HandMade

The final project for @pybae, @MikkelKim, @tsherlock, and @iancai.
Our team name is HandMade!

Our members:

- Paul Bae
- Mikkel Kim
- Tim Sherlock
- Ian Cai

Pitch

There is a growing market in drawing tools, such as Paper and Sketchpad. These tools don't seek to replace the power of heavy applications, such as PhotoShop, but instead seek to provide a simple interface to allow users to quickly sketch up their ideas.

There have been several burgeoning ideas in this region as well. Rocketboard, which allows users to draw on a whiteboard to share thoughts in real time, FlockDraw, and much more.

Our app seeks to do the same, but in a unique way.

Rather than relying on mice or touch surfaces for input, our app will take input from the user by a camera. By using real-time hand tracking, we can track the motions of the hand to simulate drawing on a virtual whiteboard.

We can come up with a set of gestures to represent each action:

- An index finger pointed to the camera will represent the drawing state.
- A palm moving will cause the current whiteboard view to move.
- A closing fist represents zooming out
- A opening fist represents zooming in

And these gestures are hypothetical as of the moment, and will likely changed based on user input.

Our end goal is to create a desktop application that relies on the laptop's camera as input and provides a virtual whiteboard that users can write on, zoom in, zoom out, move around, and share (if time permits) to other users.

References

A good portion of this project was spent researching various methods and papers regarding hand detection.
The main paper we referred to throughout the project was the following:

- Yeo, Hui-Shyong, Byung-Gook Lee, and Hyotaek Lim. "Hand Tracking and Gesture Recognition System for Human-computer Interaction Using Low-cost Hardware." *Multimed Tools Appl.* (2013): n. pag. Web.

This paper describes a detailed approach to detecting hands in real time on commercially available hardware. This approach is what we mainly followed throughout our project, excepting some caveats.

The current state of the art in hand gesture detection can be judged closely from the above paper, in which robust hand detection for most backgrounds is supported in real time with high accuracy.

We seek to emulate this model, as well as improve and adapt it to our project.

Objectives

Objective 1: Preprocessing

- Compute background
- Background subtraction
- Convert to YCrCb
- Threshold on each channel
- Morphology (erode and dilate) on each channel
- Merge channels
- Bitwise and operator with original frame
- Threshold final image
- Morphology (erode and dilate) final image
- Gaussian Blur
- Result: A binary image of the hand with most background noise

Objective 2: Detect hand motion

- Contour extraction
- Polygon approximation
- Max inscribed circle and radius

- Find region of interest
- Convex hull
- Convexity defects
- K-curvature
- Min enclosing circle
- Result: Contours, convexity defects, and circles of the hand from Objective 1

Objective 3: Gesture Recognition

- Simple heuristics to detect static gesture
- For example:
    - 0-1 fingers detected, then close palm
    - 4-5 fingers detected, then open palm
    - 1 finger detected and no thumb, then pointing
- More research to be done here
- Result: Coordinates of where the gesture is and a flag corresponding to the gesture

Objective 4: OpenGL Whiteboard

- Design a modular system for gesture recognition
- Link up the x, y coordinates to drawing on an OpenGL canvas
- Implement an interface for multiple brushes, colors, and the like
- Result: A GUI showing an OpenGL whiteboard (built on Qt as a backend)

Objective 5: HTML5 Canvas (optional)

- Link up the OpenGL whiteboard in step 4 to HTML5 Canvas
- Make the updating real time
- Optimize the speed
- Launch on a server (most likely heroku)
- Result: Real time mirroring to a server of the image

Final Report

The tools we've used in this project are the following:

- C++
- OpenCV
- Google Test
- Git

Our implementation is as follows:

For our first objective (preprocessing), our implementation is as follows.

First we detect faces inside of the original frame given to us by the VideoCapture. This indeed filters out the face properly, but increases the efficiency by a significant amount. We do this detection by a well trained Haar Cascade Classifier, which OpenCV provides for us. For every face detected, we simply draw a bounding box around those faces and fill that bounding box with black.

Next, we convert the frame to the YCrCb color space. We do this mainly to better filter out for skin detection since the YCrCb color space relies on luminance, which can better reflect skin color.

Once, we've converted the frame to YCrCb (which is a single method call), we then use background subtraction to filter out the new objects in the frame. Of course this requires the background to be computed and in hand.

We compute the background as the first step in this program as a rolling average using addWeighted. The actual background subtraction is done with an absolute difference.

Once we've subtracted the background, we split the result into three channels and do the following operations for each:

1. Thresholding using Otsu's method
2. Binarization
3. Erosion and DIlation

This is mainly filtering out noise from the image itself.

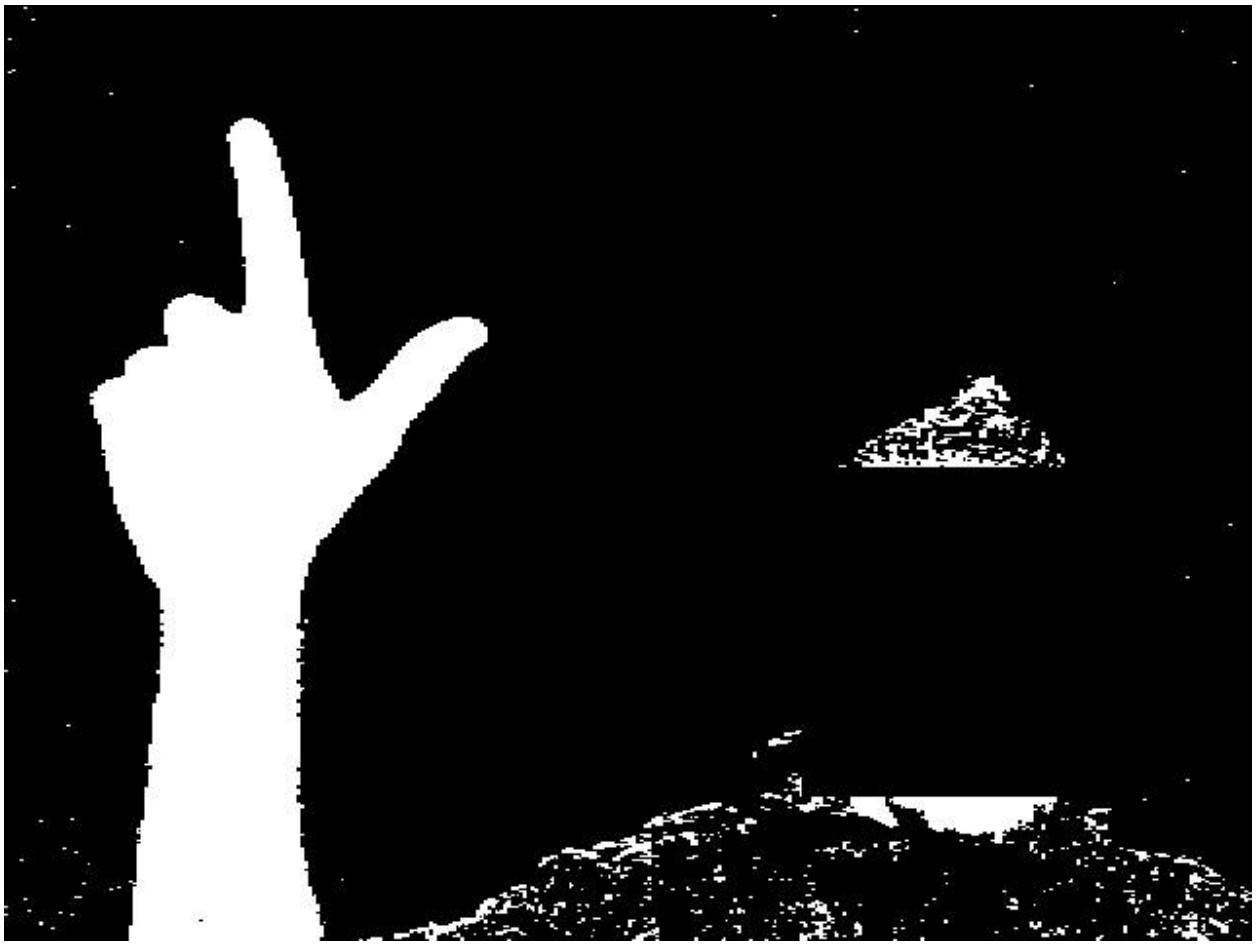
After, we merge the three channels together.

To finish things, off, we remove the regions detected by the face detection earlier here. And finally, we run an erosion and dilation operation once again.

At the very end, we find skin regions in the original image and run a bitwise and on that and the preprocessed image we have so far.

We do this in order to help produce more consistent results. The preprocessing method we followed was very close to that of the paper we mentioned earlier, and this method results in consistent output. Given more time, however, we wish to come up with a preprocessing method that does not rely on the background being computed, and instead that uses optical flow. The main difficulty to do so, however, is to find the correct points to detect and know when to recompute those points (when the hand is out of the frame).

The following is a screenshot from that process:



And here is a video demonstrating it:
https://www.youtube.com/watch?v=EHd1GC5R-2k

Now for our second objective, hand detection, our first step is to find the contours in the image.

This is done quite easily by the findContours method provided by OpenCV.

From these contours, however, we find a polygonal representation of these contours. We do this mainly to speed up computation in the later steps for the convexHull. Another side effect of this, is that we can filter out some contours that are not able to be reasonably formed as a polygon (within some error limit).

And once we have these done, we find the max inscribed circle of the curves. We do this to find where the palm lies in the curve. This is a heuristic and is observed to work for most people. The algorithm to find the max inscribed circle is to sample points with a step limit (set as ten in this example), and to then find the max inscribed circle's radius for each of these points (by the closest point to the polycurve), and we take the maximum of these radii. However, this is quite a computationally intensive process.

After finding this circle, we filter out all points and curves not within 3.5 * radius of this circle. The main reason being, we wish to filter out the arm of the hand as well as any noise that exists.

Next we find the minimum enclosing circle of the contour, which is trivial in OpenCV.

Finally, we detect the convex hull and convexity defects. Given all the defects we try to filter some extraneous ones out, as we only are concerned about those surrounding a finger (for now). This filtering is challenging. Defects between two fingers are easily recognizable by their depth which is approximately the length of the finger. Defects between a finger and another part of the hand, however, are not so easily recognized. This makes it hard to filter out defects when a gesture consists of just one finger. As a result, detecting just one finger is not as accurate as we would like.

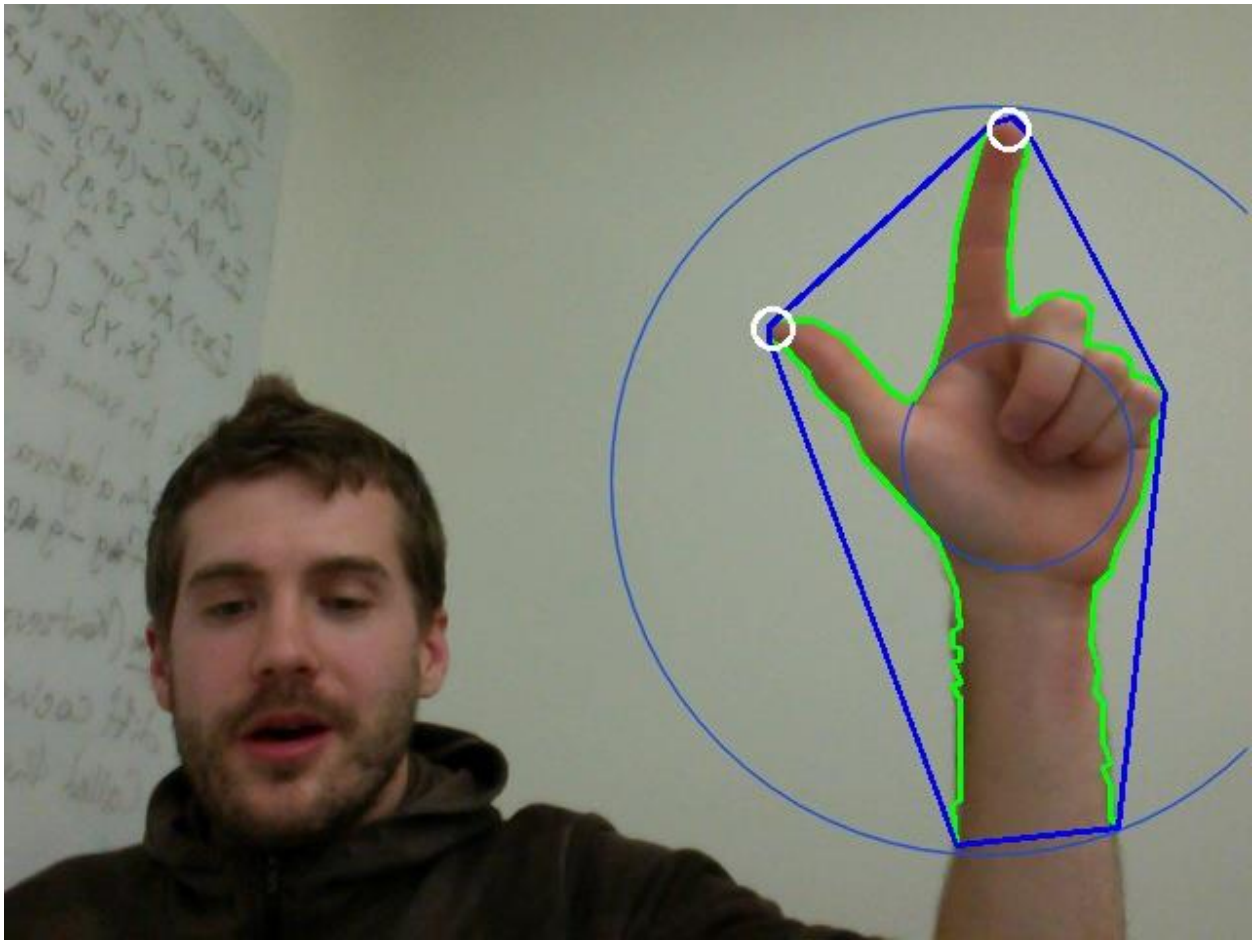
We find the top of the finger tips by comparing the start and end of a convexity defect and if the distance between the two is within some error margin, than this is a match. This algorithm in its current form is fast but not very accurate. It should be changed to find the best match (nearest neighbor) instead of the first match which meets the distance metric.

So in total, we've accomplished the total of our original objective. We have the palm, hand, fingertips, and the entire curve itself.

If given more time, we would likely spend it optimizing the process we've detailed. In particular the maximum inscribed circle is a time consuming process and we find that most of our bottleneck in the code is in this section. In terms of accuracy, however, we still can improve

detecting finger tips, as we do not account for the possibility of multiple hands or oddly oriented hands as of now.

A snapshot of the code running is as follows:



And a video:
https://www.youtube.com/watch?v=_TzWp8fBpM0

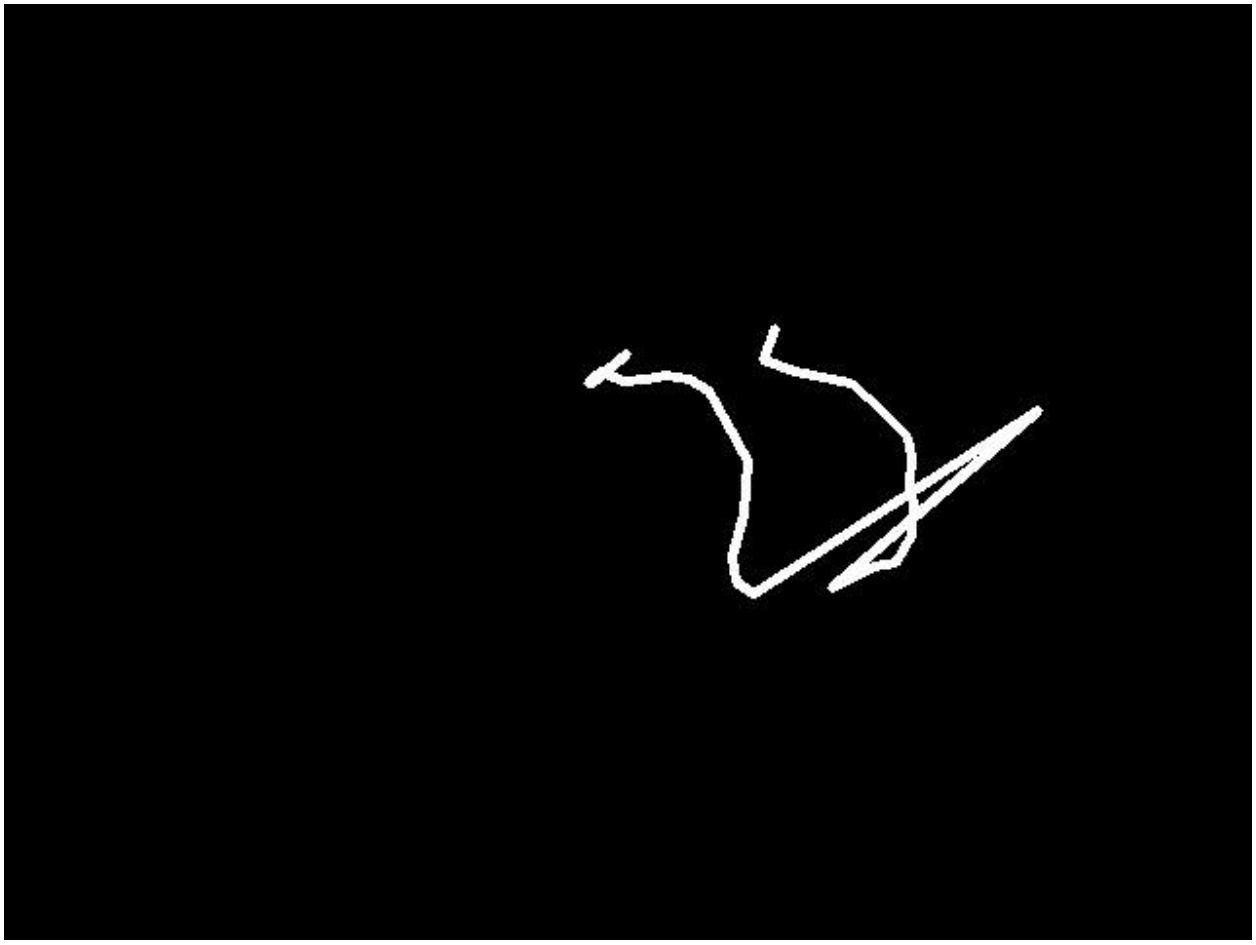
For our final objective, we intended to use OpenGL, but soon found that integrating the two was more of a challenge than we expected (given OpenGL's feedback loops not synchronizing correctly with OpenCV's frame calls).

So, we simply used OpenCV's drawing functionality instead, creating a buffer of points returned per frame, and drawing a line if we deem it to be of some margin.

If we have one finger up, we draw a line from that finger, and if we have five fingers, we can erase in that region.

This is one of the weakest points of our assignment so far, and given more time, I wish to properly integrate the two. OpenCV innately uses a Qt backend, but it is difficult to extend the GUI interface for OpenCV, requiring a user to embed an OpenCV application inside of Qt if wishing for custom interfaces, and we ran out of time while working on this portion.
A snapshot of the drawing procedure is as follows:

The following is what could be drawn with our program:



As you can see, it's quite clunky due to the speed of the program.

And a video:
https://www.youtube.com/watch?v=H2D7M0L82OA

Results

Objective 1:

- **Completed**
- **Can be more robust**
- **Works for solid backgrounds**
- **Does not work for moving backgrounds, unsteady camera, or backgrounds with colors similar to the hand**
- **Does not work for cameras that have auto adjusting features.**

Objective 2:

- **Completed**
- **Can be more efficient**
- **Works for most cases**
- **Does not work for oddly shaped gestures, such as the back of your hand, and a claw**

Objective 3:

- **Partial**
- **The data is available to us to detect more sophisticated hand gestures, but we ran out of time, and chose not to implement these**
- **Only implemented two gestures**

Objective 4:

- **Partial**
- **OpenGL proved to be a challenge to work with effectively, and we resorted to OpenCV functionality**
- **Clunky in movement and drawing**
- **Not as fluid or multi-featured as possible**

Objective 5:

- **Nothing (it was listed as optional)**

Overview

This project had an unforeseen amount of complexity. Yet, we found this as a welcome surprise. Before embarking on the project, we researched a multitude of papers regarding the same problem. Various techniques were presented to us, and we found that choosing which one to use was a formidable challenge.

Among the challenges, we faced during this project, optimization was a recurring issue. Our initial implementation of face removal lead to a significant bottleneck, compelling us to look for alternative solutions. Furthermore, finding the maximum inscribed circle proved to be difficult to do efficiently. More generally, there was a trade off between accuracy and efficiency, and for most cases, we've chosen the former. However, this isn't to imply that our program lacks room to improve in terms of accuracy. For noisy backgrounds, characterized by moving objects or a scene with differing depths, precision quickly deteriorates. There is also a limitation in terms of user experience. Requiring the user to compute the background is a tiresome step.

As the project matures, we hope to address the issues presented above. In particular, finding a more robust algorithm to draw points onto the whiteboard is of notable concern. We also have much to desire in future prospects. Integrating the application with a web interface, as mentioned earlier in our Objectives, is still to be done and supporting cross platform compatibility are all potential goals.

What we've accomplished so far is noteworthy, and without doubt, we are all proud. Yet, we realize that there is still much to do. We will continue to progress and hope that HandMade is fruitful in the coming future.

Sincerely,
Team HandMade