Simulated Robot

For FRC 2017 “Steamworks” Game

Team 159

Table of Contents

Motivation

1 Overview

As was the case for the previous two seasons a Gazebo robot simulation project was developed in parallel with the team's competition robot code for this year's "Steamworks" game. Using the simulation tools supported by First has the following benefits:

- The ability to test the performance of a model as it interacts with a simulated FRC field without needing to construct a working robot and physical field elements
- The ability to test and develop software used to control a robot model prior to the availability of a working physical prototype
- A mechanism to continue testing software strategies between bag day and the days of competition (or while the physical robot is being worked on or assembled)
- A way to explore new sensor technologies (e.g. image recognition) and subsystem ideas (e.g. special drive-trains or manipulators) during the off-season

Tools and Resources

1 Components

The current simulation environment supported by First includes the following principle components:

1. Solidworks "sdf" exporter
    - An application "plugin" that is used to produce a set of files from a Solidworks model that can be exported to the Gazebo simulator
2. Eclipse linux_simulation build
    - A special Eclipse build configuration that generates a "FRCUserProgram" binary from robot software source code that runs in the Linux Gazebo simulation environment
3. Gazebo simulator
    - Developed for the DARPA supported "DRC" challenges. Provides a graphical simulation environment to test the interaction of the mechanical model created with the Solidworks exporter and the Software model created from the Eclipse linux_simulation build
4. GRIP vision processing subsystem
    - GRIP was introduced by WPI last year as a Java based application to process camera images and isolate vision targets in the FRC field
    - It was designed as a replacement for NI's proprietary "NIVision" subsystem which was neither free (outside of First teams use) nor open sourced

- Because the original Java based version was found to be too slow when running on the RoboRIO arm processor (and also required installing a significant number of third party libraries) this year the recommended deployment method is to use a special tool in the GRIP user interface to generate native OpenCV source code (Java, C++ or Python) and then port that code to the target platform (e.g. by direct compilation into the robot program)
- In order to get this new approach working with the gazebo simulator (running in a Linux environment) it was necessary to first build several components from source (cscore, CameraServer etc.) that were supported by pre-compiled arm libraries when deployed on the Roborio but were not available for simulation. (A separate write-up about going about this is available on the MentorRepository github site:FRCSimulation2017.odt)

5. SmartDashboard
   - In addition to the Default dashboard (a pre-compiled NI application) FIRST/WPI provides two alternative Java based dashboards: “SmartDashboard (2.0)” and “SFX (3.0)”. SFX is based on JaxaFX and has a better looking UI but unfortunately we couldn’t manage to get it to display any of the camera windows so we opted to use the older 2.0 dashboard

Solidworks

1 Models

1. Robot

   The team’s Solidworks robot model was completed fairly early this year so a useful simulation prototype was available to test autonomous strategies etc. using software models well before the end of the build season.

2. FRC Field

   Unlike in previous years, a simplified Gazebo model for the competition field wasn’t provided by First this season so it was instead necessary to generate a facsimile of the field by exporting a Solidworks CAD design that was made available as a free download. However, owing to the large size and complexity of the field model (even after removing several components and “defeaturing” it in Solidworks) if was found to be necessary to also disable the collision property of most of the elements that were not directly required to test the behavior of the robot model’s interaction with the field. The collision components retained included the target “spikes” and gear placement panels

3. Gear

   A Solidworks assembly for the Gear was available as part of the Field download mentioned above and was exported as a separate model using the Solidworks Gazebo sdf exporter

   - The compression properties were modified to reduce the hardness of the surface so that the collision with the “harpoon” spring was more accurately modeled

4. Balls (Fuel)

   A simple 5” diameter yellow sphere was generated as a Solidworks assembly and exported using the Gazebo exporter plugin

5. Vision Targets

   A Solidworks sketch of the Gear placement target pattern was created (two 2x5” strips separated by 10.5” outside edges)

   - The dual tape models were place on the field panels directly behind the spikes and were suspended in place by setting “gravity” to zero in the Gazebo model
   - The color was changed to best match the color of the reflective tape illuminated by a green light ring
   - The tape was also given a certain amount of “emission” property and the ambient lighting of the Gazebo scene was reduced to better emulate the conditions seen by the targeting camera

2 Competition Robot

The final robot design included:

- An 8 wheel drive train with a pneumatic switch to select between a low and high gear ratio
  - Selected by 2 buttons (high,low)
- A fuel pusher subsystem that was designed to load balls contained in a hopper into the lower intake slot of the boiler
- A rope climbing mechanism composed of a hook and spindle.
- A release plate designed to allow the gear to drop free of side constraints after it was placed on the spike.

Sensors included:

- Quadrature encoders on two wheels (left and right side)
- Forward and reverse limit switches on the ball pusher mechanism,
- A targeting camera
- A driver camera
- An ultrasonic sensor used for distance measurements
- A digital gyro.

3 Simulated Robot

3.1 Components Modeled

The following components were modeled in simulation:

- An 8 wheel drive train with dual speed transmission
  - The gear ratio switch was emulated by limiting the max voltage applied to the wheels when in low gear
- A ball pusher subsystem consisting of a linear actuator
  - Activated by a button
- A mechanism to release a gear after placement on the spike
  - Activated by a button (teleop) or automatically at the end of autonomous
- A targeting camera used for autonomous mode
  - mounted 5” off-center on the right side of the robot at the height of the center of the gear
- A secondary camera
  - Mounted directly behind the gear and used for autonomous mode debugging
- An ultrasonic sensor for distance measurement
  - this feature was ultimately disabled because the distance measurements returned were found to be erratic and inaccurate (looks like there’s a bug in the sensor plugin code)
- A simulated gyro to measure turn angle
  - In simulation only analog gyros are supported (the competition robot employed a digital gyro)

3.2 Components not Modeled

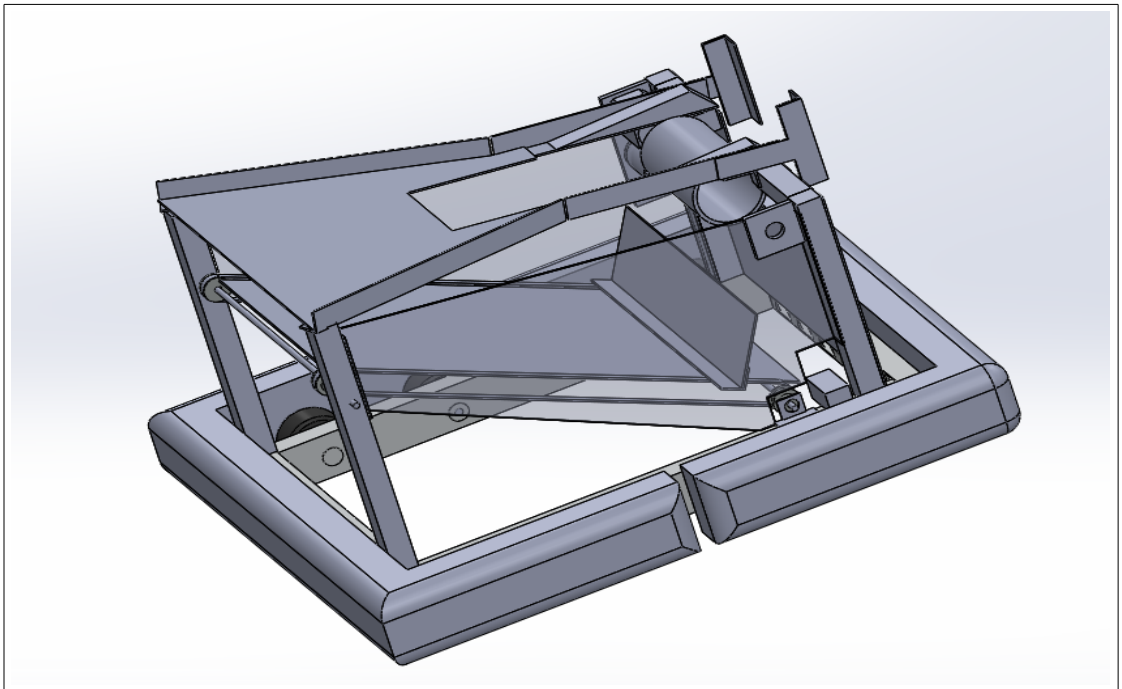
- Electronics board,pneumatics, battery and wiring
- Rope climbing subsystem

3.3 Simulation problems and workarounds

Because of limitations in the current Solidworks exporter and Gazebo simulator some of the mechanical and sensor components couldn't be modeled

1. The rope climbing subsystem couldn’t be modeled because the gazebo simulator doesn’t directly support deformable objects such as springs and cables
2. The simulated ultrasonic sensor was found to be unreliable (sometimes returning inaccurate or even negative values) and so was not used in simulation

3.4 Screen-shot of final simulated robot



Solidworks Model of the Simulated Robot

3.5 Sources

The Solidworks design files that were used in the 2017 Simulated Robot are available in the github repository “Team159/MentorRepository” in the sub-directory Solidworks/Models/

https://github.com/FRCTeam159/MentorRepository/tree/master/Solidworks/Models/2017-SimRobot

The “.sdf” and mesh files generated by the Solidworks exporter are also available for deployment in the Gazebo simulator and can be downloaded from the Solidworks/Exported/FRC2017_Exported directory in the Team159/MentorRepository:

https://github.com/FRCTeam159/MentorRepository/tree/master/Solidworks/Exported/FRC2017_Exported

Software

1 Software paradigms

The First/WPI enhancements to the Eclipse IDE provide an opportunity to base a Robot design using either the “Integrated” or “Command” style paradigms.

The “Integrated” paradigm runs in a single thread with direct callbacks from the user's code to the state machine that services periodic (20 ms) control updates in Teleop and Autonomous modes.

In the “Command” paradigm the source code is organized into separate classes that extend the Command and Subsystem library base classes. In Autonomous mode Commands can be linked together to run sequentially or in parallel to produce complex behavior. In Teleop mode each subsytem can specify a special “Default” command that is run by an external scheduler once per 20 ms cycle.

2 Competition Robot Project Software

The 2017 Competition software was c++ based and was developed using the Eclipse Integrated Development Environment (IDE) with plug-ins provided by First and WPI that are used to generate binary files that run (as applications) on the RoboRio's embedded Arm processor.

The overall software design included independent state machines for three separate subsystems as well as a vision based auto-targeting system. Software control was needed for 6 motors, 2 cameras, a Ultrasonic Rangefinder, 2 encoders (on the drive wheels) and 2 limit switches.

Code development was carried out in the Windows(10) operating system environment which allowed for direct deployment of the compiled binary file from Eclipse to the robot hardware over wireless Ethernet. This year the team also made extensive use of the Java based “SmartDashboard” application particularly for tuning the camera and targeting parameters used for Autonomous operation.

2.1 Paradigm

This year's software was based on the “Command” paradigm for the first time

2.2 Architecture

Following the “Command” paradigm the software design was centered around several subsystem classes, each with a special “default” command that controlled the subsystem’s operation during Teleop mode

Mechanical subsystems included a drive train controller (DriveTrain), fuel delivery system (FuelSubsystem), gear delivery system (GearSubsystem) and climber (ClimbingSubsystem).

A separate subsystem class was also created for the ultrasonic rangefinder (UltraSonicSubsystem)

Vision based targeting employed it’s own subsystem class (Vision) as well as a separate Utility class (GripPipeline) that was a modified version of the c++ code snippit produced by the GRIP application. The GripPipeline class was executed in a separate thread from the main robot application and was responsible for carrying out the image processing operations on the camera input stream using native OpenCV library calls running on the RoboRio. Communication between the GripPipeline and Vision application was done using NetTables PutNumber and GetNumber calls.

2.3 Code sources

Source code is available for viewing or download from the “HughBot” project in the github Team159/2017-Robot-Code repository directory at:

https://github.com/FRCTeam159/2017-Robot-Code

3 Simulation Robot Project Software

The source code used in this year's simulated robot project (SimHughBot) was almost identical to that employed for the competition robot. In many cases the student's code was simply copied over and recompiled using the linux_simulate build target. The following differences however are noted:

- Since simulation classes for CANTalon motor controllers have never been supported by WPI a separate project was developed (CANTalonLib) to allow most of the functionality of those controllers to be emulated. Source code for this library is available on line at the team's MentorRepository github site.
- The digital gyro used in the competition robot was emulated using the simulated AnalogGyro class
- Several bugs and other problems exist in the current WPI simulation libraries (e.g. see section 3.3 below). These were fixed in the allwpilib code and the library was rebuilt

3.1 Software paradigm

As for the competition code this year's simulation software was also based on the "Command" paradigm

3.2 Code sources

Source code is available for viewing or download from the "SimHughBot" project in the github Team159/MentorRepository repository directory at:

https://github.com/FRCTeam159/MentorRepository/Software/workspace/ SimHughBot

3.3 Patches for WPI library bugs

PIDController and Notifier classes

Last year several problems were discovered in the simulation code related to PIDControllers.

A bug report (artf4830) was submitted for these problems though the TeamForge website on 3/15/2016 and has been assigned for work with the status "Open". Progress can be tracked at the following URL:

https://usfirst.collab.net/sf/go/artf4839?returnUrlKey=1460655219989

motor controller classes

In the simulation build all of the motor controller classes return a zero value for the Get or GetSpeed functions. This problem appears to have a very simple fix (a class variable was just not getting updated) and a Bug report was submitted on 3/17/2017 and can be tracked at:

https://usfirst.collab.net/sf/go/artf4840?returnUrlKey=1460655771424

Update: as of 3/2017 no work appears to have been done at WPI to fix any of these reported problems

4 Launching a simulation

Although there is (reportedly) a plan to provide Windows support in the future, simulations currently need to be carried out in a Linux (specifically Ubuntu) OS environment. Details on how to set up a virtual or native Linux environment with all the necessary support tools have been described in previous reports (see the software/docs directory in the team's MentorRepository) and so wont be repeated here. A good reference on how to get started with this can be found at the following link: http://wpilib.screenstepslive.com/s/4485/m/23353

The procedure for running a simulation for this year is as follows:

1. Execute one of the 3 start-up scripts located in the SimHughBot project directory

```
> cd ~/Robotics/MentorRepository/Software/workspace/SimHughBot
> ./start_simulation-center
or:
> start_simulation-right
or:
> start_simulation-left
```

2. Each of these scripts will cause the following series of applications or command scripts to be launched in separate "roxterm" tabs:
   - start_gazebo_<center,left,right>
     - starts command script: gazebo-2017-field-<center,left,right>
     - starts up gazebo application with center,left or right ".world" file
   - start_targeting_camera
     - starts up "mjpg_streamer" application for targeting camera
   - start_driver_camera
     - starts up "mjpg_streamer" application for driver camera
   - FRCUserProgram
     - executes the simulation robot program in ../SimHughBot/linux_simulate
   - sim_ds
     - starts up the simulation driver-station Java application jar file in ~/wpilib/tools
   - start_smartdashboard
     - starts up Java "SmartDashbord" jar file in ~/wpilib/tools

5 Autonomous mode

5.1 Strategy and Requirements

The Autonomous mode strategy chosen for this years challenge was to be able to place a gear on any of the three airship spikes (left, right or center). The decision to do this (as opposed to delivering fuel to the boiler for example) was primarily based on the high point value (60) awarded for getting a rotor started in Autonomous mode (requiring only a single gear). If at least 2 robots in an alliance could deliver a gear 2 rotors could be activated resulting in a very large 120 point starting score. It was also considered important to be able to deliver a gear from a side, leaving the somewhat easier center position available for other alliance teams

5.2 Design Plan

The software design used to implement the Autonomous mode strategy involved activating up to three Commands in sequence, depending on the initial placement of the robot. For all 3 placements the first Command was just to drive forward for a fixed distance (somewhat different for the center and side positions). Next, for the two side positions a second Command was executed to turn the robot a fixed angle inwards towards the airship, Finally, a vision targeting command was executed for all 3 positions to drive the robot to the stake in an attempt to place the gear. If the gear was placed on the spike the gear plate would then be dropped to allow the airship pilot to load the gear into the center rotor slot.

In order to select which Autonomous mode was deployed a special “AutoMode” entry field was added to the SmartDashboard UI, and the drive team was tasked with entering either the text “Center”, “Left” or “Right” depending on where the robot was placed before the start of a match

5.3 Command Implementation

DriveForward (DriveForTime)

Originally, the plan was to use the encoders on the drive train to specify a travel distance and a command called “DriveToPosition” was developed to accomplish this (using a PID controller) Unfortunately, there wasn’t enough time to test and debug this command on the competition robot prior to the end of build season so a simpler “DriveForTime” Command was used (which employed a timer and a fixed drive voltage). The time and voltage where empirically determined so that the robot moved ~ 5’ from the back wall in the side positions and ~4’ in the Center position.

The Simulated robot was able to use either the DriveToPosition or DriveForTime Commands

TurnToAngle (TurnForTime)

In this Command (which wasn’t executed for the center position) the gyro was used to specify a turn angle depending on what position the robot was placed in. The ideal angle that would turn the robot to be directly facing the spike was calculated to be +/- 60 degrees, but for the competition robot it was found to be somewhat less (45-50) and was also different for each side (we’re not sure what the source of this discrepancy is). A PID controller was used to apply a differential voltage to the wheels on opposite sides of the robot to effect an in-place turn

After the Lubbock competition it was discovered that the gyro had been inadvertantly bagged with the robot and (since we didn't have a spare) a TurnForTime command was used instead for continued testing using the practice bot. This command was identical to the DriveForTime command described above except that the wheels on opposite sides of the robot where driven in different directions

DriveToTarget

This command was executed for all 3 positions and was designed to use the target location information returned by the vision processing subsystem (running in a parallel thread) to drive the robot the final distance to the spike. The vision targeting subsystem was tasked with determining the residual distance and offset angle dynamically as the robot made it's final approach to the target.

5.4 Vision processing

The Vision Processing and targeting strategy used for both the simulated and physical robot involved the following steps:

- GRIP was run as a stand-alone application to tune the HSV filter and other parameters needed to identify the target tape pattern
    - Using images captured from either a real camera and target pattern (composed or retro-reflective tape) or a simulated equivalent (captured by mjpeg_streamer from the simulated Gazebo camera output)
    - After the rectangles associated with the target were successfully isolated, the GRIP "Generate Code" feature was used to produce a C++ code segment that contains calls to native OpenCV library functions
    - The following GRIP pipeline functions were used:
        - HSV segmentation
            - Primarily uses Hue to isolate the blue-green ring-light color
        - Identify Contours
            - finds edges of solid blobs
        - Convex Hulls
            - finds external contour boundaries
        - Filter Contours (Hulls)
            - filters and rejects contours based on minimum area, ratio (width/height), etc.
- The OpenCV GRIP generated code was incorporated into a subsystem (GripPipeline) that output a vector of bounding rectangles for all of the captured contours
- An independent thread was started up as part of the robot program initialization that processed

the vector of rectangles produced by GripPipeline and generated a single bounding target rectangle centered on the target (spike)

    - The output of the Vision processing thread (bounding box & number of rectangles used to produce it) was published to a NetTables server (launched by FRCUserProgram running on the RoboRio or Linux)

- During Autonomous mode the “DriveToTarget” Command read from the NetTables server the data sent by the vision thread and adjusted the trajectory of the robot to drive to the center of the target
    - corrections were made to compensate for the placement of the camera and handle the special problem cases described in “targeting details” below

5.5 Port of Image processing code to a Raspberry Pi 3

Because the vision processing thread running on the RoboRio seemed to be fast enough (no noticeable degradation of driving responsiveness etc.) there wasn’t a strong incentive to offload the computation code to an external processor this year. However, since it is anticipated that future vision processing requirements may be more demanding an investigation was made into porting the vision based target identification code to a raspberry Pi 3 and measuring the impact that had on the cpu load of a host computer (in this case a high end quad core Dell laptop) running a full Gazebo simulation. The steps that were taken for this are briefly outlined below (more details can be found in the document “ImageProcessing.odt” on the FRCTeam159 github MentorRepository site)

PI Setup (need to build and install the following libraries)

- libntcore.so
    - Built on PI from source using cmake from WPI github project
- libcscore.so
    - Extracted and installed from a download tar file provided by WPI
- libCameraServer.so
    - Can be built on Ubuntu development system in Eclipse using ARM cross-compiler or natively on Pi using a makefile
- OpenCV 3.0 development system
    - ref: http://www.pyimagesearch.com/2015/10/26/how-to-install-opencv-3-on-raspbian-jessie/
- mpstat (to measure cpu usage)
    - `sudo apt-get install sysstat`

C++ Vision Code Porting details

- Created a application project called “ImageProc” using modified code from GripPipeline and Vision Subsystems
  - First built as a test project in Eclipse and run as a separate processing application on Ubuntu
  - Then the source code was copied to the pi and built using a Makefile
    - Note: Also tried to build using a cross-compiler on Ubuntu but ran into too many undefined header and dynamic library files (probably would need to have a complete copy of /usr from the Pi available to avoid this)
  - In the simulation startup sequence, launch ImageProc remotely from a Ubuntu command shell (after Gazebo and mjpeg_streamer have been started on the host) using ssh
    - ssh pi@raspberrypi.local ImageProc/ImageProc
    - note: need to enable password-less ssh on Pi using .ssh keys
- Uses Network tables for all communication to and from the Robot Program (FRCUserProgram)
  - Sends target bounding box data to “DriveToTarget” command
  - Receives display attributes (showRGBThreshold etc.) captured by Robot program from SmartDashboard and forwarded to ImageProc through nettables PutValue calls
    - Note: SmartDashboard wasn’t directly supported on the Pi since it requires access to Gazebo headers and libraries when compiled for simulation
- In simulation, ImageProc uses the OpenCV “VideoCapture” class to process the images published to an http port by mjpeg_streamer:
  - To enable Gazebo image auto-save include the following elements in the robot .sdf file:

```
<sensor name="ChassisCamera" type="camera">
  <updateRate>10.0</updateRate>
  …
  <camera name="ChassisCamera">
    <save enabled="true">
      <path>/tmp/target-camera</path>
    </save>
  </camera>
</sensor>
```

    - When running, Gazebo will continuously save camera images to the specified directory at a rate given by <updateRate>
  - Start up mjpeg_streamer on Ubuntu host

        - > mjpg_streamer -i "input_file.so -f /tmp/target-camera -r -d 0.1" -o "output_http.so -w www -p 5002"
        - mjpg_streamer reads images as they appear in the save directory and publishes them to a http port (e.g. 5002). The video stream emulates a network camera and can be viewed in a web browser (set url to localhost:5002)
        - After a new image is read and published it is deleted from the save directory (-r flag)
    - VideoCapture code in ImageProc running on Pi
        - `cv::VideoCapture vcap;`
        - `vcap.open("http://Ubuntu16.local:5002/?action=stream?dummy=param.mjpg")`
        - `note: without "dummy=param.mjpg" vcap.open fails when run on the Pi but succeeds with or without it when run on Ubuntu`

Benchmarks (using > mpstat 2)

Image stream was limited to 20 frames per second by setting <upaterate> in sdf file and "-d 0.05" in mjpg_/distributestreamer

- Install mpstat
    - `sudo apt-get install sysstat`
- CPU usage on Pi
    - usr: 37.8% sys: 0.46% (mostly ImageProc)
        - decreased to ~25% when video frame buffering was disabled (see below)
- CPU usage on Host when running Gazebo with Pi support
    - usr: 68% sys: 26.2% (mostly gzserver)
- CPU usage on Host when running Gazebo without Pi support
    - usr: 73% sys: 22.5% (mostly gzserver)
- Measured time to process one image on pi: ~20 ms
    - but went up ~5x when video frame buffering was disabled (see below)
- Observations

1. There is a substantial lag (at least 1-2 seconds) between Teleop or Autonomous robot motion in Gazebo and target info and camera view data updates reported by both pi and host
    - Tests to try and isolate the problem
        - delay not noticeable when using desktop Ubuntu host for image processing
            - running as a separate thread on same host

- delay not noticeable when using DELL Ubuntu laptop for image processing
  - separate host on LAN
- latency issue not improved by disabling video streams sent by mjpeg-server
- disabling wifi on pi doesn't help
- disabling image processing on pi (i.e just return the input image stream) seems to fix the delay problem (so problem is related to processing)
  - delay begins after hsvThreshold (or any opencv function) is called in GripPipeline.cpp
  - much bigger delay if blur called (also process time → 100 ms)

- Discovered Cause
  - see ref: http://answers.opencv.org/question/29957/highguivideocapture-buffer-introducing-lag/
  - it appears that opencv `cv::VideoCapture` first fills a buffer with (5?) incoming frames before processing is started and then the buffer acts like a FIFO
    - So the first (and subsequent) frames are available for output only after a latency delay of at least 5x processing time cycles has elapsed
- Solutions
  - There's supposed to be a command to set the buffer size in code but only works with some versions of OpenCV
    - `vcap.set(CV_CAP_PROP_BUFFERSIZE, 1);`
      - but this didn't work for my system (pi 3 with OpenCV 3.0)
  - A (cludgy) work-around is to flush the buffer with dummy frame reads before processing
    - for(int i=0;i<6;i++) vcap.read(mat);
    - if the initial buffer is not empty (or framerate is too high) it is now cleared (excess frames are discarded) and then vcap.read will wait for next incoming frame before unblocking
    - after this work-around the observable delay disappeared but image processing time increased from 20 to 70-90 ms (Q: was opencv using batch processing before this fix ?)
  - without buffering it looks like image processing on the Pi is limited to about 10-12 FPS

- also saw a visible slew delay in target camera SmartDashboard window when using the RoboRio for processing on the practice bot

5.6 Port ImageProc to a Jetson TK1

Porting ImageProc to the Jetson TK1 followed a similar path to that described for the Pi-3. Additional difficulties were encountered because the TK1 has a 32 bit architecture and an older version of OpenCV was installed. Additional porting details can be found in the document ImageProcessing.odt in the MentorRepository/Software/docs directory. The following briefly lists the problems encountered and work-arounds

1. OpenCV
   - The packaged NVIDIA version OpenCV4Te is based on OpenCV 2.4 and doesn't support opencv_vidioio which is needed by CameraServer and cscore
   - Solution was to build opencv 3.0 from github source
2. cscore
   - The pre-compiled Maven artifact (libcscore.so version 1.02) lead to run time crashes when CameraServer.GetInstance() was called
   - Solution was to build cscore natively using an arm toolchain

Benchmarks

1. Interface to Gazebo simulation
   - modified start_image-script_remote to:

     ssh ubuntu@tegra-ubuntu.local "ImageProc/ImageProc Ubuntu14.local"
2. CPU usage
   - with frame buffering
     - Rectangle window in SmartDashboard: 60 FPS
     - mpstat: usr: 50% sys: 18% idle:30% (ImageProc)
     - Image cycle time: 5-13 ms
   - without frame buffering
     - Rectangle window in SmartDashboard: 13 FPS
     - mpstat: usr: 58 % sys: 7.4 % (ImageProc)
     - Image cycle time: 70-90 ms

Notes

1. Unlike on the PI no noticeable lag was seen when frame buffering was enabled
2. Doesn't look like GPU is being used
   - e.g when simulation is running “sudo ./tegrastats” in ubuntu home directory prints:
   - ./RAM 662/1892MB (lfb 4x4MB) cpu [0%,0%,0%,0%]@1428 EMC 9%@204 AVP 0%@204 VDE 120 GR3D 0%@72 EDP limit 0
   - but with /opt/VirtualGL/bin/vglrun ./oceanFFT

     RAM 798/1892MB (lfb 1x4MB) cpu [74%,off,off,off]@960 EMC 15%@396 AVP 0%@204 VDE 120 GR3D 76%@108 EDP limit 0
   - web search suggests need to compile with cv:gpu:SetDevice(0) for gpu support but can't get this to work
   - more web searching: in OpenCV 3.0 gpu namespace was replaced by cuda
     - In cpp source code include:

       #include "opencv2/core/cuda_types.hpp"

       #include "opencv2/core/cuda.inl.hpp"
     - cv::cuda::SetDevice(0)
       - no longer causes link error
       - but “sudo ./tegrastats” still doesn't show any gpu usage when simulation is run
   - Search also suggests that code needs to call cv::cuda functions explicitly in order to get gpu assist
     - Use GpuMat vs Mat in function arguments
       - GpuMat `gpuInput`(Mat cpuMat); // constructor calls upload of cpu matrix to gpu
       - GpuMat `gpuOutput;`
     - `use cuda equivalent functions`
       - `cv::cuda::cvtColor(gpuInput, gpuOutput, cv::COLOR_BGR2HSV);`
     - Retrieve data from GPU
       - gpuOutput.download(Mat &out);
     - release GPU memory
       - gpuOutput.release();
       - gpuInput.release();

- Initially tested cuda version of ImageProc on Ubuntu14 (GTX980) and encountered a very large startup delay when GpuMat upload `was called (many seconds)`
  - solution was to recompile OpenCV 3.1 with 5.2 added to CUDA_ARCH_BIN list to support Maxwell architecture (otherwise JIT will recompile library on every use)
    - `see: http://answers.opencv.org/question/52304/long-delay-on-cvgpugpumatupload-after-upgrade-to-gtx970/`
- Added gpu support to Imageproc and retested benchmarks
  - Modified GripPipeline code to use GpuMat and cv::cuda::cvColor
    - note: cvColor is the only opencv function used that has a direct cuda equivalent
  - added -lcudart and -lopencv_cudaimageproc to library list
  - added -L/usr/local/cuda-6.5/targets/armv7-linux-gnueabihf/lib link line
    - location of libcudart.so
  - reran benchmarks and measured gpu/cpu usage
    - RAM 953/1892MB (lfb 1x4MB) cpu [68%,53%,off,54%]@1224 EMC 25%@204 AVP 0%@204 VDE 120 GR3D 21%@72 EDP limit 0
    - mpstat usr: 50 % sys: 17% idle: 31
    - cycle time (with frame buffering) : 5-13 ms
  - no noticeable difference in performance when GPU assist is added
    - but cycle time may be limited by image source frame rate
    - since only cvColor is run on the gpu there may not be enough work to do to effect performance

3. Unlike on the pi, ImageProc on the Jetson doesn't exit when the command window that invokes it on the Ubuntu host is closed. Subsequent evocations of CameraServer->PutVideo then fail with the error: CS: ERROR: bind() to port 1181 failed: Address already in use
   - use of the socket can be found using: lsof -i :1181

     COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME

     ImageProc 25934 ubuntu 7u IPv4 1384252 0t0 TCP *:1181 (LISTEN)
   - Imageproc can be killed using (e.g.)

     $ kill 25934
   - Alternatively, to kill running instances of ImageProc on simulation startup the start_image-

script_remote can be modified to:

```
ssh ubuntu@tegra-ubuntu.local "killall ImageProc;ImageProc/ImageProc
Ubuntu14.local"
```

5.7 Targeting details

Ideally, the targeting subsystem expects to see separate rectangles for the two strips of retro-reflective tape positioned on opposite sides of the spike. From the bounding box of these target rectangles a center point is determined that the robot will attempt to drive to (after compensating for the camera offset). The distance to the target can be estimated from the projection of the bounding rectangle on the 320x240 image plane since the Camera's "Field of View" (FOV) and target real dimensions are known (or can be measured).

Problems discovered in Simulation testing

In simulation testing, a number of special problem cases were discovered:

1. If the camera angle to the target is sufficiently large one of the tape rectangles can be split into two by the spike (resulting in 3 rectangles instead of 2)
    - This problem case was dealt with by identifying a group of rectangles that had certain center-to-center vs width relationships given the known geometry of the tape pattern
2. Once the camera is very close to the target one of the tape rectangles will usually disappear off-screen (This is unavoidable since the camera position is offset 5" to the right side of the gear center)
    - A solution was to assume that the center of the target was actually to the left or right of the center of the remaining tape rectangle and adjust the offset accordingly
3. At some camera angles it's possible to see one of the reflective tape patterns from a neighboring target and only one of the target tape rectangles
    - In this case both rectangles will fail the geometry test described in (1) above resulting in no residual target rectangles
    - This case (i.e. more than one candidate but no "good" rectangles) was handled by accepting the single rectangle with the largest area and applying the offset correction described in (2) above

Problems discovered during and following the Lubbock competition

At the Lubbock FRC several problems were discovered that weren't caught in simulation (which resulted in most attempts to place a gear in Autonomous mode failing). It is planned to try and find fixes and workaround to these issues before the FRC competition in Denver.

1. An image of the light ring reflected from the plastic wall behind the target sometimes appeared in the camera image which caused the targeting system to generate an incorrect bounding box.

(This problem wasn't identified in simulation since Gazebo isn't capable of tracing reflections from light sources)

possible solutions:

1. Only keep the top 2 rectangles that have the largest area
2. Reject the light-ring image using expected geometry properties (e.g. the width/height ratio should be ~1)
3. Tilt the camera/ring assembly somewhat so that the light ring image is not reflected directly back towards the camera

2. The TurnToAngle Command took too long to complete. This is probably caused by the PID controller not converging on the target angle fast enough

   possible solutions:

   1. Replace the PID and gyro-based TurnToAngle command with a simpler "TurnForTime" command
   2. Modify the PID and tolerance parameters to produce a faster response

3. The vision-assisted DriveToTarget Command often didn't drive far enough to capture the spike. This problem is partially do to the fact that it takes a certain minimum voltage applied to the drive train (at least 10%) for the robot to move forward at all

   possible solutions:

   1. Always add a minimum voltage when driving forward
   2. Decrease the target minimum distance

4. The spike often intersected one of the "spokes" of the gear and not one of the gaps (this problem was also observed in Teleop mode)

   possible solutions:

   1. Build a detection system that sends a signal when the spike has penetrated a plane behind the gear
      - Could be based on an UltraSonic sensor
      - Use two mirrors placed on opposite sides of the gear with a LASER or IR source and a detector tilted to that the beam traced back and forth between the mirrors several times
   2. Turn the robot quickly a few times left and right once the final distance is reached to try to dislodge the spike from the spoke if it hits there (shouldn't be a problem to do this even if the spike is already in one of the gaps)

5. When the robot was power-cycled the camera images returned by the targeting camera were

often too bright and resulted in many false target rectangles being generated by the vision subsystem

- Originally, the desired brightness and exposure settings were only sent to the camera when the robot program first started up or a change to their values was made from the SmartDashboard.
- The problem went away when the settings were sent every 20 ms robot update cycle whether or not a change was made from the dashboard
- A possible reason for the problem is if the firmware running on the camera starts up slower than the robot program after power is first applied and the camera application misses the initial set commands sent by the robot application
- A concern is that sending camera settings over the USB port every robot cycle might impact performance (although this was not observed). Another solution might be to add a minimum delay (or number of cycles) before sending the initial settings commands

6. It's better to skewer the gear in one of the top gaps than those on the bottom
   - When the gear plate is up (travel mode) the spike is aligned with the center gap on the bottom of the gear. If the gear is placed in this position a couple of problems can occur
     - when the plate is lowered the gear will spin around which can cause it to rotate off the spike
     - The spike is positioned below the climber spindle which makes it harder to remove the gear
   - A solution found that seemed to place the gear more reliably was was to stop a short ways in front of the spike, drop the gear plate, and then move forward (blindly) the remaining distance to the airship wall.
     - After this maneuver the gear was usually speared in one of the upper gaps
     - Another advantage in stopping earlier was that the vision distance calibration was more accurate since at least one of the tape strips could be fully observed
7. After delivering a gear, the robot needs to back up so that the airship operator can hoist retrieve it
   - This probable was addressed by adding a reverse direction "DriveForTime" command at the end of the autonomous mode CommandGroup
   - The distance of back-up travel was adjusted so that the airship operator could lift the spike if the gear was delivered but in case of a failure (e.g. the spike intersected with one of the spokes instead of a gap) the gear would still be in the possession of the robot so that the driver could have a second chance to deliver it at the start of telep mode

6 Teleop Mode

Originally, vision based targeting was only planned for Autonomous mode. But after observing how well it was performing in practice it was decided that a new feature would be added for the Denver FRC that would allow the driver to turn on vision-assisted gear placement as a result of pressing a button.

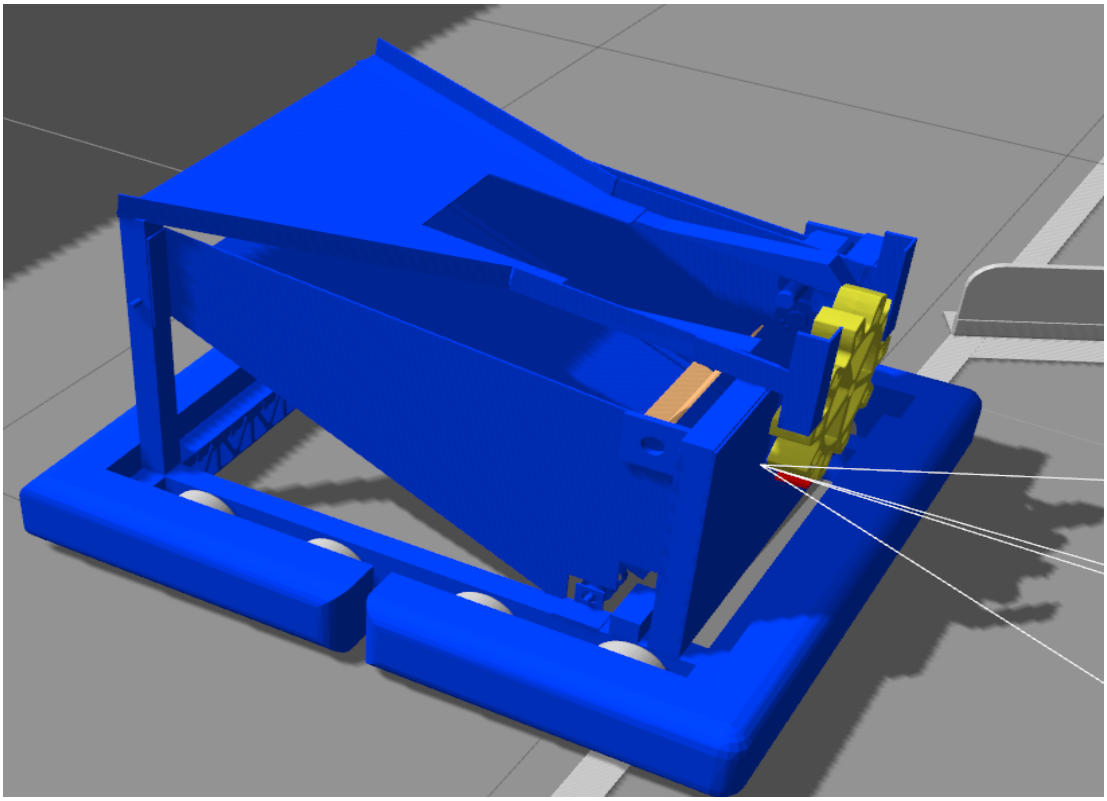
Tests and Results (Simulation)

1.1 Tests and Observations

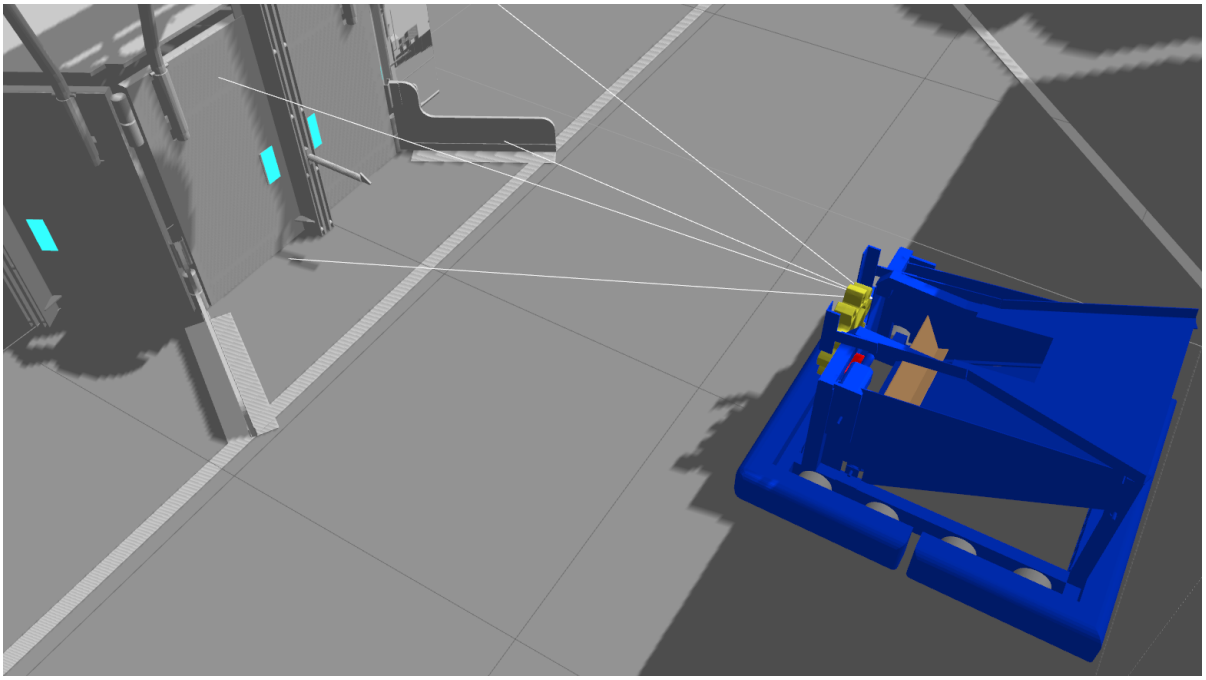
1. Driving and turning
    - The general responsiveness to accelerating and turning of the robot model in the simulator was seen to be more sluggish than that observed in the real robot. Some of the difference can probably be attributed to the fact that the simulation runs at a fraction of real-time speed and therefore is somewhat moving in “slow motion”.
    - Responsiveness was also improved by running the simulator on a system with a faster processor and graphics card (e.g. as opposed to a slow laptop or from within a virtual machine)
2. Cameras
    - In the Gazebo simulator it's possible to both project a view seen from a camera to a “floating screen” in the field and/or to a separate window that can be made visible through a selection in the “Topic Visualization” menu list
    - Two cameras were added to the Simulated robot in similar locations to those placed on the physical robot and it was possible to view the field from the perspective of either camera by switching between topics in gazebo.
        - Separate panels for each camera are are also included in the “SmartDashbord” interface

1.2 Screenshots

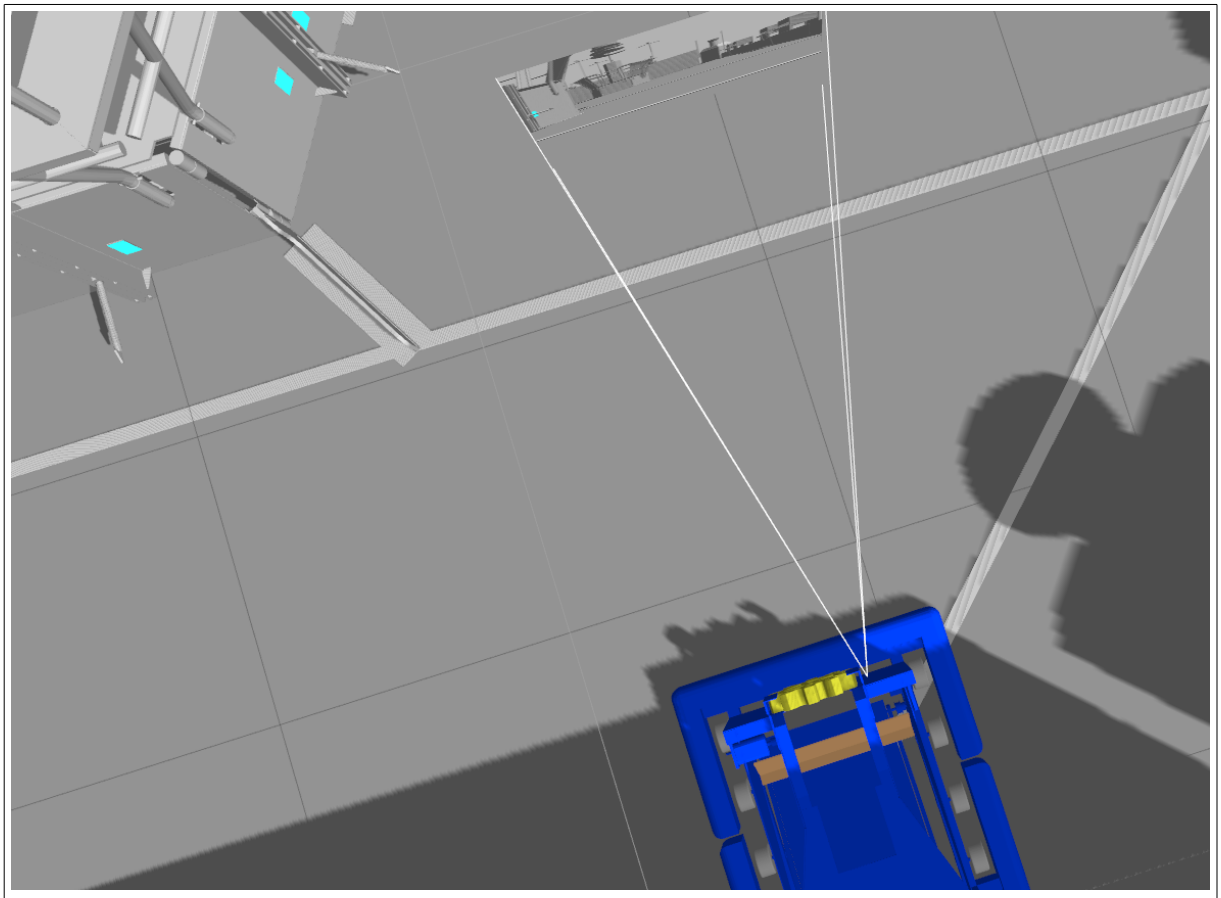
Robot Model in Gazebo



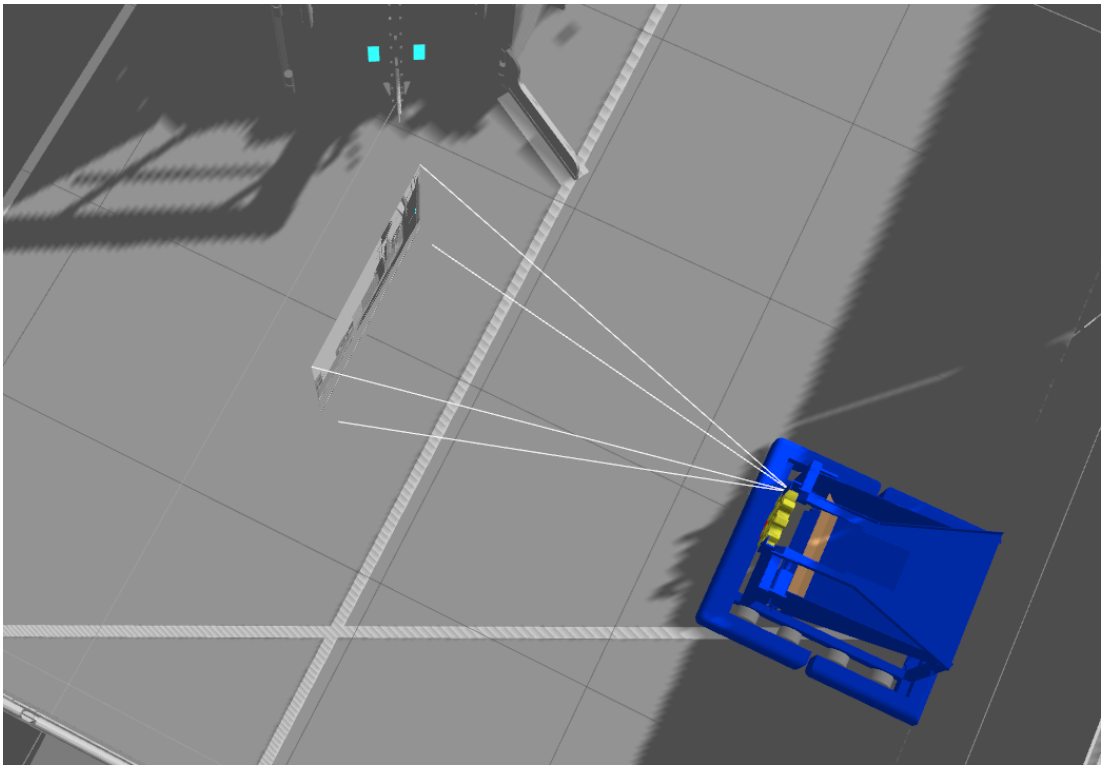
Starting Position Center



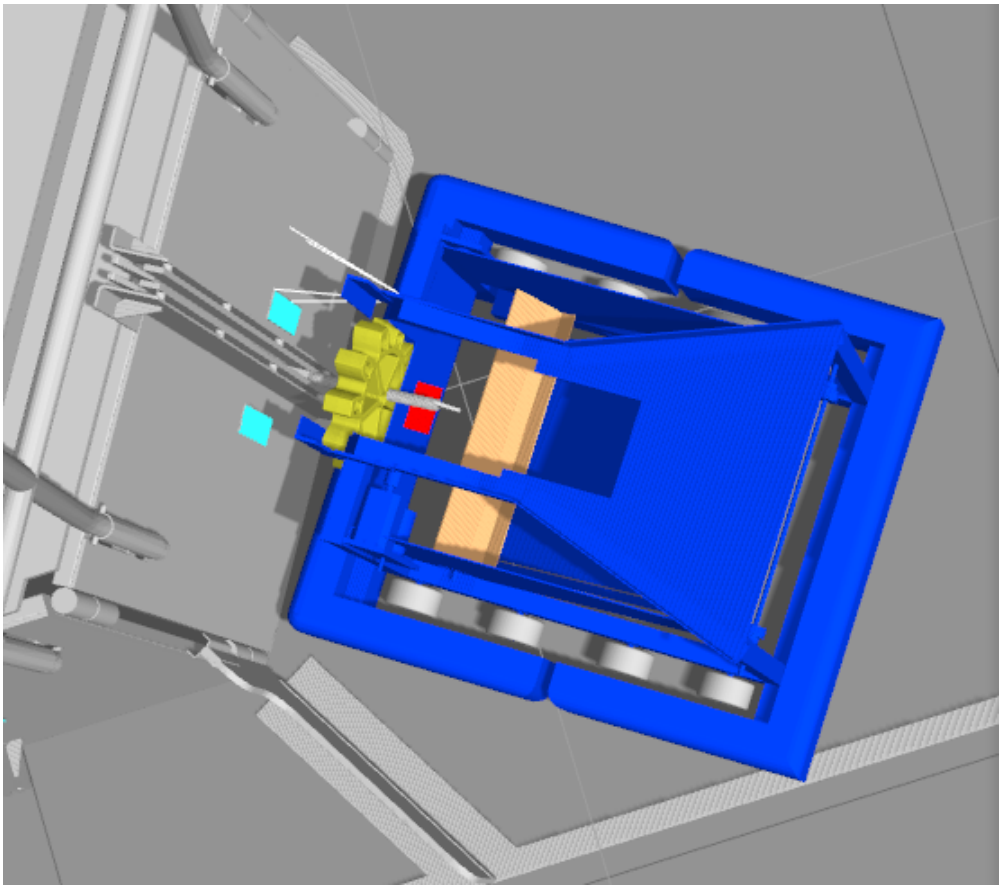
Starting Position Right



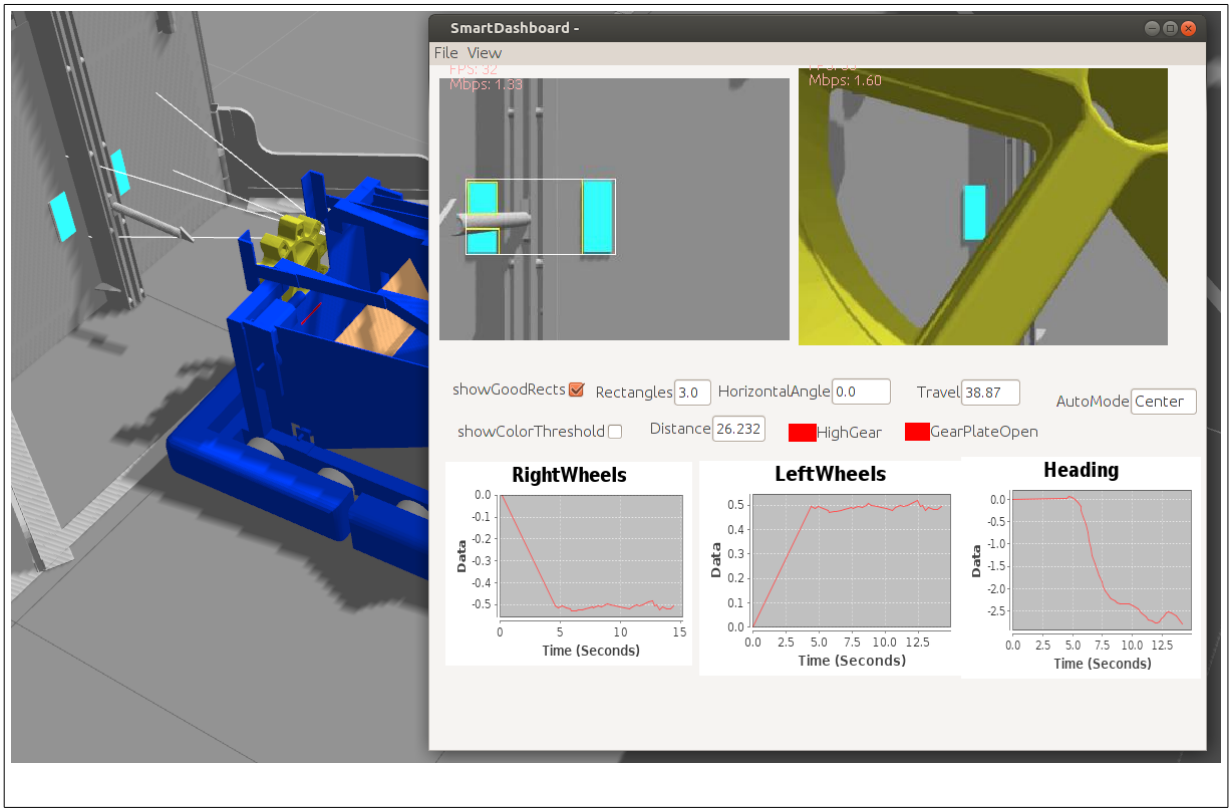
Starting Position Left



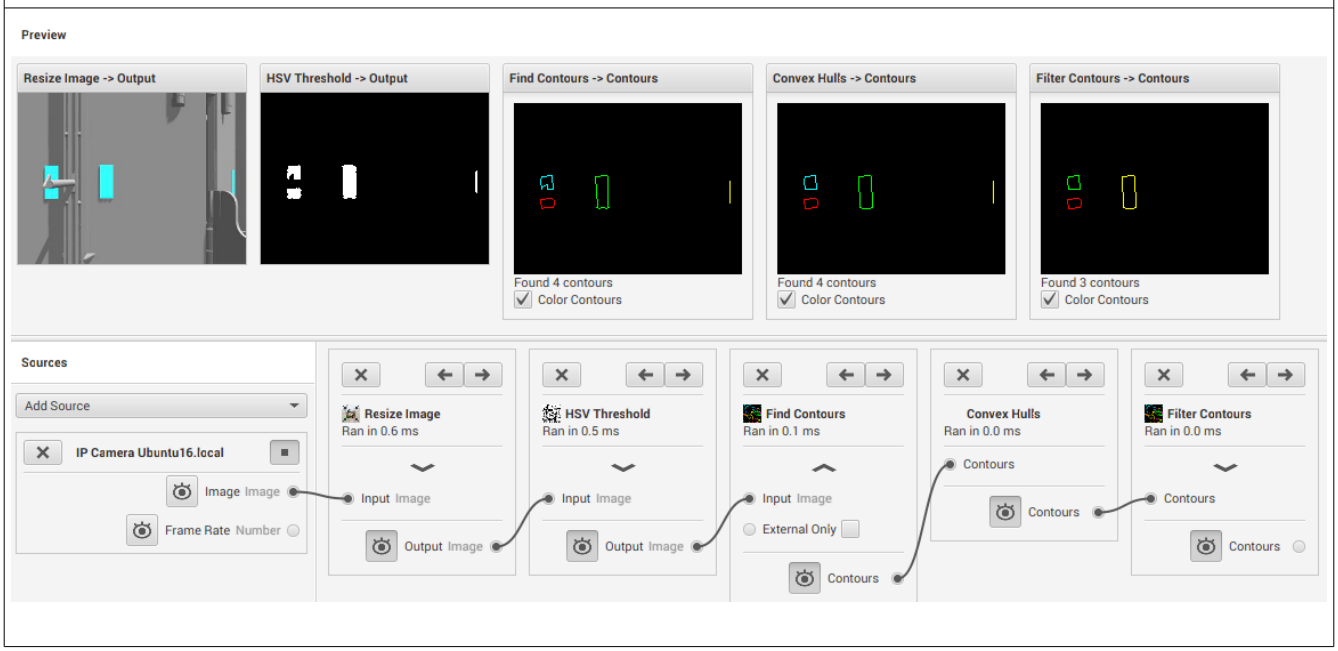
After Successful Autonomous (Right)



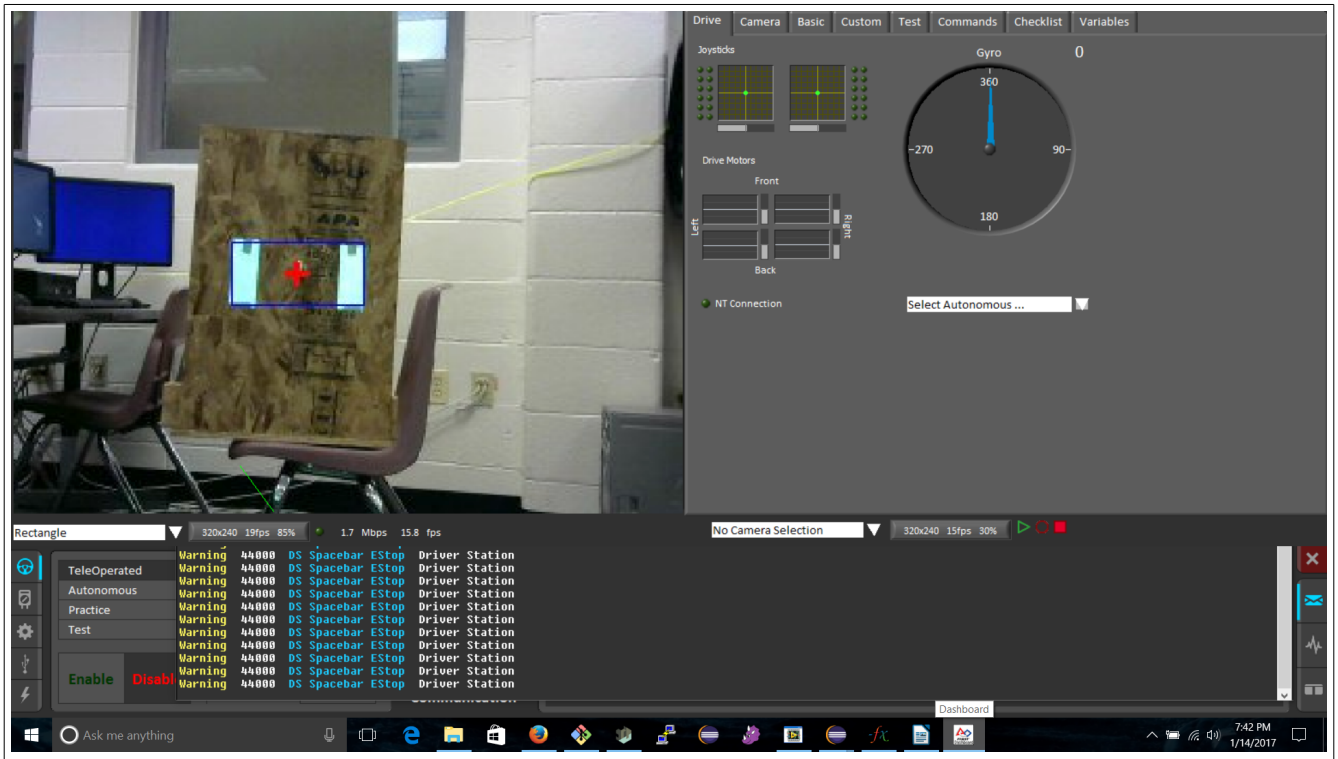
SmartDashboard Screenshot



Example of GRIP pipeline tuning in simulation



Target acquisition of test tape pattern in default dashboard


Drive
Camera
Basic
Custom
Test
Commands
Checklist
Variables
Joysticks
Gyro
0
360
270
90
180
Drive Motors
Front
Left
Right
Back
NT Connection
Select Autonomous ...
Rectangle
320x240 19fps 85%
1.7 Mbps 15.8 fps
No Camera Selection
320x240 15fps 30%
TeleOperated
Autonomous
Practice
Test
Enable
Disabl
Warning 44008 DS Spacebar EStop Driver Station
Dashboard
Ask me anything
7:42 PM
1/14/2017