Coin Detection in OpenCV

By Ian Dudder and Karran Singh

Purpose

The goal for this project is to create an OpenCV program capable of finding and identifying U.S. coins in an image. We limited our search to the four most common coin types: pennies, nickels, dimes, and quarters. We also decided to limit the coins to standard face and back (e.g. eagle-backed quarters, Lincoln Memorial-backed pennies, etc). To accomplish our goal, we divided the task into the following five main objectives, listed in order of increasing complexity:

1. Recognize elliptical objects in an image.
2. Identify which elliptical objects are coins, and which are not.
3. Determine if the coin is heads-up, or tails-up.
4. Identify which type of coin was found (i.e. penny, nickel, dime, or quarter).
5. Report on the total value of the coins in the image.

Initial Methodology

To complete objective 1, we reasoned that for any given image, coins will always appear as either perfect circles (if the image is taken from a top-down view) or as ellipses (if the image is taken from an angle) with the major axis always along the horizontal. Our initial plan to achieve this objective was to use Canny's edge detection to find key objects in the image and then use Hough's Circle-finding method to locate the circular objects.

To complete objective 2, we planned to count the number of edges inside each circle and compare that to the average amount of edges in a coin. If the number of edges in the circle is close to the average number of edges in a coin, we say it is a coin.

Our plan to achieve objective 3 was essentially the same as objective 2. We reasoned that the tails-side of a coin has more defining edges than a heads-up coin, so we planned to use the number of edges in the coin to determine if it is heads-up or tails-up.

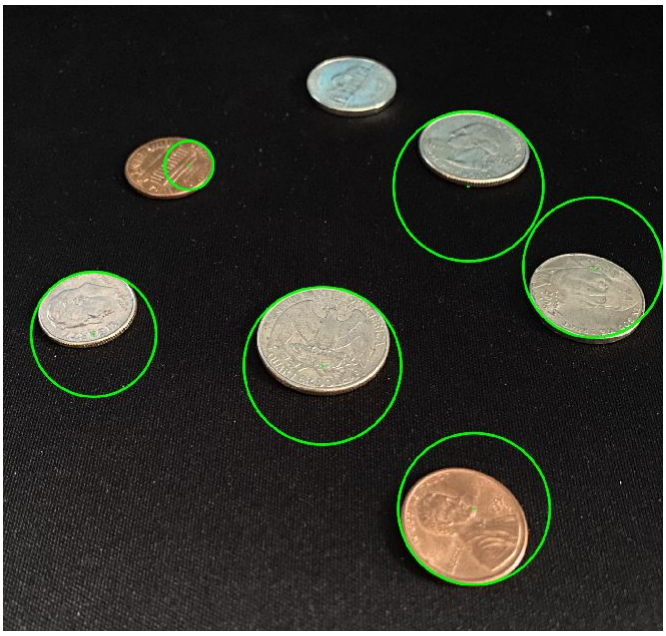
Our plan to complete objective 4 was to compute a color histogram for the pixels inside the ellipse and compute the most common color. This would help us determine what type of coin it is.

Lastly, objective 5 would be very easy considering all the other objectives were completed well. After counting up the number of coins we found, and how many of each type, we can report on the total combined monetary value of the coin collection.

Early Obstacles

As we began to work on our project, we encountered obstacles that caused our initial plan to fall through. The first flaw in our plan came from the Hough Circles method performing poorly when the test image was at an angle. Because Hough Circles always finds a perfect circle, if the coins were at an angle at all, the circle would be larger than the coin itself, causing the circle to be polluted with pixels from the background. This issue is illustrated in Figure 1. It was also challenging to compute a bounding rectangle around the circles to extract a sub-image from.

Figure 1: Hough Circles Inaccuracies



The next obstacle we encountered was that sheer count of edges was not enough to determine if a circle is a coin or not, nor was it able to determine if a coin is heads-up or tails-up.

The third obstacle came from our color comparison. Even after computing the color histograms and finding the most common colors for each detected circle, we found that color varied from image-to-image too heavily to find acceptable values to help identify a coin. Our settings were either too relaxed, resulting in a large number of false positives, or too strict, resulting in too many false negatives.

Accomplishments

While these early obstacles did prevent progress for a while, they were eventually overcome. After doing some research on OpenCV methods, we discovered some alternative approaches we could take that were more effective in producing accurate results.

Finding Ellipses with Contours

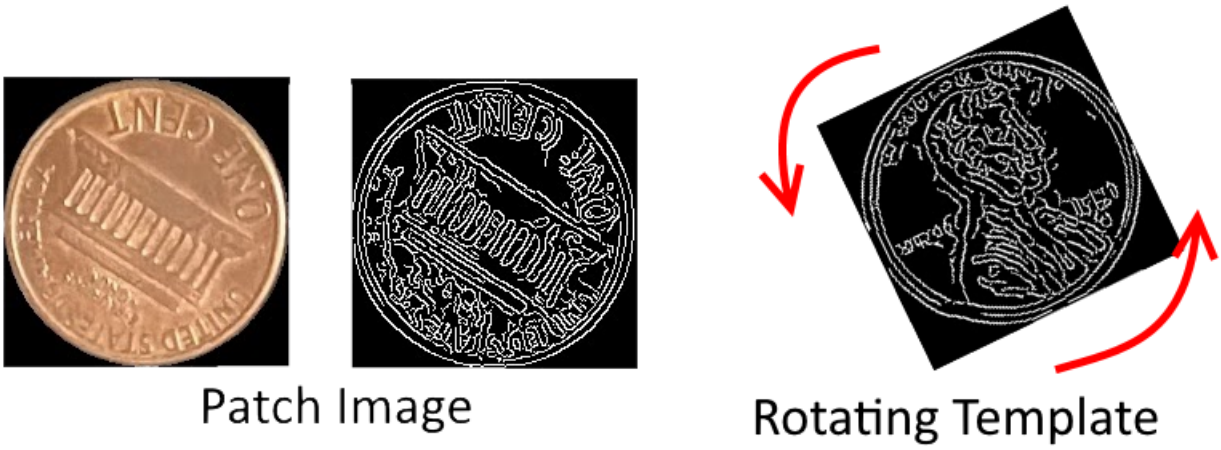
One alternative we found to Hough Circles that worked much better for finding elliptical objects rather than just circles was the “findContours” method. This method, when given a binary image, can identify objects of any shape in an image and return an array of the contours it finds. OpenCV also has a set of methods built in for contours, such as computing a best-fit ellipse or rectangle around the shape. Using the best-fit ellipse, we were able to eliminate objects that were not ellipses based on how much area they occupy in the best-fit ellipse. Elliptical objects occupied 99% of the area, while non-elliptical objects occupied less, allowing us to eliminate them outright. Additionally, the best-fit rectangle allowed us to find the exact pixels from which to extract a sub-image from, allowing us to perform more sophisticated analysis to the elliptical objects.

One challenge we encountered was that finding contours required a continuous edge in order to be considered a contour. This means that any object with a break in its enclosing edges would be missed. In certain images, we found that the Canny-edge image had too many broken edges, resulting in a low rate of detecting ellipses. This was resolved using the dilate function to build up the broken edges enough to reconnect them into one continuous shape, allowing them to be detected as a contour.

Template Matching

Our next major change in methodology was to use template images to compare the contours to, helping complete objectives 2-4. We used templates for both the heads-up and tails-up side of each coin, resulting in 8 images total. For each contour, we used the bounding-rectangle to cut out a patch image of just the desired contour. After computing the edges of both the patch image and the template image, we compared the patch image to every rotation of the template image, keeping track of the number of edges that match. Using the best-matching-rotation for each template, we can determine which coin the patch image is most similar to. Figure 2 helps illustrate this process.

Figure 2: Template Matching Process



Initially, we only considered the total number of matching edges to determine what type of coin each contour was. However, this approach caused contours to only match with a tails-up quarter, since that template has the highest number of edges. This meant our template matching was only useful for determining whether or not a contour is a coin (objective 2), but could not determine if it was heads or tails, or which type of coin it was (objectives 3 and 4). This was corrected by instead calculating the percentage of edges matched for each template. This way, templates with fewer edges were able to produce matches as well rather than just the templates with lots of edges. With this new approach, our program became significantly better in determining which type of coin each contour is, and whether it is heads or tails.

Determining the Value of Coins in an Image

Our final objective, determining the total value of the coins in an image, was very simple. We simply kept a count of how many of each type of coin was found, and use that number to calculate the monetary value.

Revised Methodology

In the end, our methodology for finding coins follows these steps:

1. Resize the source image to a hard-coded value to ensure that our selected kernels work effectively.

2. Use the Canny method to detect edges, and dilate the image to connect any broken edges.
3. Find the contours in the image.
4. For each contour, find the best-fit ellipse and bounding rectangle. Disregard any contours that occupy less than 99% of the ellipse area.
5. Compare each contour to each template, considering every rotation of the template. Keep track of the percentage of edges that match.
6. Based on the percentage of matching edges, make a guess at what type of coin the patch most resembles.
7. If a patch matches 38% or more of the edges, we say it is a coin. If not, draw the best-fit ellipse and move on.
8. For each conclusive coin, draw a bounding rectangle and annotate the image with the name of the coin we believe we have found. Provide the percentage of matching edges as well.
9. Calculate the value of the coins we have found, and display that across the top of the image.

Results

With our new methodology, we were able to complete our objectives to an acceptable rate of success. We were not able to meet all of our objectives with 100% accuracy, but a margin of error is to be expected for a program without machine learning.

Output Images

Here are some output images that our program produced. Below each image is a report of 3 things:

1. Coins Found: the ratio of coins found to coins not found.
2. Heads/Tails Accuracy: the percentage of correct heads/tails identification for the number of coins found.
3. Coin-Type Accuracy: the percentage of coins correctly identified (Note: heads/tails not taken into account here).

Output Image 1

Coins found: 7/7 (100%)
Heads/Tails accuracy: 7/7 (100%)
Coin-Type Accuracy: 4/7 (57%)

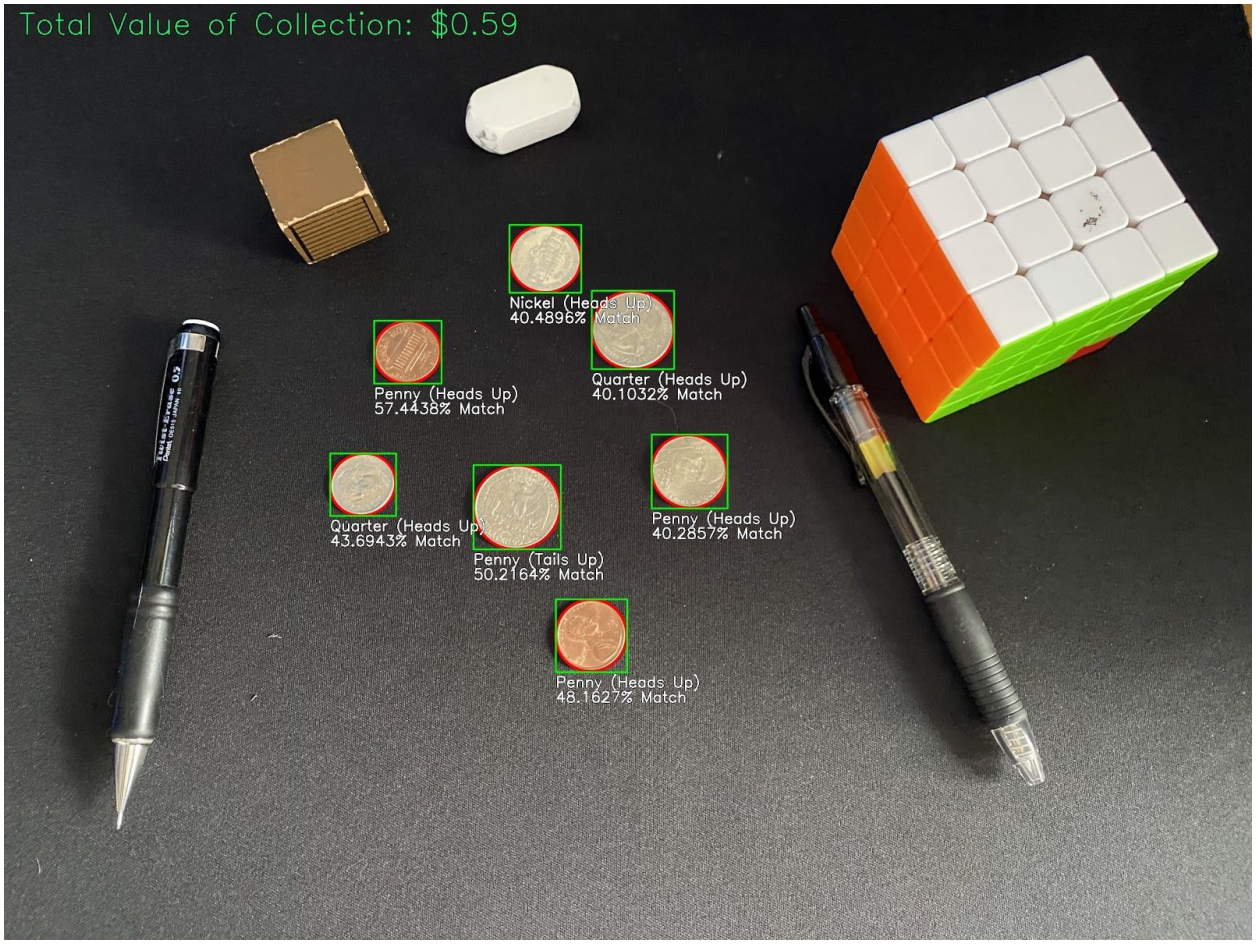
Output Image 2

Coins Found: 7/7 (100%)
Heads/Tails Accuracy: 7/7 (100%)
Coin-Type Accuracy: 7/7 (100%)

Output Image 3

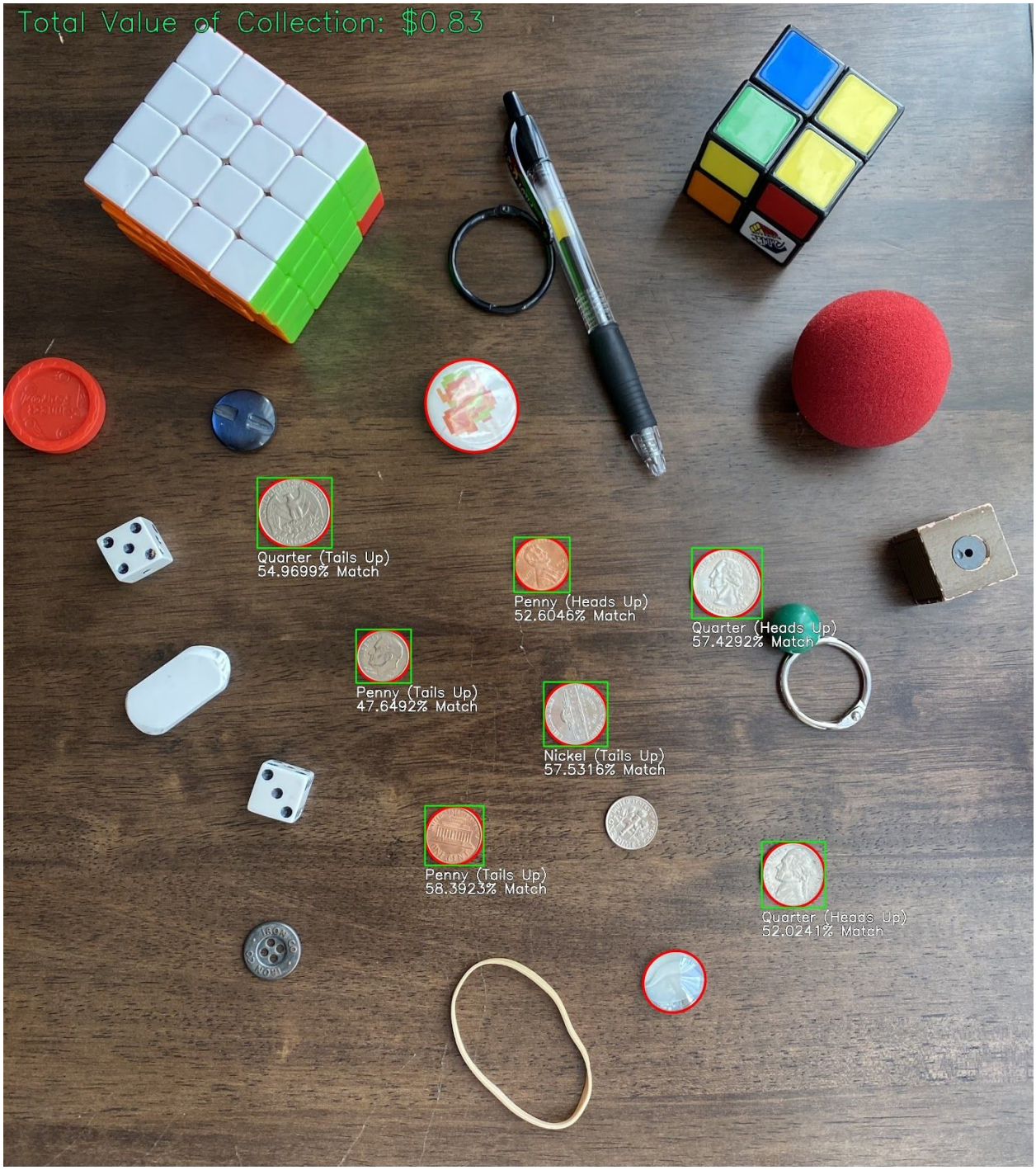
Coins Found: 7/7
Heads/Tails Accuracy: 4/7 (57%)
Coin-Type Accuracy: 3/7 (42%)

Output Image 4



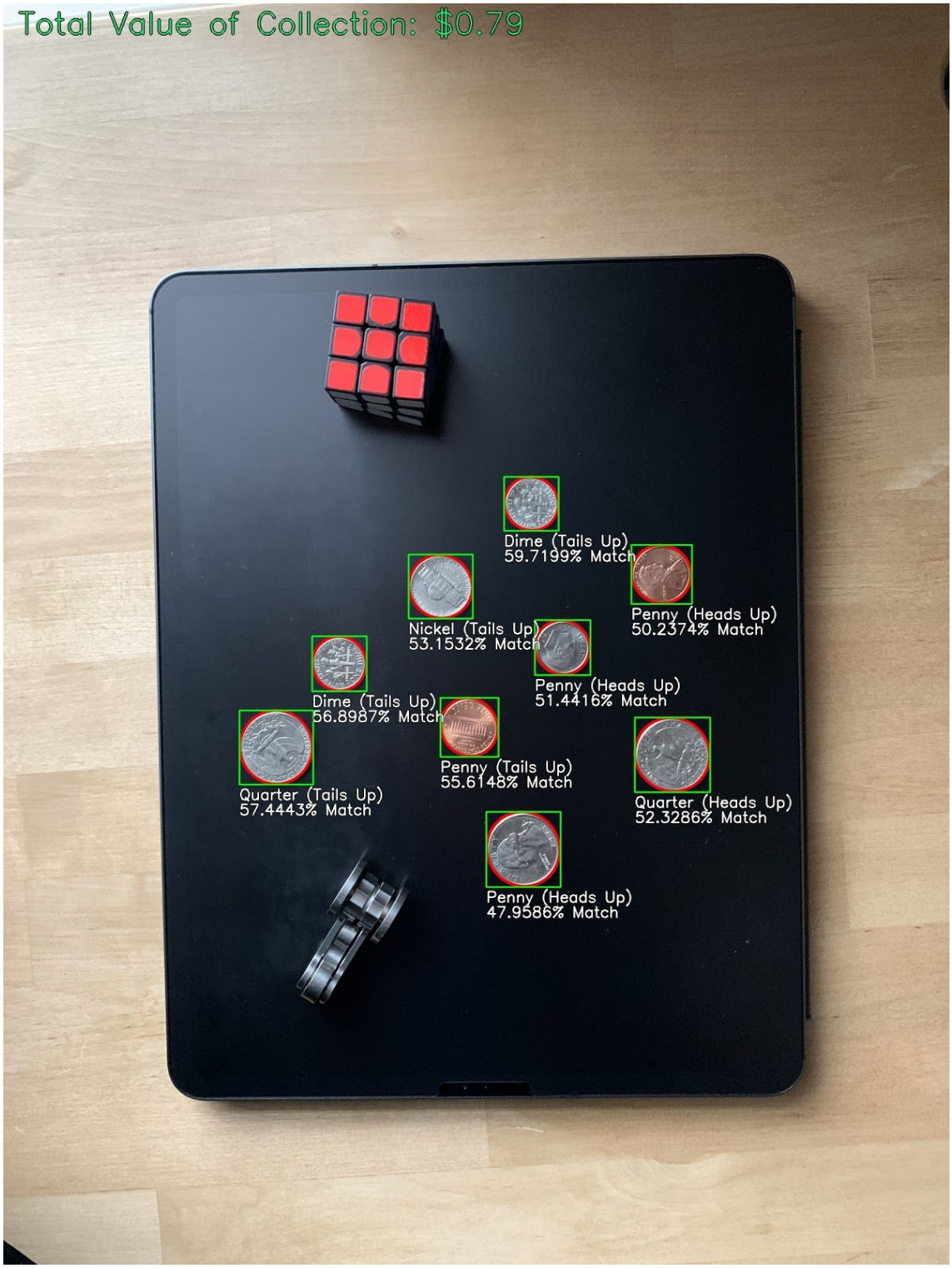
Coins Found: 7/7
Heads/Tails Accuracy: 5/7 (71%)
Coin-Type Accuracy: 4/7 (57%)

Output Image 5



Coins Found: 7/8
Heads/Tails Accuracy: 6/7 (85%)
Coin-Type Accuracy: 5/7 (71%)

Output Image 6

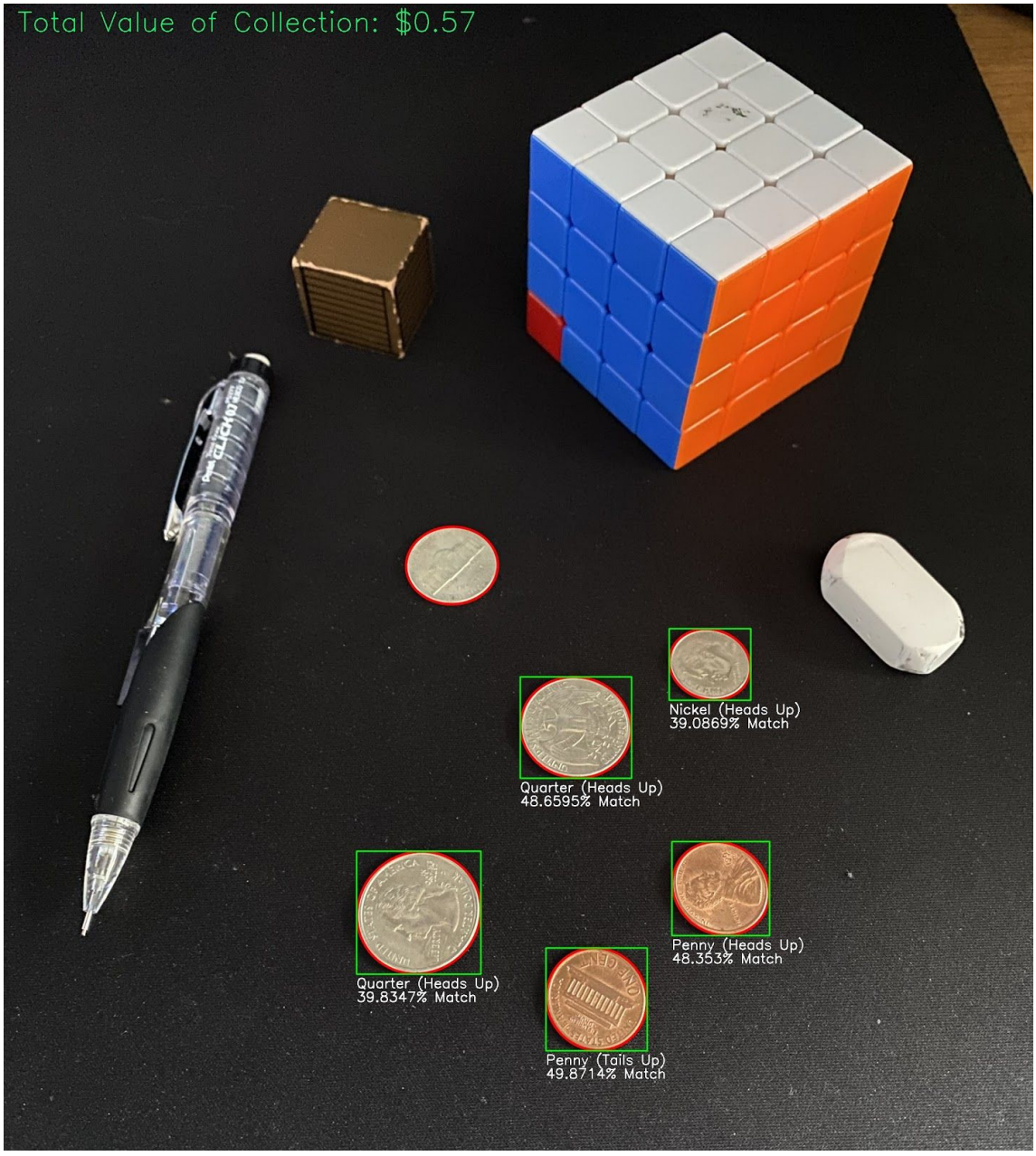


Coins Found: 9/9
Heads/Tails Accuracy: 8/9 (89%)
Coin-Type Accuracy: 7/9 (78%)

Output Image 7

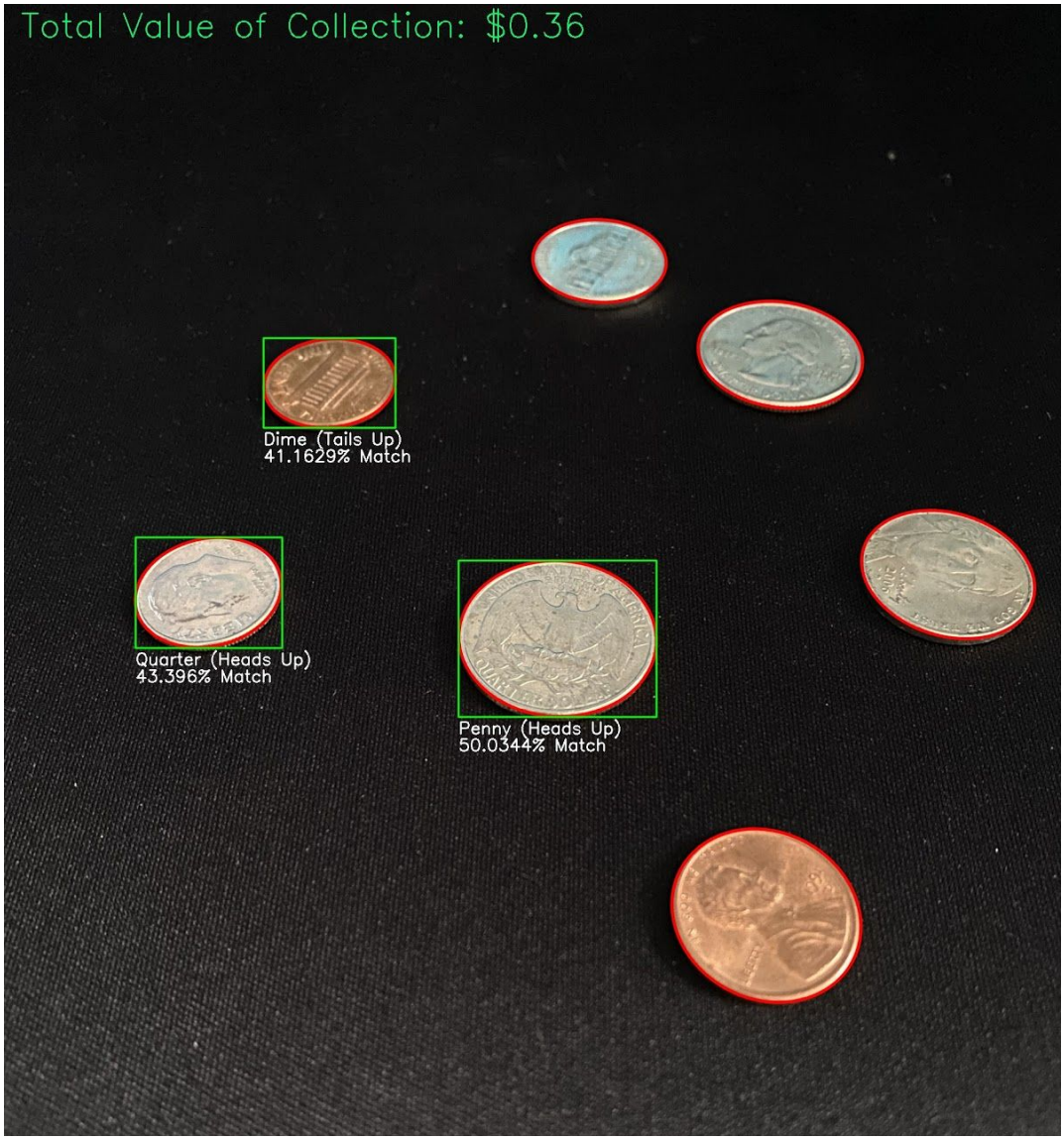
Coins Found: 4/7
Heads/Tails Accuracy: 2/4 (50%)
Coin-Type Accuracy: 2/4 (50%)

Output Image 8



Coins Found: 5/6
Heads/Tails Accuracy: 4/5 (80%)
Coin-Type Accuracy: 4/5 (80%)

Output Image 9



Coins Found: 3/7
Heads/Tails Accuracy: 2/3 (67%)
Coin-Type Accuracy: 0/3 (0%)

Output Image 10

Coins Found: 2/7 (with one false positive)
Heads/Tails Accuracy: 2/2 (100%)
Coin-Type Accuracy: 1/3 (including button) 33%

Summary of Results

Overall, our project has a coin recognition rate of 81%, a heads/tails recognition rate of 81%, and a coin-type recognition rate of 63%. Images taken from a top-down view of the coin yield the best results, as seen most notably in Output Image 2, which has a 100% success rate. Angled images, such as output images 9 and 10, tend to blur the edges on the coins as they get further back in the image, leading to decreased edges to match against.

Other factors that seem to impact coin-recognition are noise and lighting. In output image 5, a dime is missed even by the contours due to the large amount of surrounding noise. This is especially apparent when looking at the image's Canny edge image. Additionally, output image 10 has a yellowish light that seems to drown out the edges on the coins. In fact, not even the contours method was able to find a few of the coins.

The areas we could improve this program is increasing accuracy in angled images by shearing the template image to have the same tilt as the coin after each rotation but before edge matching to allow a better match to be found. Additionally, if there was some way to normalize the colors, we could use the colors to make more educated guesses on what the coins are, and never mix up pennies with any other coins.

Lessons Learned

This project required us to use everything we have learned from this class so far, and required us to combine all the techniques we learned from the past programming assignments. This project has taught us a lot; in fact, there were even many methods that we did not use that taught us a lot about OpenCV such as adaptive thresholding, Hough Circles, and close morphology. Here is a brief list of methods that this project exposed us to and helped us learn:

- Using Hough Circles to identify circular objects.
- Using dilation morphology to build up edges and reconnect broken lines.
- Using findContours to find key shapes in an image.
    - Learning to construct a bounding rectangle from a contour.
    - Learning to construct a best-fit ellipse from a contour.
- Using adaptive thresholding as an alternative method to Canny edge detection to find objects of interest in an image.
- Using close morphology to close edges inwards (pairs nicely with findContours).
- Use templates to find a match in an image.
- Annotating images using the putText method to write text on the image.
- Using filesystem in C++17 to feed our program an entire folder of images.

Additionally, here are some other concepts that we learned from this project:

- Learned that color is not always a reliable variable to consider.
- Learned to consider normalized values such as percentages over values that can vary like total numbers.
- Learned to use threads to simultaneously process images.
- Learned to run trial and error experiments to problem solve in opencv

Notable Topics

Find Contours: This method was the most important one to our program. Without it, we may not have even completed objective 1. After examining the contours drawn on an image, it is clear that finding the contours can reveal far more than just ellipses from an image. It found quite a few other objects in our program that we ignored, but that in another project may have found useful.

Additionally, we were exposed to many other contour-supporting methods such as finding bounding-rectangles, best-fit ellipses, or even a minimum enclosing triangles. Also, this method

appears capable of detecting concavity of an object, something that could be highly useful in understanding 3-dimensional objects in an image.

Adaptive Thresholding and Close Morphology: This was a concept that we explored as an alternative to Canny edge detection, but in the end we were not able to get better results than Canny and dilation. Below is an example that we tested.

Original Image

Image after Adaptive Thresholding

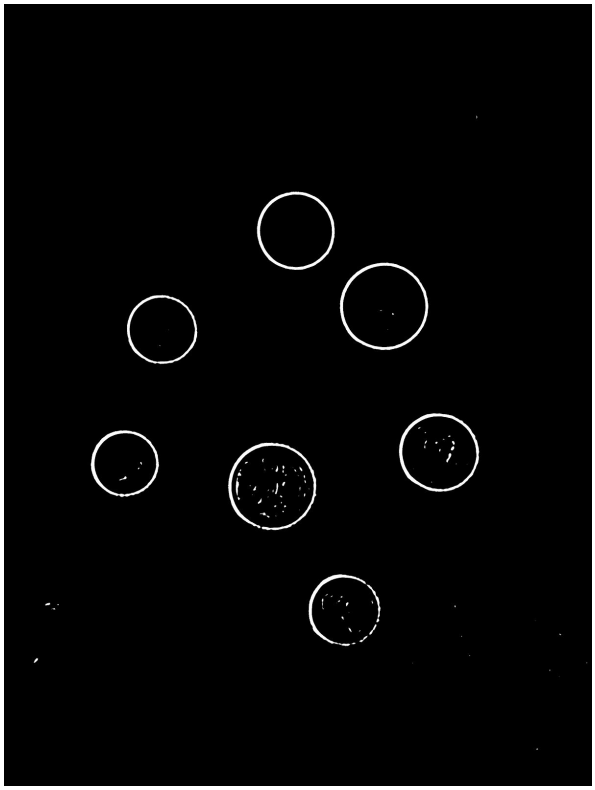
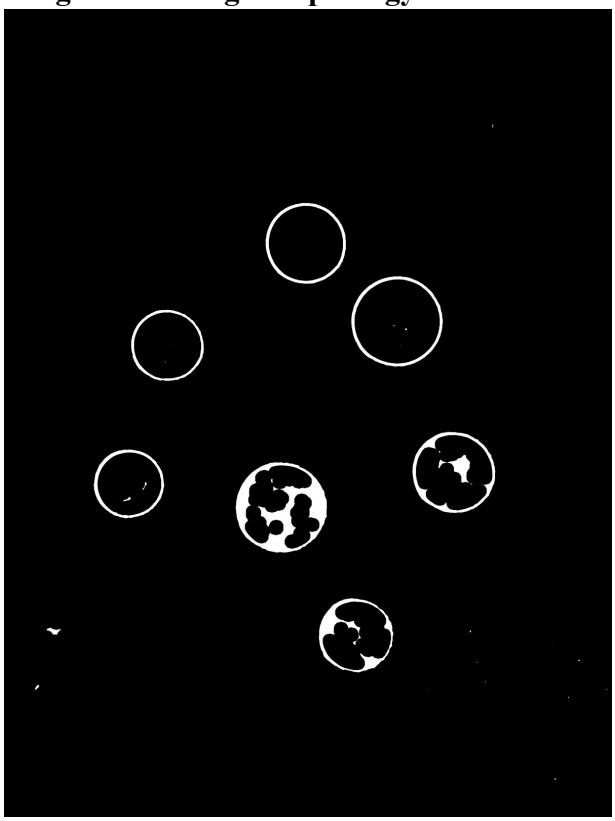


Image after using MorphologyEx with a Closing Kernel



After experimenting with a test image, it is clear that all noise must be eliminated before close morphology is attempted. With fewer iterations of blurring, we noticed that noise would bleed into the objects, corrupting the smooth shapes. We found that it was most useful to smooth with a large kernel multiple times to eliminate noise, and then allow large block sizes in the thresholding.

Perhaps if we had discovered adaptive thresholding and close morphology sooner, we would have been able to fine-tune the results to produce better contours than the Canny image could. However, given the time constraints, we were only able to learn the basics of the methods and appreciate their value for other future projects.

The Unreliable Nature of Color: One significant lesson this project taught us is that color is very unpredictable, and can vary greatly between two different images, even when the images have the same objects. Differences in lighting play a huge role in changing the color values in an image. Initially, we hoped that color would help us rule out objects that are not coins and help distinguish between pennies and the silver-colored coins. However, even when using a histogram of colors to compute the most common color within an object, we could not calibrate appropriate thresholds to rule out objects based on color.

Using Percentages Instead of Total Values: This concept was the final step between just being able to identify coins, and being able to identify which type of coin we had found. When comparing the matches between our patch and template, if we used the total number of matched edges, we would always match to a tails-up quarter because that template has the most edges. However, after altering our method to consider the ratio of edges matched to the total number of edges in the template, we were able to eliminate the unfair advantage that templates with a large number of edges had. This was the key in distinguishing the types of coins.

References

"Contour Features." OpenCV, https://docs.opencv.org/3.4/dd/d49/tutorial_py_contour_features.html.

"Finding Contours in Your Image." OpenCV, https://docs.opencv.org/3.4/df/d0d/tutorial_find_contours.html

"Hough Circle Transform." OpenCV, https://docs.opencv.org/3.4/d4/d70/tutorial_hough_circle.html.

Perone, Christian S. "Simple and Effective Coin Segmentation Using Python and OpenCV." Terra IncognitaI, 22 June 2014, https://blog.christianperone.com/2014/06/simple-and-effective-coin-segmentation-using-python-and-opencv/.