Dice Pip Detection

James Young

***Abstract*—Inspired by table-top games, this project aimed to develop a program that would provide a meaningful application of machine vision techniques learned in this semester. The program itself takes an input image of rolled dice, and outputs where dice are located, as well the number of pips on the upwards face. In its current state, the program is capable of detecting 89.13% of rolled dice, with 100% accuracy in counting the pips of detected dice. Beyond this point, the program has many avenues of continuation, such as integration into actual applications, to improving the underlying code.**

***Index Terms*—machine vision, dice detection, table-top games**

I. Introduction

THIS project was first thought of as both a fun and challenging application of machine vision as well as an extension to another program I have been developing for the past year.

To give some background, I greatly enjoy table-top gaming as a hobby and have regularly played with a group for years. However, since progressing into graduate school, as well as the majority of the group also going their own ways, we no longer had an easy way to continue playing. Not one to give up, a solution proposed was to develop a chat client bot that would facilitate many of the physical aspects of table-top gaming; the first and foremost being dice rolling. This application became known as GoblinBot (affectionately named from the fantasy creature found in some of the more well known table-top games).

The primary purpose of GoblinBot is to provide the ability to roll dice in a manner such that all players can see the result easily. Thankfully, having this based as a chat client bot, all results are returned via a message from the bot. This solved the most practical problem of playing these games over the internet, but some issues still remained. One element that makes these games exciting is making critical rolls (no pun intended) and watching the dice settle- for better or for worse. Even with GoblinBot, there is a loss of this anticipation or excitement, as at the end of the day, the bot is still just an electronic program.

This is where this program comes into the picture; a melding between the analog and digital. In the lens of GoblinBot, longer standing goal of this project is to eventually tie it into the already existing program with hopes that it provides an improvement to the overall experience. Outside of this, this project seemed like a reasonable challenge to undertake. Having completed the work, it certainly was and I believe that it was a good summation of much of what was taught in the Machine Vision course.

II. Project Goals

This project had two major goals to achieve in order to produce an ideal program.

- Be able to detect dice and count the pips.
- Be able to do detection within a reasonable frame rate.

III. Program Overview/Approach

The main problem needed to be solved by this program is to be able to take in an input image, and through a series of transformations/thresholds/etc. be able to isolate both the dice as well as it's pips. Every step of note in this process will be discussed such that it can be easily understood and re-implemented.

A. Grab Input

To begin the detection process, the program needs a background image (no dice present) (Fig. 1) from which it can use to differentiate from the foreground image (dice present) (Fig. 2).

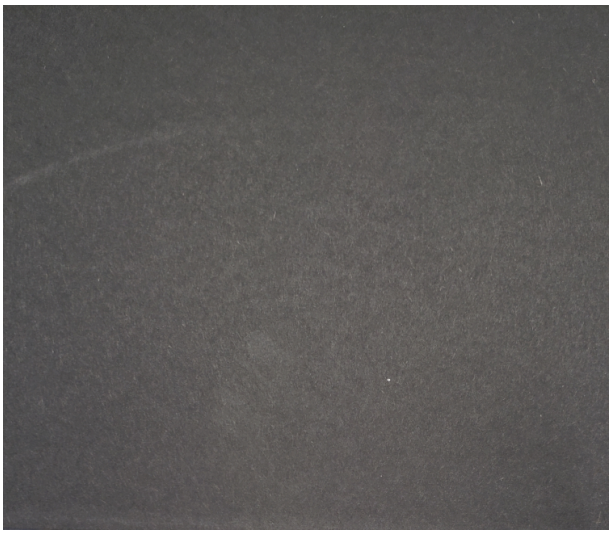


Fig. 1: The background supplied

B. Obtain Difference

Once the foreground and background images have been read in, then a difference between the two images is performed (Fig. 3). This begins the process of finding the regions of interest where the dice are located.

Fig. 2: The foreground supplied

Fig. 3: Difference between background and foreground

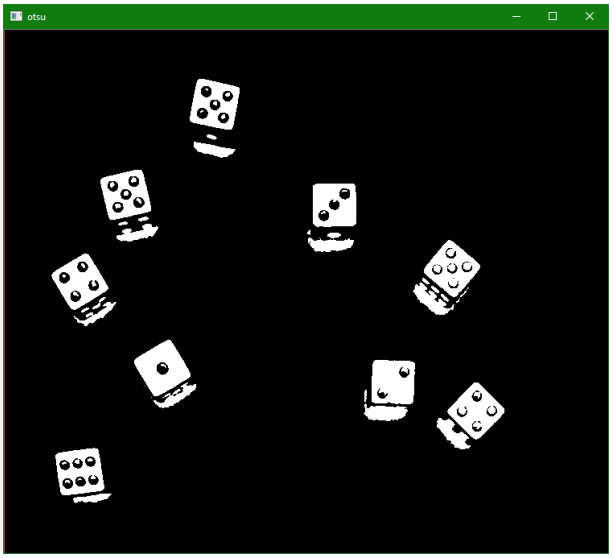


Fig. 4: Otsu's Threshold

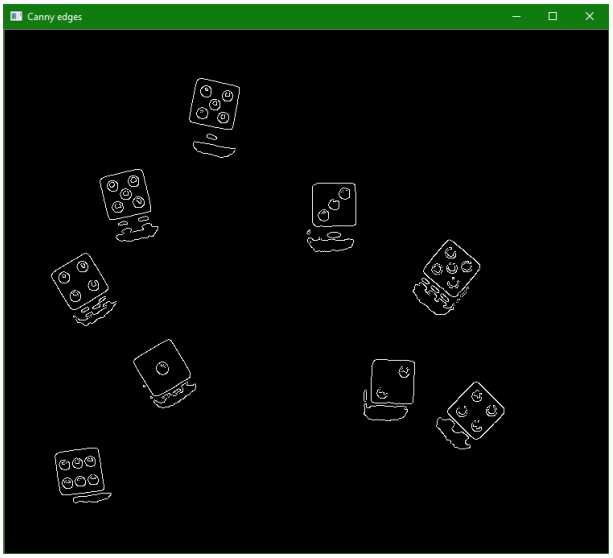


Fig. 5: Canny Edge Detection

C. Gaussian Blur and Otsu's Threshold

Gaussian Blur and Otsu's Threshold (Fig. 4) is applied to the image to prepare it for Canny Edge detection.

D. Canny Edge Detection

Canny Edge detection is performed (Fig. 5), and at this point the actual dice detection begins.

E. Find and Iterate Through Contours

With the Canny Edge detection performed, the program begins to look at all of the closed contours found in the image. If the closed contour found is larger than a specified size (to avoid grabbing singular pips, along other small noise), it will then pass that contour onto a sub-process which will extract and count the pips.

1) Use Contour to Obtain Bounding Box: Once a contour of sufficient size has been found, a bounding box is calculated.

2) Crop Out Sub-Image: Using the calculated bounding box, the region is then cropped out of the original foreground image. By cropping from the original input image, it provides the ability to look at more detail on each individual dice by using more specialized parameters.

3) Resize: With the isolated and cropped out dice, the image is then resized to amplify details and ease the detection of pips (Fig. 6). In this case, the size was doubled.

4) Gaussian Blur: Like in earlier steps, Gaussian blur is applied to improve performance of subsequent operations (Fig. 7).

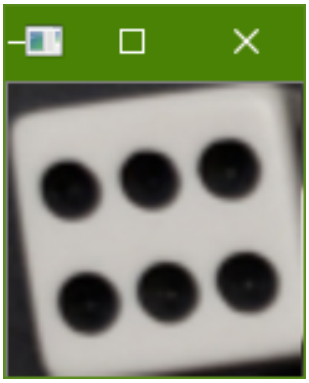


Fig. 6: Cropped and Enlarged Die

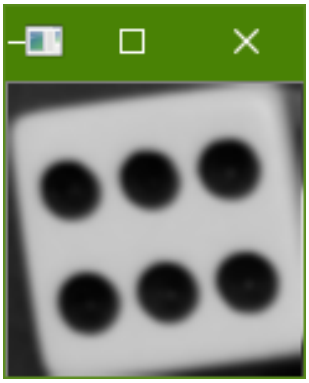


Fig. 7: Gaussian Blur Applied

5) Otsu's Threshold: Application of Otsu's Threshold to further bring out pips from the surrounding dice face (Fig. 8).

Fig. 8: Otsu's Threshold Applied

6) Bucket Fill Corners: At this point, blob detection can be performed, but the application of bucket filling the corners ensures that the pips, and only the pips, are left when blob detection is run. As the name implies, the program takes the four corners of the image and fills with white pixels (Fig. 9).

7) Use Blob Detection to Count Pips: Finally, with the pips being completely isolated, blob detection is used, and the number of detected blobs is counted, giving the number of pips on the die (Fig. 10).

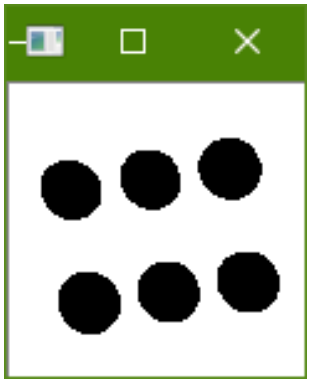


Fig. 9: Bucket Fill Corners

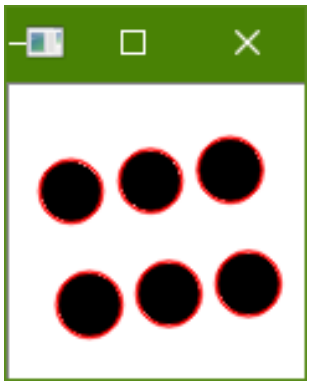


Fig. 10: Count Pips via Blob Detection

F. Draw Bounding Boxes and Pip Count

After running on each closed contour, the program then draws each bounding box and corresponding number of pips, resulting in the final output (Fig. 11).

IV. Results

The immediate results of this project show that the program can detect 89.13% of dice, with a 100% accuracy rate when counting the pips on detected dice. It should be noted however that these results were taken from a small set of five test images, four of which consisted of rolled dice, and one being manually set up. In addition, two additional images were provided containing edge case scenarios: Immediately Adjacent Dice and Overlapping Dice. The program was unable to detect dice in these two images.

Performance-wise, frame rate was investigated by looping the operations and recording how long it took to process each loop. When testing this, frame rate never dropped below 5 frames per second, even with a great deal of debug code running in tandem.

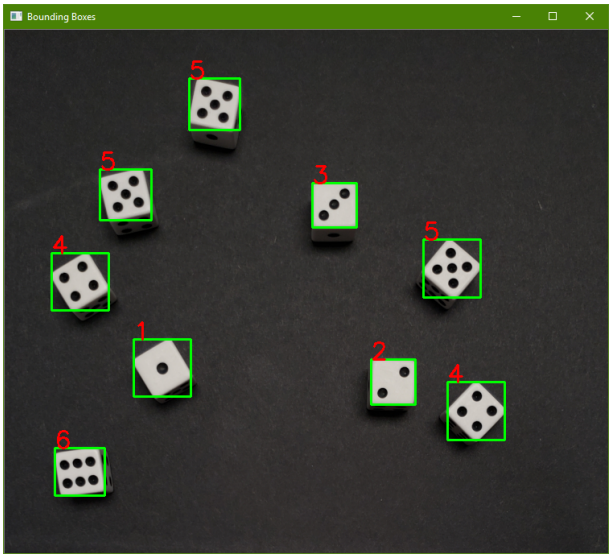
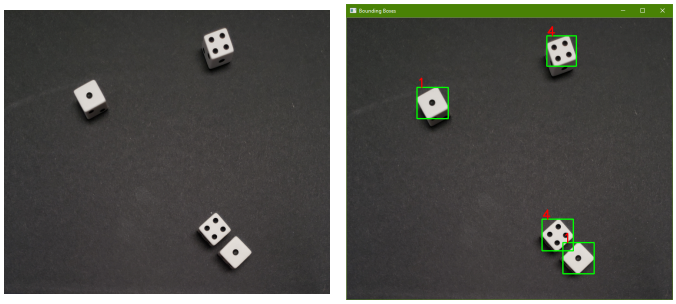


Fig. 11: Final Output with Bounding Boxes and Number of Pips Drawn

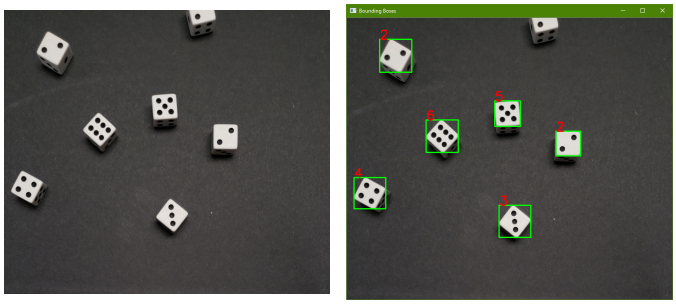


(a) Input (b) Output

Fig. 12: Input and Output using test image 1

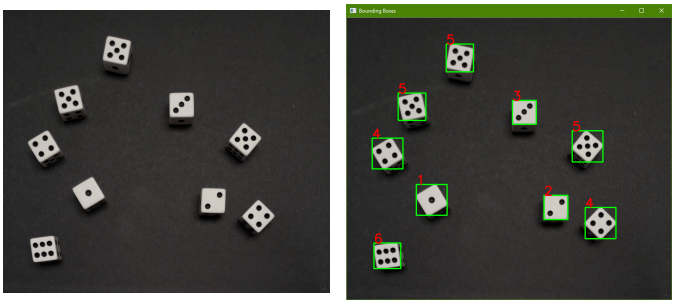
V. Discussion

Having started work on this program from scratch, these results are quite exciting. Drawing from figures 12 to 16 we can see many different, interesting aspects. As mentioned earlier, the first four test images were of actual dice rolls, with the fifth being a more controlled setup to better gauge the performance of multiple dice.



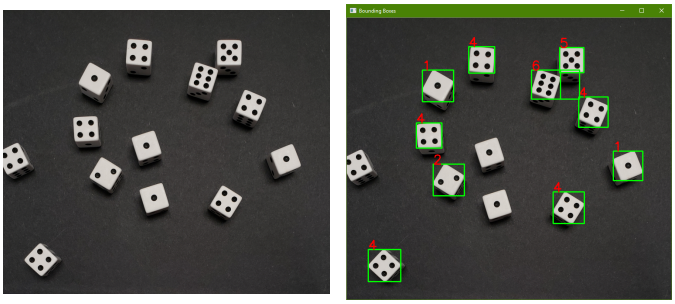
(a) Input (b) Output

Fig. 13: Input and Output using test image 2



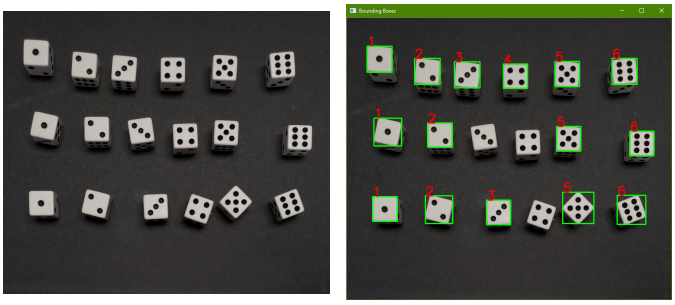
(a) Input (b) Output

Fig. 14: Input and Output using test image 3



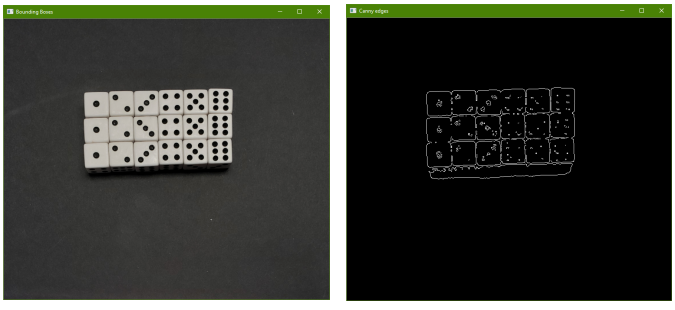
(a) Input (b) Output

Fig. 15: Input and Output using test image 4



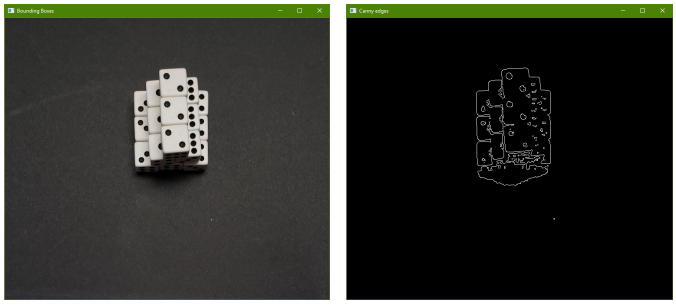
(a) Input (b) Output

Fig. 16: Input and Output using test image 5



(a) Input (b) Canny Edge Detection

Fig. 17: Input and Canny Edge using test image 6 (edge case)



(a) Input (b) Canny Edge Detection

Fig. 18: Input and Canny Edge using test image 7 (edge case)

Fig. 19: Canny Edge detection from test image 5

In figure 12, we can see that the four dice were all correctly located and counted, with two dice in the lower right being relatively close. While maybe not as interesting as what follows, this shows a good baseline application/use case of the program

(a) Fourth Dice, Last Row (b) Third Dice, Middle Row

Fig. 20: Zoomed in image of un-closed contours from test image 5

In figure 13, things start to get a little more interesting. All dice that were rolled within frame were able to be located and counted, leaving out the die out of frame. When looking over the results of this image, it was quite encouraging that the program did not miscount the dice as only having one pip, and instead ignoring it. This exclusion is primarily due to the contour not being closed, which could be an issue should the closed contours no longer be a criteria for detection. For the time being however, it does its job and produces ideal results.

Figure 14 is the best result produced by the program, in that it is able to get 100% detection, 100% accuracy on counting, as well as being rolled naturally. Not much else needs to be said about this figure, other than that this is the ideal performance for the program.

Figure 15 starts to show some of the issues in the dice detection. First, the two dice in the center, showing ones, are not picked up by the program. Second, the bounding box on the six in the upper right is heavily stretched out in comparison to the intended box shape. Since the non-detection problem is so large, discussion of that will be in a later section, and for now the focus will be on the misshapen bounding box. To the best of my understanding, this is a result of the program picking up the side of five dice right next to it, thus increasing the size of the box. Thankfully, likely due to the lighting, the pips on the side of the die are not counted, and it still gets the correct value of six.

Finally, figure 16 shows the similar issues of non-detection. This being the case, the program is still able to detect the large majority of dice in the image, as well as correctly counting the pips.

Moving onto the topic of non-detection, figure 19 shows the Canny Edge detection from test image 5, and figure 20 (a) and (b) show a zoomed in version of a couple of the non-detected dice. As described earlier, the program checks for closed contours to consider the region to be of interest or not, and as can be seen here, there are small gaps in the contours. While the program is fairly robust in avoiding noise (mostly due to the heavy constraints used to handle the input) it cannot fully avoid it even in more controlled environments, and certainly not in other environments. The take-away from this is that this would be one of the next steps in developing this program- figuring out a solution to close, or circumvent, the gaps in contours.

Currently, the biggest issue this program faces is its inability to detect 100% of the dice in the image. Unfortunately, given the time frame of this project I was unable to fix these issues in time, however below are some of the potential solutions to this problem.

- Use better parameters to improve each step involved in the edge detection/contouring process.
- Use morphology to attempt to patch the small gaps.

- Suggested in the presentation portion of this project, use the fact that the dice face is square to allow for a more flexible detection scheme (so long as the contour is mostly a square, it should be able to detect)

Beyond the initial set of test images, two additional images were taken and used as input for the program. The goal with these two edge case images were to see if any insight could be gained from how the program handled the images, as well as to sate simple curiosity. In figures 17 (adjacent dice) and 18 (overlapping dice), the input image, as well as its Canny Edge detection are shown. The biggest take away from these sets of images is to see how the contour is shaped, which by extension explains why the detection failed. While it is fairly clear that the overlapping input would have a great deal of trouble reaching a state where it could be successfully detected, the adjacent input is quite close to forming closed contours. Detection could likely be achieved by additional fine-tuning of the various parameters.

Finally, as far as the frame rate is concerned, I believe it reaches the goal of being reasonable. The frame rate was never a problem during the development phase, and if it was, there is still a great deal of room for optimization.

VI. Conclusion

As far as I am concerned, this project can be seen as a success. While the goal of detection was not achieved 100%, the ground work to reach that point is absolutely present. The project itself was quite enjoyable to work on as well, which is always a plus. Beyond that, the following two subsections discuss directions that this work could go in, both internally and externally.

A. Future Work

This section describes potential extensions of the work done in this project.

1) Improve Detection/Bug Fixes: The immediate steps to be taken with this project would be to fix the errors with the dice detection, namely fixing the issue of small gaps in the contours. Beyond this, there are small improvements to parameters and other aspects that could be undertaken as well, such as removing the constraint of requiring a background image for detection, or modifying the program that it is more flexible in what it considers a valid contour that denotes a die.

2) Broaden Detection Scope: Beyond direct improvements to the program, another area of future work would be to achieve colored dice and numbered dice detection. Initially, this project was to detect colored dice due to their widespread use, but this was constrained due to the additional difficulty it posed when trying to detect them. While the program currently only covered dice with pips, another extension of this project would be to detect dice with numbers on them. The proposed approach to this problem is to isolate the dice in a much, if not completely, similar way, but then use more advanced methods than blob detection to read the number from the die face. One such method might be to use a machine learning model, trained on detecting numbers to read the value.

3) Machine Learning Approach: While this is slightly covered in the previous section, this would be a machine learning implementation of the whole process. One of the larger problems with this however is the lack of a good data set, but with a good enough data set, this approach would presumably be much more flexible. Some of the major gains would be the loosening of the heavy constraints placed on how the input images have to be taken, as well as being able to easily train the model on both pips and numbers.

B. Applications

While similar to future work in the sense that this would be what would naturally following the completion of this project, this section will focus more on the external aspects of how this program can be taken and used, rather than internal improvements.

1) Augmented Reality Dice Games: Probably the easiest to work with, one application of this program would be to apply it to simple dice games such as Yahtzee or Farkle. Since the program is able to read in many different aspects of rolled dice and then use them in a digital manner, a natural progression would be to use this as input to a game. While it would depend on the game, the only elements that would need to be added is game states/turns, as well as keeping score. Beyond this, additional flair and aspects can be programmed in to produce a more interesting and exciting display.

2) Phone Application or Tool: Closer to the original goal/inspiration for this project, this program could also be used as a tool to assist table-top gaming. In the most general case, the ability to quickly count, as well as potentially sum, a large amount of dice is invaluable given many complex table-top games. Rolling up to eight dice at a time is often common, and there are situations in some games that require up to rolling as many as forty dice, all at once, to perform an action. Being able to speed up the process of counting dice would almost certainly translate into faster games.

3) Bringing Physical Aspects to Network Play: Closest now to the inspiration of the project is the idea of bringing aspects of in-person play to an online session. One of the larger issues for playing traditionally table-top games over the internet is the inherent lack of having all players in one spot. One of the biggest appeal to table-top gaming is being able to have face-to-face interaction with the other players, which is wholly removed when playing over the internet. This application of the program in this way would be doing no more than providing some helpful tools to otherwise a

webcam focused on some dice, but the intent is that it adds the sense of physicality to an otherwise wholly virtual game session.