CombineTap: Combine custom marker patterns in single marker to perform series of actions in one tap

Ruoteng Ma
University of Waterloo
Waterloo, Canada
r48ma@uwaterloo.ca

ABSTRACT

Most of the current methods of interacting with computer systems utilize keyboard and mouse control. With the increasing availability of cameras, both built-in and USB connected, many studies focus on allowing users to interact with virtually projected or physically printed buttons using cameras and image processing techniques. This allows users to interact naturally with computer systems by pressing buttons outside the confines of keyboards. But users still need to use complicated shortcut combinations that involve the tapping of multiple keys or buttons. In this paper, I present CombineTap, a system that allows users to create custom, hand-drawn markers to be used as buttons. It utilizes a basic USB web camera and applies image processing techniques to detect finger-tap on the buttons. The main contribution of the system is its ability to create shortcuts buttons by combining predefined marker patterns into a single marker. The accompanying video can be accessed at https://youtu.be/Zvt_TU3kKcs

ACM Classification Keywords

H.5.2. Information Interfaces and Presentation; User Interfaces - Input devices and strategies

Author Keywords

Human Computer Interaction; Computer Vision; hand-drawn element detection.

INTRODUCTION

Computer users use the keyboard-and-mouse combination to perform almost all interaction tasks with their computer. In order to help users navigate complicated menus, applications employ keyboard shortcuts to provide users with quick access to desired functionality. However, as the amount of available shortcuts grow, keyboard combinations also become increasingly complicated. Users need to not only be able to perform interaction in their custom space outside the confine of keyboards, they also need to be able to quickly and comfortably perform complicated keystroke combinations that can be easily forgotten. Current systems solve the first problem, but not the second problem.

Past research has been done on different methods that can be used to facilitate Human-Computer interaction, and many among them involved the use of virtual and physical buttons that users can tap, hold, or drag. Different systems were designed to provide these interactions. Bonfire [12] projects its interface on tangible surface around laptops and detects interactions using a pair of web cameras and existing accelerometers on a laptop. OmniTouch [11] projects its interface with a shoulder mounted projector and detects interactions using a depth camera. RetroDepth [13] detects the above actions and pressure using IR cameras and reflective surfaces. Canesta Keyboard [19] projects a virtual keyboard to replace the physical keyboard. While other systems can also create buttons for users to interact with, I have so far not seen any system that allows users to combine custom button patterns to create shortcut buttons that would execute a combination of actions or a keystroke combination when pressed.

Thus, I present CombineTap, a system that allows users to perform complicated tasks with single taps, and provides a simple way to create buttons that allow these tasks to be performed. CombineTap uses a RGB video camera to detect user interactions on custom designed, printed or hand-drawn markers that act like buttons when placed in the view of the camera. Users can define custom patterns for their buttons, and combine these patterns in certain ways that a single marker can perform the combined actions of all the patterns it contains. This allows users to quickly create shortcut buttons for any combination of actions. Once a pattern is recognized and associated with an action, it can be reused in new markers without the need of reintroducing the association. Figure 1 shows example markers and how they can be combined. With CombineTap, a LaTex user using TeXstudio can combine the marker patterns for Ctrl, Shift and Up buttons into a single button that performs the keystroke combination of 'Ctrl + Shift + Up'. When the user taps this custom button during editing, TeXstudio receives the keystroke combination and goes to the next LaTex error automatically.

The main contribution of CombineTap is that it allows users to combine custom created markers to perform complicated controls and keystroke combinations by interacting with hand-drawn or printed buttons. CombineTap also offers methods to

CHI'16, May 07–12, 2016, San Jose, CA, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 123-4567-24-567/08/06. . . $15.00

DOI: http://dx.doi.org/10.475/123_4

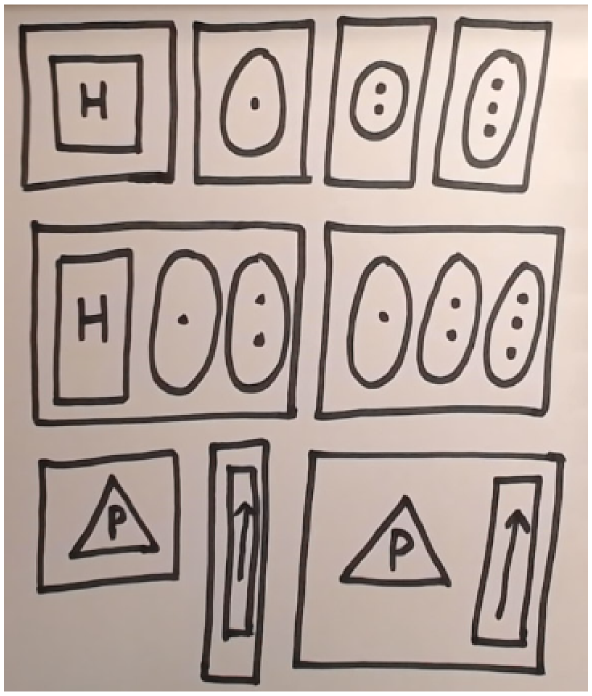


Figure 1. Markers in the first row of the figure, as examples, represent H, Left Shift, Left Ctrl and Delete respectively. The two markers in the second row show how markers can be combined to allow users to perform different shortcuts. The first and second markers in the third row represent the actions pause and turn up volume, the third marker should allow the users to toggle the pause and turn up the volume in one action, however, the feature described by row three is only partially implemented

easily produce new buttons from existing patterns with only marker pen and paper.

RELATED WORK

There are many approaches to design tangible user interface. OmniTouch [11] and Bonfire [12] project visual user interfaces onto real world objects and allow users to interact with them. VistaKey [23] and Canesta Keyboard [19] use projected virtual keyboards to replace physical keyboards.

CombineTap draws inspiration from many works in this area. CombineTap allows users to interact with a computer system by touching physical buttons. Compared with other forms of interactions stated above, touch control benefits from its similarity to the keyboard-and-mouse control system. It is natural to switch from typing on a keyboard to touching physical buttons to execute commands.

Button Design

CombineTap lets users design custom buttons following specific rules, these can be seen as fiducial markers. Fiducial markers are widely used in augmented reality systems, they are placed in the physical environment and detected using image processing techniques. Patterns of the markers are recognized and translated into information that can be used to facilitate human computer interaction. Many works have been done on the design of fiducial markers, most markers have either a circular or a square shape. Circular fiducial markers, such as [15] and [5], utilize the implementation of Contrasting Concentric Circle (CCC), which facilitates the detection of the fiducial markers since all circles must have the same centroid position. Square is a more commonly used marker shape, the lines and corners of the square can assist in determining the relative position between the camera and the marker. Square markers are used in systems such as [18], [1], [8], [14], [6] and [7].

Since CombineTap allows hand-drawn markers, it is unrealistic to expect its users to draw concentric circles accurately or always have colored marker pen available. The only goal of the CombineTap markers is to be used as buttons, this means it is not used in determining the position of the camera or projecting virtual objects following the orientation of the markers. As a result, CombineTap uses rectangular markers and determines the validity of the markers following topological rules that will be described in a later section. In summary, instead of following popular fiducial designs which use square shaped markers or the circular marker, CombineTap uses its own rule for its marker design.

Many markers generate IDs or dictionaries according to the binary pattern of the cells they carry ([18], [8], [14]); some markers use the appearance of sectors on the data rings to generate IDs ([15]). The marker design presented by Neto et al. [7] uses the HSV values of the marker cells to generate a large number of IDs. Furthermore, Garrido-Jurado et al. [9] presented a study that automatically generates reliable markers while Garrido-Jurado et al. [10] also explored the automatic generation of dictionaries. However, the Visual Marker system designed by Costanza and Huang [6] and ARToolKit [1] are more relevant to the CombineTap system since the markers create by the users of CombineTap are bound to individual user and are only used as buttons. Costanza and Huang[6] uses the topology of the marker to generate IDs, this allows the users to create custom and meaningful markers on their own. The identification of CombineTap markers will be discussed in a later section. In summary, many markers generate IDs using their binary patterns ([18], [8], [14]), sector information ([15]), and HSV values in their patterns ([7]). Studies were also done on the generation of IDs and dictionaries ([9] and [10]). CombineTap found its inspiration from systems that allow users to create meaningful markers like [1] and [6].

Fingertip and Touch Detection

CombineTap allows users to touch the button use their fingertip, users perform a finger tap action by resting their finger inside a button for a short period of time. The system needs to detect user's fingertips and decide whether the detected fingertips performed a tapping action against any detected buttons. Systems such as OmniTouch [11], DIRECT [26], ShadowHands [25], and the research presented by Wilson [24] use depth cameras such as Kinect and Leap Motion to detect user's fingertips. This allows the system to detect whether the user's fingertips have touched the surface of the interface. While depth cameras have become more accessible, they are still uncommon among regular users. Various systems also detect hand motions and fingertips by employing Electric Field

Tomography (Electrick [27]), Magnetic Sensing (Finexus [4]), Active Sonar (FingerIO [16]), and Infrared cameras or sensors (RetroDepth [13], Canesta Keyboard [19] and DIRECT [26]). These approaches run into the same problem faced by the depth camera approach, they all require addition equipment that users are unlikely to have. Cai et al. [3] proposed a system that tracks the fingertips with a video camera and uses the shadow of the fingertips to determine if the fingertips have touched the surface. Song and Cheng [22], and Niikura et al. [17] also presented similar systems. This approach is usually implemented in combination with projectors or IR camera and IR lights. In summary, while many systems allow accurate touch detection ([11], [26], [25], [24], [27], [4], [16], [13], [19]), they require additional hardware that is uncommon to the users. Some systems detect touch using the shadow of the fingertips ([3], [22], [17]) but require certain lighting condition. CombineTap will use basic camera and image processing techniques to define and detect touch events.

Character Recognition and Marker Combination

CombineTap supports the recognition of simple characters. Libraries such as Tesseract [21] and Theano [2] are open-sourced and widely available.

Finally, I have so far not seen any research that interprets a combination of marker patterns and maps the interpretation to actions in the operating systems. I believe this is the main contribution of my work since it takes a different approach to the design of the interface then previous research.

IMPLEMENTATION

Besides a computer with Windows operating system, the only additional hardware used by CombineTap is a USB web camera. CombineTap is implemented in Python using OpenCV.

Basic Implementation

In this section, I describe an basic implementation of CombineTap.

The system of CombineTap is divided into two parts: a recorder that allows users to define custom markers and bind them to different keystrokes; a detector that runs in the background to detect buttons and finger-tap events, and executes the actions associated with the tapped buttons. All videos are captured in 1280x960.

There are some basic rules on how should the buttons be created. All buttons should have either a square or a rectangular outer border; all buttons that are recognized using optical character recognition(OCR) should have a rectangular inner border and is assumed to be either a character key or a number key; all buttons that are recognized by topological features should have a circular inner border and can be any possible keys. These rules can be observed from figure 1.

CombineTap uses shape and topological information obtained from contours to detect valid markers, it uses Contour Approximation to detect shapes and uses Contour hierarchy to detect topological information. It uses the OCR library Tesseract [21] to identify characters in marker patterns. It generates IDs for markers by combining recognized characters with shape and topological information.

The recorder asks the user to perform a keystroke and asks the user to display only the desired button in the camera's field of view and start the detection process using a key press. If the button is valid, recognized elements or ID from the button will be displayed on screen. The user can choose to create the association or resupply a new button. If the user choose to create the association, the keystroke that the user performed will be mapped to the detected ID. Once a button is created, the user can reuse its inner pattern to create custom combination buttons.

Once a button is loaded into the system, the CombineTap's background program can identify the marker patterns and interpret the actions associated with them. It tracks the user's fingertips to detect when a tap event occurs. If a fingertip entered and remained in the contour of a button for a certain amount of time, it would be counted as a tap. The system then executes the series of keystrokes associated with that button. CombineTap detects fingertips using skin detection and contour convexity.

Once the basic implementation of the basic system was completed, the following feature was attempted.

A marker pattern could be associated with not only a keystroke but also system actions. For example, a user can create a pattern that un-pause his or her music player running in the background, the user can then create another pattern that turn-up the system volume. These two actions can be done independently with single keystrokes and the user can even combine the two patterns to un-pause the music and turn up the volume in one action. While not implemented in current version of CombineTap, this example can be seen from figure 1. While there are Windows automation tools that allow users to create scripts that perform these tasks, I have not discovered works on how to record those actions automatically and translate them into scripts or allow them to be replayed in anyway. Also, different software are required to perform different system actions on Windows. With these limitations, the current system is implemented with manually coded python Windows UI automation scripts that rely on the pictures of GUI elements.

Recorder Interaction and Implementation

In this section, I describe the interaction process and the implementation of CombineTap's recorder component.

Upon the program execution, user is asked to enter the keyboard button to be mapped, this process is repeated until the user is satisfied with the selection. The program asks the user to present the marker that will be associated with the keyboard button under the camera. The camera is then opened, and the user needs to press the "Space" key once the marker is in place to retrieve the ID of the marker. This process can be repeated until the user is satisfied with the result of the detection, which is displayed in a newly opened window where the detected marker is highlighted with a green outline and identified with its ID. See figure 2 for an example. In this step, it is assumed that only one marker is supplied at a time and the marker is not a combined marker.

User presses "ESC" to confirm the detection and close the camera capture. With both pressed key and detected marker

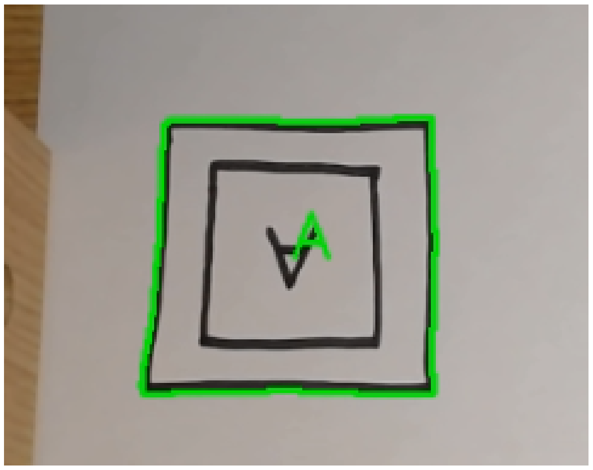
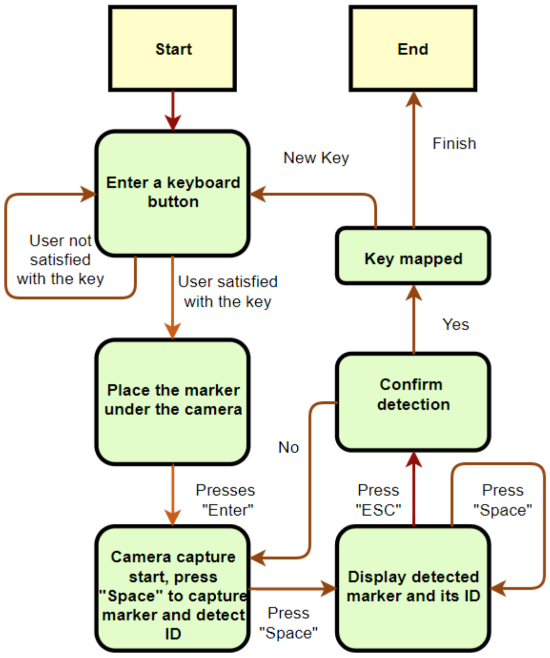


Figure 2. Recorder detects a marker with ID "A" and displays the detected marker in a new window by highlighting the border of the marker and identifying the marker with its ID

ID, the recorder asks the user to either repeat the process or confirm the association. If the user confirms the association, the ID-key pair will be stored in a local text file for future use. The user then has the option to either build a new key-to-marker mapping or to exit the program. The above describes the interaction flow of the CombineTap's recorder program, see Figure 3 for a flow chart of this interaction.



Start
End
Enter a keyboard button
New Key
Finish
User not satisfied with the key
Key mapped
User satisfied with the key
Yes
Place the marker under the camera
Confirm detection
No
Presses "Enter"
Press "ESC"
Press "Space"
Camera capture start, press "Space" to capture marker and detect ID
Display detected marker and its ID
Press "Space"

Figure 3. Flow chart describes the interaction process of the CombineTap's Recorder component

The keyboard input is obtained using the **keyboard** package, and mapped key-ID pairs are written to a text file called **keyToID.txt**. Each line in the text file represents a pair, and the key and ID for each pair are separated by a comma. The detailed implementation of the marker detection algorithm will be described in later section.

Detector Interaction and Implementation

In this section, I describe the interaction process and the implementation of CombineTap's detector component.

The detector contains two major components, a fingertip detector and a marker detector, the two components are linked together by a tap detection component, and another component parses the ID of tapped marker and executes the parsed command.

The work flow of the detector is simple, once the program is started, it shows the video captured from the camera in real time and highlights all detected markers and their IDs in the frame. The user can press the "Space" key to confirm the detection at any time, presumably when all markers are correctly detected. This stops the marker detector component and saves all detected markers and their locations for touch detection until the "Space" key is pressed again. With the locations of detected markers saved, the detector starts its fingertip detection component. The fingertip detection component detects the user's fingertip with the assumption that there is only one hand in the view of the camera and only one finger is extended on that hand. The tap detection component is also activated and is actively comparing the location of the detected fingertip with the locations of all saved markers. If the fingertip is within the confine of a marker for more than a certain number of consecutive frames, a button tap event is considered to have happened on this marker. Once a tap event occurs, the ID of the button tapped is parsed according to the requirements of the **keyboard** package and is executed using the package. It is worth noticing that the current CombineTap system uses predefined IDs for system actions other than key presses and these actions are invoked using hard-coded scripts without any interpretation. The user can at any time press "Space" again to stop the hand detector and restart the marker detection process, he or she can also press "ESC" at any time to exit the detector. Figure 4 visualizes the work flow of the detector component.

The detector first reads the stored key-ID pairs from **keyToID.txt** and stores the pairs in a dictionary for later use. The detector then runs the marker detector, which will be described in later section, to display all detected markers and their ID per frame. Markers are detected every 10 frames and are displayed until next detection. Once satisfied with the detected markers, the user can press "Space" to save the border contours and IDs of all detected markers into two corresponding lists. At this stage, the marker detector no longer detects markers and the fingertip detector becomes active. The algorithm for the fingertip detector will be described in the next section. The fingertip detector returns a single point in the frame, this point is checked against all saved marker contours using **cv2.pointPolygonTest** and the ID of the marker the fingertip point resides in is stored. The ID is checked every frame and a tap event is considered to have occurred if the same ID is selected for more then 15 consecutive frames. The number of consecutive frames required to detect a tap event can be

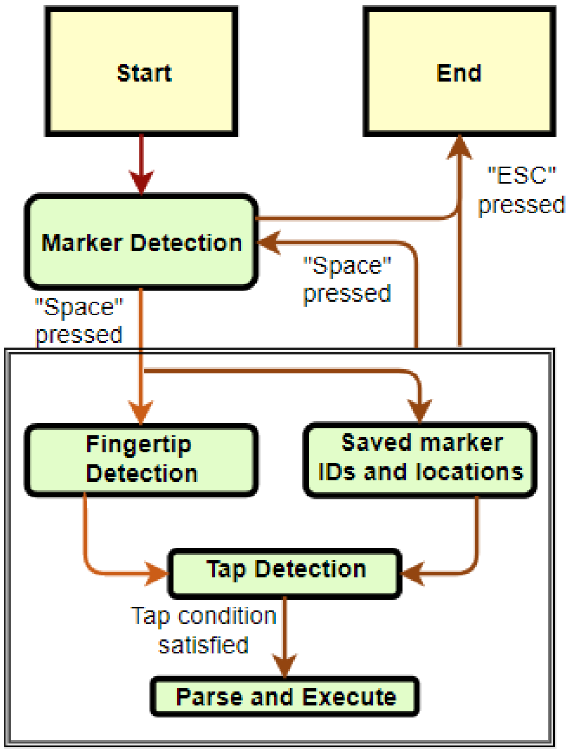


Figure 4. Flow chart describes the work flow of the CombineTap's Detector component

changed to customize the speed of the detection, and the current setting detects a tap event in about 1 second on the device it is tested on. Once a tap event occurs, the ID of the tapped marker is parsed into command in the form of "Control key + Character/Number key", Control keys such as Ctrl are parsed to always be in the front of the command. This is achieved by putting the IDs of topological markers, which are assumed to be any possible keys, in front of IDs of OCR recognized markers, which are assumed to be either character or number keys. The IDs are checked against the dictionary obtained from the key-ID pair file to retrieve corresponding keys. "+" character is then added between all keys to form the command, such as "Ctrl + A". The command is invoked using the **keyboard.send()** method.

For the hard-coded system actions, package **Lackey** is used. Pictures of the UI elements to be clicked are provided and used in sequence to perform the desired actions. While there exist ways in python to automate parts of the Windows Operating System without clicking UI elements, different operations often require different libraries and packages. As a result, they are not used in the current version of CombineTap. In current implementation of CombineTap, ID "n9" is predefined to trigger a mute audio action, this is a proof-of-concept implementation that requires pictures of the UI elements involved in the series of action, these pictures need to be taken from the device it is operating on.

Fingertip Detection

The goal of the fingertip detection algorithm is to provide a single point that represents user's fingertip. CombineTap assumes that only one hand is in the view of the camera and only one finger is extended. It also assumes that the hand is oriented towards the bottom of the frame. CombineTap first performs skin detection by converting the color space of the frames from BGR to HSV and filter the frame using a predetermined HSV range. A HSV ranger finder program [20] is used to determine the appropriate HSV range in which only the user's hand can be detected. Different approaches including HSV histogram and YCrCb color space were used and in current setting the HSV range selection achieved the best result. It is worth noticing that in current setting, different HSV values are needed for different lighting conditions, backgrounds, and different users. The range is currently hard-coded and needs to be changed directly within the python code. **cv2.findContours** is used to detect all contours from the detected regions, the Contour retrieval mode used is **cv2.RETR_TREE** and the Contour approximation method used is **cv2.CHAIN_APPROX_SIMPLE**. The algorithm retrieves the largest contour from all detected contours and applies Contour Approximation using **cv2.approxPolyDP**, the **epsilon** value used in **cv2.approxPolyDP** is calculated as 0.1 times the arc length of the contour. The algorithm finds the bottom-most point in the approximated contour, given the assumptions that only one finger is extended and the hand is oriented towards the bottom of the frame, the bottom-most point is always the tip of the extended finger. This detection process is visualized in figure 5.

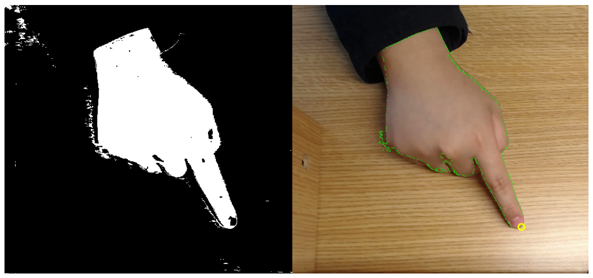


Figure 5. Top left figure shows the detected region, top right figure shows the frame with hand contour highlighted in green and detected fingertip denoted with a yellow circle

MARKER DETECTION

In this section, I describe the implementation of the marker detection component. The goal of the marker detection algorithm is to provide outer contours for all valid markers along with their IDs and it is used in both the Recorder and the Detector.

The detection process starts by applying Gaussian Blur on the frame using a 5x5 kernel, it then uses canny edge detector to retrieve all edges in the frame. **cv2.findContours** is used to detect all contours from the edges, the Contour retrieval mode used is **cv2.RETR_TREE** and the Contour approximation method used is **cv2.CHAIN_APPROX_NONE**. The algorithm loops through all detected contours and checks for

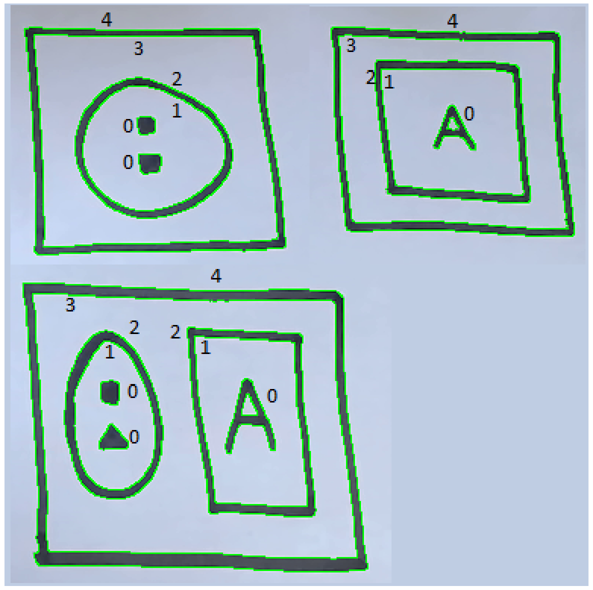


Figure 6. Top left figure shows the contour hierarchy for a topological feature based marker; top right figure shows the contour hierarchy for a character recognition based marker, note that contours within level 0 are ignored; the bottom figure shows the contour hierarchy for a combined marker

rectangular contours. For every rectangular contour, the algorithm checks for their validity, if they are valid, the algorithm also returns their IDs.

To check if a contour is the outer border of a valid marker, the algorithm uses contour hierarchy obtained from **cv2.findContours**. The algorithm traverses the hierarchy by going from parent contour to its child contour while checking the requirements for each level of contour to be valid. The contour levels of all types of valid markers are presented in figure 6. For all types of markers, the algorithm starts from level 4, and finds its level 3 child contour. The level 3 contour is checked for shape and the rectangular shape is the only valid shape. From level 3 contour, the algorithm reaches its first level 2 child. There can be more than one level 2 child and the shape of a level 2 contour can be either rectangular or circular. The algorithm checks the validity for all level 2 children and combine their IDs if they are valid.

For a rectangular level 2 contour, the algorithm checks to make sure it only have one level 1 child. From the level 1 contour, the algorithm retrieves its level 0 child contour and uses it as a region of interest(ROI) mask to retrieve the character to be recognized from the original frame. The ROI image that contains the character is flipped to make sure the character is correctly oriented and is converted to gray scale. Figure 7 shows the ROI image for the bottom marker in Figure 6. The Google Tesseract OCR library is used to recognize the character in the image. **pytesseract**, a python wrapper for Tesseract, is used to achieve this. The method used is **pytesseract.image_to_string**, the language for the detection is English, and the page segmentation mode is set to single character(10). It is worth noticing that the recognition process

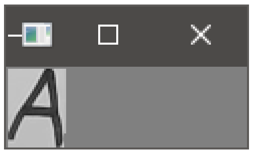


Figure 7. ROI image for the bottom marker in Figure 6

is relatively slow. Once a character is detected, it is returned to be used as the ID for the level 2 contour. If the level 2 contour has no siblings, this ID is also the ID for the entire marker. In figure 6, the top-right marker has the ID “A”.

For a circular level 2 contour, the algorithm again checks to make sure it only have one level 1 child. From this level 1 contour, the algorithm checks for its first level 0 child and checks the number of siblings for that child. The number of level 0 contours is concatenated after character “n” and used as ID for the level 2 contour. If the level 2 contour has no siblings, this ID is also the ID for the entire marker. In figure 6, the top-left marker has the ID “n2”.

If the level 2 contour has siblings, the IDs for all its sibling level 2 contours are concatenated with commas between them. This new ID is the ID for the combined marker. In figure 6, the bottom marker has the ID “n2,A”.

At any steps, if the contour failed to meet the requirement, whether it does not have a child before reaches level 0 or its shape is not what is expected, the marker is considered to be invalid. For a combined maker, if any sub-markers failed to meet the requirements, it is considered to be an invalid marker.

All contours inside a valid marker are stored as used contours, and all detected rectangular level 4 contours are checked against the list of used contours before further detection to make sure they are not already considered as part of a valid marker.

It is also worth noticing that some edges produce two contours, one from the perspective of region outside of the contour while the other from the perspective of region within the contour. To avoid counting them as two distinctive levels, the area of every child contour is checked against the area of its parent contour, if the absolute difference of the two area is within a certain amount of pixels, they are considered to be the same contour. The current minimum area difference is set to be 75.

Once a contour is considered a valid border for a marker, it is appended to the rear of a list of all valid markers, and its ID is stored in another list at the same index position. These two lists are the results of the CombineTap’s marker detection component.

DISCUSSION AND EVALUATION

In this project, I implemented CombineTap, a system that recognizes custom buttons from user-defined marker elements and preforms actions that are associated with the buttons. The system allows users to combine previously defined buttons to form new shortcut buttons with ease and does not rely on printed markers. The ability to reuse user-defined contents

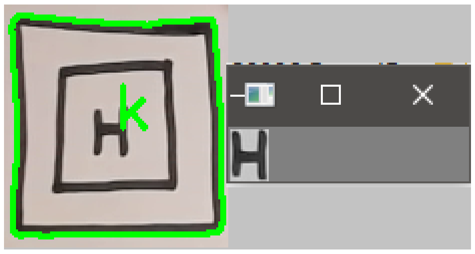


Figure 8. The left image shows the result of the OCR, the right image shows the region of interest used in the recognition. The result of the recognition is wrong although the image provided to the OCR algorithm is clear

should be of interest to designers of similar systems. The ability to allow users to draw their custom markers with marker pen also makes the system mobile and accessible. CombineTap also does not rely on additional devices, it only need a RGB webcam to function.

However, CombineTap also has many limitations. Although it allows users to draw custom buttons following simple rule, the drawing need to be clean and precise, this is especially true in the case of OCR. If the lines inside the marker are drawn too close to each other, CombineTap will likely not be able to separate them. It also requires the lines to be thick, so a thin pen drawing would likely have missing edges. There are also limited space with in the marker, topological feature based marker can not reach high ID number without taking up more space. Also, the OCR library Tesseract is not designed for handwritten character recognition, without extra training, the detection suffers from low accuracy and unstable detection performance. Figure 8 shows an example of wrong recognition. As a result, the detector component asks the users to choose a frame where all markers are correctly detected and use marker locations and IDs detected in that frame for all future frames until the users initiate new detection. Another limitation is the shin detection algorithm, using HSV space require per-condition adjustments. I have attempted to use different color spaces and histogram approach, but none of them had good performance that were stable across environments. I believe the best approach would be to either ask the users to wear gloves or to use an IR camera instead of a RGB camera.

An informal user study was performed with the help of two friends. In the study, I performed a completed recording and detecting process and asked both user, separately, to create their own markers following the rules. One user created three individual markers and one combined marker, while two of the individual markers and the combined marker can be stably recognized, the third marker, while following the rules, could not be stably recognized. The other user recreated the markers I presented to him as examples, CombineTap successfully recognized the markers but had difficult initially in detecting his fingertip due to the HSV value setting. While one user liked the idea of drawing marker by hand, the other user considered this process to be tedious. Both users liked the idea of combining existing marker elements into new markers, but both complained against the adjustment process of the fingertip detection.

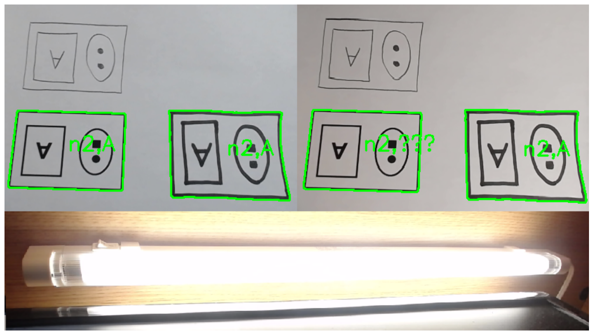


Figure 9. Top-left image shows the detection result with the light turned off, the top-right image shows the detection performance with the light turned on. The bottom image shows the overhead light used in this test. In the lighter condition, the detection of printer character A becomes less stable and is not detected in this the frame presented in the image

Another test was performed with the goal of comparing the performance of printed marker, hand-drawn marker using marker pen and pencil under different lighting conditions. Three markers with the same ID were produced and presented to the detector together, first with an overhead light bulbs slight above the camera turned off then with it turned on. The results of the test can be seen from Figure 9. The pencil drawn marker was never detected under any lighting conditions while the marker pen drawn marker was stably detected under all lighting conditions. The printed marker was detected stably in darker condition, however when the light was turned on, to my surprise, the detection became less stable. This could be the result of the larger character size in the marker-pen drawn marker.

Overall, I am positive about the result of the CombineTap project, the performance of the system should see much improvement given more time. Although some of the issues are resulted from hardware limitations, the idea of combining marker elements can be applied to many different applications.

FUTURE WORK

First of all, ability to record and reply system actions could be implemented. The current CombineTap system only has a proof-of-concept implementation of system automation, I will certainly attempts to fully implement this feature. The fingertip detection could be improved with background subtraction. Some background subtraction techniques were attempted during the implementation of the fingertip detector, they could not achieve good performance as skin pixels can be hard to distinguish from each other. But I believe background subtraction can be successfully implemented in ways that can assist the skin detection. Furthermore, the methods of interaction with created buttons can also include drag and hold. Users could hold down a button to perform a task repeatedly or drag along a volume button to turn up the volume according to the distance dragged. Tap events could also be detected using the shadow of the fingertips, this would offer a more natural interaction when compared with holding fingertip in position, as can be seen from past works([3], [22], and [17]). However, past works used projectors or IR devices to facilitate the detection. I believe that it is possible to achieve this using a flashlight. Finally, an IR camera could be added to assist skin detection, although this requires an additional hardware, an IR camera could be used in both skin detection and touch detection. New devices such as Microsoft's Surface laptops already include IR camera, in the case of Surface laptops, for Windows Hello's face sign-in feature. Other then the implementation of the system, a formal user study should be performed. The users should be asked to produce both hand-drawn markers and printed markers. Assuming skin detection no longer uses hard-coded HSV values, each user should be asked to perform the tap action in different lighting conditions to evaluate the performance of the fingertip detector. Hand-drawn markers should also be tested with different thicknesses and different colors to assess the performance of the marker detection algorithm.

CONCLUSION

In conclusion, CombineTap allows users to create custom buttons on paper and combine them to create shortcut buttons that can be interacted with via fingertip contacts. While there are many limitations, some resulted from the hardware used by the system, CombineTap makes contribution in allowing marker elements to be combined into new markers that combine the meaning of individual marker elements. It offers users a mobile and accessible method to interact with computer system outside the confine of keyboard and allows natural interactions on tangible interface with fingertip tapping.

REFERENCES

1. 2018. open source augmented reality sdk. (2018). `https://artoolkit.org/`
2. James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf.* 1–7.
3. X. Cai, X. Xie, G. Li, W. Song, Y. Zheng, and Z. Wang. 2014. A new method of detecting fingertip touch for the projector-camera HCI system. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. 526–529. DOI: `http://dx.doi.org/10.1109/ISCAS.2014.6865188`
4. Ke-Yu Chen, Shwetak N. Patel, and Sean Keller. 2016. Finexus: Tracking Precise Motions of Multiple Fingertips Using Magnetic Sensing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 1504–1514. DOI: `http://dx.doi.org/10.1145/2858036.2858125`
5. Youngkwan Cho, Jongweon Lee, and Ulrich Neumann. 1998. A Multi-ring Color Fiducial System and an Intensity-invariant Detection Method for Scalable Fiducial-Tracking Augmented Reality. In *In IWAR*. 147–165.
6. Enrico Costanza and Jeffrey Huang. 2009. Designable Visual Markers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 1879–1888. DOI: `http://dx.doi.org/10.1145/1518701.1518990`
7. V. F. d. C. Neto, D. B. d. Mesquita, R. F. Garcia, and M. F. M. Campos. 2010. On the Design and Evaluation of a Precise Scalable Fiducial Marker Framework. In *2010 23rd SIBGRAPI Conference on Graphics, Patterns and Images*. 216–223. DOI: `http://dx.doi.org/10.1109/SIBGRAPI.2010.37`
8. M. Fiala. 2005. ARTag, a fiducial marker system using digital techniques. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, Vol. 2. 590–596 vol. 2. DOI: `http://dx.doi.org/10.1109/CVPR.2005.74`
9. S. Garrido-Jurado, R. MuÃśoz-Salinas, F.J. Madrid-Cuevas, and M.J. MarÃ■n-JimÃľnez. 2014. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition* 47, 6 (2014), 2280 – 2292. DOI:`http://dx.doi.org/https://doi.org/10.1016/j.patcog.2014.01.005`
10. S. Garrido-Jurado, R. MuÃśoz-Salinas, F.J. Madrid-Cuevas, and R. Medina-Carnicer. 2016. Generation of fiducial marker dictionaries using Mixed Integer Linear Programming. *Pattern Recognition* 51 (2016), 481 – 491. DOI:`http://dx.doi.org/https://doi.org/10.1016/j.patcog.2015.09.023`
11. Chris Harrison, Hrvoje Benko, and Andrew D. Wilson. 2011. OmniTouch: Wearable Multitouch Interaction Everywhere. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 441–450. DOI: `http://dx.doi.org/10.1145/2047196.2047255`
12. Shaun K. Kane, Daniel Avrahami, Jacob O. Wobbrock, Beverly Harrison, Adam D. Rea, Matthai Philipose, and Anthony LaMarca. 2009. Bonfire: A Nomadic System for Hybrid Laptop-tabletop Interaction. In *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology (UIST '09)*. ACM, New York, NY, USA, 129–138. DOI: `http://dx.doi.org/10.1145/1622176.1622202`
13. David Kim, Shahram Izadi, Jakub Dostal, Christoph Rhemann, Cem Keskin, Christopher Zach, Jamie Shotton, Timothy Large, Steven Bathiche, Matthias Niessner, D. Alex Butler, Sean Fanello, and Vivek Pradeep. 2014. RetroDepth: 3D Silhouette Sensing for High-precision Input on and Above Physical Surfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 1377–1386. DOI: `http://dx.doi.org/10.1145/2556288.2557336`
14. Rafael Munoz-Salinas. 2012. ARUCO: a minimal library for Augmented Reality applications based on OpenCv. *Universidad de Córdoba* (2012).
15. Leonid Naimark and Eric Foxlin. 2002. Circular Data Matrix Fiducial System and Robust Image Processing for a Wearable Vision-Inertial Self-Tracker. In *Proceedings of the 1st International Symposium on Mixed and*

Augmented Reality (ISMAR '02). IEEE Computer Society, Washington, DC, USA, 27–. http://dl.acm.org/citation.cfm?id=850976.854961

16. Rajalakshmi Nandakumar, Vikram Iyer, Desney Tan, and Shyamnath Gollakota. 2016. FingerIO: Using Active Sonar for Fine-Grained Finger Tracking. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 1515–1525. DOI: http://dx.doi.org/10.1145/2858036.2858580

17. Takehiro Niikura, Takashi Matsubara, and Naoki Mori. 2016. Touch Detection System for Various Surfaces Using Shadow of Finger. In *Proceedings of the 2016 ACM International Conference on Interactive Surfaces and Spaces (ISS '16)*. ACM, New York, NY, USA, 337–342. DOI: http://dx.doi.org/10.1145/2992154.2996777

18. J. Rekimoto. 1998. Matrix: a realtime object identification and registration method for augmented reality. In *Proceedings. 3rd Asia Pacific Computer Human Interaction (Cat. No.98EX110)*. 63–68. DOI: http://dx.doi.org/10.1109/APCHI.1998.704151

19. Helena Roeber, John Bacus, and Carlo Tomasi. 2003. Typing in Thin Air: The Canesta Projection Keyboard - a New Method of Interaction with Electronic Devices. In *CHI '03 Extended Abstracts on Human Factors in Computing Systems (CHI EA '03)*. ACM, New York, NY, USA, 712–713. DOI: http://dx.doi.org/10.1145/765891.765944

20. Adrian Rosebrock. 2016. imutils. https://github.com/jrosebr1/imutils/blob/master/bin/range-detector. (2016).

21. R. Smith. 2007. An Overview of the Tesseract OCR Engine. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, Vol. 2. 629–633. DOI:http://dx.doi.org/10.1109/ICDAR.2007.4376991

22. Chengqun Song and Jun Cheng. 2017. A robust projectorâĂŞcamera interactive display system based on finger touch control by fusing finger and its shadow. *Journal of the Society for Information Display* 25, 9 (2017), 568–576. DOI: http://dx.doi.org/10.1002/jsid.608 SID-04-17-0514.R1.

23. Nadeeka Samanthi Wijewantha and Chanaka Amarasekara. 2004. VistaKey: A Keyboard Without A Keyboard–A Type Of Virtual Keyboard. *Final year project thesis, Informatics Institute of Technology, Wellawatta, Sri Lanka* (2004).

24. Andrew D. Wilson. 2010. Using a Depth Camera As a Touch Sensor. In *ACM International Conference on Interactive Tabletops and Surfaces (ITS '10)*. ACM, New York, NY, USA, 69–72. DOI: http://dx.doi.org/10.1145/1936652.1936665

25. Erroll Wood, Jonathan Taylor, John Fogarty, Andrew Fitzgibbon, and Jamie Shotton. 2016. ShadowHands: High-Fidelity Remote Hand Gesture Visualization Using a Hand Tracker. In *Proceedings of the 2016 ACM International Conference on Interactive Surfaces and Spaces (ISS '16)*. ACM, New York, NY, USA, 77–84. DOI:http://dx.doi.org/10.1145/2992154.2992169

26. Robert Xiao, Scott Hudson, and Chris Harrison. 2016. DIRECT: Making Touch Tracking on Ordinary Surfaces Practical with Hybrid Depth-Infrared Sensing. In *Proceedings of the 2016 ACM International Conference on Interactive Surfaces and Spaces (ISS '16)*. ACM, New York, NY, USA, 85–94. DOI: http://dx.doi.org/10.1145/2992154.2992173

27. Yang Zhang, Gierad Laput, and Chris Harrison. 2017. Electrick: Low-Cost Touch Sensing Using Electric Field Tomography. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 1–14. DOI: http://dx.doi.org/10.1145/3025453.3025842