Playing Card Recognition

Computer Vision (EE4H) — Final Report

Yousef Amar (1095307)
Chris Lewis (1234567)

2014-04-28

Abstract

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

1 Introduction

1.1 Task Assigned

1.2 Method Used

1.3 Findings

2 Review

2.1 Identify sub-tasks

(Flowhart breakdown from final PPT?)

2.2 Review of methods

(existing literature (HIPR2?)) Should each include limitations if not chosen, justification in 3.2

2.2.1 High-level Isolation

Contours Perspective transform (That OpenGL citation?)

2.2.2 Low-level Classification

Basic morphology HoM Transform

3 Proposed Method

3.1 Logical progression from whole image to classification

Program flow, flowchart

3.2 Justify each stages techniques

Identify strengths and weaknesses

4 Implementation

Justify custom implementations vs. OpenCV pre-made function ˆ(interleaved and when relevant)

4.1 Image IO/Webcam IO

4.2 Card Isolation

Once an image is passed in as an input, the first step is to find one or more cards in the image and, if any were found, extract them from the image and prepare them for processing. This is done in a number of steps.

4.2.1 Card Detection

In order to detect cards within the image, the function `find_cards()` in *isolation.cpp* (appendix B.1, page 13) is called from *main.cpp* (appendix B.1, page 13). It takes in an empty vector of objects of type `Card` as an inputs which it then populates with the detected cards for later processing.

Cards are detected by finding quadrilaterals with a method not unlike the one used in the sample file *squares.cpp* in the official OpenCV repository (Kamaev, 2013-04-08) albeit with some decisive distinctions. The backbone of the algorithm is the OpenCV function `cv::findContours()`. This function retrieves contours from a binary image using the algorithm developed by Suzuki et al. (1985) via border following. It was decided not to implement a custom function since it would not have added any flexibility or speed advantage over the OpenCV implementation. Indeed it would only have increased code complexity.

This function requires a binary image as an input. As such, input images undergo a series of processing before hand. First, they are converted to greyscale. Then CLAHE — as explained previously (section **??**, page **??**) — is applied to these images. The size of the CLAHE window varies between 32x32 and 28x28 pixels depending on if the user has specified with the `--multi` command line flag that the program should run in multiple card mode. CLAHE operations using OpenCV functions run on the GPU through CUDA; they cannot be optimised by writing custom code for CLAHE without utilising CUDA, which would only introduce system-level bugs that have been long since resolved within OpenCV's code base. For this reason, the OpenCV CLAHE implementation was used.

After performing CLAHE, the image is blurred. The amount of blur is set as a function of it image's dimensions. More specifically `(image_size.width + image_size.height)>>8` or in plain English, the perimeter of the image divided by 512. This prevents small images from losing too much detail through blurring while large images do not remain too sharp. Blurring is done to filter out noise for the next step.

The resulting image is then converted to binary using the threshold parameter before `cv::findContours()` is called. Contours are found for processed binary images for thresholds ranging from 150 to 255 (both inclusive) in increments of 2. This bright range was experimentally determined to be the best to create binary images from.

The found contours are then approximated into geometry using with the function `cv::approxPolyDP()` and then go through a number of rigorous checks. Contours that do not pass these checks are discarded and not added to the final contours collection that the function populates. These include:

1. Does the contour have four corners?

   Although card contours can have more than 4 approximated corners if the cards are bent or folded significantly, most will not, so the design does not handle card contours with more than four corners.

2. Is its area greater than 4% of the whole image?

   This is to filter out noise.

3. Is it convex in shape like card borders?

4. Is its area less than 75% of the image?

   This is to filter out borders.

5. Is it not inside another previously detected contour or another inside it?

   Calculated mathematically from vertex coordinate information. This is to filter out symbol (e.g. diamonds) and picture card misclassification. Previously detected contours in the list that turn out to be contained by newly detected contours are replaced by the newly detected and discarded.

6. Is it significantly dissimilar to previously detected contours?

   The criteria for this check are that every rotation of the four contour vertices are at least a certain total distance away from their counterparts (summed up manhattan/city block distance in pixels). This minimum total distance is a function of the image size to normalise pixel units into units relative to the image dimensions. This relationship is `(image_size.width + image_size.height) * 0.5F * min_square_diff` or in other words, the mean image side length (to account for disproportionate image aspect ratios) multiplied by a constant factor set to 0.1, i.e. 10% of the mean image side length.

Once all contours have been iterated through and the ones that fail these criteria have been filtered out, the result is a collection of contour quads that surround at least every card in the image (in the form of a `vector<vector<cv::Point> >` — a vector of vectors with four points each). *At least* every card in the image; the pruning process is not yet over!

The assortment of contours go through a secondary, later filtration process that requires processing the pixel data within these contours. To do this — as well as to construct the card Mats for subsequent classification — the contour quad data is used to perspective

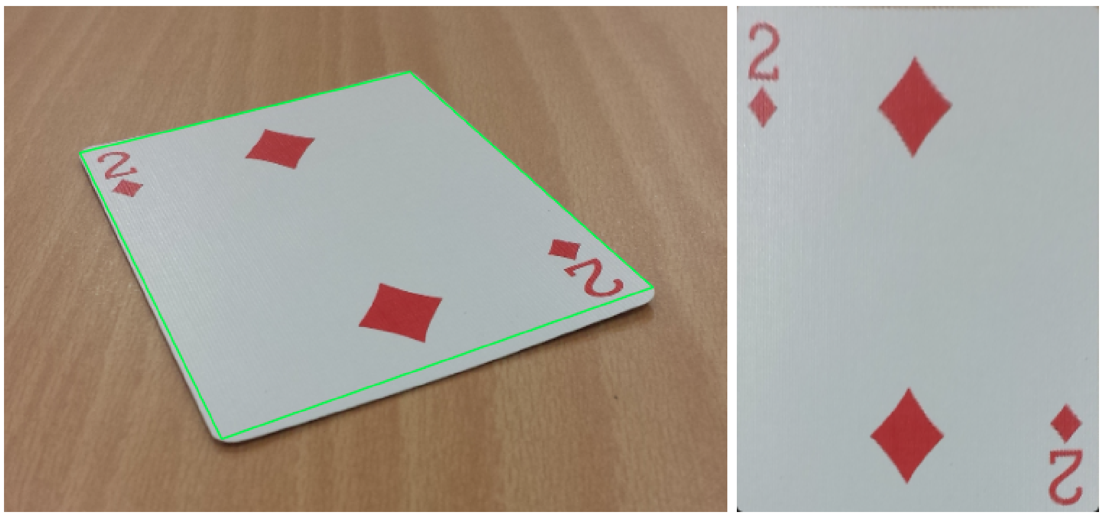

Figure 1: Output of Perspective Transform of Contour Quad from Test Image

transform the original image into a rectangular image of the card, an operation reviewed previously (section ??, page ??).

The perspective transform is low-level enough that its implementations do not vary much at all. The maths involved in calculating the 3x3 matrix of a perspective transform that can be used to map pixels at a perspective to other shapes/perspectives is standard but cumbersome to implement (unlike an affine transform). For these reasons, the OpenCV function `cv::getPerspectiveTransform()` was used. It takes two quads as inputs and returns the map matrix. The source quad is the detected contour and the destination quad is a simple 250x350 rectangle which matches the standard 5:7 poker card aspect ratio like in figure 1.

The transformation matrix is then used in the function `cv::warpPerspective()` which deforms the pixel grid of the source image and maps it to the destination image while interpolating transitional pixels; a process that is both non-trivial as well as distant from the task at hand. The Mat produced through this process is subsequently used to instantiate a new `Card` object.

On instantiation, the `Card` constructor additionally generates an 8x8 CLAHE version and a 140 threshold binary version of the card Mat, which are publicly accessible from a `Card` object. Once instantiated, the cards binary Mat is finally used for the last phase of isolation filtering.

The last criteria for what makes a card is its whiteness measure. The measure is simple; all non-zero (i.e. white) pixels in the card Mat are counted and the final value is divided by the card's area giving a float that describes the proportion of whiteness in the card. The whiteness threshold is set to an experimentally practical `0.5F` or 50% bearing in mind that error due to lighting inconsistency is a non-issue following CLAHE. Cards that are not white enough are discarded. This solves false detections such as that in figure 2.

Finally, there is one more processing step that needs to be undertaken before a detected card is deemed worthy enough to join its brethren in the detected card vector and

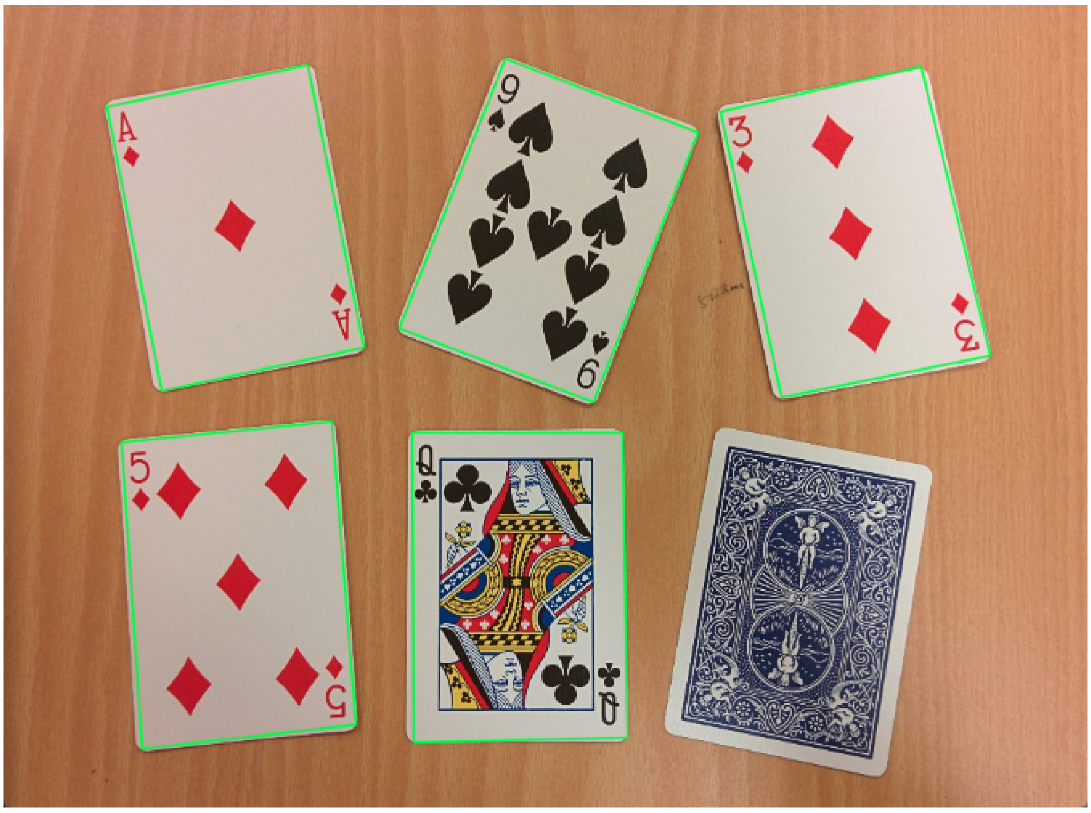

Figure 2: Example of Whiteness Check Ignoring Overturned Card

`find_cards()` can return...

4.2.2 Rotation Correction

One deal-breaking issue was the problem of a sequence of contour quad vertices starting at the wrong corner and, as a direct consequence, causing the final card Mat to be perspective transformed incorrectly. This problem became evident became evident when processing cards where the top-most corner in the image is numbered, such as the lower examples in figure 3; the card could just be at a very strange perspective that projects to that shape.

The solution developed is simple and works very well. The same measure of whiteness as is used for the whole card is applied to both the top left and the bottom right corners of the card Mat (the bounds for which are defined as static constants in the `Card` class based on physical measurements). If corner whiteness surpasses 80%, the card quad is rotated by one vertex — the direction or rotation being arbitrary due to a card's symmetry — and transformed into a new Mat.

The corner whiteness for this new Mat is also measured, and compared against old corner whiteness. If it has a lesser corner whiteness measure than the old, that means that the old card Mat was indeed oriented incorrectly and so the new one is used instead. If the opposite holds true, then the first overall whiteness threshold was passed incorrectly (perhaps due to some bizarre specular lighting that the CLAHE did not quite catch)

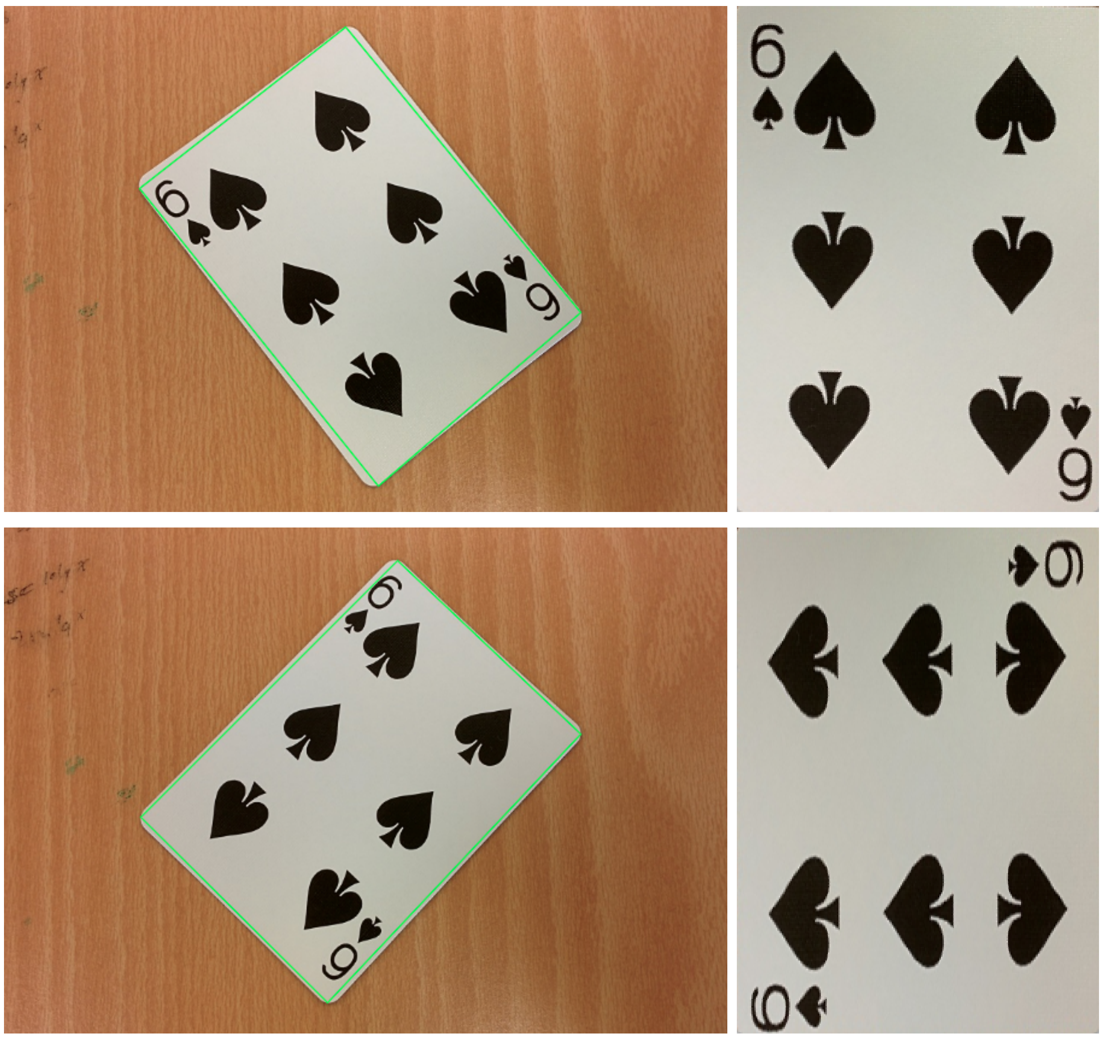
Figure 3: Example of Erroneous Perspective Transformation

and so the new mat is discarded and the old one's usage is continued as if nothing ever happened.

At long last, the `Card` vector contains all detected cards — to a high degree of accuracy — ready for processing and classification by the rest of the program, and the function can return.

4.3 Color Detection

4.4 Type Detection

4.5 Symbol/Rank symbol Isolation

4.6 Number Detection

4.6.1 Value cards

Morphological opening/closing Floodfill count

4.6.2 Picture cards

Application of HoM

4.6.3 Custom HoM Implementation

4.7 Suit Symbol Detection

Colour Integration HoM

4.8 Results output

GUI Console output

5 Evaluation

5.1 Test images, increasing complexity

5.2 Webcam tests

5.3 Confusion Matrix

6 Conclusion

6.1 Summary

6.2 Improvements

Appendices

A Contribution Agreement

(I have both of these and can scan them)

B Source Code Listings

B.1 main.cpp

```
/**********************************************************************\
| Main code file for the EE4H Assignment (Playing card recognition)      |
|                                     |
| Authors: Yousef Amar and Chris Lewis                   |
|                                     |
| Dependencies: OpenCV-2.4.8                         |
|       - opencv_core248.dll                     |
|       - opencv_imgproc248.dll                    |
|       - opencv_highgui248.dll                    |
|       - tbb.dll (Built for Intel x64)                |
\**********************************************************************/

#include "../include/stdafx.h"

using namespace std;

int process_image(cv::Mat input)
{
  cv::Size input_size = input.size();

  //Check size is greater than zero
  if(input_size.width > 0 && input_size.height > 0)
  {
    //Show image details
    cout << "Image is " << input_size.width << " by " << input_size.height << " 
    pixels." << endl << endl;
    //cv::imshow("Input", input);

    //Hack to deal with super large, hi-res images
    if (input_size.width > 1000)
      cv::resize(input, input, cv::Size(1000, 1000*input_size.height/input_size.width));
    if (input_size.height > 500)
      cv::resize(input, input, cv::Size(500*input_size.width/input_size.height, 500));

    //Find cards in image and populate cards vector
    vector<Card> cards;
    if (find_cards(input, &cards)) {
      cerr << "Could not find any cards!" << endl;
      return -4; //No cards found
    }

    for(size_t i = 0; i < cards.size(); i++)
    {
      Card *card = &cards[i];

      //Detect suit colour
      detect_colour(card);

      //Detect suit type (number/picture)
      detect_type(card);

      //Find symbols
      find_symbols(card);

      //Get card value
      if (!card->is_picture_card)
      {
        detect_value_number(card);
      }
      else
      {
        detect_value_picture(card);
      }
      //Find suit
      find_suit_sym(card, 0.95F);
    }

    //Show results until key press
    show_cascade(cards);

  }
  else
  {
    cerr << "Image dimensions must be > 0!" << endl;
    return -2;  //Image size zero code
  }

  //Finally
  cout << "Processing finished successfully!" << endl;
  return 0; //No error code
}


/**
  * Program entry point.
  *
  * Arguments
  *    int argc: Number of arguments
  * char** argv: Array of arguments: [1] - Image path to open
  *
  * Returns
  * int: Error code or 0 if no error occured
  */
int main(int argc, char **argv)
{
  cout << endl << "----------------------------------------------" << endl
         << " EE4H Assignment - Recognising playing cards   " << endl
         << " By Yousef Amar and Chris Lewis                " << endl
         << "----------------------------------------------" << endl << endl;

  //Check image is provided
  if(argc < 2)
  {
    cout << "Arguments error. Check image path/format? Did you mean to use --cam?" <<
    endl;
    return -1;  //Incorrect arguments code
  }

  cv::Mat input;

  bool from_cam = !strcmp(argv[1], "--cam"), should_quit = false;

  if (argc > 2) {
    cout << "Multi mode activated!" << endl;
    multi_mode = !strcmp(argv[2], "--multi");
  }

  do
  {
    if(!from_cam)
    {
      input = cv::imread(argv[1], CV_LOAD_IMAGE_COLOR);
    }
    else
    {
      cv::VideoCapture cap(CV_CAP_ANY);
      cv::waitKey(1000);

      if(!cap.isOpened())
      {
        cerr << "Unable to access webcam" << endl;
        return -3;  //Incorrect arguments code
      }
      cout << "Press space to take a photo" << endl;

      char key;
      do
      {
        key = cvWaitKey(10);

        cap >> input;

        cv::imshow("Webcam", input);

        // Break out of loop if Space key is pressed
      } while (char(key) != 32);

      cv::destroyWindow("Webcam");
    }

    if (process_image(input))
      return EXIT_FAILURE;

    if (!from_cam)
    {
      cout << "Press any key to quit" << endl;
    } else {
      cout << "Press Esc to quit, any other key to try again" << endl;
    }

    should_quit = (from_cam && cv::waitKey(0)==27) || (!from_cam && cv::waitKey(0));

  } while(!should_quit);

  return EXIT_SUCCESS;
}
```