarching objectives were to successfully detect mappable planar surfaces in a single 2D image in real-time, to track and map models to these surfaces, and to paint and motion-track vector objects onto non-planar surfaces.

In the following sections, we describe our accomplishments in implementing several surface detection and tracking algorithms, a model mapping procedure, and a user painting and motion-tracking interface that uses this system in a more interactive and free-form manner.

This report describes the technical aspects of our implementation. Please refer to the README.md file in the code folder for details of how to run our code, including command line switches (of which there are many) and keyboard commands.

II. Background & Inspiration

The Klein-Murray paper above describes an approach for parallel tracking and mapping (PTAM) of a small workspace containing a prominent planar surface. They use epipolar geometry and corrective disparity mapping to perform SLAM with the workspace and is very robust to varying camera angles and degrees of zoom. We attempt and had reasonable success with such a planar surface tracking result, though our implementation is far less robust.

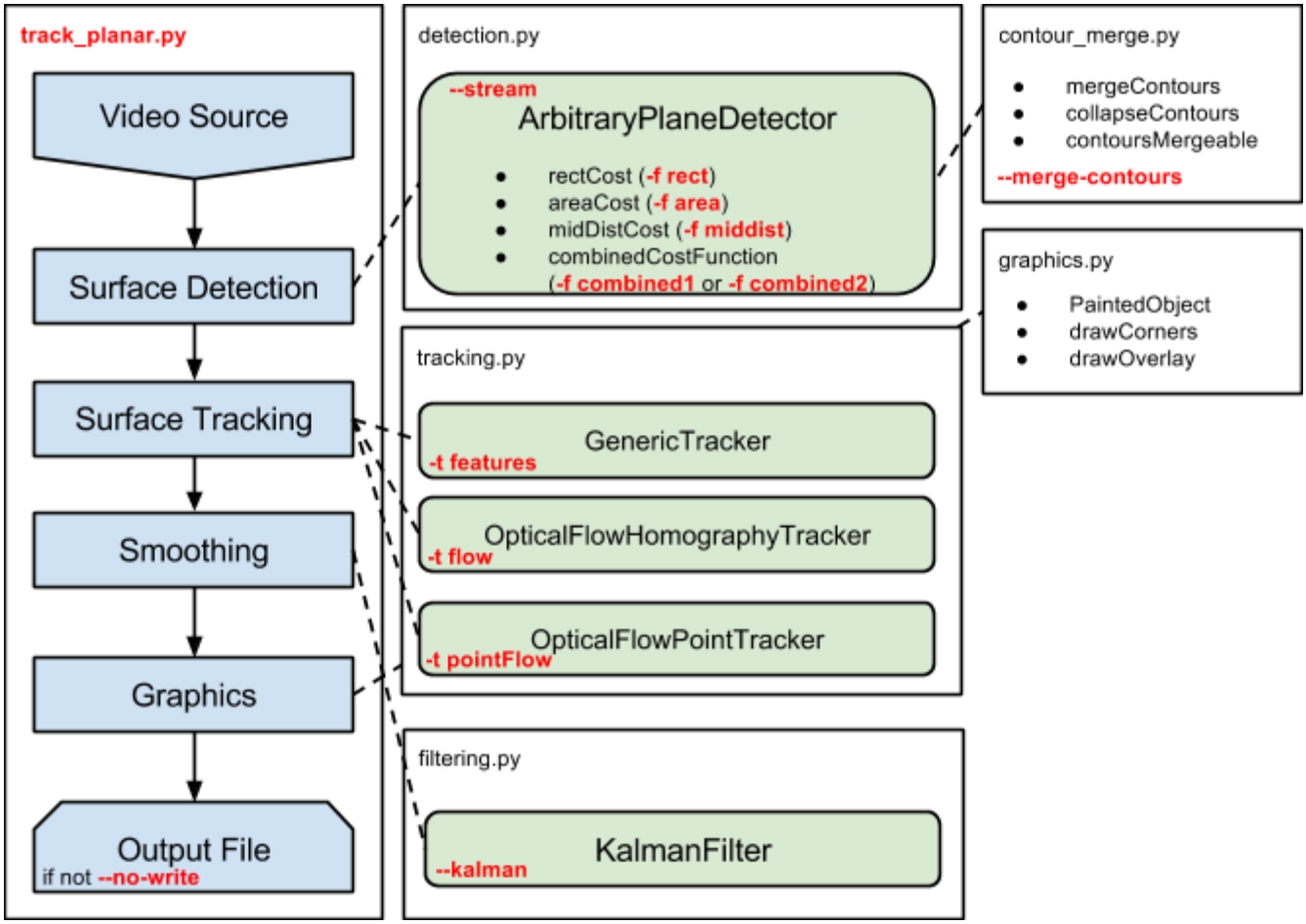
Meander, an animation system developed at Disney (Paperman, the video linked above, is one of the first proof-of-concept works using the technology) is able to track the motion of nonplanar surfaces and layer on a user-provided vector drawing. We implemented the tracking of user-provided drawings, but didn't get very far with tracking nonplanar surfaces.

III. Methods

Libraries, Languages Used: OpenCV (Python)
Hardware: Consumer Webcam

Our system consists of a primary user interface that iterates over a stream of static or real-time images and dispatches actions to several independent modules. Our main modules comprise a suite of detection, tracking, graphics, vector math, and contour merging classes and methods that make it easy to combine different algorithms for a variety of different video and image cases that may arise. Another module, smoothing and filtering, is included but not completed. This module is intended to reduce the error contributed by bad per-frame tracking measurements.

This diagram shows the structure visually:



Planar Surface Detection

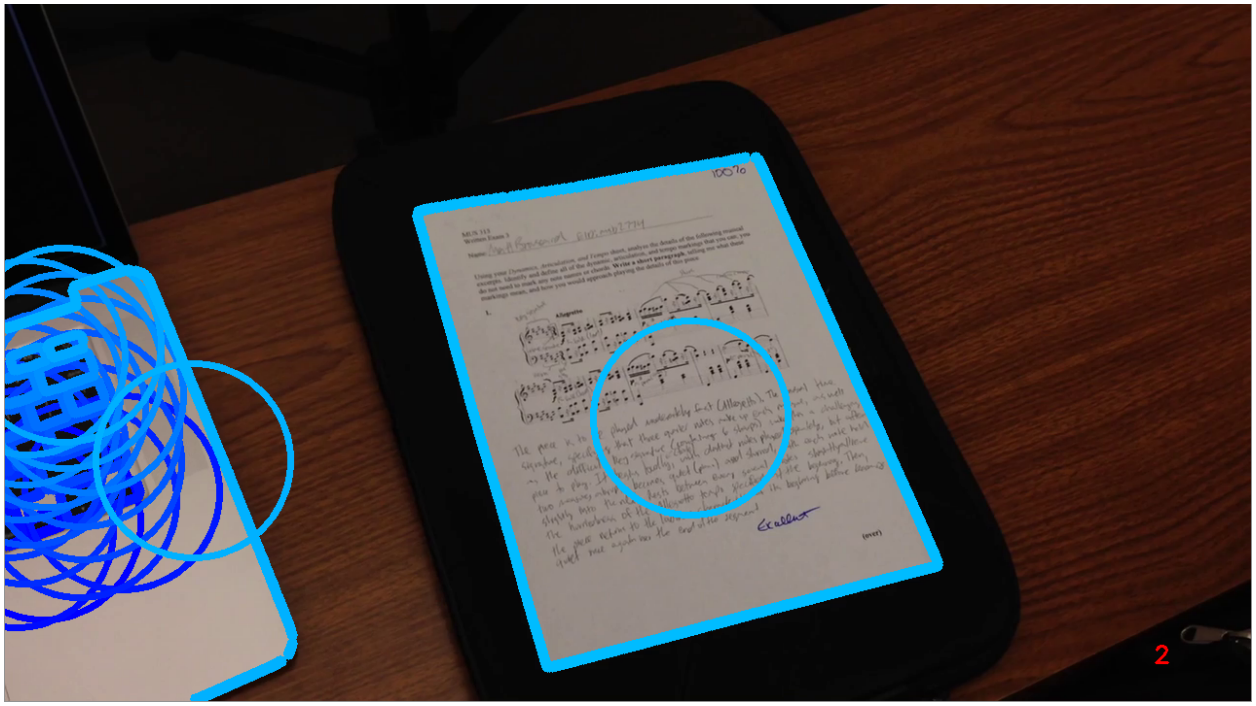
Our surface detection focuses mainly on rectangular surfaces. However, if no rectangular surface is immediately present, the system will give a best effort to return an approximate quadrilateral surface.

Overview:

- Contour Detection
- Contour Merging
- Weighted Scores/Cost Functions
- Quadrilateral Approximation and Corner Estimation

Contour Detection

The first step in detecting surfaces from a single 2D image is to extract the image's primary contours. The input frame is converted to grayscale, and a gaussian blur is applied to reduce the number of secondary edges detected by the following Canny edge detection application. The edge map is then dilated slightly before cv2.findContours() is applied to find external contours using the simple chain approximation algorithm.



In this image, multiple detect contours are differentiated by varying shades of blue. To further distinguish close together contours (particularly when we were implementing the contour merging described below), a circle is drawn around the average point of the contour.

Contour Merging

If contour merging is enabled, we attempt to combine multiple contours adjacent to each other in the list of contours. We determine if a pair of contours is a good candidate to be merged by computing a scalar entity called the “Hu moment”, a measure of how similar to a sample rectangle the contour is. If the contour resulting from merging the pair is deemed more rectangular than either of the constituent contours, the merge is retained. Otherwise, the original pair of contours is kept.

One problem with this approach is that it’s very sensitive to the ordering of contours in the list returned by cv2.findContours. In practice, physically adjacent contours usually show up adjacent in the list, but this is not an invariant that can be relied on. To really find the best subsets of contours to combine regardless of their position in this list would require at least $O(n^2)$ time.

Contour Cost Functions

Although ideally users should be able to choose which surface they wish to track, it is preferable to recommend users with a contour that is likely to be successfully tracked and mapped to. To that end, several cost functions were implemented to predict the effectiveness of tracking and mapping the desired planar surface in consequent frames of video.

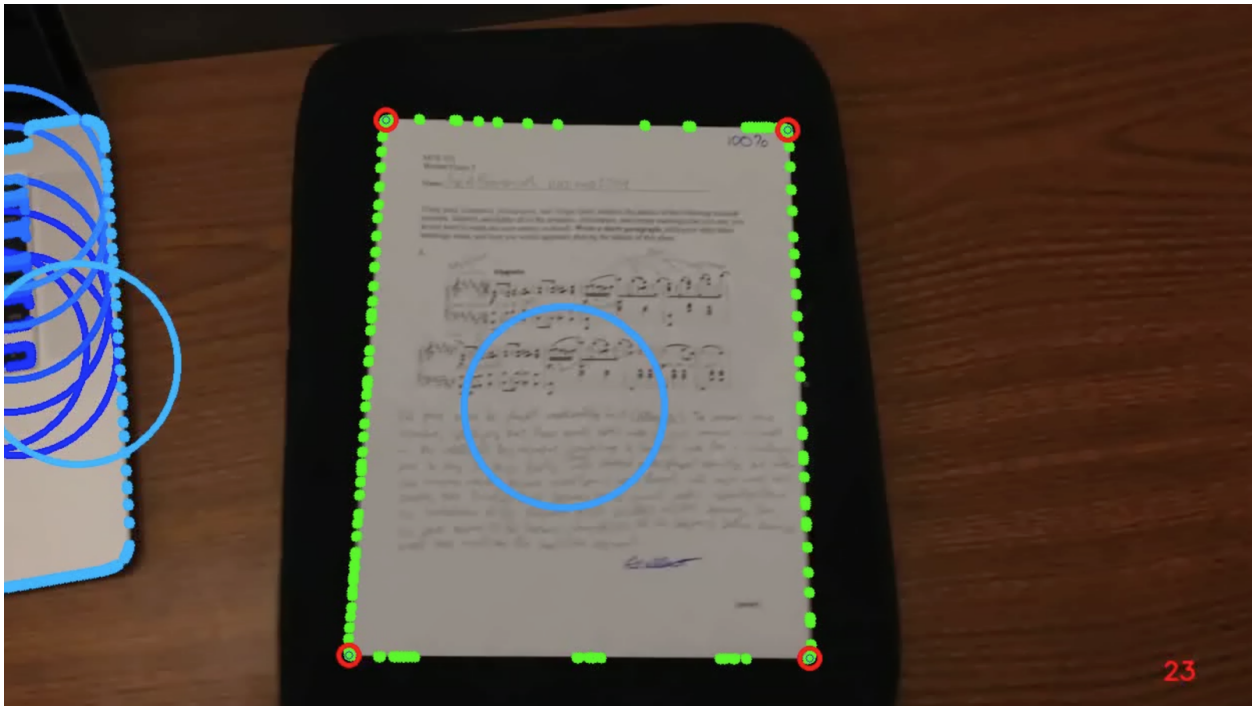
1. Rect: The aforementioned “Hu moment” similarity score between the contour and a sample rectangular shape. This is implemented using cv2.matchShapes().
2. Midpoint Distance: The average contour corner distance from the frame’s midpoint. This intuitively punishes contours that are farther away from the frame’s center and are therefore less likely to be the user’s desired surface.
3. Area: This cost is proportional to the contour’s area. This punishes contours that are farther away from the camera, or are small noisy details, since the desired surface is likely to be prominent in the frame.
4. Weighted: This is a weighted linear combination of the other cost functions.

After optionally merging contours, the cost function is computed for each remaining contour and the contour with minimal cost is used for the remaining corner estimation calculations.

Quadrilateral Approximation and Corner Estimation
To more easily track and map the planar surface, the contour is approximated as a quadrilateral with four corners. The algorithm starts with the basic cv2.approxPolyDP() function to approximate the contour as a closed polygon. The next step is removing duplicate corners that may arise during approxPolyDP, which is done using a simple pairwise distance test.

At this point, the contour has been approximated as a polygon with distinct definition points. Edge pairs are then continuously filtered, combining adjacent edges with nearly collinear endpoints in an attempt to approximate the contour even further. Finally, last-resorts are used to return a resultant quadrilateral.

If the polygon consists of two opposite corners, the other two corners can be approximated using the x-y distances between those two corners. If the polygon consists of three points, the last corner can also be approximated by finding the hypotenuse and estimating the corner location. If the polygon still consists of more than four points, only the first four points are kept. This is a final option and does not occur often in practice.



In this image, the blue dots and circles represent the detected contours as before. The green dots represent the contour selected by the cost function. The red circles show the corners of the quadrilateral approximated from the selected contour.

Surface Tracking

Once a surface has been selected, it must be tracked in consecutive frames. This is started with the “t” keyboard command. We incorporated a number of different trackers, and evaluate their effectiveness in later sections.

Overview:

- SIFT Feature Matching
- Sparse Optical Flow
- Dense Optical Flow
- Naive (Continuous Detection)

SIFT Feature Matching

Our first approach was to utilize feature matching to track features, either from a provided training image or from the initial tracked frame. This method is fairly robust, but is prone to failure in live-stream situations due to motion blur, dynamic lighting, feature occlusion. It also introduces heavy latency when live-streaming.

Sparse Optical Flow

This approach uses the Lucas-Kanade method of computing the optical flow of a sparse set of features, namely the corners of our quadrilateral. This is incredibly fast and works well when the corners have a good amount of contrast from its background, even when blur is present in the frame. Although it is less effective with faster-moving objects, its computational speed allows frames to be quickly fed into the system, generally minimizing the amount of change that can occur between frames. However, this is highly susceptible to corner occlusion.

Dense Optical Flow
Similar to LK Sparse Optical Flow, dense optical flow allows us to determine localized movement. However, dense optical flow is less feature-dependent and is generally more accurate. In particular, dense optical flow is useful for tracking points on more bland, textureless surfaces. As we will see, this can be used to great effect to track user-painted objects on animated and streaming video.

Naive Tracking (Continuous Detection)
Our final tracker is naive and merely runs the planar surface detector on each frame, rather than trying to track a previously detected surface. This can work well on clean inputs, where the detector is very accurate at detecting the primary planar surface, even in textureless environments.

Smoothing

We attempted to use a Kalman Filter to smooth some of the noise associated with tracking errors on individual frames. We defined a state that included positions and velocities of the 4 corners of the tracked quadrilateral. Unfortunately, this smoothing approach was too resistant to change and even with various parameter tuning we were unable to get it to perform better. Perhaps a different transition model would have helped.

Model Mapping

Once we have the corners of our quadrilateral, we can map a three-dimensional model onto our surface.

Model Normalization
On program load, we load our mesh model, in this case stored as OBJ files. We normalize our points between [0,1] and, for each face, determine their surface normal and resulting shade, assuming a uniform light facing [0, 0, -1].

Mapping
To correctly map our model, we require several things: the corners of a rectangle representing the default plane state, the corners of our plane in the current frame space, and the camera focal length. We can use the solvePnP method along with a perspective camera homography matrix to solve for the mapping from our default plane space to the current frame space.

Since our model vertices are normalized, we can easily map them to the default plane state, scale them as needed, and project our vertices into the frame space using the calculated mapping.

Finally, we can render each face using any common rendering interface. Since the lab machines do not support OpenGL bindings in OpenCV python, we decided to manually iterate over each face.

User Painting Interface

Beyond explicit detection, tracking, and mapping of planar surfaces, we also allow the user to paint onto the scene and track their object's motion in the world space.

Painted Object Definition and Integration

Our painted object is defined to be easily integratable with the preceding pipeline for surface tracking and mapping. When creating an object, we store the initial painting state as well as the corners of the bounding box. We then run these corners through a dense optical flow tracker on each frame, find the homography map, and map our original painting into the frame space.

IV. Results

Surface Detection

Arbitrary detection using a combination of Canny edge and contour detection + polygon approximation.

1.1. *Attempt to merge related contours in the image so that a rectangle can be more easily detected*
1.2. *Define cost function for selecting a contour or linked group of contours (see 3.2) to track*
1.3. *Result: Planar surface tracking no longer requires training image, detects surface correctly in most "good" frames (little blur or clipping).*

There's still a lot of room to improve. The weights of the cost function need more tuning, sometimes contours are merged improperly (or not merged when they should be), and sometimes the corners are improperly selected from the contours.

We had originally aimed for 90% frame accuracy, but had to reduce this to 70%. It can easily reach 90% once our corner choice algorithm is fixed, and some location

smoothing is introduced. We recently realized an error in our corner choice that rejected valid corners in several frames.

Textured Plane Video:
extreme blur, occluded corners, and secondary rectangular planar surfaces

Weighted Func without Merged Contours (Equal weights between Rect and MinDist):
- 72.5% Primary Surface detected (79/109 frames)
- 21.5% Wrong Corners selected (21/109 frames)
- 6% Wrong Surface detected (7/109 frames)

Weighted Func with Merged Contours:
- 61.5% Primary Surface detected (67/109)
- 17.5% Wrong Corners selected (19/109)
- 21% Wrong Surface detected or Bad Merge (23/109)

Rect Cost Func:
- 33% Primary Surface detected (36/109)
- 6% Wrong Corners selected (7/109)
- 61% Wrong Surface detected (66/109) [Picked secondary surface]

Area Cost Func:
- 54% Primary Surface detected (59/109)
- 17.5% Wrong Corners selected (19/109)
- 28.5% Wrong Surface detected (31/109)

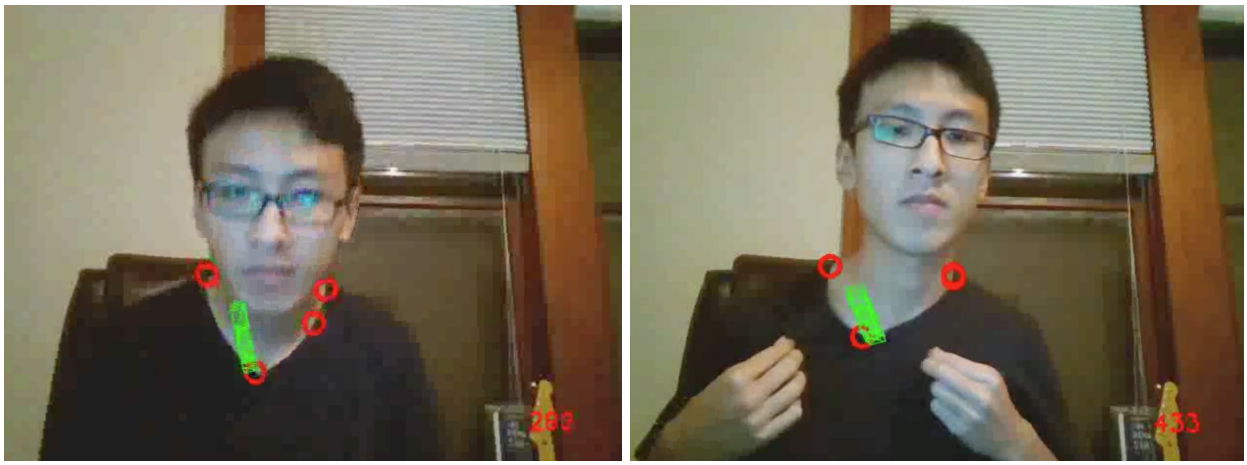
MidDist Cost Func:
- 75% Primary Surface detected (82/109)
- 19% Wrong Corners selected (20/109)
- 6% Wrong Surface detected (7/109)

Textureless Plane Video:
clean input, blank sheet of paper, no occlusions, no other distracting background

Weighted Func without Merging:
- 97.7% Primary Surface detected (384/393)
    - About 6 frames saved by 3-corner quad approximation
- 2.3% Wrong Corners selected (9/393)

Surface Tracking

Sparse Flow on Blank Paper: 100% Accuracy

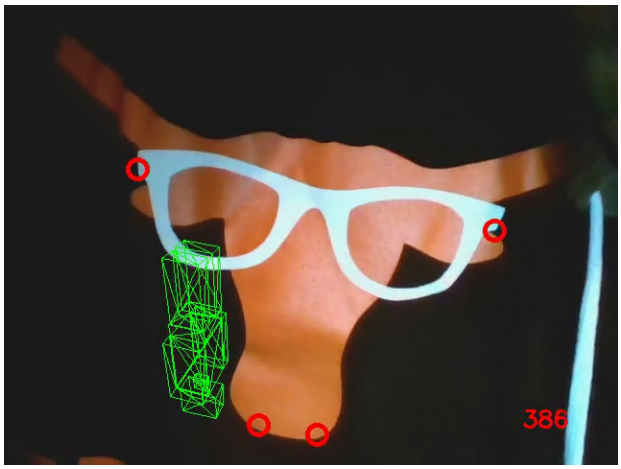


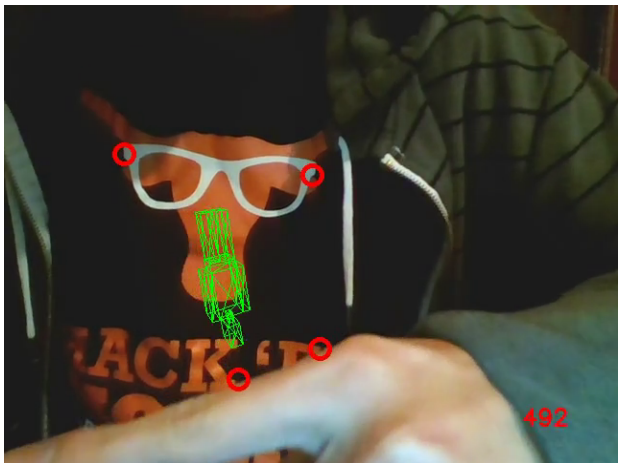

166


HACK 'EM
HORNS
188

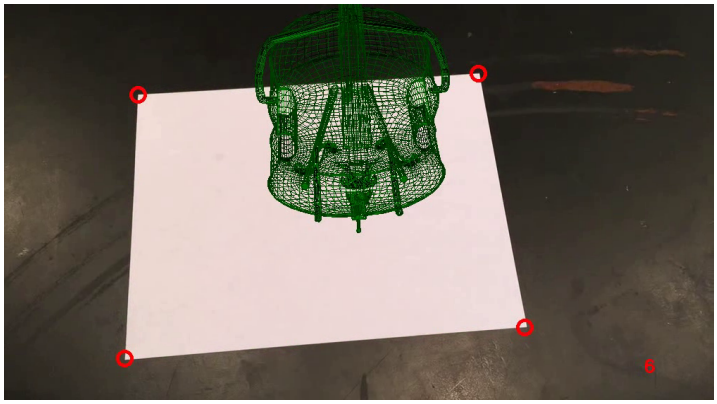

213

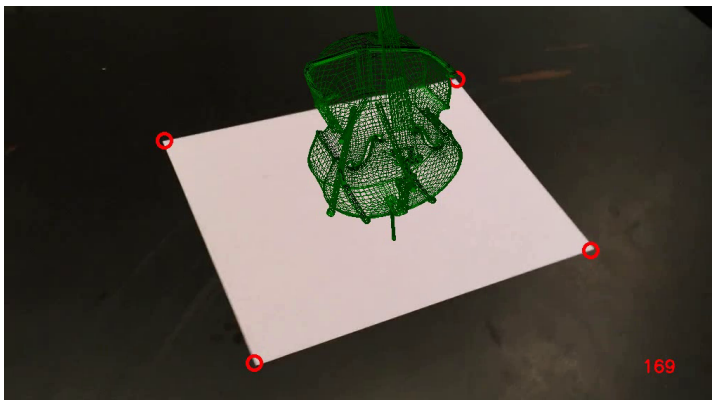

293

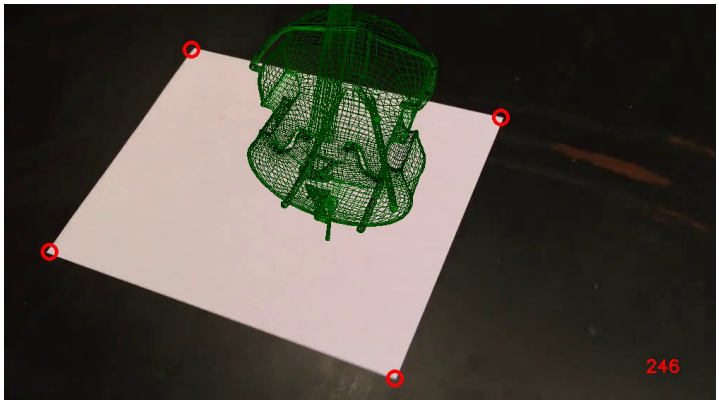

386

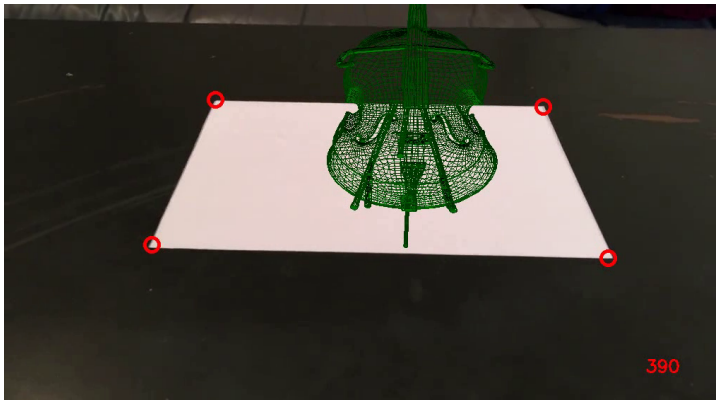

492

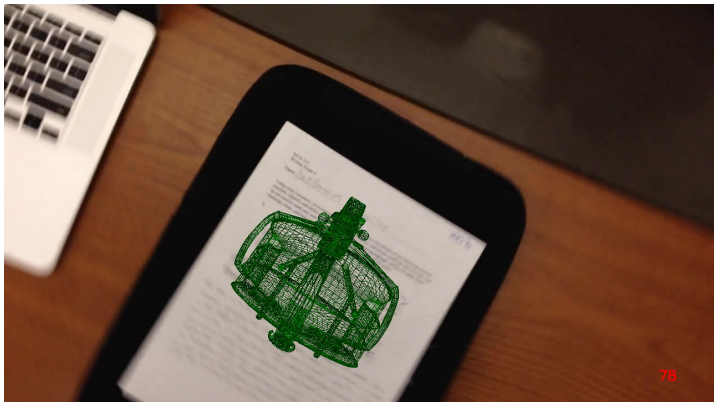
Model Mapping

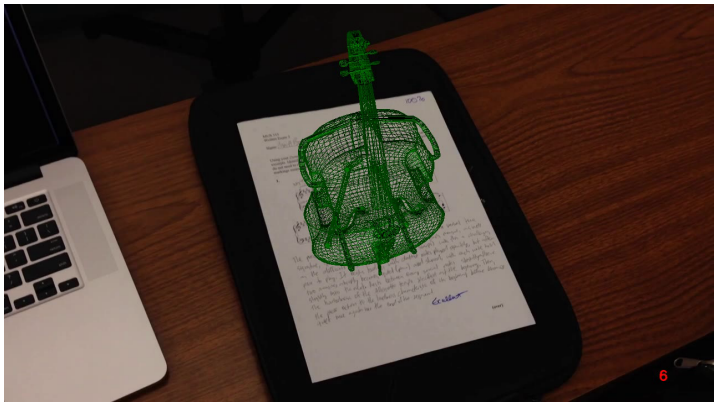




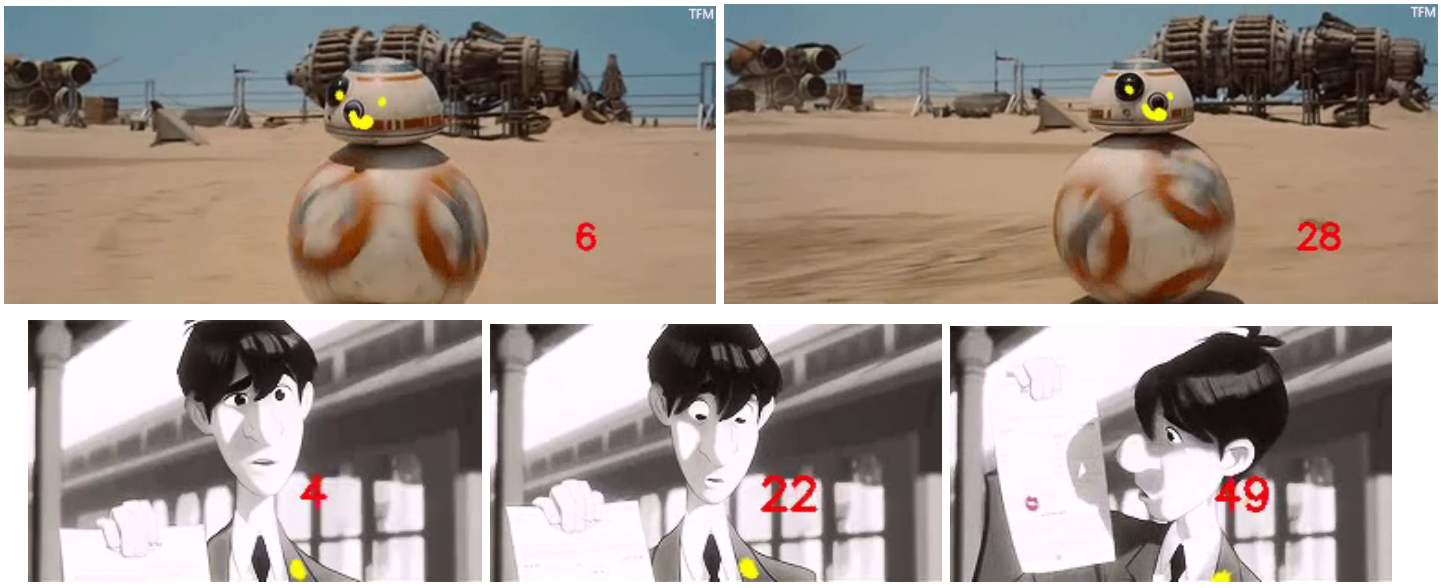








User Painting and Motion-Tracking



See the videos folder for the videos these screenshots are from.

V. Challenges

- Corner Intuition from colinearity (This still doesn't work about 10% of the time...bug)
- List of contours returned from cv2.findContours() not necessarily in the order that we thought; adjacent contours are not necessarily adjacent in the list, rendering our solution to merge the contours to be a bit of a fallacy.
- Difficult to come up with the right weights for the linear combination of features. This resulted in problems with detecting other unwanted rectangular/planar surfaces at the edge of the video (surrounding the surface of focus).
- Kalman filter doesn't work very well because it's too resistant to change. We tried tuning the parameters but couldn't get it into a good state.

Possible future avenues of exploration

- Decompose user-painted objects into points using findContour, track using those points (or find contour points with SIFT, SURF). This is how the Meander motion-tracking is implemented.
- Explicit detection of common non-planar shapes (e.g. cones, cylinders, spheres)
- Smoothing and speeding-up feature-based tracking using optical flow between feature-matching intervals