Sampling Vision Software

A crucial component of our autonomous programs is our custom vision pipeline to determine the sampling field order at the beginning of each autonomous route. This is done by utilizing the phone’s rear camera for observing the sampling field. The frames from the phone camera are then analyzed by the aforementioned vision pipeline, which uses OpenCV as a foundation.

The vision pipeline consists of many different steps to analyze each frame from the camera. The first step resizes each frame to a standardized resolution, and then draws points at three predefined locations on the frame. These locations correspond to the approximate expected locations of the three elements. Circles are then drawn using the previously drawn points as the center point of each circle, resulting in three separate circles. These circles are used as defined regions of interest in the image. Everything outside of these regions is “masked over” and entirely ignored by the rest of the vision pipeline. This process serves two purposes. The first is that the consistency of the system is drastically increased, since other objects that could potentially trigger false positives are never analyzed by the rest of the pipeline. The second is that the performance of the pipeline is increased (because fewer pixels of the image are being analyzed in every loop of the pipeline), which is very valuable, because the available system resources are severely reduced in this application.

After the image has been resized and masked, the pipeline begins color filtering to find regions that fall within a specified HSV range (HSV is used because it’s a color space that is more resistant to changes in lighting. This is important for our applications, because lighting conditions can and do change drastically between event venues). The HSV range is intentionally wider than necessary, which helps reduce effects of lighting even further. After the image is filtered for color, “contours” are found within the image. Contours represent any line segment (or collection of line segments) on the edges of detected regions. Once the contours have been found, we filter the contours based on many different criteria. These criteria include minimum and maximum area, width, height, perimeter, and vertices. By filtering the contours, we can be reasonably confident that we have successfully filtered out any source of noise that made it past the other filtering steps.

Once the contours are filtered, a rectangle is drawn around the remaining contours, and the center of that rectangle corresponds to the center of the element. This point is then used to determine whether the cube is located in the left, right, or center sampling field position.

Driver Enhancements

Our driver controlled software contains a number of automated enhancements to make driving easier and more consistent for the robot drivers. These include automated arm setpoints (defined as one dimensional angles in this application) and automated extension setpoints (defined as one dimensional translation in this application). These enhancements allow the drivers to consistently return to the same angle and extension height for scoring elements every time without any effort on their part, which vastly improves driver performance and decreases driver error.

However, we've learned in past seasons that while automation is very helpful for increasing performance, it can also be harmful. If the automation reduces a driver's freedom of control, problems tend to arise when the robot is not behaving nominally. Situations where this is applicable include hardware failures such as belts skipping or inconsistent encoder zeroing. If this occurs, the automation systems can be severely impacted, and can prevent scoring from occurring at all during the course of a match. To prevent this, our automation systems are driver adjustable. If our arm extension is drifting in one direction during a match, the driver can press a button to reduce or increase the extension setpoint. This setpoint reduction/increase is then remembered for the rest of the match, allowing the drivers to focus on scoring.

Hardware Superstructure

Each subsystem on our robot has an associated software class, separated from each other. This allows us to rapidly debug specific subsystems, easily make modifications to existing subsystems in software as the hardware changes, and add new subsystems as they're added to the robot. These subsystems are finally packaged together into a robot hardware wrapper, which instantiates each subsystem as an object, allowing us to give each subsystem control commands. The updating of these subsystems is handled by a method in the robot wrapper object, which passes the relevant commands to each individual subsystem (which allows the programmer to simply call a single update method and send all of the relevant data through that method, rather than individually updating each subsystem manually). This results in a more robust framework for electronics integration and provides more features and versatility to the programmer.

Autonomous Navigation

Our autonomous navigation software has many separate systems within it. The first contains algorithms for calculating one-dimensional “motion profiles” (defined as a graph representing the robot’s position, velocity, acceleration, and jerk over time) with given constraints. These constraints include maximum velocity, acceleration, and jerk. This allows us to customize specific behaviors of the drive controller. However, a motion profile alone isn’t enough for a two dimensional drive controller. Motion profiles work perfectly fine in one dimensional applications (such as the arm rotation and extension on our robot), but a drive train moves in two translational axes, and a rotational axis. As such, a path must be created with a series of points. These paths generally have two or more points associated with them. These points define (at least) the starting (x, y, rotation) point for the path, and the final (x, y, rotation) point. From here, more points are interpolated based on predefined drive constraints, and then a motion profile is fit to this path. Once this is done, the profile is followed by translating the position, velocity, and acceleration values from the motion profile to motor powers, and then the error between the profile values and the actual values (measured through the motor encoders for translational position and the BNO055 IMU in the REV Expansion Hub for heading) are minimized via three separate PID controllers. The first minimizes velocity error, the second minimizes translational position error, and the third minimizes rotational error. This system results in an exceptionally easy to use interface for the programmer, allowing them to simply define a set of points and constraints for a given path, and the algorithms do the rest.

Sensors Used

We use a variety of sensors on our robot. The most critical are the motor encoders, which are utilized on 7 out of 8 motors on the robot. These provide position data for each motor. Additionally, we use a potentiometer geared 1:1 to our arm rotation for the most accurate angle data possible. This solution is also more resistant to non-nominal behavior.

Additionally, we use an internal IMU to determine heading in autonomous. This is also cross-checked with another heading value, which is computed via kinematic equations that utilize measured data about the robot.

Finally, the phone camera is an integral part of our autonomous strategy, as it allows us to determine the sampling field order while hanging from the lander (which provides more consistent robot placement).