HoloPanel: Interactive Augmented Reality Panel Interface for Android

Abraham Botros
SUNet ID: abotros
abotros@stanford.edu
Stanford University, Winter 2015 - CS231M Final Project Report

1 Introduction

Augmented reality (AR) promises to bridge the gap between the physical and virtual worlds. AR would allow users to interact with mobile devices in novel and intuitive ways, and would allow better integration into everyday life, as users would not have to look strictly at a handheld device with limited screen and input space, and could instead essentially interact with the world in front of them.

Especially in recent years, AR has become more and more of a popular topic even in the public eye, with numerous appearances in media such as in films such as "Minority Report" and the "Iron Man" series. We have even seen the initial stages of formidable commercially-available AR devices for the near future, such as Microsoft HoloLens. Especially with the advent of ubiquitous smartphone ownership and wearable heads-up displays (HUDs) such as Google Glass and Microsoft HoloLens, utility-focused AR applications may become increasingly more practical and useful. Applications include such things as AR panels when walking, driving, or otherwise in-motion; elegant interfaces for seamless interaction without handheld devices in the case of wearable displays; intuitive interaction with AR objects, such as 3D models; and in general allow us to take baby steps towards some of the related AR or hologram-based technology prevalent through science fiction.

While many current AR applications for mobile phones and wearable devices focus on providing graphical entertainment, the current project aims to experiment with more practical implementations for every-day use of an AR interface. In particular, we aim to create an AR-based panel-like interface that the user can interact with using hand and finger motion to do things such as easily gleaning summary information (time, system information, location, weather, recent activity, etc.) or performing simple shortcut-like actions.

Overall, the proposed system here is implemented on an NVIDIA Shield tablet running Android OS, and uses continuous camera image frames to detect hand and finger location in front of the device in real-time. This first involves a calibration step, where the user shows their hand to the system, and the system extracts color/hue information to use for future reference for detection. In subsequent frames, the system finds similar hues to those seen in the hand in the calibration step, and uses this to detect hand/finger locations in these new frames. These locations are then used to interact with the AR panel interface displayed on the device screen, also in real-time. We observe reliable hand/finger location and accurate interaction with the panel interface in simple and even changing backgrounds, but see failure when backgrounds contain similar hues as the foreground hand/fingers. In general, these results suggest this system is a simple yet effective combination of hand/finger tracking and AR interaction, though there is certainly room for future improvement before such a system would be practical in real-world applications for end users.

2 Comparison to previous work

2.1 Previous work

In [4], the authors evaluate performance on a mobile AR system using marker-based finger tracking and the Qualcomm Augmented Reality (QCAR) SDK. Their work mainly focuses on comparing this AR system to normal touch-screen interaction, with results indicating that the AR-based approaches they implement are usually more fun and engaging than more traditional touch-screen interfaces, but not as always as quick or efficient for performing mundane on-screen object manipulation. Since their approach is focused on marker-based tracking, and we intentionally want to avoid markers to increase robustness and practical relevance, this work presents more an overview of the current near-upper-bound of what AR interfaces can do and how they compare with traditional screen interfaces.

In [3], Bradski introduces a now-common algorithm called CAMSHIFT, based on the mean shift algorithm. As we will see is similar to our approach for modeling hand/flesh probability, CAMSHIFT is applied to face tracking by forming histograms of hues in reference images to create a reference distribution of probabilities relating hues to flesh probabilities. This mapping is then used in subsequent frames to find the most likely region containing a face, with the main contribution of the CAMSHIFT algorithm being tracking of high-probability-density regions making small movements between frames using an adapting search window.

In [5], the authors present a robust system for markerless fingertip tracking for manipulating AR objects. They present a complex yet real-time system for robustly deriving a mapping from camera position to a single hand pose (with relatively-assumed fingertip positions) over successive frames. Using this single pose, fingertips are then segmented from the hand, and tracked in subsequent frames. Estimation of hand and finger position allows derivation of the camera location and orientation, which in the end allows for relatively reliable use of hand and finger location in the AR interface. However, we specifically did not want to assume a fixed hand/finger pose, as in real-world applications we would hope a system would be able to intelligently detect user intent given varied hand poses.

Numerous academic works, such as [6], and commercial products, such as Microsoft Kinect, provide accurate hand-tracking for often controlled environments with the help of depth information. This allows robust segmentation from background, potentially even with changing backgrounds, poor lighting, similarities in color between foreground and background, etc. While these could all lead to relatively high performance AR systems due to the accuracy in tracking body parts, depth-estimating devices are not currently widely available in mobile platforms (and certainly not available in devices like smartphones and tablets at the moment), so we intentionally avoid assuming depth sensor information input for our system.

Works such as [6] and [2] also require extensive setup and specific positioning of table spaces, cameras, users, etc. Similar to depth information above, this leads to large constraints on practical usability, and in particular makes such systems irrelevant to mobile computing applications.

2.2 Differences

The current work aims to combine several of the techniques and approaches used in the literature to develop a simple yet efficient system for end-to-end tracking and AR interaction. As hinted at above, though, the goal of this work is to implement a system that works with minimal assumptions and inputs. In particular, we:

- Allow both different and changing backgrounds. Users do not need to be sitting at a specific desk with extensive setup (as in [6] and [2]), and only need the single camera on the back of an NVIDIA Shield tablet, for example. There are no specific assumptions on the background, and therefore any background is technically permissable; however, as we see in Section 4, certain backgrounds do end up performing better/worse than others due to similarities in hand and background hues.
- Do not require a depth sensor. As mentioned above, many systems use depth information, but since mobile devices rarely have such a sensor, we do not assume it to be present.
- Do not require a specific hand pose, as in [6]. Users should be able to use the interface with a variety of deformable hand poses.
- Do not use markers, as in [4]; we intentionally want to avoid markers, as we cannot realistically assume end users would carry/wear markers throughout the day if using such a system.
- Require more than just the location of a high-probability-density region, such as is output by the CAMSHIFT algorithm discussed in [3]. In particular, we want to be able to find the actual locations of the fingertips, along with the number of convexity defects to get an estimate of the number of fingers being shown. As a side note, we also do not assume that the hands/fingers only move a small amount between frames; while this seems a reasonable assumption, this requires reliable initial estimates, which we did not assume at this point in the project (see Section 5).
- We require our system to function in real-time, using only the local processing power of a commercially-available Android tablet. More complex modeling of hand structure and other complex computations (including finger classification, etc.) were, as a result, not considered.

3 Technical approach

Our system uses the NVIDIA Shield Android tablet as our testing platform, with code implemented in C++ using OpenCV 2.4.8 and in Java/Android using the Android NDK interface. We first provide a quick, general summary of our approach in Section 3.1, and then break down the pipeline into more detail in Section 3.2.

3.1 Technical summary

Our approach involves first performing hand/finger tracking, and then using those tracked locations to allow the user to interact with an AR panel interface. We can therefore break our overall approach down into those two components as follows in this subsection.

3.1.1 Hand/finger tracking

We adapt an approach somewhat similar to that used in the CAMSHIFT paper ([3]) for finding the general hand location, and then augment slightly using standard computer vision algorithms to get finger locations. This is outlined as follows:

- Extracting reference hues from the user's hand on an initial calibration image.
- Creating a normalized histogram of these hues to proxy the probability of a given hue value being flesh.
- Back-projecting hue-flesh probabilities onto the image space.
- Using these back-projections to find the largest convex hull of pixels above some flesh probability, giving us the approximate hand location.
- Estimating the main pointing finger's location by evaluating corners of the largest convex hull.
- Estimating the number of fingers extended by evaluating convexity defects of the largest convex hull.

3.1.2 AR panel interface

Using the estimated locations of the user's hand and fingers, we then use these locations to allow the user to interact with the AR panel interface:

- The main pointing finger's location is used for selecting objects in the panel interface.
- The number of fingers is currently only displayed as auxiliary information, but could be used for varying actions and selections based on the number of fingers showing at the given time.
- Object selection triggers corresponding actions in the Android system, such as displaying information to the user, or potentially launching other applications (the latter was not yet implemented).

3.2 Technical pipeline

3.2.1 Hand/finger tracking - Initialization/calibration

To begin tracking, we first must initialize on a calibration image. Once the HoloPanel application is started on the Android system, the user uses the on-screen camera display to place their hand in a rectangular box in the image. Since most of the action in the HoloPanel application will be via pointing with the palm of the hand facing away from the camera, we find it is best if the user fills the entire rectangular box with the full back of his/her hand. The box outline surrounds the center 36% of the image (60% along each dimension). The user then taps the screen to initialize on a given frame. Our initial frame (and all subsequent frames) are of size 720x480 pixels. An example calibration frame is shown in Figure 1.

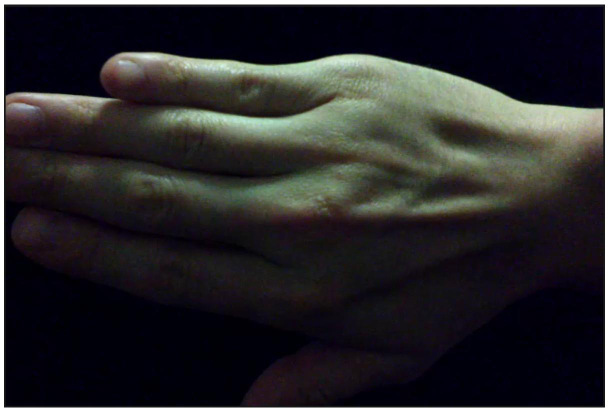
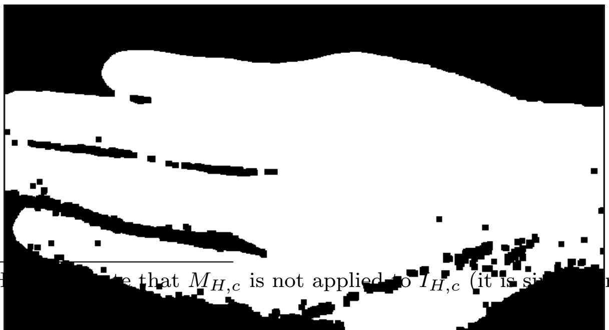


Figure 1: Example of initial calibration frame against simple black background.

Ideally, the user places his/her hand in front of a totally black background for calibration. Assuming this, we first convert the RGBA image to grayscale, and threshold out pixels with intensities less than some value (empirically, masking out pixels with values less than 10 of 255 seemed to work best, followed by eroding the mask around 3 times or so to eliminate some scattered noise). An example of such a binary mask is shown in Figure 2. The user is not required to use a completely black background, though, as this only marginally helps performance, if any; it is most important that the user simply fill up the entire calibration box with his/her hand, so hopefully no background would be showing anyway (see Section 5).



We then convert the RGBA of the center box to HSV color space, getting an image matrix $I_{H,c}$ (I for image, H for HSV, c for center region) that represents the HSV values in the center of the original calibration frame. Similarly, we can extract only hues to get $I_{h,c}$ (lower-case h to indicate only hue values as opposed to full HSV tuples). Using $I_{h,c}$, we create and store a mask $M_{H,c}$, which indicates if a given pixel has a valid HSV value or not; we use ranges from OpenCV CAMSHIFT examples, where the HSV value must be between the tuples (0, 60, 32) and (180, 256, 256). As explained in [3], this allows us to exclude pixels with poor S/saturation and V/values, which correlate to noisy H/hue values.[1] Lastly, we apply a very generous Gaussian blur to the hue values in $I_{h,c}$ (in practice, a kernel size of (21, 21) seemed to work best), which helps in smoothing our some of the noisy hue values we consistently get from the low-quality images we are capturing from our camera video stream. Without such smoothing, we are very prone to forming a flesh model based on noisy and misleading hue values. An example of the output of the hue extracted from the full and cropped (only the center region) calibration images are shown in Figure 3.

We then compute a histogram based on the hues we have extracted from the center of the calibration image. As in most CAMSHIFT implementations, we use 16 histogram bins. We also apply the mask $M_{H,c}$ we computed in the previous step when creating our histogram, completely avoid counting pixels that have invalid HSV values. We then normalize the histogram, giving us a normalized histogram $H^*_{h,c}$ (initial H for histogram, subscript h since based only on hue), where the * corresponds to the fact that this is used as our reference histogram for hand/finger/flesh probabilities for all subsequent steps. As a result, the exact histogram we form here essentially determines the performance of the rest of the runtime of this particular calibrated instance of our system. We hope that hues that have higher histogram counts/probabilities reliably correspond to pixel hues of ground-truth flesh pixels, and use this assumption going forward to measure the probability of future pixel hues corresponding to flesh, too. Figure 4 shows an example of the histogram formed from the initial image center $I_{h,c}$ in Figure 3 and using the mask $M_{H,c}$.

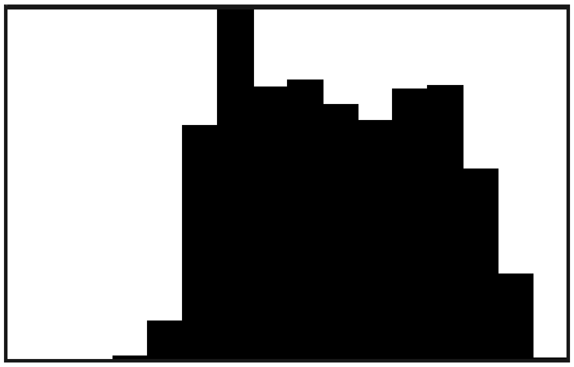
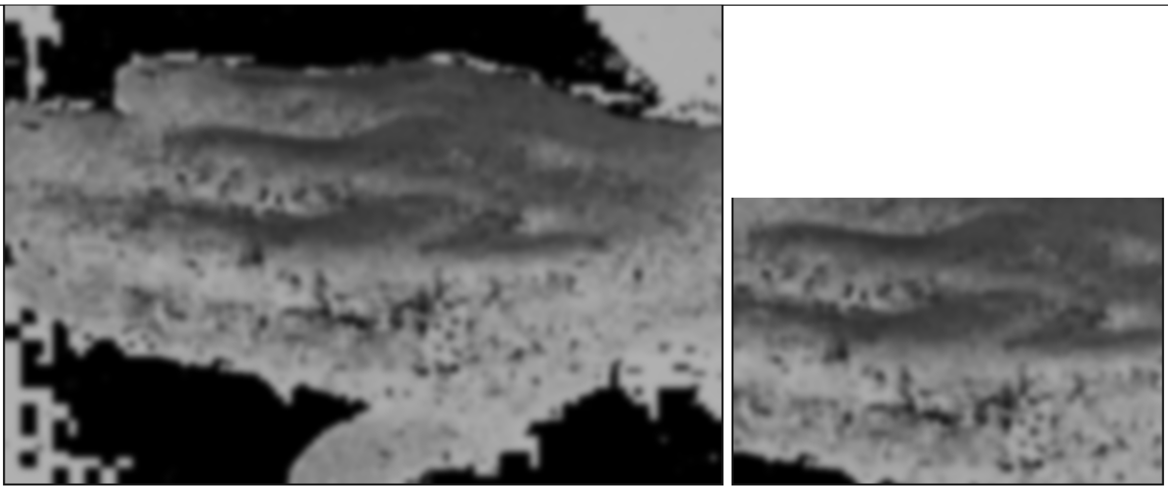


Figure 4: Example of reference histogram $H^*_{h,c}$ formed from $I_{h,c}$ (Figure 3b) using mask $M_{H,c}$.

[1]H[illegible]e that $M_{H,c}$ is not applied to $I_{H,c}$ (it is si[illegible]reated from $I_{H,c}$) and will only be used in later steps.



(a) (b)

Figure 3: Example of initial hue values extracted from the full calibration image (a; I_h) and the center calibration image region (b; $I_{h,c}$).

3.2.2 Hand/finger tracking - Tracking after initialization

Now that we have completed our initialization/calibration phase using our initialization/calibration image, we are ready to proceed with all subsequent image frames where we want to track hands/fingers for AR panel interaction.

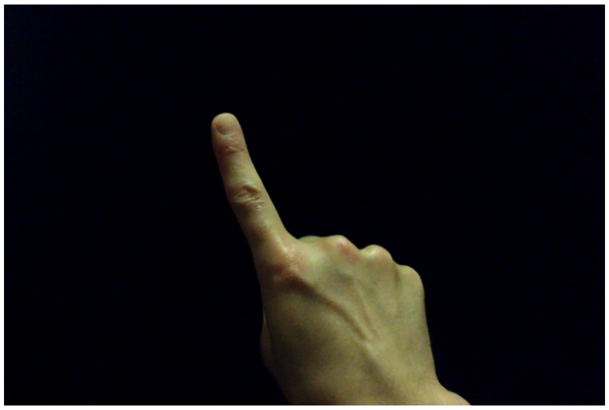


Figure 5: Example of new frame I' against simple black background.

Given a new image frame I' (we will use the Figure 5 as our running example), we convert to HSV (I'_H) and extract hues over the entire new image frame to give I'_h (shown in Figure 6).[2] When computing the new image's hue values, we store a new mask M'_H (mask based on HSV tuples; $'$ to indicate this is for our new image) representing whether each pixel has a valid HSV value or not based on the conditions explained earlier. We note again that M'_H is not applied to I', I'_H, or I'_h.

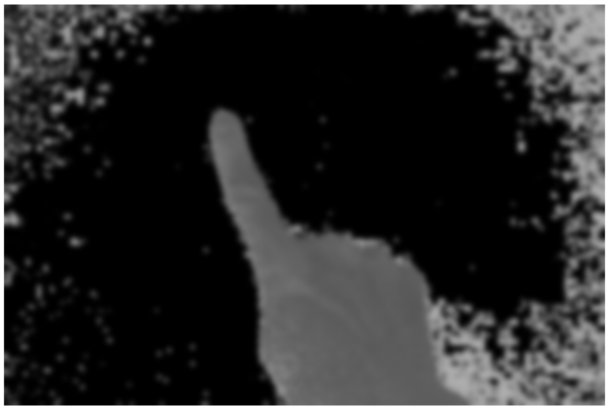


Figure 6: Example of new frame's extracted hue values, I'_h.

Given that we already computed our reference histogram $H^*_{h,c}$ in the previous section, and now that we have our new image's hues I'_h for each pixel location, we can compute back-projection probabilities for each pixel in any new image I'/I'_h.[3] Thus, for the hue for each pixel location in I'_h, we look up that hue's corresponding bin in our normalized reference histogram, assign it the probability of that bin, and store it in a back-projection probability matrix P' (P for probability). Again, the assumption is that, if our initial calibration image was able to create a reliable initial reference histogram with higher probabilities assigned to flesh-colored pixels, we can then assign reliable probabilities to any future pixels by using our histogram. We note that P' uses probabilities from our reference histogram $H^*_{h,c}$, but scales values to the full range of hues (0 to 180 in the case of applications using OpenCV).

Given that P' has the same size as our hue image I'_h, P' has a corresponding value for each pixel location that is some scaled value between 0 and 180 depending on the probability of the hue in that pixel location being flesh. We can then apply our previously-computed mask M'_H on our image-like P' matrix to zero out pixels corresponding to invalid HSV values. From here, we threshold our back-projection prob-

[2]Note that the word "entire" is specifically used here to indicate that we no longer are looking at only the center of an image; use of image centers was only for initialization/calibration, where we assumed the user's hand was contained in said center.

[3]Note that the reference histogram $H^*_{h,c}$ was formed on only the center region of our initial calibration image, but since it was normalized, it can serve as reference for any new entire image I/I'_h.

abilities (taking only pixel locations with a value of 50% of the max 180 value seemed to work best empirically), apply another generous Gaussian blur (again, with a kernel size of (21, 21)), and perform a single erosion on the image to again eliminate some unwanted, scattered noise. An example of P' for I' is shown in Figure 7.

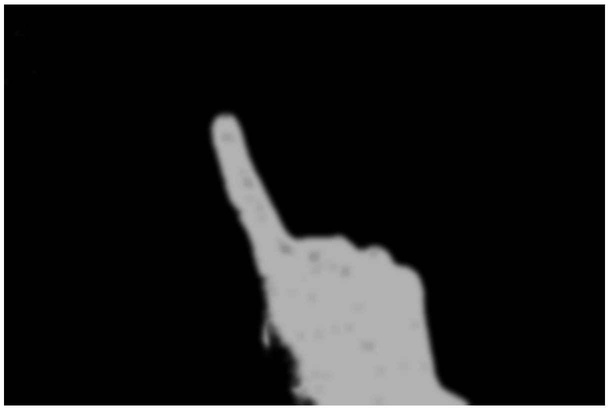


Figure 7: Example of back-projection probabilities P' computed from I' in Figure 5 and I'_h in Figure 6.

The back-projection probabilities in P' are then used extensively to compute the remaining unknowns for our system for this new image frame I'. First, to locate and isolate our prediction for the user's entire hand in the image I', we compute the contours C' of P', and use these contours to find the largest convex hull CH' of P'. We restrict that the largest convex hull must have an area larger than some minimum size (we used 0.15^2 the size of the full image area) and yet smaller than some maximum size (80% of the full image area); these restrictions help us eliminate detecting hulls that are either too small (likely the hand is not in the picture and we are picking up some other noise) or too large (likely detecting the hand and large portions of flesh-hue-like background instead of just the hand). An example of an accurate detection of the largest convex hull from P' is shown in Figure 8.

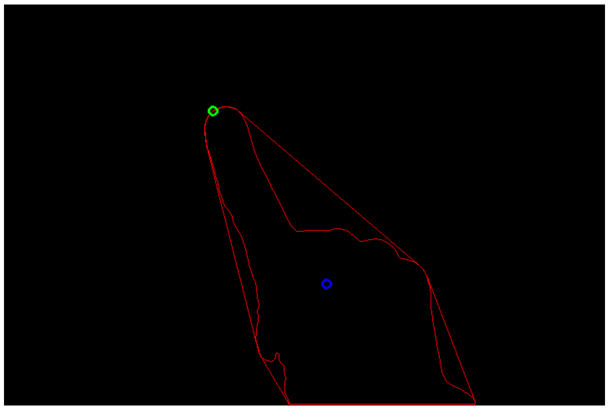


Figure 8: Example of largest convex hull CH' computed from P' in Figure 7. The blue circle indicates the location of the centroid $\overline{CH'}$, and the green circle indicates the location of the farthest corner F'.

Given CH', we then want to approximate the centroid $\overline{CH'}$. As in CAMSHIFT, we can do so by using image moments, and setting the location of the centroid to:

$$\overline{CH'}_x = \frac{M_{10}}{M_{00}} \tag{1}$$

$$\overline{CH'}_y = \frac{M_{01}}{M_{00}} \tag{2}$$

Where M_{00} is the zero-th order moment, and M_{10} and M_{01} are the first-order moments with respect to x and y directions of the image. The centroid of CH' is shown in blue in Figure 8.

Given the largest convex hull CH' and the hull centroid $\overline{CH'}$, we can then compute the estimated location of the finger the user is most likely using to point with and therefore to interact with the AR panel interface. To do so without restricting that any certain finger or pose is used, we simply find the farthest convex hull corner from the centroid; we restrict that this corner must be some threshold distance away from the image edges to avoid detecting corners along the arm or wrist, for example, when the hand is outstretched into the center of the image. This gives us the location of the main finger we want to track, F', as shown in green in Figure 8.

Lastly, we compute convexity defects on CH' to estimate the number of fingers being shown in I' (note that we will be using a different I' example just for the illustration of convexity defects). We use the contours C' and hull CH' corresponding to the largest convex hull, and count the number of convexity defects that have a convexity depth exceeding some minimal threshold and not having a contour contour that lies too close to an image edge (these would likely correspond to the defect between the thumb or pinky finger and the forearm). Counting the number of valid defects, we then take the number of fingers to be:

$$(\#\text{ extended fingers}) = \min\Big(5, (\#\text{ convexity defects}) + 1\Big) \tag{3}$$

An example of correct detection of 5 fingers is shown in Figure 9.

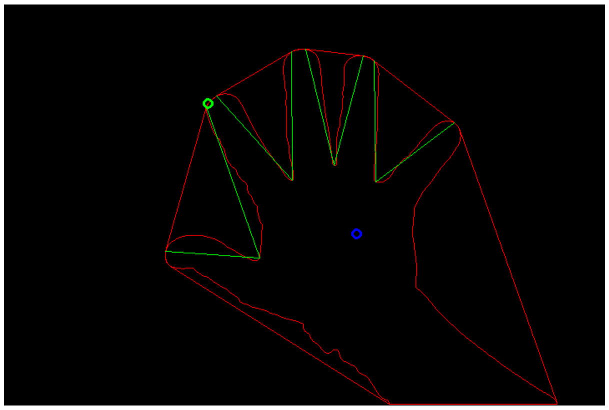


Figure 9: Example of convexity defects computed on some largest convex hull CH' for some image frame where the user has extended all five fingers.

3.2.3 AR panel interface - Interaction

Now that we have extracted all relevant summary information from our new frame I', we hopefully have a decently-accurate representation of where the hand and fingers are in the image, and in particular where the user might be pointing (F'). We have five possible panel selections/actions, as outlined in Section 3.2.4; therefore, we choose the corresponding five points in the image to serve as our action points $A_1, \ldots, A_5$ (A for action). If the user's finger (again, approximated by our point F') enters some radius around one of we then trigger the corresponding action.[4] Visualized on our running example I', we can think of circles of some fixed radius around our action points $A_1, \ldots, A_5$, as seen in Figure 10.

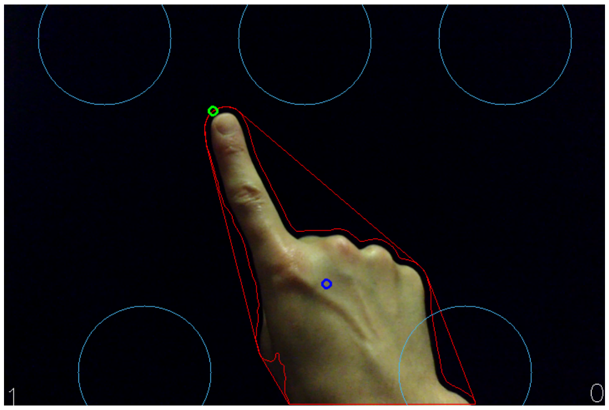


Figure 10: Example of circles of fixed radius around action points $A_1, \ldots, A_5$.

3.2.4 AR panel interface - Actions

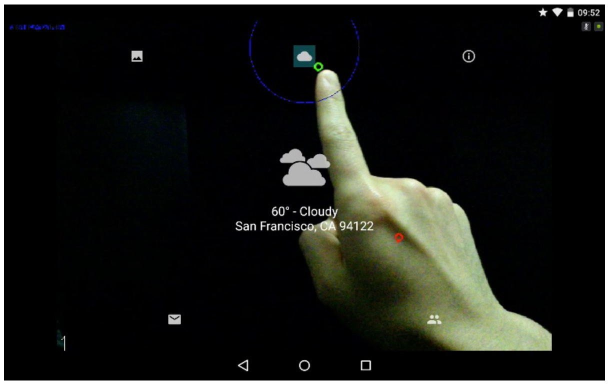


Figure 11: Example of AR panel interface. The main extended fingertip (at point F' in green) is shown selecting action A_2, which corresponds to checking the current location and weather.

An example of the panel interface running on the NVIDIA tablet in real-time is shown in Figure 11. The five actions are:

- A_1: (Top-left) Show a pre-selected saved image on the device (currently pre-compiled and not changeable by the runtime user).
- A_2: (Top-center) Show the current location and weather of the user.
- A_3: (Top-right) Show general system information, including current date, time, and battery level.
- A_4: (Bottom-left) Show current number of unread email messages (currently just displays a fixed number).
- A_5: (Bottom-right) Show current number of social media (Facebook and Twitter) notifications (currently just displays fixed numbers).

Note that the actions are simple so far, as we were focusing on the computer-vision aspects of the AR system; they could easily and certainly be extended as discussed in Section 5.

Each action point $A_1, \ldots, A_5$ always contains some icon representing the corresponding action, as seen in Figure 11.

When an action is triggered, a blue circle is shown surrounding the relevant icon (and corresponding to the circles shown in Figure 10), and a translucent Android view is brought to the foreground to overlay the screen with the relevant information. Icons, views, and other panel information are all drawn in Android, not in C++ or OpenCV (with the exception of the blue circles around selected icons), and allows us to avoid manually redrawing such complex graphics with each camera frame.

In particular, for action A_2, user location and weather are computed when the Android activity starts up, using the device's last known location and querying the Yahoo Weather API [1]. Weather information is parsed to prepare a corresponding weather image. For action A_3, time, date, and battery status are queried from the device on action selection.

4 Results

Videos of results can be seen at:

- `https://www.youtube.com/watch?v=oqX5HqUQyog` - Video ID 1 in this paper.
- `https://www.youtube.com/watch?v=qMgkO652SM0` - Video ID 2 in this paper.
- `https://www.youtube.com/watch?v=NZFL_Qf-qcQ` - Video ID 3 in this paper.
- `https://www.youtube.com/watch?v=_V23NV6LwnA` - Video ID 4 in this paper.
- `https://www.youtube.com/watch?v=6dmQIckfjDg` - Video ID 5 in this paper.

Preliminary results that visualize a bit more of the system (though in its earlier stages) can be seen at:

- `https://www.youtube.com/watch?v=WpzBGLEidbk`
- `https://www.youtube.com/watch?v=5rIsTagGW6I`

On the NVIDIA Shield tablet running Android 5.1, OpenCV 2.4.8, and using the Android NDK, our system obtains frame rates of approximately 25FPS on calibration (serving as a benchmark for our resolution and code, as we are simply drawing a single rectangle for calibration frames), and approximately 10FPS after initialization/during processing of new frames for hand/finger tracking and AR panel interaction. All work is done locally on the device, and no computation is off-loaded from the device to any server elsewhere.

The testing procedure was done on the actual device, and each time roughly involved a series of motions after initialization to primarily test tracking:

(1) Only index finger extended, moving successively between each region $A_1, \ldots, A_5$, with brief pausing at each action region. Note that due to the author's right-handedness and constraints on screen size, the thumb was often used to select specifically A_5.

(2) Only thumb extended, repeating the movements above,

[4] In practice, to avoid spurious selections, we enforce that the user's finger must remain in a given action point's radius for these points, at least five consecutive frames, or about half a second in a 10FPS system.

with slightly farther distance from camera.

(3) Test of detection of all five fingers extended, showing both palm and back of hand towards camera successively.

(4) Testing of detected/non-detected finger locations when the hand was removed from the camera view.

(5) Testing of detection upon the hand re-entering the camera view with index finger extended, and selecting A_2.

For the duration of each testing session on the different backgrounds (discussed below),

Table 1 shows results against different backgrounds and runtime conditions. Relevant notes for each runtime condition are then included afterwards. Each session took roughly 45-55 seconds (just depending on variations in the author's movements), and were only subsets of the linked videos given. The author manually counted image frames where the estimated main fingertip location F' (see Section 3.2) was or was not either touching the fingertip or within approximately 1 centimeter from the finger (in real-world, not image, coordinates); frames where the estimated fingertip location was not accurate according to these criterion were labeled as error frames. As a side note: this was a rather stringent criterion for error analysis, as even estimating the fingertip was instead along the finger, somewhere inside the hand region, or on a different edge of the hand would constitute error frames, though relaxing this criterion would not have improved results too significantly.

- For the black chair background (video ID 1), all errors occurred when the hand was totally out of view, and were due to spurious detections of small convex hulls of light reflecting off the back of the chair.
- For the white wall background (video ID 2), all errors occurred due to the estimated fingertip location F' switching to an erroneous hand edge when the finger was bent towards the hand and the hand was near the edge of the image.
- For the desk background (video ID 3), all errors occurred when the hand and finger were in compromised conditions, as in the white wall background condition.
- For the wood floor background with some background movement (video ID 4), results were quite terrible. See Section 5 for a discussion of these results.
- For the backyard/outdoor background with significant camera movement and background changing (video ID 5), results varied greatly depending on initialization, and best results for best initialization in the full test video were shown (in the video ID 5 link, this was approximately from 00:41 to 01:34 in the video). However, results can dip down to the 50% error/success level or lower depending on the initialization and specific backgrounds.

We lastly note that no formal testing of the panel selection process itself was done, as this always appeared to work flawlessly (if the finger was detected to be in an action region for the minimum amount of consecutive frames, the action was always activated as expected).

5 Discussion

5.1 Backgrounds, initializations, and lighting

The system works quite well in simpler backgrounds, such as solid, non-flesh-colored backgrounds (a black chair or white wall), or a somewhat-crowded desk background (varying colors, objects, and lighting in the background, but few flesh-colored pixels in the background). The system also works decently in the case of a moving camera against a cluttered backyard background, though results vary depending on the specifics of the initialization and the items in the background. Unfortunately, the system fails miserably when the background has similar hues to the user's flesh hue.

In simpler backgrounds that do not contain flesh-hue-like pixels, we can more easily segment the hand out from the background (as discussed earlier, we do not put any constraints on the background and instead assume it might be constantly changing, so segmentation does not come as easy as background subtraction). As long as initialization is decent and does not include much of the background, we can form a reliable reference histogram that places high probabilities on hues that are quite different from the background; if subsequent frames also contain hand/finger pixels that have different hues than the background, then the task becomes fairly easy and our tracking works effectively.

In the case of changing backgrounds (such as the user walking around, or objects in the background scene moving), the task is a bit more difficult, especially because flesh-hue-like pixels that pop up in the background can degrade clean segmentation of the hand and fingers from the background. In addition, changes in lighting have large effects on our segmentation and tracking, as relying on hues alone makes us vulnerable to unreliable hues obtained from high or low saturation or brightness pixels that were not masked out by our HSV thresholds.

When the background contains significant amounts of flesh-hue-like pixels, the system fails because it cannot distinguish the foreground hand/fingers from the background objects/scene. This is a major problem that comes with relying strictly on hue - if the hues are not differentiable between foreground and background, we have little hope of recovering. This was especially apparent in the case of the wooden floor background, which had similar hue to the author's hand; the system simply believes that most or all of the image is the user's hand. While some constraints were placed on minimum and maximum convex hull size for identifying the hand region in the image (see Section 3.2.2), modifying these constraints to be a little harsher (such as restricting convex hulls to not cover either the entire width or height of the image) might help a bit. However, even if we do better and excluding such convex hulls, we would still need to have a way to distinguish the hand from the surrounding background image pixels with similar hue. To this end, we did put some work into augmenting back-projection probabilities with 0-valued contours created from edges detected in the new frame's RGB colorspace, but this did not improve performance (see Figure 12).

If instead we had depth information via depth sensors im-

Video ID	Condition summary	Errors	Test frames	Error (%)	Success (%)	5 fingers detected
1	Black chair, inside	12	550	2.2%	97.8%	Yes
2	White wall, inside	9	550	1.6%	98.4%	Yes
3	Crowded desk, inside	6	450	1.3%	98.7%	Yes
4	Cluttered wood floor, inside	485	550	97.0%	3.0%	No
5	Cluttered backyard, outside	150	550	27.3%	72.7%	Shaky

Table 1: Table of results. Note that numbers are approximate.

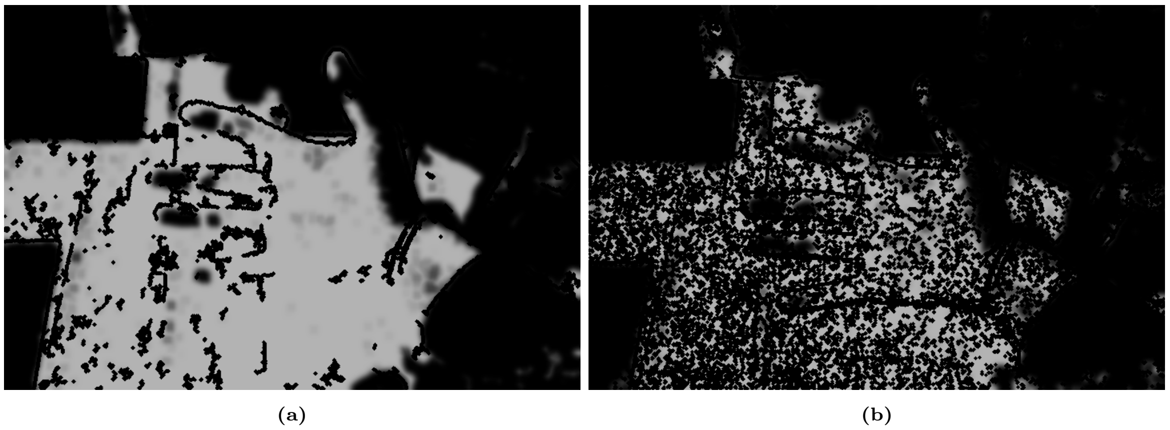


Figure 12: Example of attempts to augment back-projection probabilities for a given image with edge contours detected from RGB space (grayscale was also tried, to no avail). We used edges detected from RGB colorspace for a given image to draw 0-valued contours on top of the back-projection probabilities, which would ideally prevent convex hulls from being created through the edges of the hand. However, we found that either (1) gaps existed between edge contours, precluding forming perfectly-segmented convex hull regions, as in (a); or (2), too many edges were detected as in (b; though a rather extreme example), causing only very small convex hulls to be formed, usually along only a small portion of the hand.

plemented on mobile platforms, we could then easily segment a foreground user hand/fingers from the farther-away background, which would almost certainly drastically improve the robustness of the system and allow for this to be used in a variety of environments. However, depth sensors are not currently widely available on commercial mobile devices like smartphones and tablets, so this is not a viable option at the time, despite the potential it possesses.

In addition, we also observed that sometimes the HSV range checks we applied in Section 3.2.1 masked out some HSV pixels that actually belonged to the hand/fingers, usually when specific lighting conditions (such as shadows or unusual light cast from nearby monitors, etc.) caused variations away from normal-lit hues. However, the used ranges seemed to be the best choise for overall use, and the problematic cases for these HSV ranges were relatively few and far between.

Lastly, we should note that initialization quality has an extremely large effect on system performance. As can be seen in the outdoor background video (video ID 5 in Section 4), various initializations are tried in the beginning and end of the video that result in very poor performance, while an initialization in the middle of the video (and used for test results for this condition) led to much better hand/finger segmentation/tracking. Instead of using a rectangular box to capture the center of the calibration image, we might in the future attempt to use a hand-and-finger-shaped region shown to the user so that the user could show exactly the pose they would mainly be using to select panel items (usually a closed fist with only the index finger extended).

5.2 Number of outstretched fingers

We also did not yet implement any actions depending on the number of outstretched fingers, which could be a potential extension. For example, different icons and actions could be shown based on the number of fingers detected at the time, or different numbers of fingers could be shown to indicate user intent for more general actions, such as turning on/off the system tracking, swiping between panels, or switching between icon sets.

5.3 CAMSHIFT

We also experimented with using CAMSHIFT to detect a CAMSHIFT window around the user's hand and fingers, as can be seen in Figure 13 and in an earlier prototype of the system at `https://www.youtube.com/watch?v=WpzBGLEidbk`. However, the CAMSHIFT window often only covered the main region of the hand, leaving the extended finger (which is exactly what we wanted to track in the first place) outside of the box. Still, CAMSHIFT provides a good estimation of the main high-probability-density region where the hand might lie, so our system could be further extended to include information from CAMSHIFT (such as the location of the CAMSHIFT window center and size) to compute the likely location of the finger (such as requiring that the finger belong to the same convex hull that eclipses with the CAMSHIFT window).

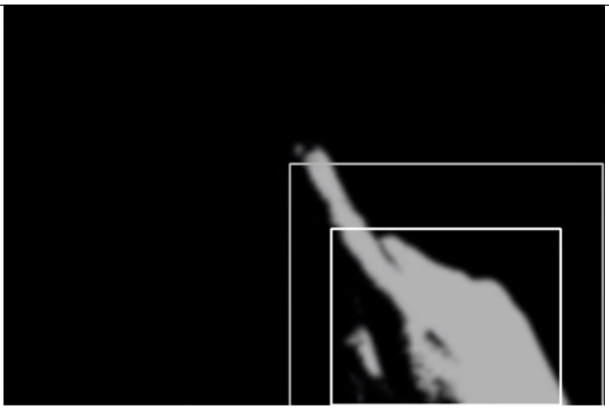


Figure 13: Example of CAMSHIFT applied to a new image. Note that the CAMSHIFT window (the inner box) only surrounds the main hand area, and excludes the finger area. The outer box represents the next CAMSHIFT search window.

5.4 Panel interface

We first note that the panel interface does a decent job of giving the user basic information in real-time. Users can fairly quickly select action items in the panel (limited primarily by the threhsold number of consecutive frames to select a panel item), and the information is concise and translucent enough that this does not get in the way of future selection of panel items or of events occuring in the background of the image, lending to real-time heads-up display use.

However, the panel interface is currently quite minimal and could certainly be vastly improved and extended. In particular, actions such as checking email and social media notifications could be fully implemented, and more information could be shown in the information-based actions, such as the location/weather action and the system information action. In addition, panel selection areas could be moved around, as selecting the region at the bottom on the side of the user's hand is quite awkward, as the user must struggle to keep the hand and fingers in view (and, under the hood, above the minimal convex hull area) while still putting an extended finger into the panel region. The bottom regions could instead be accessed at the sides of the image/screen, which would provide for better user experience.

5.5 Runtime

The system was able to function in real-time at 10FPS, and did not seem too slow for prototype performance. However, frame rate would need to be increased before any widespread use. This would likely involve minimizing use of heavier functions, such as generous Gaussian blurring, and might include setting more fast-fail constraints on search spaces, such as limiting the number of contours searched for convex hulls, or minimizing the number of contours generated in the first place through better hyperparameter tweaking. More analysis could also be done using tools such as NVIDIA's PerfHUD ES.

5.6 Future platforms

Implementing such a system on platforms such as Microsoft HoloLens or Google Glass would be ideal, as these systems have heads-up displays that could fully benefit from such an AR interface. Users would potentially be able to call up the system when wanted, and the system would recognize finger position and gestures to interact with the device without ever having to use hardware input on the device itself. This is optimal as it would allow users to have a fully AR-like experience in real-time, and would potentially have real utility.

5.7 Conclusion

Overall, we present a system that performs real-time tracking of the user's hand and fingers from a mobile camera and that allows the user to interact with an augmented reality panel interface. The entire system can be run on a single NVIDIA Shield tablet (or device with similar specifications), and thus allows interaction with the underlying Android operating system via panel selections and actions. This work provides a utility-focused AR application that could potentially be used in everyday life to glean quick information from a mobile device (and ideally one with a heads-up display) with minimal intrusion into current activities. This would create a novel, intuitive, and efficient user experience that would augment, instead of distract from, real life.

References

[1] Yahoo weather api. `https://developer.yahoo.com/weather/`.

[2] Integrating paper and digital information on enhanced-desk: A method for realtime finger tracking on an augmented desk system. *ACM Trans. Comput.-Hum. Interact.*, 8(4):307–322, December 2001.

[3] G.R. Bradski. Computer vision face tracking for use in a perceptual user interface. *Interface*, 2(2), 1998.

[4] Wolfgang Hurst and Casper van Wezel. Gesture-based interaction via finger tracking for mobile augmented reality. *Multimedia Tools and Applications*, 62(1):233–258, 2013.

[5] Taehee Lee and Tobias Hollerer. Handy ar: Markerless inspection of augmented reality objects using fingertip tracking. In *Proceedings of the 2007 11th IEEE International Symposium on Wearable Computers*, ISWC '07, pages 1–8, Washington, DC, USA, 2007. IEEE Computer Society.

[6] Robert Wang, Sylvain Paris, and Jovan Popović. 6d hands: Markerless hand-tracking for computer aided design. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 549–558, New York, NY, USA, 2011. ACM.