# Introduction

This is a brief summary of how to control and use the Raspberry Pi Vision Processor to assist in various driving modes for the FRC Deep Space 2019 Challenge. Since the Vision Server was adopted from FRC Team 2877 (Ligerbots) it makes sense to see how they interfaced with the Vision Server running on the Raspberry Pi coprocessor. Some minor changes in Network Table labels were made for operation clarification and are noted below.

Refer to Team 2877 source on GitHub (<https://github.com/ligerbots/DeepSpaceRobot2019>) for the details of the software excerpts below.

Also refer to the README.md on the Team 293 Vision Server implementation on GitHub (<https://github.com/Team293/video-2019>) for our Vision Server interface modification details.

# Operator Interface

File: OI.java

Key Imports: import org.ligerbots.robot.Commands.DriveToVisionTarget;

Key Methods:

JoystickButton xboxB = new JoystickButton(xbox, 2);

xboxB.whenPressed(new DriveToVisionTarget());

Description:

A joystick button is declared to invoke the DriveToVisionTarget() command further detailed in the following section.

# Commands

Description:

1. In the DriveToVisionTarget.java file there is an import to include the SmartDashboard class. The SmartDashboard is used to access the Network Table (NT) elements associated with control of the Vision Server mode.

import edu.wpi.first.wpilibj.smartdashboard.SmartDashboard;

1. The DriveToVisionTarget “initialize()” method calls SmartDashboard.putString() to put a string value of “rrtarget” to the NT entry “vision/active\_mode” which is effectively under “SmartDashboard/vision/active\_mode”. The string value “rrtarget” puts the Vision Server into the retroreflective target acquisition mode, which in turn outputs target distance and angle values once the target is acquired. The target must be within the maximum range of some 80-90 inches in order to acquire the target pattern.

// Called just before this Command runs the first time

@Override

protected void initialize() {

System.out.println("STARTED DRIVETOVISIONTARGET COMMAND");

Robot.driveTrain.setLEDRing(true);

SmartDashboard.putString("vision/active\_mode", "rrtarget");

}

1. The DriveToVisionTarget “execute()” method calls SmartDashboard.getNumberArray() looks at the table entry “vision/target\_info” to read the Vision Server target information. The array target\_info contains the following information.

target\_info[0] = timestamp of image in seconds.

target\_info[1] = success (1.0) OR failure (0.0)

target\_info[2] = mode (1='driver\_pov', 2='driver\_alt\_pov', 3='rrtarget')

target\_info[3] = distance to target (inches)

target\_info[4] = angle1 to target (radians) -- angle displacement of robot to target

target\_info[5] = angle2 of target (radians) -- angle displacement of target to robot

While the Acquisition Success/Failure Flag (target\_info[1]) and the Active Mode (target\_info[2]) value for “rrtarget” (3) should be verified before reference other target values, the key target information is the distance and the two robot/target angles used in the subsequent calculations for drive control. The highlighted Team 2877 drive control Robot.driveTrain.allDrive() illustrates how they use these target heading values.

// Called repeatedly when this Command is scheduled to run

@Override

protected void execute() {

visionInfo = SmartDashboard.getNumberArray("vision/target\_info", empty); //refetch value

distance = visionInfo[3]; //reset distance and angle

angle = visionInfo[4] \* (180/Math.PI);

deltaAngle = angle + (visionInfo[5] \* (180/Math.PI));

Robot.driveTrain.allDrive(Robot.driveTrain.driveSpeedCalc(distance), Robot.driveTrain.turnSpeedCalc(deltaAngle), Robot.driveTrain.strafeSpeedCalc(angle));

}

1. The DriveToVisionTarget “isFinished() method is supposed to check if the distance to the target is within an acceptable margin to stop the robot. While the highlighted comment suggests this, it appears this calculation is not yet complete in this version of the software.

// Make this return true when this Command no longer needs to run execute()

@Override

protected boolean isFinished() {

return /\*distance <= 40 && Math.abs(angle) <= 1\*/false; //This isn't totally done...

}

1. Finally, the DriveToVisionTarget “end()” method closes out the command action by reverting the Vision Server back to driver POV mode of live streaming video. A call to SmartDashboard.putString()puts the string value of “driver\_front” to the NT entry “vision/active\_mode” to switch the processing to live stream. However, our version of the Vision Server software was modified to use the term “driver\_pov” (string) instead of “driver\_front”. This in part was linked to some camera naming changes and clarification of the function. A provision has been made for an optional additional camera that would provide a driver alternate POV (hence, “driver\_alt\_pov” mode). Activation of this alternate live streaming mode requires the second camera and some minor Vision Server software modifications to enable the camera usage.

File: DriveToVisionTarget.java

// Called once after isFinished returns true

@Override

protected void end() {

System.out.println("COMMAND ENDED");

//Robot.driveTrain.setLEDRing(false);

SmartDashboard.putString("vision/active\_mode", "driver\_front");

}

# Drive Control

The Subsystem DriveTrain.java contains Team 2877’s design for their drive control. The details are spread over the file so it should be viewed to see how driveSpeedCalc(distance), turnSpeedCalc(deltaAngle), and strafeSpeedCalc(angle) make use of the vision target information in their context. Clearly the Team 293 drivetrain is different by description and would require modifications to make use of the vision target information.