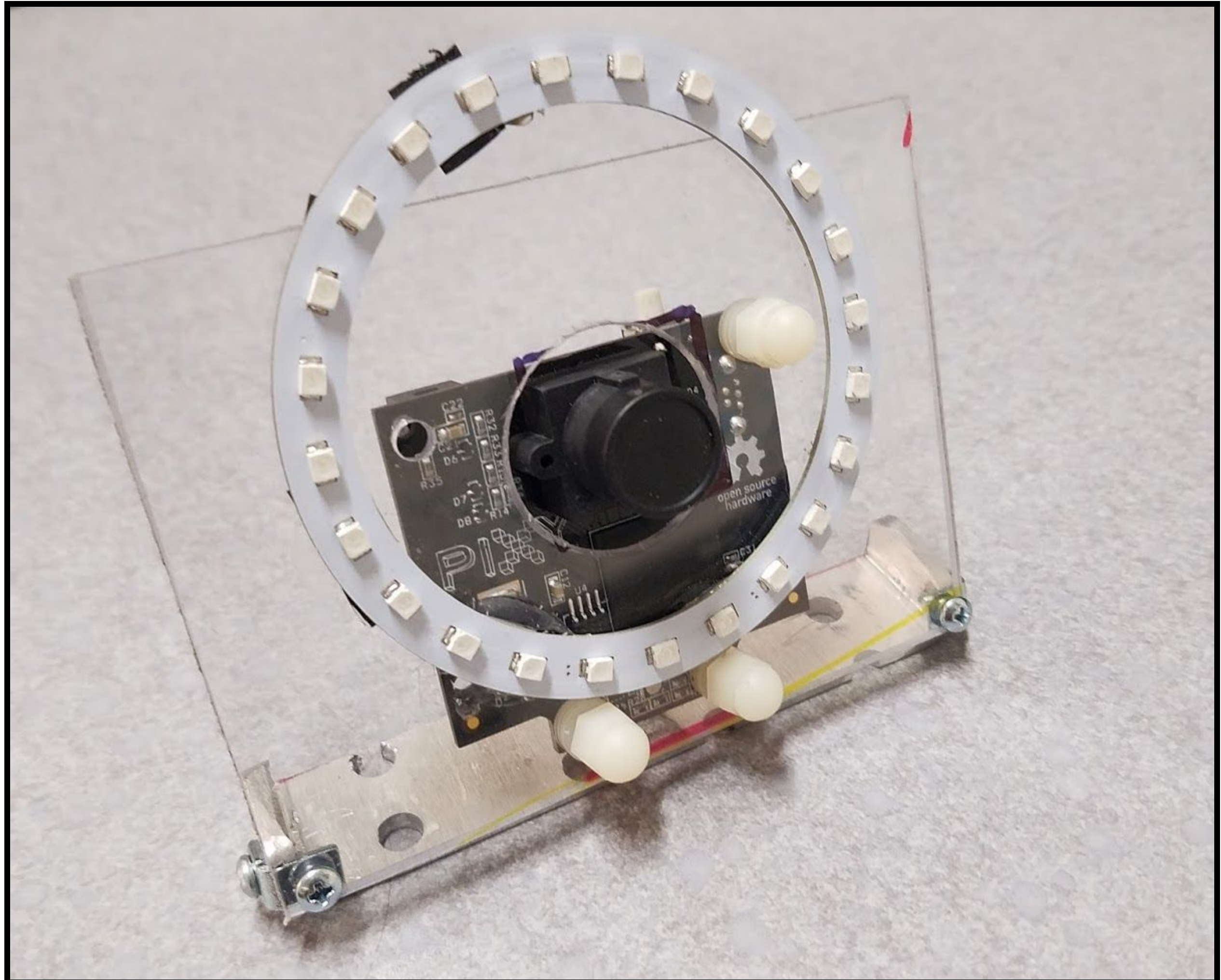
FRC Vision Targeting with CMU Pixy

Abby Lambert - Team 2035



Mechanisms

The Turret

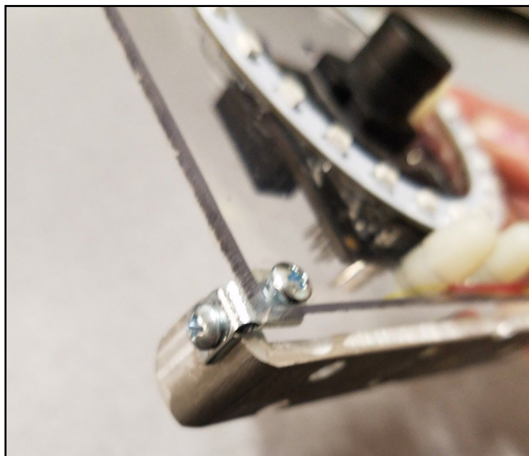
In order to create a shooting mechanism that will be able to accurately aim the fuel towards a target, we decided a shooter would be our best bet. Inspired by 254's design from 2016, we built a turret that would be manipulated by one motor. Inside sits a single flywheel that will propel the fuel in whichever direction the turret is facing.

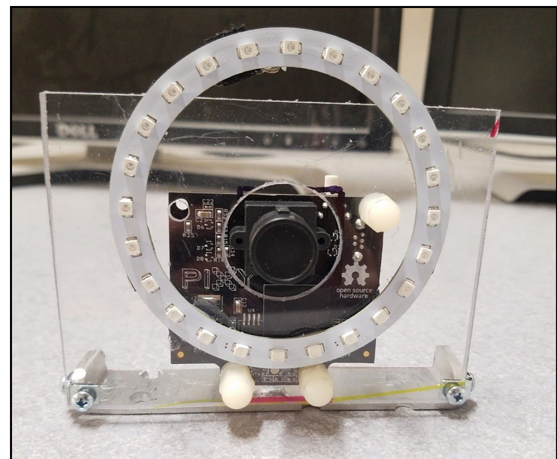
The Eyelid

The eyelid attaches to the center of the turret to determine the vertical angle in which the ball will be shot. This is independent from the vision code, as the angle is determined based on an ultrasonic distance sensor and a physics equation developed with the help of Mr. Nixon. One issue we had is the placement of the ultrasonic sensor, as it needed to be both parallel to the floor and above the rest of the robot to avoid interference.

The Mount

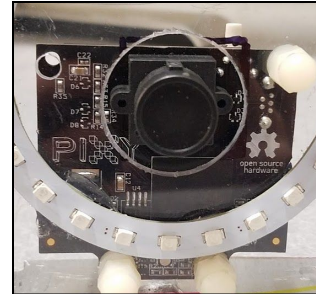
It is important that the Pixy camera is properly attached in order to maintain accurate vision information. We created a plastic mount that utilizes the L brackets that come with the Pixy in order to create a hinge. A hole was cut into the plastic for the lens and the green LED ring was glued around the outside.





Hardware

CMUcam5 Pixy



Pixy is a hue-based vision sensor developed by Carnegie Mellon University and Charmed Labs. It can be programmed either with PixyMon (the camera's GUI) or with a button on the camera itself.

roboRIO

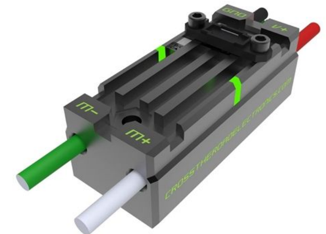
The roboRIO is the robot's central processor that carries out commands from the FRC Driver Station and user written code.

Raspberry Pi 3

Since the Pixy has no way of talking to the roboRIO, a middleman is needed to send vision information to the robot. The additional processing power is very helpful so that with each frame coming from the Pixy (at about 50 frames/sec), calculations can be made to determine the angle of the robot from the target without bogging down the processing power on the roboRIO.

Talon SRX

We chose to use two Talon SRXs (one for the turret mechanism and one for the eyelid mechanism) for two main reasons: the ability to use the CTRE Magnetic Encoder and the implemented closed-loop capabilities.



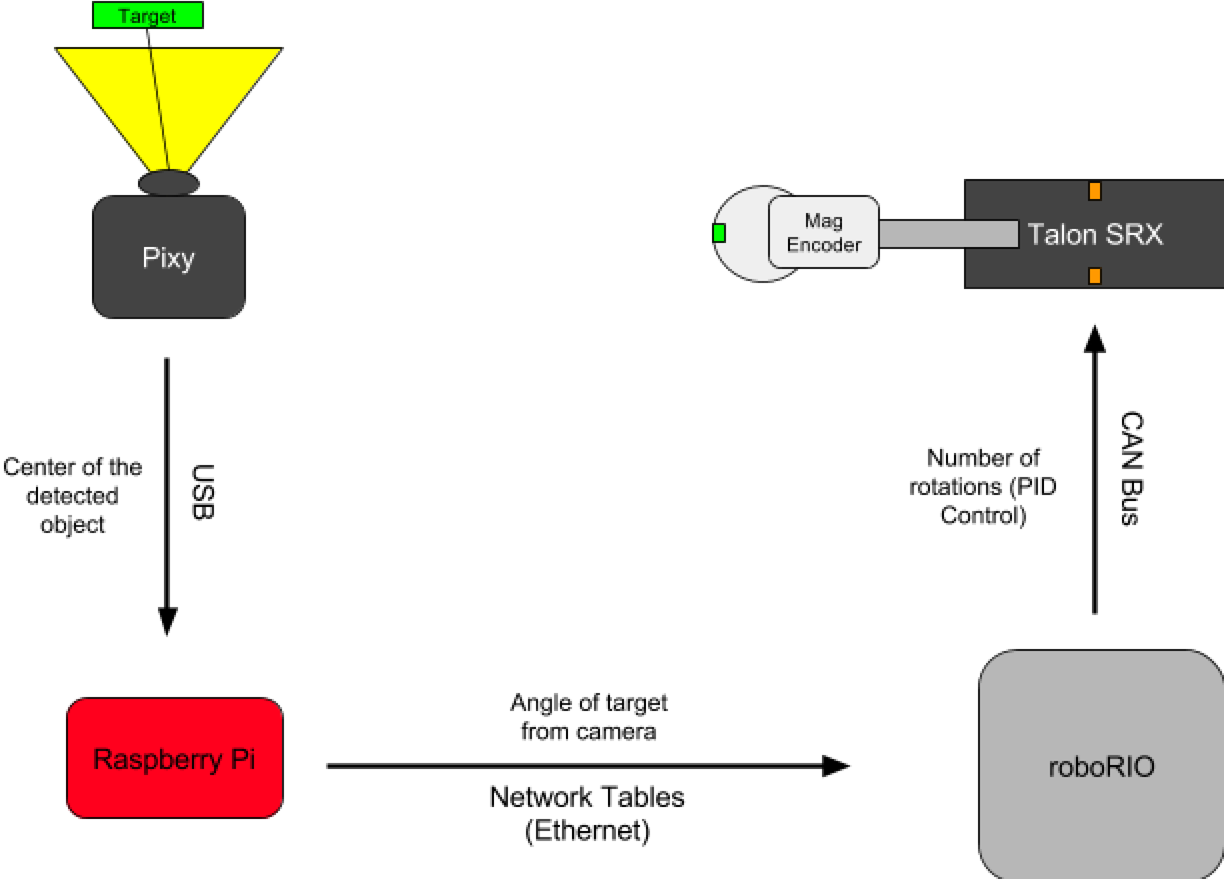
CTRE Magnetic Encoder

This encoder is used with the Talon SRX to determine the position of the motor shaft in order to mainly implement PID control in a closed-loop. The encoder is mounted on the end of a motor shaft with a magnet sitting inside. Its quadrature design is able to detect 4096 ticks per rotation and works in both a relative and absolute mode. It worked perfectly as our team has had trouble with optical encoders in the past.

Super Bright LEDs Halo Accent Light - Green, 80mm

As a crucial aspect of FRC vision processing, this green LED ring shines on the fields retroreflective tape in order to create a unique color signature that can be recognized by the Pixy.

Information Flow



Software

Pixy

While no software had to be written for the Pixy specifically, it did have to be calibrated and programmed to detect a certain hue. We had a few issues at first, including the focus of the camera, camera lag, and overexposure. We did this in PixyMon, the GUI meant for Pixy. The image quality in PixyMon was only 320x200 resolution, which was disappointing as it is advertized as 640x400 at the least (1280x800 at 25 frames/second). This made it difficult to determine whether or not the camera was focused, as the lens had to be manually adjusted. However, we focused the camera to the best of our ability. PixyMon also often had a significant lag (up to 7 seconds) if left running too long on the Raspberry Pi. This may have been simply due to the lack of processing power on the Pi, but had to be fixed each time by restarting the program or running it on a desktop computer. Lastly, the camera was very overexposed at first while trying to detect the green light. It was so bright that the Pixy detected the green light reflected from the tape as white! However, we discovered that the brightness on the pixy can be set, both in software and in PixyMon, and we found that 20 was the correct brightness to detect the tape.

Once it was all configured, we could teach Pixy the color green in order to detect the tape. By following the instructions listed on their website, we used the button on the camera to program the Pixy to detect the retroreflective tape. It worked very well, and we could see the object being detected in PixyMon. The difficulty with Pixy is that it would be impossible to adjust the color while in a match, or without dismantling the Raspberry Pi setup. Therefore, it must be thoroughly tested before a match.

Raspberry Pi

hello_pixy

All of the code we wrote and manipulated for the Pi was written in C and C++. Following the instructions for the Pixy online, we installed the libpixyusb-0.4 library (https://charmedlabs.github.io/pixy/) in order to run the example code written for the camera, called "hello_pixy." This is a program that establishes a USB connection with the Raspberry Pi and then receives any objects the Pixy sees. We modified this code so that only the x value is received from the package and is then transformed to a horizontal angle from the camera's position. For this calculation we used a linear model; we attempted the pinhole camera model suggested by 254, but the equations were not working in our favor. We hoped it would be accurate enough for our purposes.

NetworkTables Client
Originally, we piped this angle from the hello_pixy program to a separate NetworkTables client. This client used the C++ version of NetworkTables, which had to be compiled for the Raspberry Pi. After having difficulties building the library on a desktop computer using both gradle and g++, Team 5104 pointed us to the prebuilt maven repository from FIRST:

http://first.wpi.edu/FRC/roborio/maven/release/edu/wpi/first/wpilib/networktables/cpp/NetworkTables/3.1.6/

This allowed us to easily compile the library and use it to build our client file with g++ on the Pi. We created the client which received the angles from hello_pixy and published them to the NetworkTable accordingly.

The Merge!
While having one program pipe to another was great for testing purposes, the biggest issue was lag. While testing, we noticed a lag of up to 2 seconds, which is complete nonsense for a 15 second autonomous match. One way to easily fix this lag was to combine both programs, which created a new host of issues. The code was much easier to write, because instead of outputting and inputting at the right times the angle could be sent as soon as it's calculated. However, hello_pixy was meant to be compiled using CMake, and the client was being built with g++. Therefore, we had to track down all of the libraries that hello_pixy depended on, copy them into our includes and lib files for NetworkTables, and link all of them while building the program in g++. It took lots of trial and error, but when it came together, the program ran with very little lag. It has just about enough speed to run as the robot moves. However, we did not have time to implement corrective software, such as latency correction.

roboRIO

The roboRIO runs all of the control code for our robot, and our team uses Java. The main framework for how the vision processing runs is that there are two modes toggled by a button: "manual" and "vision." In manual mode, the driver has the ability to move the turret manually, with a button to target and shoot. In vision mode, the turret is constantly searching for the target. The turret needs to utilize the encoder's capabilities in order to stop before the mechanical stops and wherever the target happens to be. The Talon SRX libraries implement PID control, which we used to control where the turret stops.

When the program receives an angle, it has to do multiple unit analysis calculations in order to convert that angle to a certain number of rotations for the motor to handle. We needed to determine how many degrees the turret would spin in one rotation. We did this by first calculating how many degrees were in one gear tick by measuring how many gear ticks were in 180 degrees (to find degrees per gear tick) and then multiplying that by 15 (the amount of teeth on the motor shaft). This can be used as a factor in the rest of the calculations.

Here is the method for calculating the correct setpoint for the PID controller:

```
public double calculateSetPoint() {
      //turn degrees into fraction of rotation
      double angle = Double.parseDouble(receiveAngle());
      double rotations = angle/(RobotMap.DEGREES_PER_ROTATION);
      double current = ((turret.getEncPosition())/4096); //current encoder position
in rotations (not ticks)
      point = rotations + current; //adds the current position (in rotations) to
"zero" the shooter
      //Does not allow the shooter to turn more than 90 degrees in either direction.
      //Check to keep motor from overturning and harming mechanics without a limit
switch
      if(point > rightLimit) {
            point = rightLimit;
            return point;
      }
      else if(point < leftLimit) {
            point = leftLimit;
            return point;
      }
      return point;
}
```

Team 2035 would like to thank Team 5104, especially Zachary Staples and Henry Loh, for setting us in the right direction when we started this project and allowing us to hop on board with their plan. We would also like to thank Team 254 for providing so many resources on vision targeting algorithms for the public.