FANUC iRVision System Implementation

Dawson Burgess
Computer Science Department
University of Idaho, Moscow, ID
Moscow, ID, United States
burg1648@vandals.uidaho.edu dawsonhburgess@gmail.com

I. Introduction

In the constantly shifting landscape of industrial automation, the use of advanced vision systems in conjunction with robotic technology has become a cornerstone for boosting efficiency and precision. Vision-guided robotics allows machines to perform predefined tasks and adapt to dynamic environments by interpreting visual data. The Fanuc iRVision system is part of this technological advancement. It offers a solution for enabling robots to perceive and interact with their surroundings intelligently and autonomously.

This study explores the implementation of the Fanuc iRVision system on the CRX-10iA collaborative robot to detect and pick up large block dice. The CRX-10iA robot is known for its safety features and user-friendly interface and it serves as an ideal platform for exploring vision-guided manipulation tasks. Detecting and handling dice presents a practical challenge that encapsulates key aspects of machine vision - object recognition, localization, and precise manipulation.

The primary objective of this project is to develop a system where the CRX-10iA robot can accurately identify the position and orientation of dice using the iRVision system and execute pick-and-place operations with high precision. This project has the unique constraint of needing the whole process to be implemented solely on the teaching pendant (TP) of the robot. There exists a Python API for this robot, but it will not be made use of, and I don't think you can access the robot mounted camera via Python. This involves creating and calibrating a vision system, programming the robot for coordinated motion, and optimizing the integration between the vision data and robotic control.

The insights gained from this project are intended to contribute to the broader understanding of vision-guided robotics and its potential applications in various educational contexts. This paper will cover what was done, what was learned, and how useful the system is. Most of the difficulties of this project involved the lack of documentation provided on usage of the vision system, and robot movement in general. Several roadblocks came up, and each of them will be discussed as they arise.

II. Materials and Methods

This section outlines the systematic approach taken to integrate the Fanuc iRVision system with the CRX-10iA collaborative robot for the purpose of detecting and picking up dice. The process encompasses the setup of the iRVision camera object, followed by creating an operational vision process, and finally implementing it all into a TP program. There are many different setups and methods to getting this all done, but for our specific use case there was one best path. This system uses a camera mounted on the robot, just above the end of arm tooling.

A. iRVision Object/Camera Setup

The initial phase involved the installation and configuration of the iRVision camera. In this subsection we will explore all the steps in the process of setting up a camera to use in the subsequent vision process of the program.

1) Camera Setup

The camera is mounted securely to the robot near the end of arm tooling to ensure an unobstructed view of the area where the dice would be located. Proper lighting conditions were established to minimize shadows and glare, which could adversely affect image processing. This process requires all lights to be on in the room, and if one of the two switches is turned off, the dice will not be detected. As seen in Figure 1, the first menu offers a few options:

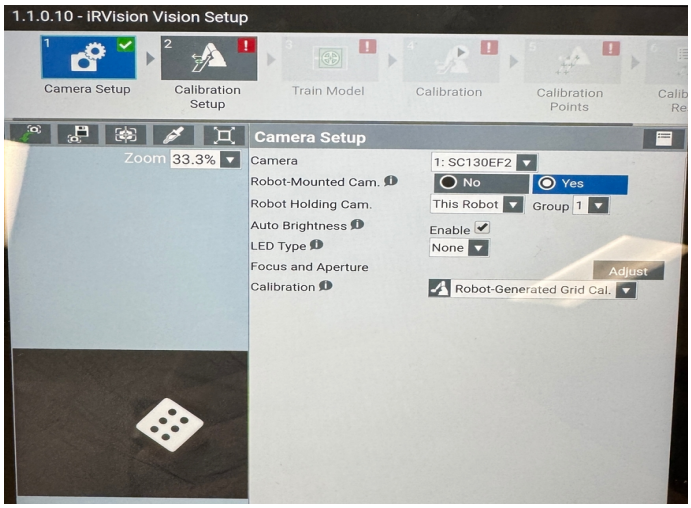

Figure 1: Camera Setup

a) Camera: Where a camera device that's plugged in will be detected and used.

b) Robot-Mounted Cam: The option to dictate whether or not the camera is mounted to the robot. This determines what the rest of the steps the robot will take in order to

calibrate. In a fixed camera set up, a grid of points is required to calibrate and undistort the camera. In our case we choose yes, and the robot will run its own special calibration.

c) Robot Holding Cam: Used to determine which robot is holding the camera. Can be configured to use one camera in a two robot setup. For our use case, Beaker is holding the camera so we choose this robot.

d) Auto Brightness: Self explanatory – auto brightness of the image (exposure, brightness related settings). It's best to leave this on as the program will optimize this. Otherwise, you can manually adjust these settings under the "Adjust" button.

e) LED Type: For use with an external LED camera to help provide extra light to the workspace. Not used in this setup.

f)Calibration: Choose the calibration type the robot will use. This is an important setting. We will be using "Robot-Generated Grid Calibration." This will take photos in a grid like pattern at many configurations to calibrate where the object is located. Other options are used in static/fixed camera setups.

2) Calibration

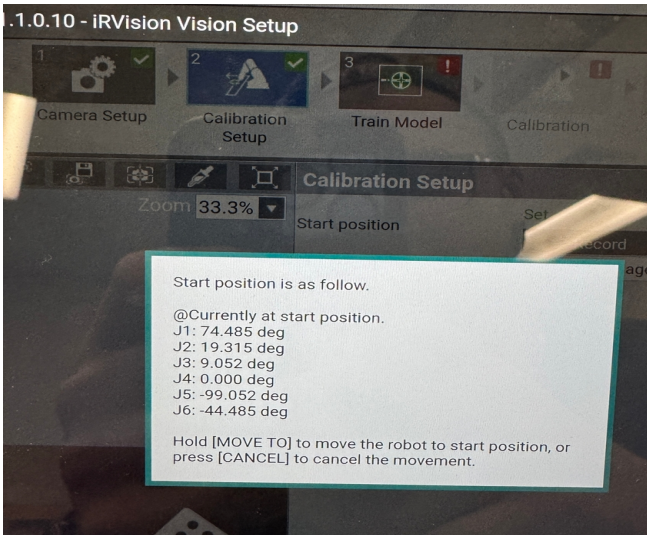

Figure 2: Setting Start Point for Calibration

As seen in Figure 2, this is a relatively simple screen. All that is needed is where the robot will be starting its calibration from. Move the robot to the desired position and hit the "Record" button (next to "Start Position"), and the robot will auto save the position in joint coordinates. **WARNING**: Make sure to set a start point that is *NOT TOO HIGH* off of the workspace. The robot will move to various positions during the calibration phase. If it attempts to move to a position outside its reach, you will get a limit error and have to *RESTART*. Calibration *WILL NOT* be able to finish, and you will need to pick a new start position. I had this happen to me twice.

3) Train Model

This model training process is very similar to the process used in the "Vision Process Setup" portion of the guide, and will in turn be covered there. Please refer to Figure 9. Essentially, you are just taking a quick snapshot, and giving the program information about what it needs to train on. For our case, taking a picture of the dice and masking out the pips is all that was needed. Edge detection is automatic.

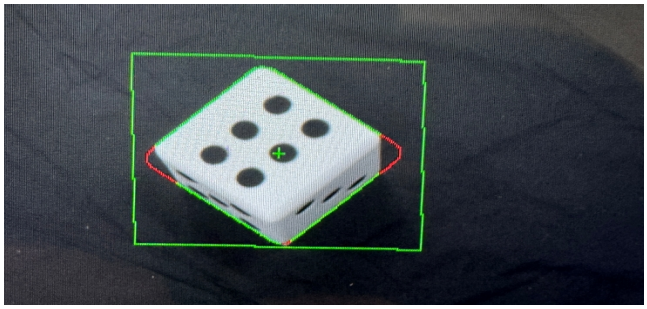

Figure 3: Picture of Dice During Calibration and After Train Model. See Figure 5 for view of Beaker

4) Calibration

This is the most painful part of the entire process. The calibration portion. For this section, the user must simply hold the green checkmark button as the robot calibrates itself. It will move to ~20 different positions orbiting the object in the workspace and take measurements at each position. See Figure 5. **WARNING**: Your finger *MUST NOT COME OFF THE BUTTON*. See Figure 4. If you do release it, there is a chance you will lose a lot of your calibration data, or even have to restart entirely. I had this happen. This process can take up to 30 minutes or longer.

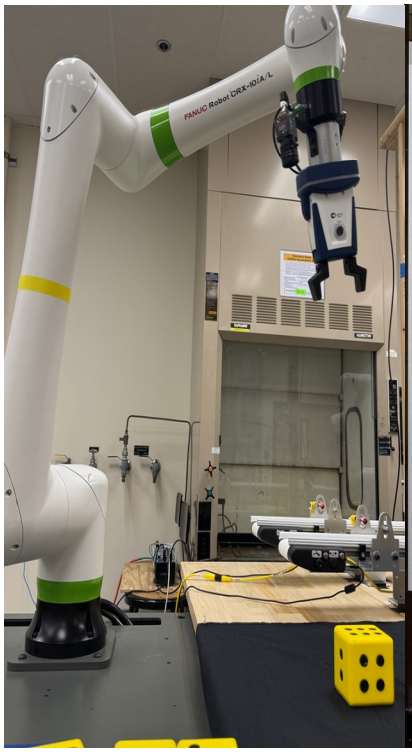

Figure 5: Beaker Orbiting Around Workspace Object During Calibration

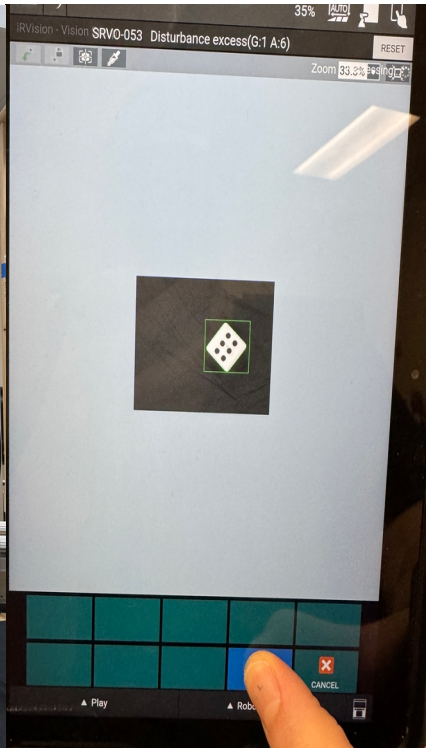

Figure 4: Holding the Green Check Mark Button for Calibration Process

5) Calibration Points and Review

Once the calibration process has successfully been completed, the program will show you a comprehensive list of calibration points. It prompts you to check for points with large errors that do not fit to the "grid" of points that was created. This can be seen on the left side of Figure 6. All of these

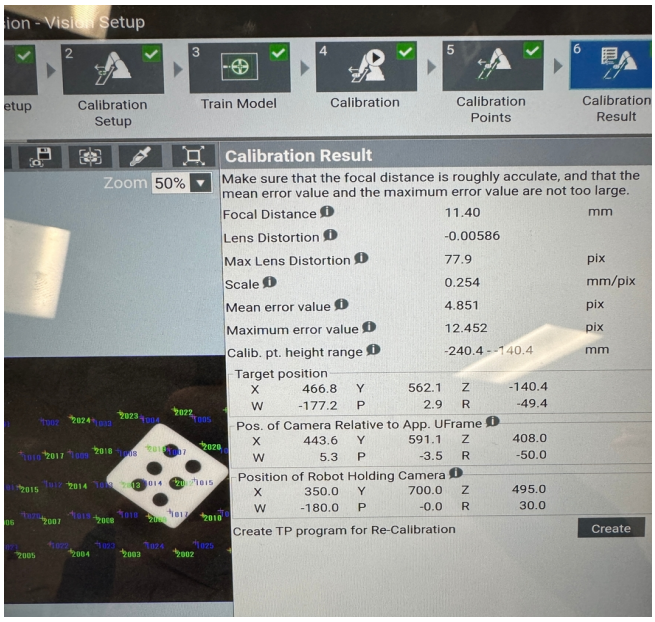

Figure 6: Calibration Results

settings seen in Figure 6 are auto generated and assigned. They are fine to be used as is, unless something seems very wrong. If that is your case, it is best to repeat all the steps for calibration. In our case all the settings were fine as is. As seen in Figure 6, there is also an option in the bottom right to *Create a TP program for Re-Calibration*. This is useful for recalibration without having to write a whole new camera module.

B. iRVision Vision Process Setup

Now that the somewhat tedious initial phase of configuring the iRVision camera is done, creating the Vision process task can be started. In this subsection we will explore all the steps in the process of setting up a iRVision process task to use in your program to find and detect dice.

1) 2-D Single-View Vision Process

There are several different types of vision processes that can be chosen for varying applications, but the one that fits our use case best is a 2-D Single-View process. This type when called will take one photo and store it in a Vision Register (VR). Make sure you have *FULLY* completed the camera calibration section before continuing, or the process cannot be set up. See Figure 7 for a view of this menu.

a) Camera

Drop down section where you can choose camera objects to use in the vision process system. For our case, we will use the camera object we just walked through making – I named the one I used "BURGESSCAMERA." Make sure the camera being used is calibrated!

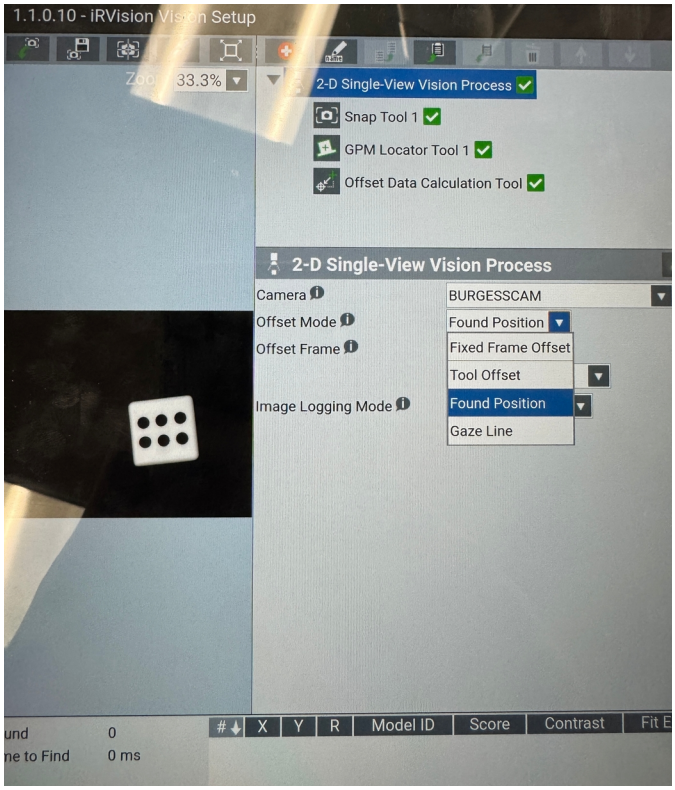

Figure 7: 2-D Single-View Vision Process Configuration

b) Offset Mode

This is one of the *MOST IMPORTANT SETTINGS.* This determines how the data for the dice positions will be calculated and saved. This is also where my project differs from how this has been implemented in previous times (from what I could find on old TP programs). The "Fixed Frame Offset" is what is recommended and used by the tutorial on vision processes via FANUC. This setting is best used for a fixed camera position (it will expect the same frame every time), like the one above the entire robot workspace. It determines how you will move to the dice too – the fixed frame makes use of vision moves and VOFFSET. I discovered that the "Found Position" option works significantly better in this use case. See Figure 7. It will take the calibration data, the height you set as the pieces in the workspace, and it will calculate the cartesian coordiantes of the dice in the offset frame specified. You can then use the coordinates found to make regular joint moves instead.

c) Offset Frame

This setting determines what frame the calculations will be done in. It is also very important to set this to the write frame – for our use case either 0 (World Frame) or 1 (Machine 1 Frame) will work fine.

2) Snap Tool

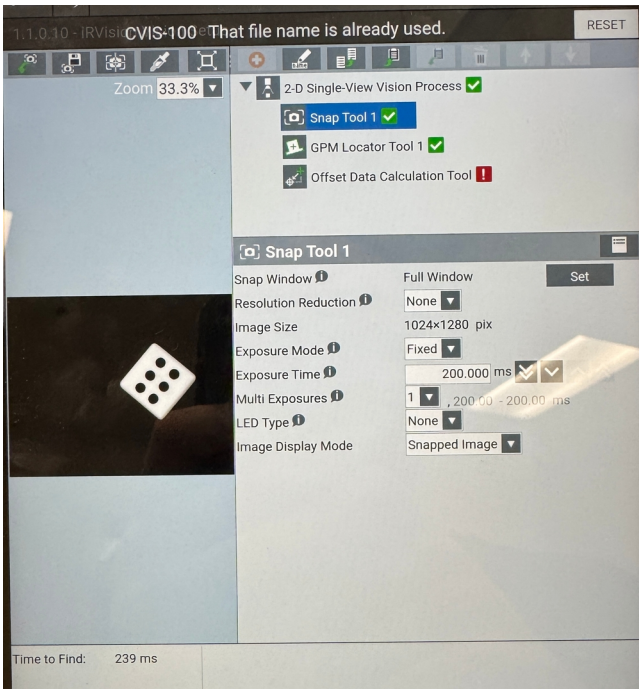

Figure 8: Snap Tool Settings

The snap tool is simply the settings for the photos that will be snapped upon process execution. Most of these settings are fine left as default, with the exposure time being the only setting that needs changed. Set it to whatever makes the dice nice and bright/visible so that the locator can easily find workspace pieces. 200 ms worked perfectly for this project. Multiple snap tools can be configured at once. See Figure 8.

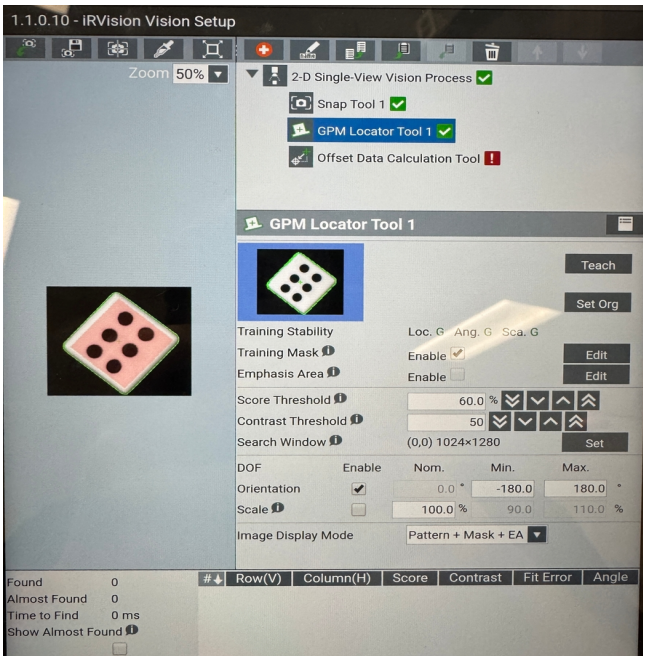

Figure 9: GPM Locator Tool Settings

3) GPM Locator Tool

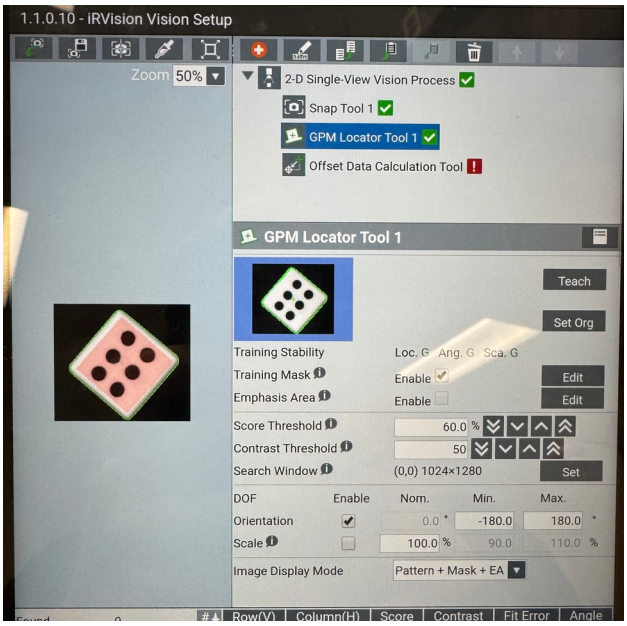

Figure 10: GPM Locator Tool

The GPM Locator Tool is one of the many workspace piece locating tools. Its job is to find the objects that match its training/teaching image. This can be seen in Figure 10. After capturing an image to use during teaching, there are several actions that can be performed:

a) Training Mask: This is any user defined area for the proprietary vision system to mask out of training data. For instance, the red area over all the pips in Figure 10 is a training mask. The canny edge detection will ignore any of that highlighted area. This is useful when pieces have anything that might get flagged by the detection software that needs to be ignored.

b) Score Threshold: This is the score required to pass the object detection filter. An object is scored against the teaching image and rated based on how much it matches. Anything that doesn't meet this requirement will not be flagged properly. In this use case, ~60% score threshold worked perfectly for detecting dice at any angle.

c) Image Display Mode: This simply decides what will be displayed to the user if an image is requested to be shown. This setting doesn't matter, but for debugging purposes, the option that shows all fields was used. This can be seen on the left side of Figure 10.

4) Offset Data Calculation Tool

This is a menu where you set the offset of either the tool from the camera, or the piece from the workspace depending on which offset mode was selected (as seen in Figure 7). For this use case the dice are 80 mm tall, so a Z value of 80 mm

was set. The workspace is the table in this case, and it wants the height of the piece in Z coordinates (mm).

C. Implementing it all into a Program

```
 1:  LBL[2]
 2:J  P[2] 100% FINE
 3:  LBL[1]
 4:  CALL IPL_SCHUNK_EGPC_OPEN(0,1,1,.5)
 5:L @P[1] 250mm/sec FINE
 6:  VISION RUN_FIND 'DBURGESS_VISION'
 7:  VISION GET_OFFSET 'DBURGESS_VISION' VR[1] JMP LBL[99]
 8:  PR[8]=VR[1].FOUND_POS[1]
 9:  R[50]=250
10:  R[51]=(-95)
11:  R[52]=180
12:  R[53]=0
13:  R[44]=PR[8,1]
14:  R[45]=PR[8,2]
15:  R[49]=PR[8,6]
16:  PR[9,1]=R[44]+15
17:  PR[9,2]=R[45]+25
18:  PR[9,3]=R[51]
19:  PR[9,4]=R[52]
20:  PR[9,5]=R[53]
21:  PR[9,6]=R[49]-10
22:L  PR[9] 250mm/sec FINE Wjnt
23:  PR[9,3]=R[51]-30
24:L  PR[9] 250mm/sec FINE Wjnt
25:  CALL IPL_SCHUNK_EGPC_CLOSE(0,1,1,.5)
26:J  P[2] 100% FINE
27:J  P[3] 100% FINE
28:  CALL IPL_SCHUNK_EGPC_OPEN(0,1,1,.5)
29:  JMP LBL[2]
30:  LBL[99]
31:  JMP LBL[1]
[End]
```

Figure 11: TP Program Code for FANUC Vision System

1) Calling and using the Vision Process

There are four lines of code that do all of the heavy lifting for this vision process implementation. These can be seen in Figure 11 and they are:

a) VISION RUN_FIND <FILE>: This method will use the vision command to find and run a specified vision process in the program – in this case it is the "DBURGESS_VISION" process we walked through creating earlier.

b) VISION GET_OFFSET <FILE> VR[1] JMP LBL[99]: Like the last method, this line will run the vision command with the parameters of *GET_OFFSET,* which in turn will get the offset we chose in offset mode during the vision process creation. It will attempt to store this offset in vision register one (VR[1]), and if unsuccessful, it will jump to label 99. Label 99 is placed at the end of the program and has a jump to label 1, which will restart the vision process. This acts as a pseudo for loop until a dice has been detected.

c) PR[8] = VR[1].FOUND_POS[1]: This line of code takes the found position from the vision register (X, Y, and R) and then saves it in a position register in the appropriate locations (1, 2, and 6).

*d) PR[9] 250 mm/sec FINE **Wjnt**:* This line of code has the robot move to a position register where all dice coordinates have been loaded into, but with the special parameter *Wjnt.* The wrist joint motion instruction specifies a path control operation that does not control the attitude of the wrist. If the configuration of a start point is different from that of a destination point, the robot cannot perform any Cartesian motion (Linear, circular, etc). In this case, a joint motion or adding wrist joint motion instructions will permit Cartesian motion. Essentially – the program was calling the instruction in a "flipped" state, and we had to lock the motion to be in an "un-flipped" state. See figure 12 and 13 for more information.

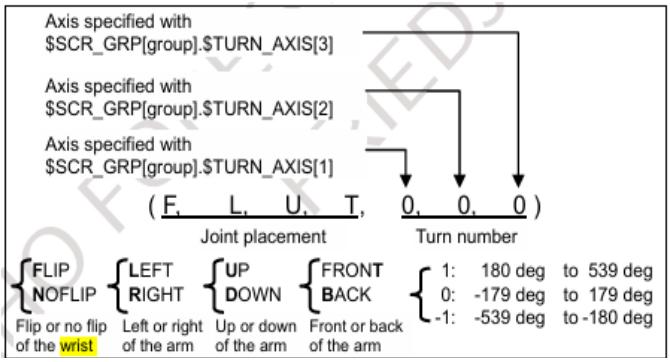

Figure 12: Robot Movement Configuration

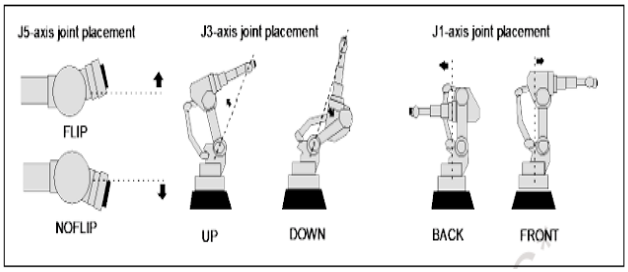

Figure 13: Joint Placement Diagram

2) Applying Coordinates and Grabbing Dice

The bulk of the rest of the program is reading individual values from calculated coordinates into registers (R[]) to be then put into a new position register (PR[]) for robot movement. During implementation it was discovered that if the robot tried to move directly to a register that has raw vision data, it would switch configurations from "N U T" to "N B D" and cause the wrist to be locked in a "flipped" state. See Figures 12 and 13 for clarity. To work around this, the loading of position registers was done manually before movement calls were made.

III. Results and Discussion

A. Results

The system was successfully implemented and can properly pick and place dice. It will run until stopped and wait above the

workspace until a dice is detected – then it will move to it, pick it up, and drop it back off at the center of its table. Beaker will then return to the workspace and await a new dice. This closely follows what an industry environment would look like – picking parts off of an assembly line of similar location, and then moving the part for further operations.

B. Challenges

1) Lack of Documentation

Initially, it was very difficult to figure out anything at all. There was very little information in the student version of the iRVision system manual. It had some very basic set up information, and steps to a process that ended up not being appropriate for this use case. It was only when Jacob provided me with the FANUC iRVision manual that I was able to really get the ball rolling.

2) Frames, Frames, Frames

In short, frames still confuse me. As a computer science student, we never truly get taught anything like this. It is briefly touched on in some math classes, but after this year, I really wish it was required that we take a circuits class and a kinematics course. I learned that the problems I was facing weren't necessarily frame issues, but to me they were roughly in the same ballpark. The issues I was having was not fundamentally understanding how the robot moves, and makes decisions when moving to positions (and what the whole "flipped" thing meant). After working on this project, and with lots of explanation from Jacob, I finally understand much more of what is going on.

C. Discussion and Future Work

1) Can this be useful?

This could be useful in a scenario where there is an industry-like process that needs to be implemented. For students that don't have time to master the system and deep dive – a simple pick and place program would not be too difficult. Anything more than a single "workspace" would start to get tricky rather quickly.

2) What was learned

How to implement a a vision system for picking and placing detected objects from one workspace to another. With some more time, I'm sure I could completely replicate the pick up the dice lab, and grab dice from anywhere on the table.

I also learned a lot of mechanical engineering concepts – forward and reverse chain kinematics, how user and world frames work (I would like to take a class that explores this), and how to work with FANUC software.

Overall I learned a lot of new ideas surrounding how the FANUC robots actually work. Figures 12 and 13 really helped me grasp how the configuration settings worked and what they were, as well as how joint placement works and the fundamental differences between movement styles.

ACKNOWLEDGMENT

A special thank you is in order for Jacob Friedberg's assistance with some of the more confusing and difficult to figure out portions of this project. During some of the learning and testing process, I had completely erased the user/world view frame settings for the robot, so movement was completely changed and "ruined" from what it was set at. Jacob assisted me in resetting the robot back to its original settings, and with figuring out how to deal with "flipped" coordinates/positions.

REFERENCES

[1] *FANUC user manuals – not entirely sure how to cite these or if I am even allowed to at all.*