Hand Recognition: Real-Time Cropper, Filter, and Gesture Identifier

William Santosa*, Arrian Chi†, Allen Liang‡
Computer Science Department
University of California, Los Angeles
Email: *wsantosa@g.ucla.edu, †alienchi@g.ucla.edu, ‡aliang20@g.ucla.edu

***Abstract*—A program that crops hands in real-time by adding a green screen layer around the hand is presented. Some of the techniques utilized are Convex Hull Algorithm, Snakes (active contours), Bit Manipulation, and Hand Landmark Detection. Experimental results show the process of optimizing this application. We conclude by suggesting future improvements to our tool and use cases.**

***Index Terms*—Active Contour, Bit Manipulation, Chromakey, Convex Hull, Cropping, Graphical User Interface, Hand Landmark Detection**

CONTENTS

I. INTRODUCTION

Green screens, also known as chroma keys, were introduced in the 1940s and popularized in the 1980s when computer graphics became more affordable. Blue screens were initially used but were later swapped to green as it is less likely to be present on subjects and easier to key out in post-production. These screens can be used to superimpose videos, create virtual backgrounds, and much more. Our goal is to make a program that draws this green screen in real-time without the need of using a physical screen.

Our application is proposed to solve two main problems in real-time:

1) Cropping hands quickly and efficiently without the use of any other external tool.
2) Cropping hands without having to own a green screen. For example, people could take a video in a crowded library, and the application would crop their hands despite the background containing many different rows of books, tables, and miscellaneous objects.

Additionally, the application provides an intuitive graphical user interface that contains options for controlling the tool. For example, there are options to control the FPS, the save directory for the output images or videos, the image dimensions coming in and out of the program, as well as the display options like showing landmarks or the active contour itself.

We found a lot of inspiration for this project from childhood shows like Oob and The Addam's Family and the video game character Master Hand, and as such, our application will crop out the arm to lean toward having more of a "disembodied hand" aesthetic.

Fig. 1. Cropped Hand.

II. IMPLEMENTATION

A. Directory Structure

We aimed to modularize our code to improve code readability and allow work on portions of the code without worrying too much about merge issues. The breakdown for each file is as follows:

Fig. 2. Master Hand.

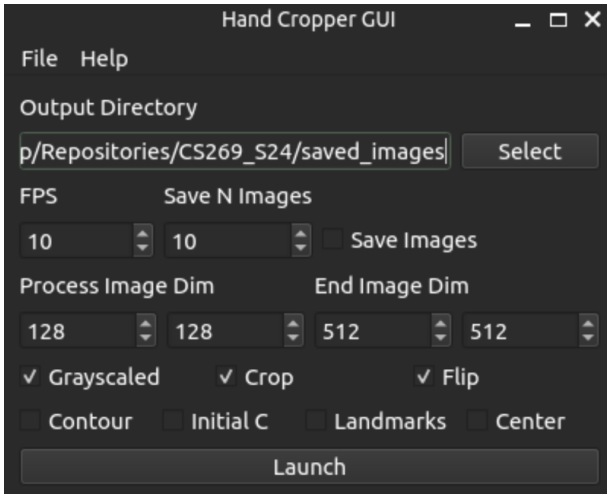

Fig. 3. Graphical User Interface.

1) **main.py:** Contains the code for initializing the GUI and contains definitions for each customizable option, ultimately passing them into the launch camera function when the launch button is pressed.
2) **hand.py:** Where the code for processing the hand and launching the camera is located.
3) **convex_hull.py:** Where the convex hull algorithm is implemented, with some miscellaneous portions in the misc.py file.
4) **paint.py:** Where our functions for manipulating the image is located.
5) **iocustom.py:** Custom input and output file for saving and loading images.
6) **misc.py:** Contains a variety of miscellaneous functions for reusable logic throughout the program.

B. High Level Logic

To accomplish this task, the following steps are taken.

1) Run `main.py` to launch the GUI.
2) Enter options like FPS, process and end image dimensions, grayscale, initial contours, and press the launch button.
3) The app opens and reads input from the camera.
4) Camera image (frame) is resized and converted to both grayscale and BGR. Both are then set back to RGB and set to writable.
5) Hand landmarks are obtained and a convex hull is created for the initial snake.
6) The image is manipulated with the options provided, applying the green screen on the part outside of the snake, and/or displaying the options provided in the GUI.
7) Display the image in our application and save the image if the box is checked.

C. Libraries

We used a variety of libraries within the project. Firstly, we used PyQt6, a set of cross-platform C++ libraries with support for GUI development, as it's supported on many different operating systems and has a variety of modules in case we want to increase the scope of our project.

We also used Google's MediaPipe and scikit-image. MediaPipe is a library of artificial intelligence and machine learning solutions, and we specifically used their hand landmark detection model. Scikit-image is a similar library and is used for image processing and computer vision. The active contour model we used comes from scikit-image.

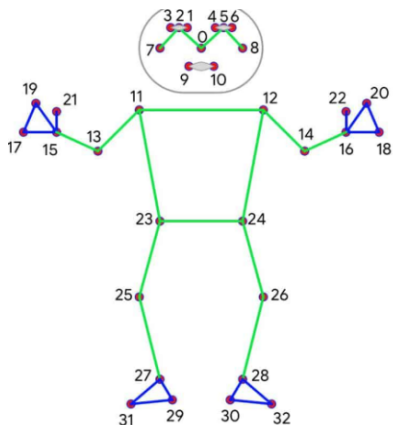

Fig. 4. Google's MediaPipe Body Landmark Detection Model.

We also used the Open Source Computer Vision Library, specifically cv2, which has real-time optimized computer vision tools and hardware.

We also used Numpy and Matplotlib. Numpy is an open-source Python library with efficient numerical operations while Matplotlib is a static visualization library. We use NumPy for computations and Matplotlib for plotting and displaying data like landmarks and contours.

III. Optimization

A. Issues

Initially, processing the image took a very long time. We could not even get 1 frame per second. This caused a delay in the processing of images as the camera captures images at a rate of at least 1 frame per second, which are stored and then sequentially processed by our application, making it so that the processing is not done in real-time. For example, 5 seconds after the application started, we could be processing an image from 3 seconds instead of 5 seconds. Also, the active contours were slow to generate and didn't accurately wrap around the hand. The first few implementations of our paint algorithm also had similar challenges where they were either inaccurate or too slow.

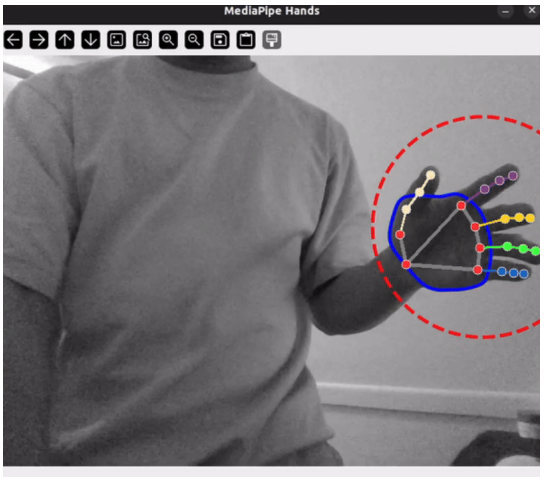

Fig. 5. One of the First Iterations of Program.

Figure 5 is an example of one of the first "working" iterations of our program. The dotted red circle is the location of the initial contour, the circles and lines on the hand are the landmarks, and the blue bounding polygon, which only shows up sometimes, is the active contour snake. The center of the red circle is the center of the landmarks, and the radius of the circle is set to 200 pixels. This setting does not work very well when the hand is either too far or too close to the camera. The program also cannot process multiple hands in an image. Trying to do so would only identify one hand. Also, the snake does not work very well, being seemingly unable to identify the hand properly.

B. Active Contour (Snakes) Parameters

Our first optimization was improving the parameters of the active contour model. MediaPipe allows us to modify their active contour model's alpha, gamma, and beta parameters. The alpha value controls the elasticity of the model, where higher values resist changes in length and favor smoothness. The gamma value controls rigidity and increasing its value results in a less jagged shape. The beta value controls the influence of external forces, which is the pull of areas of interest on the image, with higher values increasing that pull.

Refining the alpha, gamma, and beta parameters was important to the accuracy of the mode, but the most important parameter to improving runtime was the number of max iterations. This parameter determines how many iterations the spline may take per call. The default was set to 500, and we reduced it to 50, which we found to be a good balance between time and accuracy.

C. Image Resolution

Another simple optimization was reducing the resolution of the image. This made the entire contour generation process iterate through significantly less pixels. More specifically, we reduced the resolution from 512 by 512 to 128 by 128, which is approximately 94% less pixels. We could also scale up the contour size to fit the original image during post-processing as the main features of the images were still present, making this optimization have a negligible effect on the shape of the contour. However, as we can see on the before and after images on the screen, there is about a 3 to 4 times speedup after applying this, as the average runtime when running with 10 iterations before and after is 0.25 versus 0.06 on our machine.

IV. Convex Hull

Initially, we provided the active contour model an initial snake which used a circle centered on the mean of the landmarks. This gave subpar results because the circle would match to surrounding areas (expand) or overstep (shrink too much). We hypothesize that this occurs because the distance between each landmark and the closest point on the initial snake varied too much. To fix this, we generated an enlarged convex hull of the landmarks to match to the hand. This approximates the shape of the final contour using the minimum bounding volume of the group of points.

Fig. 6. Circle around mean of landmarks.

To accomplish this in the code, we implemented the monotone-chain algorithm. Essentially the algorithm builds the lower and upper hulls individually by iterating and appending

Fig. 7. (Enlarged) convex hull of landmarks.

the leftmost coordinate points (rightmost for upper) and ensuring the last 3 points added create a counterclockwise turn. If not, points are popped until the condition is satisfied.

We found that the contour model fits better when more points are added to the initial snake. This implied the need to interpolate values between each of the segments of the hull. This final step will then yield our better-performing initial snake.

```python
def convex_hull(points):
    points = [(y, x) for x, y in points]
    points = sorted(set(points))  # Sort lexographically

    if len(points) <= 3:
        return points

    # Build lower hull
    lower = []
    for p in points:
        while len(lower) >= 2 and orientation(lower[-2], lower[-1], p) <= 0:
            lower.pop()
        lower.append(p)

    # Build upper hull
    upper = []
    for p in reversed(points):
        while len(upper) >= 2 and orientation(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)

    ret = lower[:-1] + upper
```

Fig. 8. Find path that corrals landmarks together.

```python
# add points between each segment based on its length
dist = np.linalg.norm((right[0] - left[0], right[1] - left[1]))
print(dist)
pts = 0
for key in dist_pt_map:
    pts = dist_pt_map[key]
    if dist < key:
        break

# use interpolation to add points
x = np.linspace(left[0], right[0], pts)
xp = [left[0], right[0]]
yp = [left[1], right[1]]
y = np.interp(x, xp, yp)
```

Fig. 9. Interpolate each segment based on length.

V. Paint

When figuring out how to implement our flood fill function, we initially broke it down into three steps.

1) Take the camera input and process it.
2) Draw the snake on the hand and save the image to memory.
3) "Paint bucket", like on photo editing programs, on the portion of the image outside of the contour.

Unfortunately, implementing our flood fill function using this approach was not efficient (as we would later realize) and resulted in many problems.

Initially, we used cv2's floodfill function. However, we found that that it inaccurately colored pixels on our images due to the bounding color of the fill being based off the original pixel color, making it stop at boundaries. Since images can have many areas of different colors aside from the contour, the flood fill would stop before everything outside of the contour was filled in. Additionally, it is unfeasible to use the fill function multiple times on each section of the image outside of the contour as we would need to identify each section and call the fill function on each part. Even then, we would not be guaranteed to have a fully green background, as this method does not deterministically fill in pixels.

```python
def flood_fill_custom(image, x, y, fill_color=[255, 0, 0]):
    # initial coordinate
    ret = image.copy()
    queue = deque([(x, y)])

    # perform the flood fill
    while queue:
        x, y = queue.popleft()
        if (
            x < 0
            or y < 0
            or x >= ret.shape[1]
            or y >= ret.shape[0]
            or ret[y, x].tolist() == fill_color
        ):
            continue
        ret[y, x] = np.array(fill_color)
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            queue.append((x + dx, y + dy))

    return ret
```

Fig. 10. Flood fill from pixel until same color.

We decided to create a custom flood fill function that colored the outside of a bounding polygon until it reached pixels with the same color as the fill color. As seen in the code in Figure 10, the image, initial position, and fill color are passed into the function. A queue of coordinates is created, and while the queue is populated, we decide to pop the first element in the list. If the pixel is not out of bounds and not the target color, it is changed to that color, and its neighbors are added to the queue. Otherwise, it would ignore that point and continue to the next element in the queue.

This method's accuracy was good but the runtime was too long due to having to check pixels one at a time. We implemented an optimization by taking bounding boxes of 4,

then 9, pixels in 2 by 2 and 3 by 3 fashion, and randomly checked approximately half of the pixels within that bounding box for the color, and then filled it all in if it matched. This improved the runtime but it was still extremely slow, taking about 1 second to fully fill in a 128 by 128 pixel image.

```python
# Fills the bounded area with the specified color
# color is BGR format
def color_outside_contour(image, contour, color=(255, 0, 0)):
    if len(contour) == 0:
        # Handle case where contour is empty
        return image

    contour = contour[:, ::-1]  # Flip x and y coordinates

    # dilate image
    kernel = np.ones((20, 20), np.uint8)  # Adjust kernel size as needed
    dilated_image = cv2.dilate(image, kernel, iterations=100)

    # Create a mask and perform the color operation as before
    mask = np.zeros_like(dilated_image)
    cv2.fillPoly(
        mask, [contour.astype(int)], (255, 255, 255)
    )  # Fill contour with white color
    # mask = cv2.bitwise_not(mask)
    blue_mask = np.zeros_like(dilated_image)
    blue_mask[:] = color
    ret = cv2.bitwise_and(image, mask)
    ret += cv2.bitwise_and(blue_mask, cv2.bitwise_not(mask))
    return ret
```

Fig. 11. Color Outside Contour.

However, a much more efficient solution existed. This solution only occured to us the week before our presentation. The process goes as follows:

1) Pass in the image and contour into the function.
2) Dilate the image to increase the area of contour.
3) Create a mask by initializing a fully black image with the same size as the dilated image.
4) Contour is drawn and filled onto the mask as white pixels.
5) Another mask is initialized but is fully green.
6) The bitwise AND of the image and the mask is taken, resulting in the original image with black pixels outside of the contour.
7) The bitwise AND of the color mask and the negation of the mask is then taken. Negating the mask turns the filled-in contour to fill everything outside of the contour. The AND of that and the color mask replaces all the white pixels with blue.
8) Add to the previous result to layer the colored portion on top, resulting in the green-screened image.
9) Image is returned.

The processing time on a 128 by 128 image with our new function improved drastically compared to cv2's flood fill function. It originally took a tenth of a second to paint the image. Now it takes 3% of that time to perform the paint operation. Not only that, but the accuracy of the contour improved significantly as well. There is no longer any ambiguity on what should be filled in, as the bounding area is the contour itself.

VI. Results

With all our optimizations, we achieved a large speedup, running at about 5 hundredths to 6 hundredths of a second per frame, permitting our app to be used in real time! It's important to note that the snake and paint times are especially small now, having a similar time to the other steps of our algorithm. The program's runtime is bounded by the process that plots the landmarks, contours, and other options on the image, accounting for about half of the total time it takes to fully process an image from start to finish.

TABLE I
Application Metrics

Metric	Average Time (seconds)
Total	0.055
Convert	0.02
Landmark	4.1e-0.5
Snake	0.01
Plot	0.024

VII. Conclusion And Future Work

Some possible improvements we can add in the future.

1) Gesture recognition. This can be incorporated into our program by adding events when doing certain gestures, such as changing the background color, options, and filters.
2) Video filters like on SnapChat and Instagram to breathe life into the image.
3) Option to add another shading layer to look like there is natural light in the image when adding a green screen.
4) Implement further optimizations. We currently can support between 15 to 20 frames per second on our application on our devices, and improving the FPS using better algorithms would result in smoother videos.
5) Improve parameters to make the snake better wrap around the hand. This could potentially be implemented by utilizing reinforcement learning. We make alpha, gamma, beta, and number of iterations our hyperparameters and the input by the image and landmarks. We run our program with different parameters and our loss criteria will be based on:
    a) How many of the landmarks are still visible after post-processing, and
    b) the program runtime.

References

[1] J. Smith and A. Jones, "An Introduction to Convolutional Neural Networks," arXiv preprint arXiv:1511.08458, 2015.
[2] C. Brown and D. Davis, "On-device Real-time Hand Gesture Recognition," arXiv preprint arXiv:2111.00038, 2021.
[3] E. Johnson and F. White, "An Active Contour Model with Local Variance Force Term and Its Efficient Minimization Solver for Multiphase Image Segmentation," arXiv preprint arXiv:2203.09036, 2022.
[4] Active contour model. (n.d.). In Wikipedia. Retrieved May 9, 2024, from https://en.wikipedia.org/wiki/Active_contour_model