

**Introducing Computers to Blackjack:
Implementation of a Card Recognition System using Computer Vision Techniques¹**

Geoff Hollinger & Nick Ward

Geoff.Hollinger@gmail.com & nward2@swarthmore.edu

Submitted to IEEE Paper Contest: March 10, 2004

Swarthmore College Student Branch Counselor:

E. Carr Everbach

¹ Paper originally submitted as a final project to Dr. Bruce Maxwell for his Computer Vision class in Fall 2003.

Abstract

For this project, we wrote a computer program capable of recognizing all of the members of a standard deck of playing cards. With a camera pointed at a black background, the program can recognize the contents of its hand and make basic gameplay decisions. Although we set it up to play blackjack, the rules for another small-hand game could be added. The system worked fairly well, but slight changes in card orientation or overall illumination could cause bad card identification. A new version of the system could include more robust card-playing AI, or rules for a number of different card games. The card recognition framework could be coupled with a robotic system to remove the need for the human player and/or dealer to manipulate the cards. Adding a higher resolution camera would increase the number of cards that the computer can examine simultaneously.

Procedure

Image Thresholding

Our first task when presented with a frame containing cards was to differentiate between the cards and the black background. To do this, we used a simple intensity histogram in RGB space. We scanned across the image and determined the number of pixels at each point in RGB space and ran an isodata² algorithm on each color channel to determine a red, blue, and green threshold. Pixels that contained red, blue, or green channels above these thresholds counted as part of card regions, and darker regions counted as background.

Because of our setup, this was a fairly straightforward process. The only problems were with black areas on the cards, but we used the grassfire transform as described below to fix that

² See Appendix B for description of isodata algorithm.

issue. The output of this stage was a binary image with cards (with significant holes) as white and background as black.³

Grassfire Transform

To close up black regions in the cards, we wrote a grassfire transform function. The two passes of the grassfire transform label each pixel in the image with a value based on their distance from the background. For a 4-connected grassfire transform, the first pass checks pixels that are adjacent left and up and sets the value of the current pixel, if it is not a background pixel, to one plus the minimum of its adjacent labels. Background pixels are set to 0. On the second pass, it checks the adjacent right and down pixels and sets the current pixel to one plus the minimum of adjacent labels if it is less than the current label. This correctly labels each pixel with its distance from the background. To perform a shrink of arbitrary magnitude, we simply changed all foreground pixels with appropriate labels to background pixels. To perform a grow, we expanded each pixel with value of one in the four directions according to the specified magnitude.

For our card regions, we first shrunk once to eliminate small regions and then grew five times to close up holes in the cards. Finally, we shrunk five times to return the cards to their original scale. This produced a binary image with white cards and black background. We used the grassfire transform to grow and shrink because it is quick and simple.

Image Segmentation

We ran a two-pass segmentation algorithm on the binary image to segment out the different card regions and determine how many cards were in the image. This segmentation algorithm passes through the image once to check neighboring pixels and set regions. On its

³ See Appendix A for example pictures of this and subsequent procedures.

second pass, it fixes the region labels based on neighboring region labels. This produced a region map correctly labeling each card with its appropriate region number. The segmentation algorithm is in `segment.c`, and `process.c` calls it.

Card Rotation

To rotate the cards, we first needed to calculate some mathematical data. We determined the central moment of each card, which allowed us to calculate the central angle of each of the card regions, as well as the centroid in image coordinates according to accepted mathematical procedure. These values were what we needed to calculate the translation and rotation components of a coordinate transform from image coordinates to card region coordinates.

Once we knew where the card was in the image, and how it was rotated, we could put an oriented bounding box around the card region. We then created a small image the same size as the bounded card region and sampled through the coordinate transform equation to fill the card image with the correct pixels. We first tried to generate the card image from the main image pixels, but that left us with black patterns on the card image. The aliasing effect was caused because sometimes no pixel in the image would correspond to a pixel on the arbitrarily rotated card. We fixed this problem by reversing the direction of the transform and grabbing an image pixel for each card pixel.

After the card image was generated, the coordinate transform was reversed to find the four corner points of the oriented bounding box. The lines of this box were then drawn on the image for debugging purposes.

Card Thresholding

Once we had segmented out the card regions and rotated them to a vertical orientation, we needed to do more processing on each card. We cut out a ppm image of each card, and ran it through another thresholding process. To correctly threshold red cards, our program needed to first paint the red regions black. We did this by running isodata on the red channel and painting any pixels above the red threshold black. This essentially transformed red cards into black cards.

Once we had the necessary regions painted black, we ran isodata on all three color channels and did a reverse intensity threshold to classify non-black regions as background. This gave us very clean binary images with useful card attributes as white and the card background as black. We ran the two-pass algorithm again on these binary images and determined the number and placement of each useful region in the card.

Card Color Recognition

To determine the color of the card, we summed the intensities of all non-background pixels in the original card image and normalized over the number of pixels. This produced an average intensity. We empirically determined a threshold where high intensities showed red cards and low intensities showed black cards.

Card Suit Recognition

Once we successfully produced region maps of each card, we were able to determine the location of the suit and character regions. We found the two leftmost regions and classified the top one as the character and the next to top one as the suit. When trying to differentiate between suits, we had some problems. We first tried using a bounding box fill, but the spade, club, diamond, and heart all have very similar bounding box fills. It did correctly differentiate between

spade and club and diamond and heart with a fair accuracy, but we felt that another feature would help.

To better determine suit, we added a radial projection to our feature set. We started at the centroid of the suit region and moved outwards in the four diagonal directions counting pixels along each diagonal. The Spade and heart have a greater number of pixels along their diagonals, and we were able to successfully differentiate them from the corresponding competing suit. To get the best suit estimate, we combined bounding box fill and radial projection under empirically determined thresholds to create a two feature set.

Card Count Recognition

After determining suit and color, it was time to determine the card's number. The first step was to differentiate court cards from number cards. While there is no intuitive way of doing this, we realized that the court pictures were segmenting as non-background, and the interior of the court card image contained a lot of non-background pixels. We used a simple bounding box fill with an empirically determined threshold on the center of the image to correctly separate court cards.

For number cards, we simply counted the number of regions in the center of the image. This worked fine. To differentiate between the king, queen, and jack, we used a neural network trained on the character region. The section below describes this process.

Face Card Character Recognition

Since we only need to distinguish between the three face cards, we trained three simple networks with a 10x10 input layer, 5x5 hidden layer, and a single output. The output was thresholded at 0.1, so outputs between 0 and 0.1 were rejected, and outputs between 0.1 and 1 were considered a match.

The three networks were cascaded, with the default case being a Jack. That is, if all three networks rejected the character, but the bounding box fill from the previous step implied that it was a face card, we assumed it was a Jack. In general, Jacks were the hardest to recognize. This was probably due to the fact that the J is the smallest character of the three, and has a lot of similarities to the Q, at least in the font face used on our deck of cards.

To provide input for the networks, we took the upper left region in the card image, binarized it, and scaled it to a 10x10 array. We trained the networks using the Stuttgart Neural Network Simulator (SNNS), and we generated a C function that was called from our program. Although it took a long time initially to generate a training set, and to train the networks, the calculation speed was very good.

The training set contained 96 character images, with each of the 12 face cards (for all values and suits) captured in 8 different orientations. The program pattern takes a list of image files and sets accept or reject values (1 or 0) for all of the input images. It then generates a .pat file that can be used to train a neural net in SNNS. Generating three pattern files from the training set did not take very long. We performed 150 training epochs, in groups of 10, shuffling the training pattern each time.

Result Voting

After our system was up and running, we found that we were getting very good accuracy for our card recognition. The system, however, would make an occasional mistake caused by the misclassification of a region or poor segmentation. We found that often simply running the algorithms on the same setup would produce the correct result. This led us to believe that noise in the camera was the source of many misclassifications.

To fix this problem, we implemented a voting scheme where the system runs for five frames before making a decision as to the value of the card. Each frame votes for a certain card type, and the type with the most votes gets displayed on the screen. This helped to eliminate the effect of camera noise on our system.

Playing Blackjack

The blackjack playing functions assess the value of a hand that is visible to the camera, and decide what action (hit or stay) should be taken.⁴ The functions can also perform these calculations on the human player's hand. The functions also handle prompting the user for input, and determine whether we are in the game loop or not. The code considers both the current value of the computer's hand, and the value of the human player's up card.

The computer will always stay if the value of its hand is 17 or higher. If the opponent's upcard is high, the computer will hit if its hand is 16 or lower. The program also knows that the game is over if it gets five cards without hitting 21. During the end game, when it asks the human player to display its full hand, it compares values and determines who won, or if there was a push.

Discussion

Although the system worked well in the end, we had a couple of problems that required hours of trial and error, as well as repeated trips back to the drawing board. The two biggest and most time consuming problems occurred with the coordinate transform between image space and card space, used to get properly oriented images of card regions, and the face card identification neural networks.

⁴ The play decision code is based on a program written by co-author Nick Ward for his Computer Vision Fundamentals class.

The coordinate transform required a translation and a rotation, in both directions (image to card and card to image). We initially tried to implement it using the Pythagorean Theorem and direct point-to-point transforms. Unfortunately, the calculations were complex enough that it was very easy to confuse terms. The source of the problems was probably sign errors on the coefficients of the sinusoidal functions used in the calculations. We got some very interesting results, especially with the bounding boxes. One of the more amusing errors were rotationally variant bounding boxes, i.e. their angle relative to the card and their size would change as the card was rotated in the image.

We went back to the math, and realized that it would be far easier to use vector dot products. One thing that made this still difficult was the fact that the image coordinate plane is oriented with the y-axis pointing downwards, but we didn't flip the axis for the card coordinate plane. We were able to do the image transform by dotting the vectors to a pixel in one coordinate system with the unit vectors in the other coordinate system. We eventually got all of the sign problems fixed so we could produce images of the cards with proper oriented bounding boxes.

Our first attempt with the character recognition neural network completely failed. We started with too large of a test space (all of the characters, instead of just the face cards), and we had too small of a training set (only one instance of each card type). We were going to use two networks, one to identify face cards, and one to distinguish between them. This simply did not work, so we used the bounding box fill to find face cards and trained separate networks for each of the three cards.

Overall, this project was a great success, and a good comprehensive demonstration of computer vision techniques. The system itself is fairly general, and has several areas where we could expand the project or improve its performance. Specifically, we could improve the actual

card playing ability, although that has nothing to do with computer vision, or we could create an even large training set to improve our character recognition neural networks. We hope that this project helps to spur further research at Swarthmore College in the field of object recognition.

References

Dr. Bruce A. Maxwell. Computer Vision Class Website. 2003:

<http://palantir.swarthmore.edu/maxwell/classes/e27/F03/2003>

Appendices

Appendix A: Sample Images

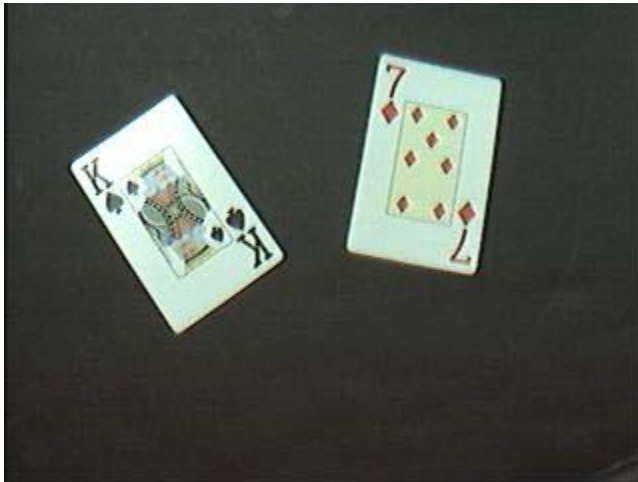


Figure 1: Raw input Image

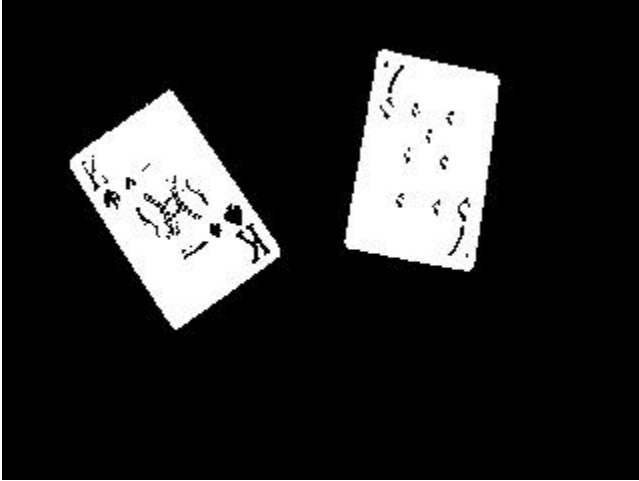


Figure 2: Binarized image

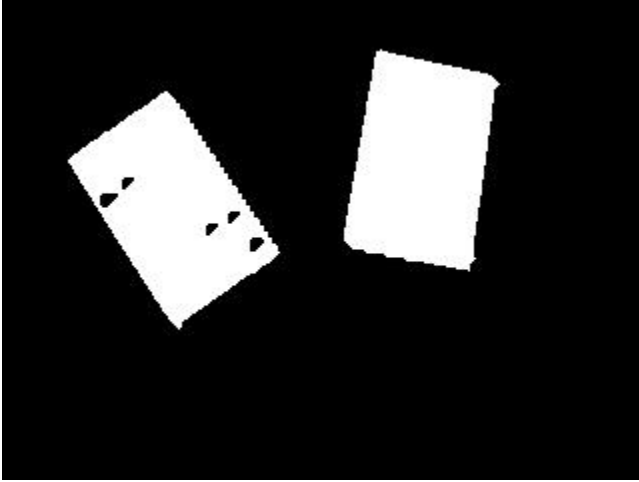


Figure 3: Binarized image after grassfire transform

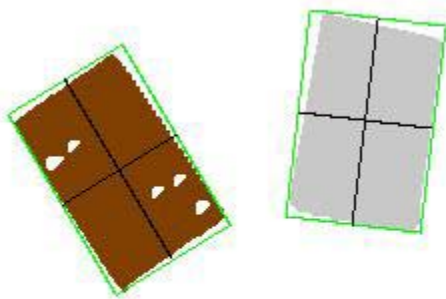
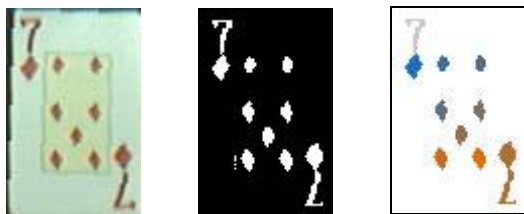


Figure 4: Image with colored regions, region axes, and oriented bounding boxes



Figures 5-7: Card images of 7 of diamonds: raw (left), binarized (middle), regions colored (right)



Figures 8-9: Binarized character image of a 7 (left) and diamond (right)



Figures 10-12: Images of king of spades: raw (left), binarized (middle), regions colored (right)



Figures 13-14: Binarized character image of a K (left) and spade (right)

Appendix B: Description of Isodata Algorithm⁵

The following pseudocode implements a clustering algorithm--the ISODATA method--to automatically pick the threshold for an image. The first thing to do is to smooth out your histogram by making a new histogram where each bucket is a weighted average of itself and its neighbors to either side.

Then, run the ISODATA algorithm is as follows:

```
thresh = range/2;
while(1) {
    cpmean = highmean = cpcount = highcount = 0;
    for all buckets i in the histogram up to thresh
        cpmean = cpmean + bucket_value[i] * bucket_count[i];
        cpcount = cpcount + bucket_count[i];
    for all buckets i in the histogram above the threshold
        highmean = highmean + bucket_value[i] * bucket_count[i];
        highcount = highcount + bucket_count[i];
    if thresh = (cpmean + highmean)/2 then
        break;
    else
        thresh = cpmean + highmean/2;
}
```

Thresh is the final value of the threshold.

⁵ Description is from Bruce Maxwell's lab summary for Computer Vision lab 1 in Fall 2003. Available at: <http://palantir.swarthmore.edu/maxwell/classes/e27/F03/labs/lab01/>