Tiago Lima

Joseph Suarez

CEE101S/201S

Counting coins presents a simultaneously irksome and tedious tax. Image processing paired with a few clever algorithms easily eliminates the problem. Coin Reader implements such a solution, using eight separate sub-methods to achieve the task at hand. These algorithms proceed as follows: take the average background color, iteratively find coin edges, use pixel color averages to ensure that an actual coin has been found, calculate the coin diameters, distinguish dimes from pennies, ensure that coins are not counted multiple times, compare relative diameters to determine coin types and amounts, and take a sum of the latter values. Hence, Coin Reader effectively implements a solution to a real-world irritation. However, as the program is a beta version, it assumes some conditions regarding the image input. Said image must have a black background, contain no touching coins, have no coins near the image’s edge, be a close-up, and contain at least one coin of each type *smaller than the largest coin*. Although most of these assumptions make intuitive sense, the last appears rather cryptic. However, it is placed to eliminate the following difficulty: if the program is fed a picture of a random coin, the coin’s diameter will vary drastically depending on the camera position. Hence, the latter assumption eliminates the problem by allowing comparison of *relative* diameters.

The first algorithm returns the approximate average background color of the input image. It achieves this by using a short piece of vectorized code to sum the color of the pixels along the image’s edge and divide by the number of pixels counted. Assuming that the image’s edges are clear and that the background is approximately uniform (apart from perhaps a shading issue, which is the point of implementing this algorithm as opposed to just using the color of the top left pixel), this will return the average background color for use later during the program.

The second algorithm operates through use of both the average background color and a configurable threshold variable. It iteratively compares the color of each pixel to the average background color. By this method, a coin is determined to be present if and only if the difference between the latter two values is greater than the threshold.

The third algorithm accounts for the second algorithm’s shortcoming, that is, scratches in the image occasionally register as coins. To account for this discrepancy, the third algorithm steps down a few pixels from the point of contact with the “coin” and then iteratively “draws” an isosceles triangle. A quick comparison between the average color of all pixels within the triangle and the background color confirms or denies the presence of a coin.

The forth algorithm proceeds with the top center pixel of a coin as its argument. From the top, the sub-method travels downward and compares each pixel–along with the two below it–to the background color. If any of these three pixels is significantly different from the background color, then it is assumed that the diameter is larger than currently known. With that assumption, the algorithm travels a pixel downwards and repeats its comparison. In a perfect image, comparison of the additional two pixels per iteration would be redundant. However, as images occasionally have small imperfections, adding the extra check often steers the program from error. When a comparison fails at the algorithm’s end, it is assumed that the bottom edge of the coin has been found. The diameter is thereby known and saved in an array.

The fifth algorithm occurs directly at the end of the forth. Its purpose is to distinguish the two most similarly sized coins–pennies and dimes. First, an average RGB color value of a small square of pixels in the coin’s center is taken. Then, considering this obtained RGB value as an array of the form [R, G, B], the blue component is subtracted such that a new array of the form [R-B, G-B, 0] is obtained. The operation is repeated, subtracting the new green value. Hence, a single value is obtained of the form [R-B-(G-B)]. As it transpires, this value is invariably 20-30 for any coin type but the penny, which has a value of at least 32. Hence, penny amounts may be distinguished from the coin array and saved in an array elsewhere for later use.

The sixth algorithm ensures that coins are not counted multiple times. This unwanted calculation would occur invariably if not accounted for because the coin’s lower pixels would be evaluated during a later iteration. Hence, the sixth algorithm uses the coin’s known top and bottom edge locations along with the fact that coins are circular to draw (using a vectorized pixel setting algorithm) a box over the coin using the background color. As the background color will never evaluate as different from itself, the coin will not be recounted.

The seventh algorithm takes an array of diameters (not including penny diameters) as input and produces an array of coin amounts as output. It does this by “binning” the largest array entries together, thereby considering them as one coin type. After removing these coins from the diameter array, the algorithm is executed again. This process repeats until no coins are left in the array. This algorithm is also the cause of the most constraining assumption about the input. While using relative diameter *ratios* in the code’s comparative statements may very well eliminate the “one coin of each type *smaller than the largest coin*” requirement, it would do so shoddily. These ratios would ideally allow the user to have any coin combinations, provided he/she inputs any one of the coin types present; however, these ratios do not vary by enough in practice to accurately distinguish coins. Therefore, in absence of coin face recognition, the constraint proves necessary.

The eighth and last algorithm takes as input an array containing the number of each type of coin and a variable containing the number of pennies present in order to output the total coin value. As the array length will vary depending on which coins are present in the original input image, the algorithm proceeds by multiplying the input array by the last *length* elements of an array of constant coin denominations where *length* is the length of the input array. For example, if four elements are present in the input array (dollar coins, quarters, nickels, and dimes), then the array will be multiplied by [1.00, 0.25, 0.05, 0.10]. By appending .01\*(number of pennies) to this array and then taking a sum of all elements, the total coin value is obtained.

From this discussion of Coin Reader’s eight algorithms, a replica should be easily constructible. However, a few particular obstacles during coding and their solutions are worth mentioning. In particular, configuration variables merit explanation. All thresholds for pixel comparison and tolerance values for other comparative operations in Algorithm 7 should be determined experimentally through evaluation of multiple images. Also, debugging the code without a formatted print system would present a herculean challenge. Defining debug variables equal to either zero (for non-debug mode) or one (for debug mode) proved to be an effective method. Simply including checks in the format if(debugVariableName) allowed for quick and helpful debug navigation. Additionally, a copy of the image should be reserved for graphical debug. In this image, all algorithms that take a sum or average of a group of pixels may be illustrated on the image. For example, the third algorithm (triangle comparison) should include a debug check to draw white triangles on the alternate image in appropriate locations. These debug checks are still present in the final code along with numerous comments explaining each algorithm briefly. Hence, Coin Reader’s otherwise cryptic code becomes quite easily understandable and reproducible.