**TEAM 1389 STRATUS Software Overview**

Arye Mindell, Rafael Metz

**Features**

- **Reuseable code repository** - many complex tasks on the robot are handled by *Ohm*, an extensive library of reusable code we wrote as a base for all future robot projects
- **Computer vision** - the robot uses an image processing routine to identify the airship's lift peg, align with it, and place gears on it accurately
- **Autonomous path following** - during the autonomous period, the robot can drive in curved paths to reach its destination in the most efficient way possible
- **Web dashboard** - during matches, a sleek dashboard interface offers a comprehensive readout of the robot's systems, and notifies the drivers of any robot failures as they come up
- **Simulation** - a custom physics simulator enables us to test any code written using our library, Ohm, without a working robot available

**Team 1389's reusable robot code library: Ohm**

Ohm is a java library for FRC robots. It is intended to augment the features of WPILib (the standard code library for FRC) with tools that we found useful in our programs. We designed the library both as a way of maintaining our accumulated programming knowledge from year to year, and as a way of sharing that knowledge with other teams. Here are some of our favorite feature from Ohm:

**Advanced motion control:**

Complex algorithms such as PID control, motion profiling, and path following make a capable robot into an unstoppable one. Despite the invaluable advantage they offer, many teams fail to use them because of how difficult they are to implement. Ohm offers simplified interfaces that handle all the advanced math, leaving the user to simply plug in the unique details of their robot and application.

**Debugging/Data logging**

Every object and controller in Ohm can be asked to display all relevant information about itself for debugging. This ability hastens the debugging process and simplifies the robot code. For example, we were able to diagnose a problem with the current draw of the drivetrain in mere minutes by adding three words to the code, and watching the debugging output. Furthermore, any data that gets displayed to the dashboard can easily be saved to a log for later examination.

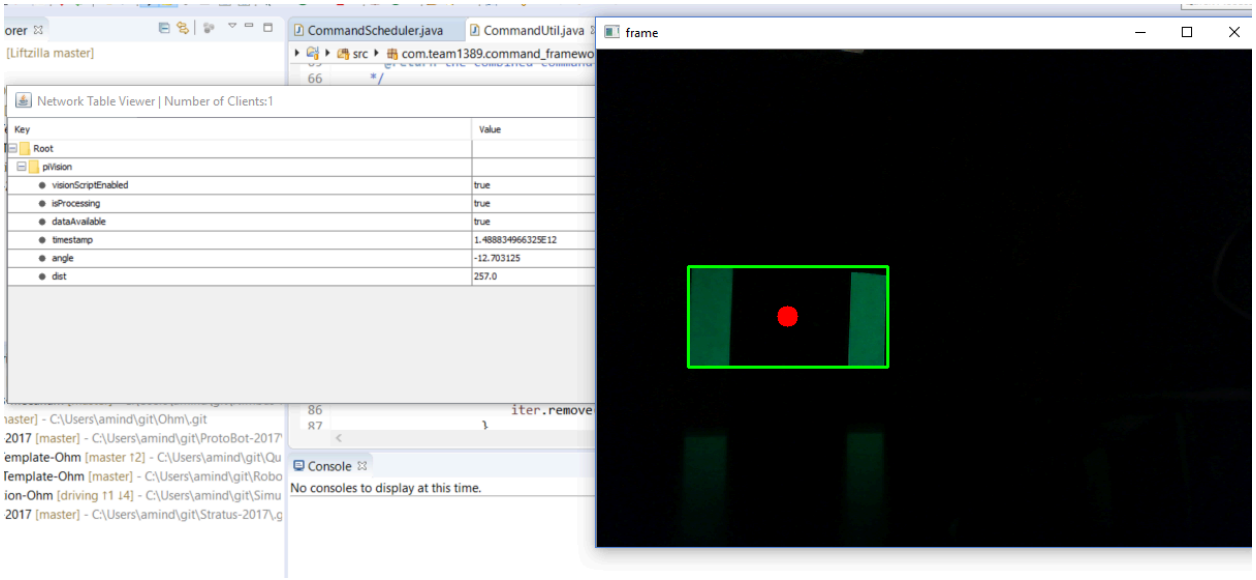
**Computer Vision**

Computer Vision (CV) has become critical component of high level FRC success. CV in a general sense is the use of sensors to understand your environment.

This year, our game strategy included placing as many gears as possible, so we chose to use CV to accurately and quickly align our robot with the gear peg. While CV provides unparalleled accuracy in robot operations, a working CV system must be well-planned, efficient, and reliable.

The first step of computer vision is acquiring some information about the robot's environment. In our case, a camera onboard the robot constantly captures images of the field as the robot drives, searching for the target with unique visual properties that surrounds the gear peg. We developed a processing algorithm which goes over each image pixel by pixel searching for patterns. The target produces a unique pattern when processed. Once this algorithm has identified the target, we do some trigonometry calculations to figure out the peg's position relative to the robot. Finally, we can calculate the direction the robot needs to move to get aligned.

This entire process must occur nearly instantly at the press of a button, so we optimized the entire vision system for speed. Rather than running on the main robot computer, the processing algorithm (where each pixel is analyzed) runs on a separate computer, known as a coprocessor, that can devote all of its power to vision calculations.



**Autonomous Path Following**

Having a successful autonomous is particularly crucial this year as the amount of points awarded for autonomous functionality are high and getting multiple rotors running is essential for success.

This year we are using a path following system that generates the most optimal path given points pre programmed into the robot. We determined the points using our simulator to find the most efficient points. We use different sets of points depending on which autonomous we would like to use. We have pre programmed eight different autonomous modes, that each utilize the path following system. The path following system generates the most efficient path possible, using, velocity, acceleration, jerk, and angle of the robot.

In autonomous every second is critical, so having the most efficient path is vital. But to optimize the speed even more, we have chosen to have the paths read off of a file rather than generated each time on the robot. This allows us to have no latency, while still having the robot follow an extremely complex path.

The utilization of the path following system has allowed us to perfect other areas of autonomous, while still having the most efficient path.

**Web Dashboard**

As each year's robots have become more and more complex, well-designed dashboards have become increasingly important to give drivers everything they need for peak performance. Team 1389's Web Dashboard is designed to provide the driver with information on all the most vital systems, as well as the status of the robot, all contained in a sleek and minimalistic shell. Every element of the UI overlays a video stream, which can switch between two cameras on the robot at the switch of a button. The Web Dashboard was designed using HTML, CSS, and Javascript, to allow for greater freedom in design.

**System Information**

The Web Dashboard keeps clutter to a minimum when it comes to systems, using transparent backgrounds and well-spaced images to allow the driver to focus on the competition. There are four widgets, representing the speed of the robot, the state of the Gear Intake, the angle of the robot, and the drive mode of the robot. Each of these systems are represented by clean, non-distracting images. All four systems are

represented together in the bottom of the screen, allowing the driver to view them with minimal eye movement from the video stream.

### Robot Status

The Web Dashboard allows drivers to monitor the status of the robot, allowing for lightning fast decision making in critical situations. The amount of time left in a match can be integral as a factor for decisions, and as such is placed at the top of the screen for easy access. There are five status lights placed in the top-left corner, representing connection, drive-train, climber, ball-intake, and gear-intake. Each of these lights turn red if the system fails, and green if the system is working. In addition, a bar graph represents the current draw of each drive-train motor, giving drivers accurate knowledge on the state of each motor.

## Simulation

Nothing is worse than having code ready to test but nothing to test it on. Especially in robotics, the process of testing code even when the robot is present can take a while; each time the code is changed it must be built again. The simulator is our answer to this problem. Most obviously, it allows us to practice driving the robot around the field. The robot simulation is extremely detailed. It is not merely an x and a y controlled with a joystick with an arbitrary acceleration. Each motor is simulated carefully, with an input voltage and output change in x. We then include factors like the distance between the wheels and the axles to accurately simulate the position of the robot. As we used an octo mecanum drive this year, our simulator could also switch between different modes of simulation: tank and mecanum. Finally, we also built a robust collision system that stopped the robot when it ran into a field object like the airship. The encoder would still increase but the actual position of the robot on the screen would not move. We also simulated gear pickups and gear dropoffs using a similar system. All of this gave us an easy and robust way to test both autonomous routes and teleop cycles. We even added support for joysticks and keyboard controls. We will also be able to reuse it in the future for further competitions.

However, driving is not the only thing our simulator was good for. Theoretically, anything can be simulated with our package. Through the simulator, we tested many features of Ohm, like watchers and streams. It was like having another robot to test with, except as many people could work on it as once as we wanted (with the nice fork ability of git). One interesting application of the simulator was a state estimator we built in a separate project. We simulated a network table server in the simulator with a local ip address, and connected to it with a client from the other project. Then, we could

practice displaying the robot on the driver station from only x, y, and theta values from where the robot *thought* it was. The chief reason we did this was to practice error correcting this display: the robot's position estimation values get off when the encoders get off, which happens when the robot's wheels slip. This can include hitting a wall, another robot, or just inevitable skidding. To combat this, we decided to reset the robot position client side whenever we dropped off a gear, as the angle stays relatively error free and the gear drop off point can be completely determined from the robot's angle. The driver can also click anywhere on the screen to tell the client side simulator that right now, the robot is *there*.