Simulated Robot

For FRC 2016 “Stronghold” competition

Team 159

Motivation

1 Overview

Over the past two or three years there has been a significant effort put forth by First and it's partner WPI to develop a set of tools to give FRC teams a user friendly simulation environment to test concepts and robot designs prior to the physical manufacturing process. In principle, this can provide many advantages including (but not limited to):

- The ability to test the inertial and geometry properties of a model as in 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 prototype
- The ability to evaluate candidate design concepts without expending the resources (time and dollars) to construct physical models
- A mechanism to continue testing software and competition strategies between bag day and the days of competition
- 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 Gazebo simulation environment

3. Gazebo simulator
    - 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

Solidworks

1 Competition Robot

1.1 Expectations

During the 2016 build season the team's Solidworks CAD model development often lagged the production of physical subsystems and assemblies due to a variety of factors:

- More emphasis and resources were given this year to manufacturing and physical prototype development
- The decision to build two complete working robots (which put demands on limited manpower resources)
- The existence of too many competing design ideas too late in the season (which might have been prevented if a working simulation environment had been available for more rapid prototyping)

1.2 Results

The final robot design included an 8 wheel articulating drive train, a bumper subsystem and two manipulators: a flywheel based ball shooter and a set of arms with rollers that could function as both a ball loader and as an apparatus designed to overcome some of the defenses in the 2016 field (e.g. the Portcullis and the Cheval de Frise

2 Simulated Robot

2.1 Components Modeled

By the end of the build season many of the critical components had already been CADed to a level that was adequate as a starting point for a simulation model, others were implemented later. Included are:

- An 8 wheel drive train with limited side-to-side frame articulation
- A shooting subsystem that included a dual fly-wheel ball propulsion mechanism
- A ball holder sub-assembly that had an arm used to stage the ball during loading and hold it prior to launching
- A push wheel used to propel the ball into the flywheels
- Bumpers with brackets

- A ball loading apparatus that consisted of rollers mounted on a set of arms

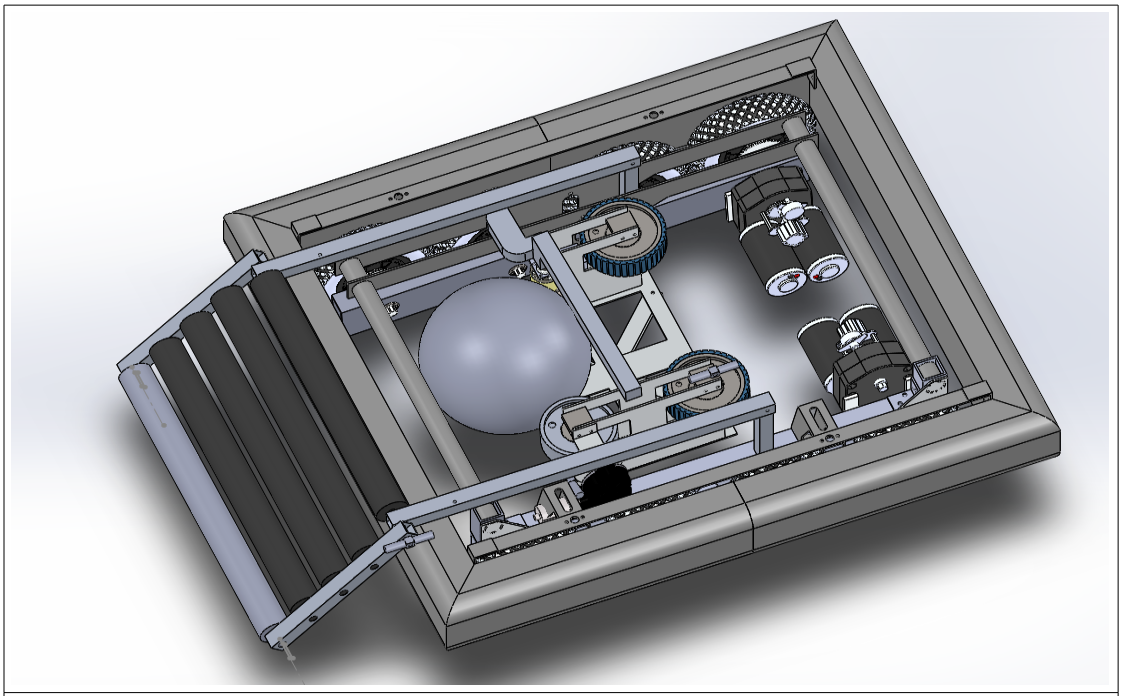
2.2 Components not Modeled

- Electronics board, battery and wiring
- Polycords, drive chains, pulleys and springs

2.3 Workarounds

Because of limitations in the current Solidworks exporter and Gazebo simulator some of the mechanical and sensor components couldn't be directly modeled but had to be emulated:

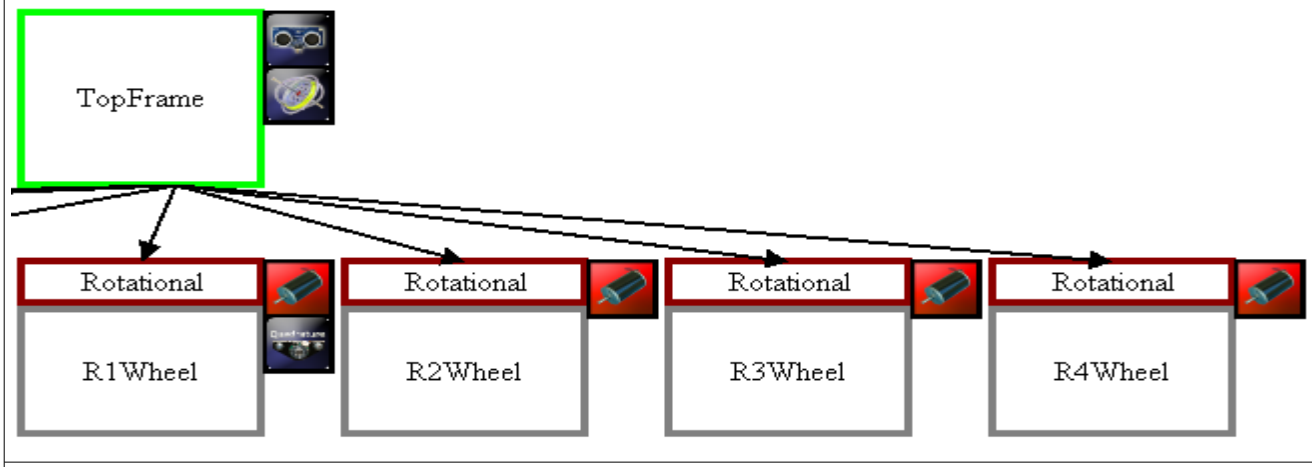
1. The Polycord bands in the Loader apparatus were replaced by 3 additional rollers
2. The infrared sensor used as the ball detector was replaced by an ultrasonic sensor (rangefinder)
    - In order to detect whether or not a ball was present in the Holder, the sensor needed to be pointed somewhat downwards. In the Gazebo simulator, the determination of minimum distance for the reflected beam requires impact with an external model (e.g. the ground plane, a wall etc.) whereas the sides etc. of the parent model are not detected
3. Accelerometers used in the shooter and loader were emulated using single axis gyros



Final Solidworks Model of Simulated Robot

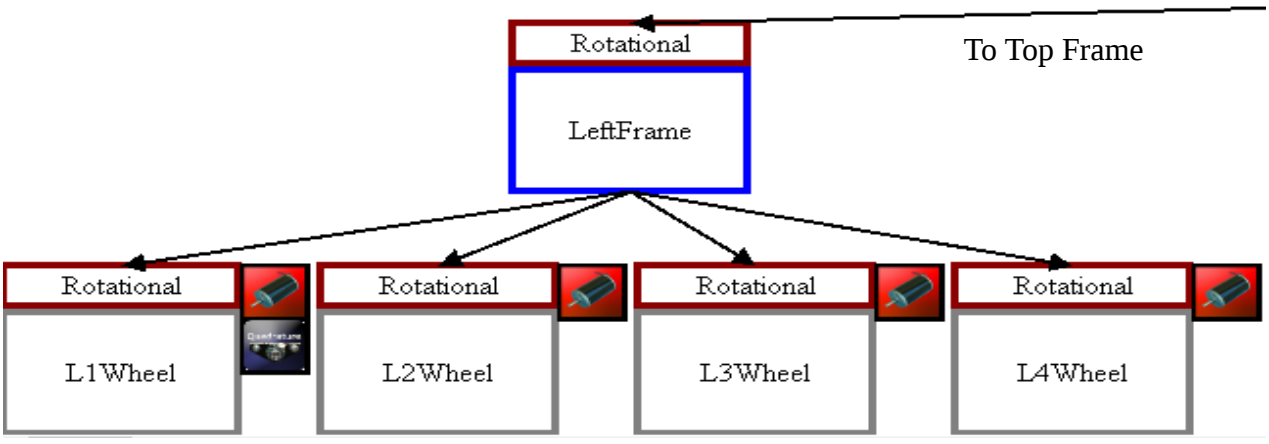
2.4 Solidworks Exporter Robot Model Joint tree

Right (Top) Frame (non-articulating side) and right wheels



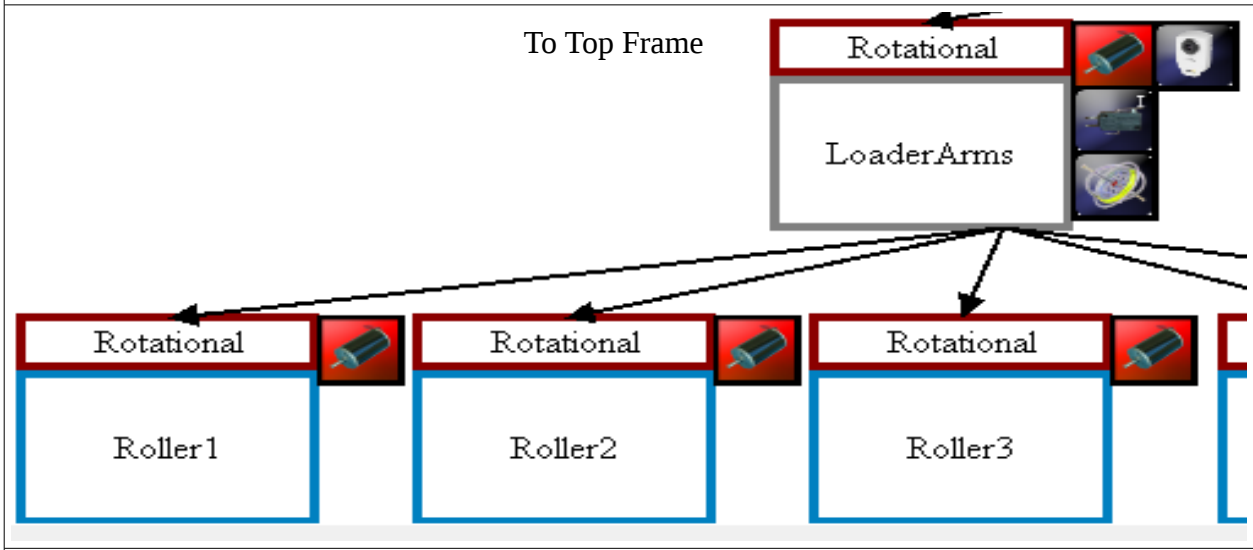
Sensors: Gyro(yaw), Rangefinder (ball detector), Encoder(on R2Wheel)

Left Frame (articulating side) and left wheels



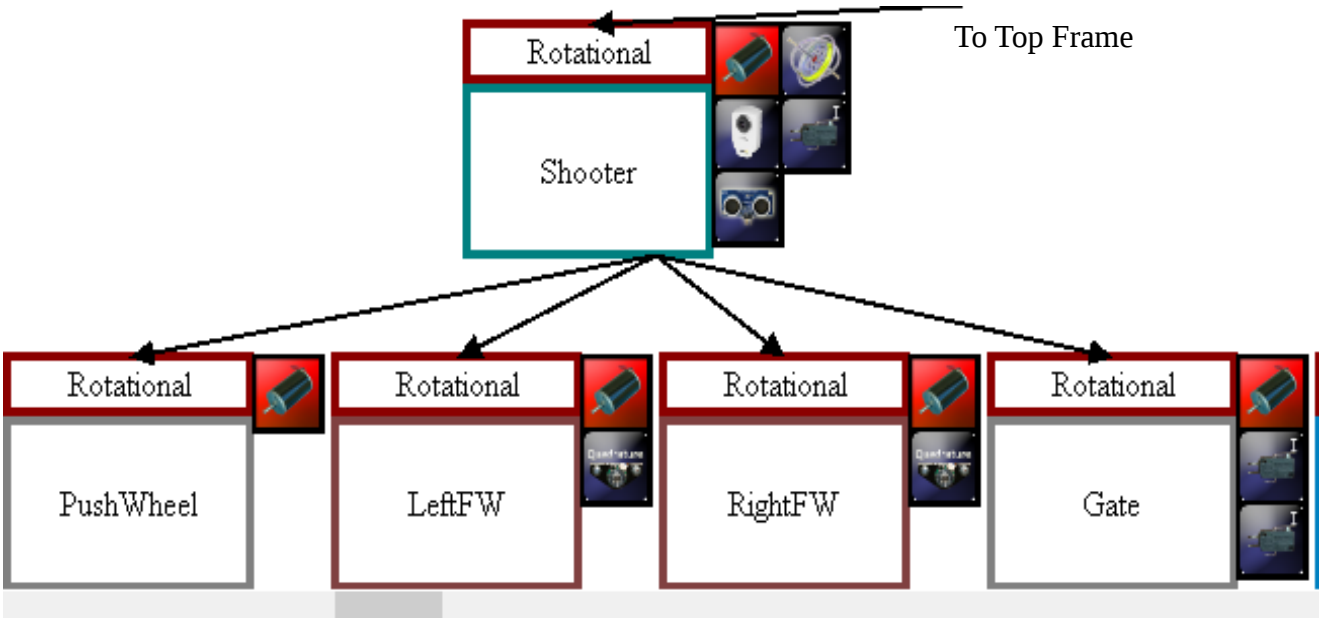
Sensors: Encoder(on L2Wheel)

Ball Loader (contains 5 rollers – 2 not shown)



Sensors: Gyro-pitch (emulates accelerometer), Camera (rear facing), Lower Limit switch on arm

Shooter/Holder



Shooter Sensors: Camera (front facing), Gyro (emulates accelerometer), Rangefinder(emulates Lidar)

Holder Sensors: Encoders (left&right flywheels) Limit switches (gate open, closed)

2.5 Sources

The Solidworks design files that were used in the 2016 Simulated Robot are available in the github repository “Team159/MentorRepository” in the sub-directory Solidworks/Models/2016-SimRobot

https://github.com/FRCTeam159/MentorRepository/tree/master/Solidworks/Models/2016-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/2016-Robot_Exported directory in the Team159/MentorRepository:

https://github.com/FRCTeam159/MentorRepository/tree/master/Solidworks/Exported/2016-Robot_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 is arguably the simplest to implement since it 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. A disadvantage of this approach is that the user code needs to be aware that it is running in a scheduler and so cannot monopolize compute resources in a given update cycle while servicing a single program aspect (like the PID control of a motor, or an image processing step) without running the risk of degrading another critical function (such as joystick control over the drive-train).

The “Command” paradigm is somewhat more complex since it requires that the source code be more carefully organized into separate classes that extend the Command and Subsystem library base classes. The “Commands” in the user program can be designed to carry out simple operations such as to toggle a lever, raise a manipulator by a fixed amount or more complex operations such as to direct a state machine that controls an entire subsystem. Commands can also be configured to run sequentially or in parallel and can be strung together into “CommandGroups” that can be used to specify a complex series of operations that might be used, for example, to implement an Autonomous mode strategy.

In general, the “Integrated” paradigm gives more explicit control over the real-time operations the software must carry out but places the burden of avoiding compute resource allocation and scheduling bottlenecks on the user. Since the “command” paradigm is “multi-threaded” it depends on the background scheduler to allocate when a particular command is serviced which simplifies the user's code but makes it somewhat less clear what is happening in a given moment in real time.

2 Competition Robot Project Software

The 2016 Competition software was c++ based and was developed by the students 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 this year was quite ambitious and included independent state machines for three separate subsystems as well as a vision based auto-targeting system. Software control was needed for 11 motors, 2 cameras, an infrared ball sensor, a Lidar rangefinder, 2 accelerometers, 4 encoders and 5 limit switches. Extensive use was also made of tuned PID control loops

Code development was normally 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. The Windows environment also included a driver station application which provided a console window that could be used for code debugging (via print statements)

2.1 Paradigm

Like the previous year, this year's software was based on the “Integrated” Robot paradigm

2.2 Architecture

The software design was centered around a single main controller class “Robot” that extended the “IntegratedRobot” base class. Utility classes were created for image analysis and auto-targeting (Particle,VerticleTargeter, Launcher) and to process sensor data (AngleAccelerometer,Lidar).

Subsystems included a ball holder (Holder), ball loader (Loader) and drive train controller (TankDrive)

Separate directories were used to hold the source files for the Holder and Loader subsystems so that they could be accessed independently from external “unit test” projects via soft links.

2.3 Implementation Details

During the build season it was possible to test and debug most of the software subsystems using electronic breadboards and such but working hardware wasn't available by bag day for full testing of critical components such as the ball loader and shooter mechanisms. We had anticipated being able to continue software development on the 2nd robot that was to constructed prior to the Denver competition but, unfortunately, unexpected delays and problems that were encountered with the angle motor in the shooter etc. prevented this from happening.

As a result, there was little or no time available prior to competition for driver practice or to test and debug the auto-targeting subsystem. The lack of a reliable shooting mechanism required that that entire functionality (including auto-targeting) had to be disabled. Problems with the drive train encountered in Denver also prevented deployment of a planned Autonomous mode strategy.

2.4 Code sources

Source code is available for viewing or download from the “IntegratedRobot” project in the github Team159/2016-Robot-Code repository directory at:

https://github.com/FRCTeam159/2016-Robot-Code

3 Simulation Robot Project Software

Originally, an attempt was made to adapt the Team's robot software to support a simulation build directly but this soon proved unwieldy for a variety of reasons including:

- The WPI Simulation libraries for Eclipse do not yet support many critical components that were incorporated in this year's robot design (e.g. CANTalon motor controllers, accelerometers, optical sensors)
- There is no simulation support for the IMAQ image processing package that was used in the auto-targeting subsystem
- Workarounds for these lack of support issues required the extensive use of “ifdefs” in the code (e.g. #ifdef REAL, #ifdef SIMULATION) which placed an additional coding burden on the student programmers who already had enough on their plates
- Testing of the simulation code requires a Linux environment which was not yet available on the laptops and desktops used by the 159 robotics team
- Several bugs and other problems exist in the current WPI simulation libraries (e.g. see section 3.5 below)

The code for simulation support therefore was developed as a separate project from the control code that was developed for the competition robot. While the robot competition code was written entirely by the students the simulation project coding was primarily mentor driven this year. To be frank, although significant progress has been made towards user-friendliness since they were first introduced last year the tools and documentation required to adequately test a robot hardware and software design in simulation are still a bit rough around the edges. Hopefully, these resources will be easier to adapt and employ before next year's competition and more student involvement will be possible.

3.1 Software paradigm

During this years build season much interest was expressed by the students in adopting the “Command” software paradigm in the future since it seems to have a better array of built in support features (Timers, button bindings etc.). It also appears to be the methodology most supported by First since all of the Demo simulation projects (PacGoat, GearsBot, etc.) are based on this coding style.

As a way to introduce students to the concepts involved the Command paradigm was chosen as the basis for the SimulatedRobot software project

3.2 Software Architecture

Further Description of the Command based Software Architecture

The Command software architecture is generally much more distributed than that of the Integrated architecture. As such, the main program entry class “Robot” contains very little code and is primarily responsible for constructing shared pointers to other objects (subsystem classes etc) that are used throughout the application. Typically, most of the project source files are arranged in two sub-

directories called “Commands” and “Subsystems”. At the top level (in the projects “src” directory) are Robot.cpp and OI.cpp. The OI class is unique to the Command paradigm and is used to assign joystick buttons and other controls to specific Command classes.

User Command classes must contain at least a Constructor and implement the following interface functions of the base class: Initialize(), Execute(),Finalize(), Interrupted() and End(). In Teleop mode the Initialize() function is typically called whenever a specific button is pressed following the bindings specified in the OI class

Once called, the Command will continue to run unless interrupted (e.g. because a different button has been pressed) which causes the program to exit through the Interrupted() function or some criteria specified in the “Finalize” method has been met (e.g. A PID controller has reached it's target). The End() function is available for cleanup operations and is called when the Finalize() function returns true. The Execute() function will be called once per command service cycle.

In the Command programming style task scheduling during Teleop and Autonomous modes is handled somewhat differently. For Autonomous mode, the usual procedure followed is to create a special Command class (actually a CommandGroup) which contains a set of other Commands that can be configured to run in parallel or sequentially. A typical sequence of lines in the Autonomous class file might be something like:

```
AddSequential(new OpenGate()); // pinch the ball
AddSequential(new DriveStraight(7,0)); // go forward
AddSequential(new Turn(-50)); // turn
AddSequential(new StepShooterAngle(36)); // set angle
AddSequential(new ShootBall()); // shoot
AddSequential(new FullStop()); // end autonomous
```

In Teleop mode, Commands are usually started up as a result of a particular action carried out in the interface (e.g. a Joystick button is pressed that is assigned to toggle a gate open or closed). A notable exception to this is that “Subsystem” classes can specify a default command that will run in parallel to other commands when Teleop mode is active. For example, the default command for a DriveTrain subsystem might be something like “TankDriveWithJoystick” which ensures that the operator will still be able to drive the robot even if other commands like “StepShooterAngle” are in progress.

SimulatedRobot Software Architecture

For instructional purposes, when designing the commands and subsystems used in the Command based SimulatedRobot project an attempt was made to try to keep as much of the functionality and structure as possible the same as that employed in the student's IntegratedRobot project. So there is also a separate class file for the “Holder”, “Loader”,”Shooter” and “DriveTrain” subsystems. The subsystem state machines that were coded for IntegratedRobot were also duplicated in a similar way but were implemented in the “Default” commands assigned to the Subsystems rather than in the Subsystem s themselves as was done in the Integrated project.

Following the guidelines of the Command based architecture operator actions such as Joystick button presses were mapped into Commands using the OI class rather than handling them explicitly in the Robot classe's TeleopPeriodic function.

3.3 Code resources

Source code is available for viewing or download from the “SimulatedRobot” project in the github Team159/2016-Robot-Code repository directory at:

https://github.com/FRCTeam159/2016-Robot-Code/SimulatedRobot

3.4 Workarounds for unsupported components

CAN Devices

A major deficiency in the current First/WPI simulation Tool-set is the lack of support for CANTalon motor controllers. As a result, these devices need to be emulated using older controllers such as Talons and Victors (which are supported in simulation). Unfortunately, the application programming interface (API) for the two types of controllers is significantly different since peripherals like encoders, PID controllers and limit switches are built-in features of the CANTalon class but need to be implemented externally for the other motor controller types.

As a workaround for this problem and in an attempt to keep the user code as much the same as possible when building a Debug (Robot deploy) or linux_simulate configuration of the c++ project a general purpose motor controller “superclass” (GPMotor) was created that encapsulated some of the most widely used functionality of the CANTalon class (like PID control and encoders) but implemented them using separate class member objects when the project is compiled with “#ifdef SIMULATE” active (i.e. when the linux_simulate configuration is built). For simulation, in both the Solidworks exporter and GPMotor class CAN bus and encoder id's need to be mapped into PWM channels.

Accelerometers

Since neither the Solidworks exporter or WPI simulation libraries currently offer support for accelerometers these needed to be implemented in the code using single axis gyros (through the AnalogGyro class)

Limit Switches

Limit switches are integrated through special functions of the CANTalon class but for simulation needed to be implemented using external DigitalInput(DIO) channels (this functionality was made part of the GPMotor class)

Optical Sensors

Infrared detectors and Lidar rangefinders were both used in the 2016 robot but because of lack of library and plugin support needed to be implemented in simulation using “Ultrasonic Sensors” and assigned to an “AnalogInput” channel

3.5 Patches for WPI library bugs

PIDController and Notifier classes

During the course of designing and testing the GPMotor class described above several problems were discovered when trying to implement external PIDControllers. These problems didn't seem to exist in last year's WPI library code which, along with the Solidworks exporter, went through an extensive revision in order to be compatible with a much newer version of the Gazebo simulator.

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

Talon and Victor classes

In the simulation build neither of the older motor controller classes return a non-zero value for the Get or GetSpeed functions (which was found to be important when trying to combine multiple PID controllers). 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

As a workaround, while waiting for WPI updates that have these problems corrected the simulation library had to be compiled manually from source (need to clone and pull allwpilib from github) and copied over to wpilib/lib in the (Linux) user's home directory.

Simulation Details

1 FRC “Stronghold” field

Early in the build season First made available a scaled Gazebo model for the 2016 “Stronghold” challenge field. This package also included models for all of the “Defense” field elements complete with articulation for gates, doors etc.

In order to construct a field adequate for simulation the various field elements (defenses) needed to be added to and positioned in the base field (which contained the towers, walls and platforms for the defenses). In addition, the exported Solidworks robot model had to be added to the scene. All this could be done using the direct editing features of the Gazebo simulator.

2 Special field elements

Ball

A model for the Ball (or “Boulder”) used in the competition was also provided by first. While the ball model had reasonable values for size, inertia and mass it was found that it didn’t provide adequate emulation for certain aspects that were needed to mimic the behavior expected by the robot simulation model. In particular, the ball originally acted as a simple hard sphere with no compression or friction

and so could not be gripped well by the rollers and gate arm in the holder and loader subsystems of the simulated robot.

Fortunately, the “sdf” format (a form of xml) provides options to set the frictional (mu,mu2) and deformation (kp,kd) properties of a model so that (although the end result was far from perfect) the “world” file could be hand-edited to give a more satisfactory simulation of real world behavior.

Defenses

When testing models in the FRC field a few minor problems were encountered, such as some of the defenses weren't exactly the right size to fit into the platforms. These problems were subsequently fixed by hand editing the model-field “.world” file

Some of the joint properties of certain of the defenses were also adjusted to give better emulation of real world behavior (by adding a little friction, damping etc.)

3 Launching a simulation

Although there is supposedly a plan in the works to provide Windows support in the future, simulations currently need to be carried out in a Linux (specifically Ubuntu) OS environment. This can be implemented either natively (as a grub boot partition) or as a Linux virtual machine running inside a host OS (e.g. Windows or Mac). Details on how to set up a virtual or native Linux environment with all the necessary support tools have been described in a previous report so wont be repeated here. However, 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

A procedure for running a simulation with this year's tools is:

1. Start up the Gazebo simulator and load the model-field “.world” file (e.g.)

```
export SWMODEL=$SWEXPORTS/2016-Robot_Exported
export GAZEBO_MODEL_PATH=$SWMODEL:${GAZEBO_MODEL_PATH}
gazebo --verbose $SWMODEL/2016-Robot-field.world &

note: for convenience, these commands can be placed inside an executable
shell script
```

2. Start up the Java-based simulated Driver Station

   > sim_ds

3. Start up the Eclipse simulation build “FRCUserProgram” either ..
   - though the Eclipse interface
     - “Run As.. WPILib C++ Simulation”
   - or from a command shell

     > cd ../SimulatedRobot/linux_simulate

> ./ FRCUserProgram

4. If using “Smart Dashboard” Start up the java based “sfx” jar file
    - java -jar ~/wpilib/tools/sfx.jar

Tests and Results

1.1 Tests and Observations

1. Traversing the Lowbar
    - It was first seen that when a ball was held in the “pinched” or gate open position the tip of the gate would often hit the bottom of the Lowbar when passing through that obstacle
    - One problem found was that the Lowbar simulation field object needed to be increased in size slightly to fit correctly on it's platform (and so leave the expected clearance). While the tolerances are still a bit too close for comfort the model then usually traversed the defense without collision after this change
    - A benefit of doing the simulation is that it pointed to a possible need to provide a better margin for clearance by (for example), lowering the shooter apparatus in the robot.
2. Side to Side frame articulation in negotiating defenses
    - Limited testing seemed to indicate that making this change to the frame may have helped in the model in traversing certain defenses such as the Ramparts. However, more testing would need to be done to verify that the effort and other compromises to the design to support this feature were justified
    - An easy way to test the effect of frame articulation in simulation would be to manually set the limits for the frame joint to zero in the model's sdf file (which would remove the ability of the joint to articulate) and then repeat the testing
3. 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)
4. Loading a ball
    - Modeling the polycord bands on the lifter was difficult to do since there isn't yet direct support for belts etc. in the current version of Gazebo. A reasonable substitute therefore was

to replace the cords with additional rollers (5 total) which we all driven at the same speed.

- In order to get the lifter to “roll the ball up over the bumper” the frictional and compressional properties of the rollers and the ball needed to be set appropriately.
- When testing the lifter on the physical robot it was found that the lifter needed to go between two angles (low then high) in a carefully timed sequence in order to keep the ball moving as it was being transferred into the holder
- In the simulator, better behavior was observed if a constant upward force was applied to the lifter arms when loading the ball. The amount of force needed wasn't enough to counteract the downward pull of gravity on the arms but was sufficient to move them up and out of the way when the ball was pushed forward against the bumpers and then upward by the rollers.
- This result suggests that it would have been interesting to try adding a constant force spring to the lifter arms on the physical robot to see if the ball loading behavior improved

5. Shooting
    - The shooter in the simulated robot was able to model the required behavior of the subsystem fairly accurately. In the end, it was possible to pinch the ball in the holder with the gate lever, raise the angle to the shooter to a high angle (>50degrees), turn the flywheels on until they reached a specified speed (using a PID loop) and then launch the ball by turning on the push wheel.
    - One issue that was seen in simulation that wasn't observed in the physical robot was that it was more difficult to keep the ball from slipping out of the gate when the shooter angle was raised. This problem was reduced by increasing the friction between the joints in contact and also by applying a slight forward rotation (proportional to the angle) to the push wheel
    - The likely cause of this gripping problem is that the default physics engine used in Gazebo does not take into account the increase in surface area that occurs when an object is compressed but rather models the contact between the ball and roller as a single point (which provides an axis that may cause the ball to “spin out” if too much force is applied).
    - It's possible to attach different physics engines when starting up Gazebo that may better model soft contact behavior, although that wasn't tried (But would be a good area to explore in the future)
6. Using the lifter to traverse the Cheval De Frise
    - By lowering the lifter arms onto the middle ramp and one of the side ramps it was possible to get the simulated robot to traverse the Cheval de Frise.
    - However, maneuvering the robot chassis and arms into position proved to be tricky and time consuming in the simulator.

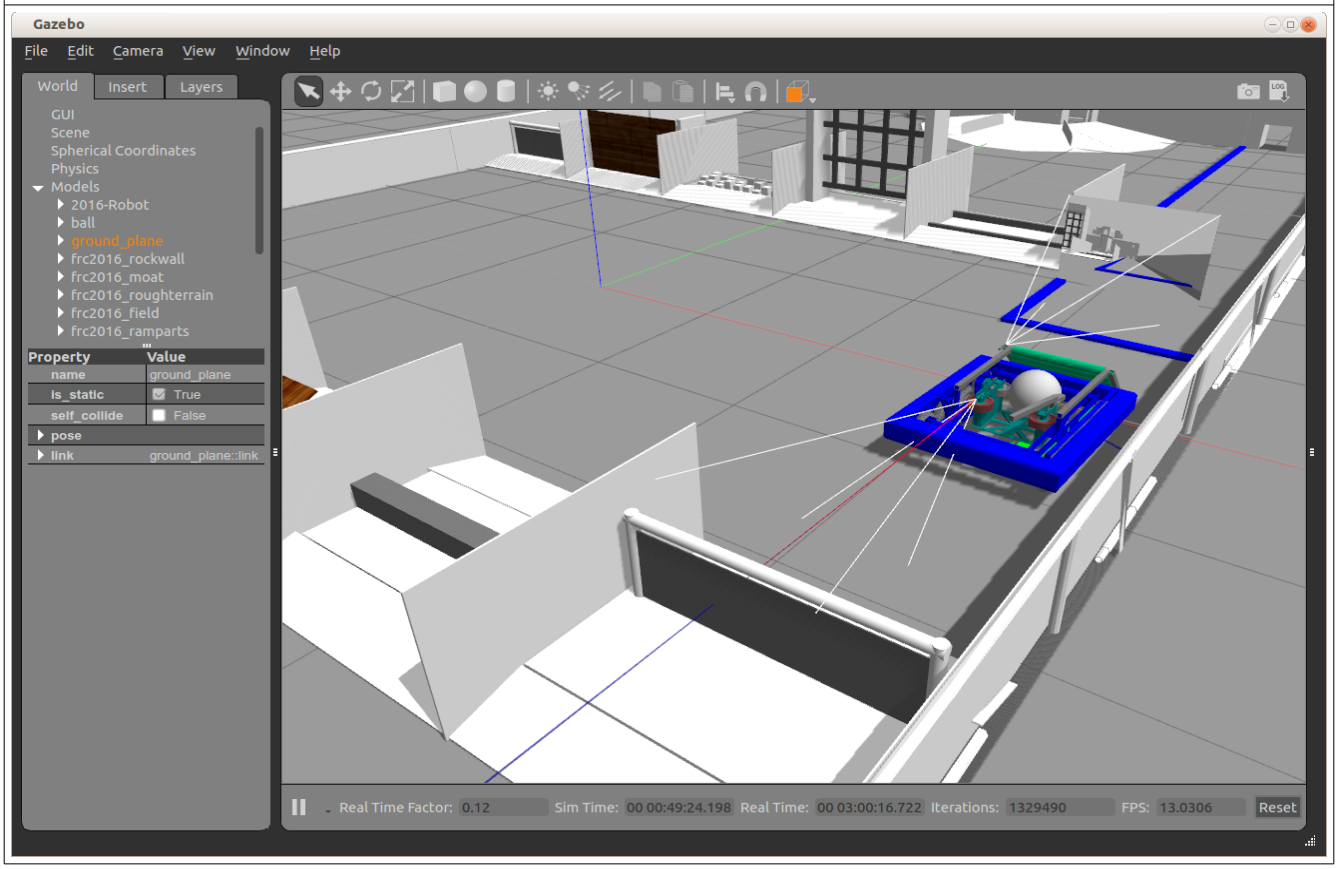
- This experience suggests that although doable, this obstacle would probably be best left alone as a strategy tactic (at least with the current design)
- As seen in the real robot, attacking the Cheval head on resulted in the center ramp springing back and getting caught up at the back of the frame under the shooter.

7. Using the lifter to traverse the Portcullis
    - Surprisingly, in the simulator at least, it was possible to traverse this obstacle quite easily by simply driving in reverse with the loader arms lowered. The Portcullis gate was lifted as it rode up on the rollers and the robot was able to drive though without it falling down and catching on the chassis or shooter
    - If there had been sufficient time for driver practice it would have been interesting to have tried this strategy in the practice field or on the Portcullis field element that was constructed in the shop
8. Autonomous mode (Lowbar breach and high goal shot)
    - As described above, an Autonomous Policy was written using the CommandGroup class that was designed to traverse the Lowbar and shoot a ball into the high goal of the opposing tower. After considerable tweaking of distances, angles etc. it was possible to score a goal about a 3rd of the time this way.
    - Since no auto-targeting was employed everything was strictly done by dead-reckoning, so accuracy would have probably been much better had it been possible to get the shooting and aiming functionality operational in the competition robot.
9. 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 the Integrated robot software this view switch was carried out automatically when a button on the Joystick was pressed but this was not duplicated in simulation
    - Viewing a camera window “Full screen” when driving in the simulator might be a good way to practice using a camera in the DriverStation interface (since it doesn't allow “cheating”)
    - A serious problem that was found with Gazebo camera views was that unless the computer system has a decent graphics card and the simulator is run in a native Linux environment (i.e. not in a virtual machine) the objects in the scene do not occlude correctly. This has all

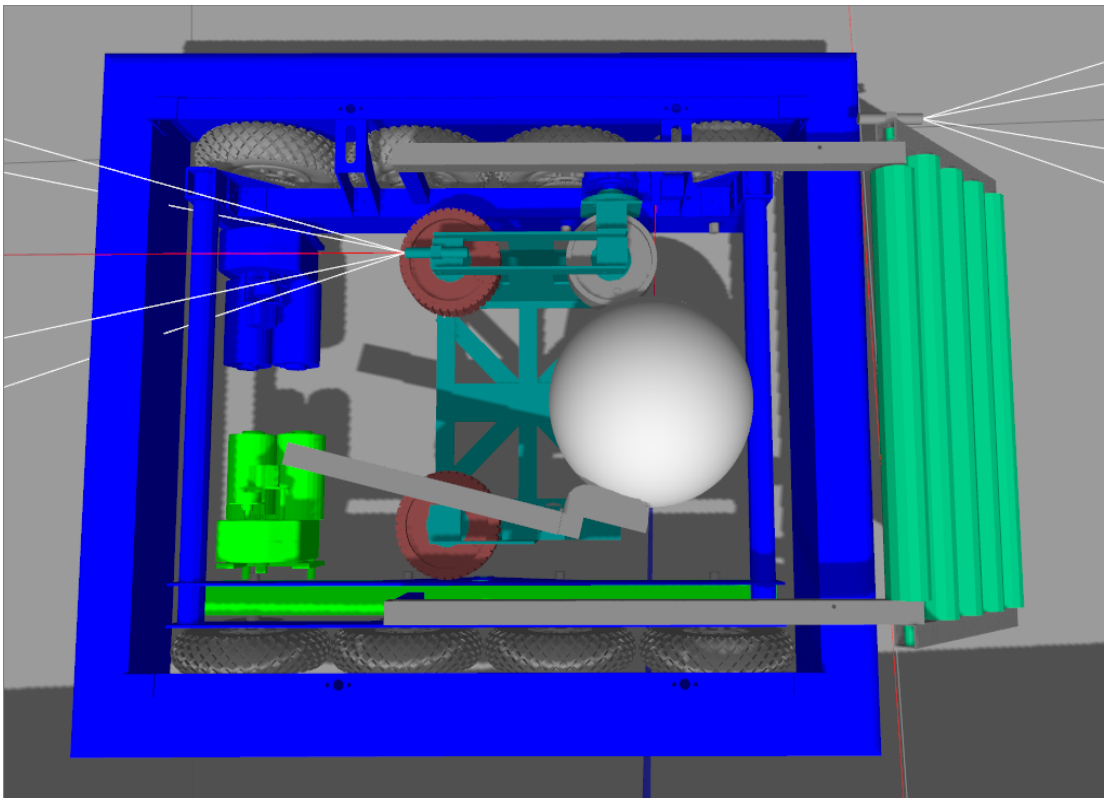
the symptoms of a situation where the off-screen rendering of the camera images is not including a depth buffer.

1.2 Screenshots

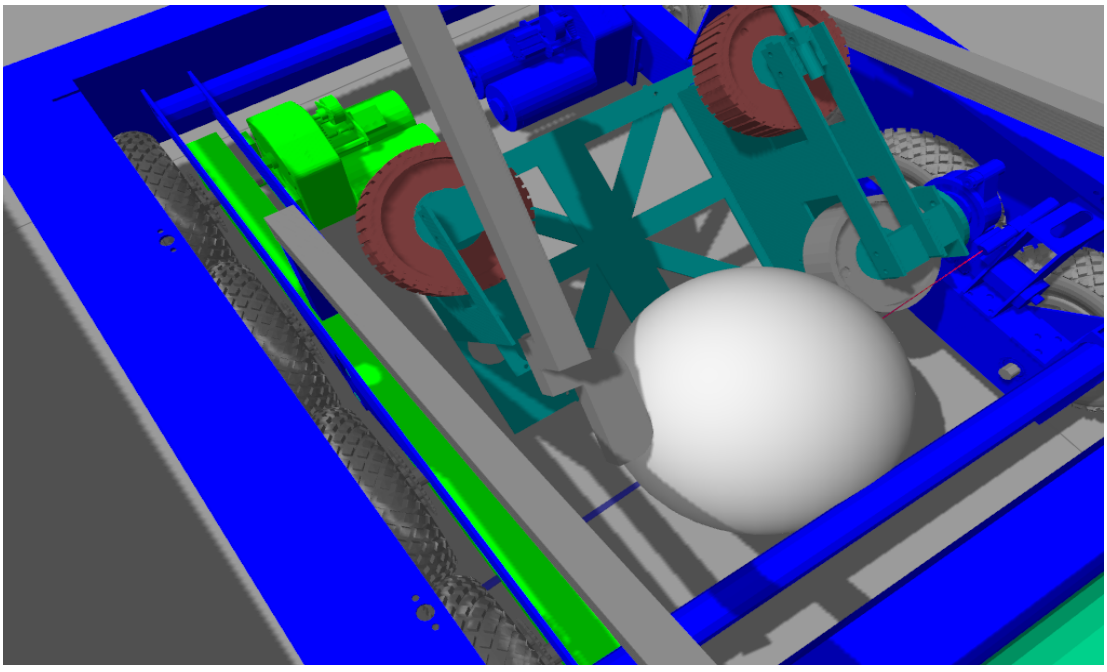
FRC 2016 field with Robot (“Artemus”)



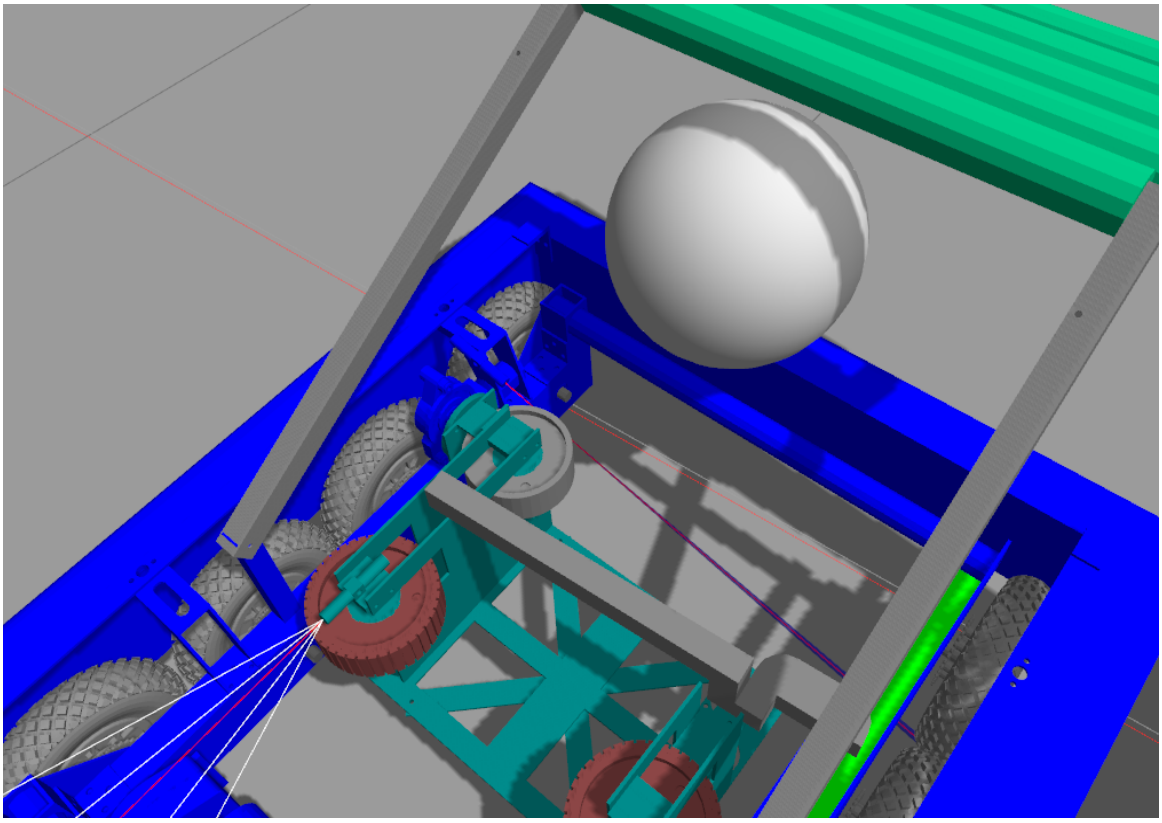
Robot Model in Gazebo (Top view)



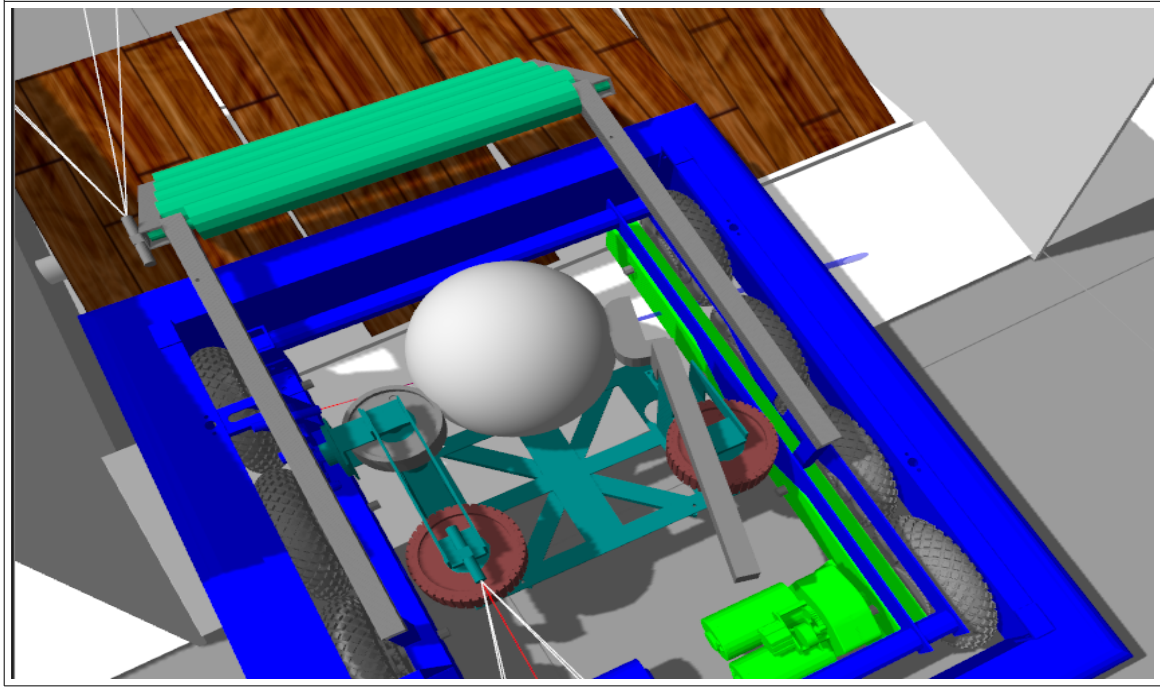
Shooter at 40 degree angle ready to fire



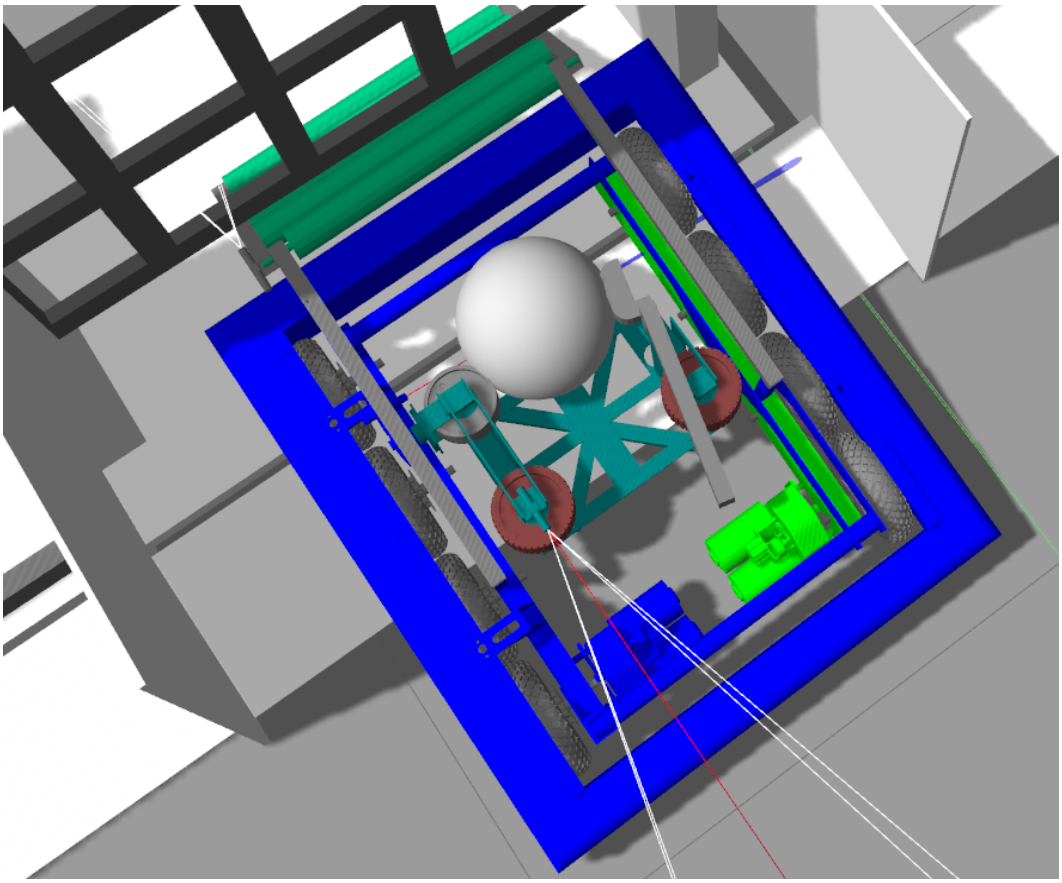
Loading a ball



Traversing the Cheval De Frise



Traversing the Portcullis



Looking Forward

1.1 WPI Simulation Library

1. Debugging and Testing Involvement

   In looking at the TeamForge website and noticing the small number of people that seem to be involved with programming at WPI and First it seems clear that any help that could be provided from the outside (mentors, students or others) to improve the feature set and behavior of the simulation software would be appreciated. Possible ways to assist might be to:

   1. Evaluate Simulation Tools and submit Bug Reports and Feature requests through the TeamForge website
   2. Help WPI debug problems by tracking down the source of coding errors etc. by building and testing the executables generated:
      - git clone https://github.com/robotpy/allwpilib
      - follow instruction in README.md on how to build the various projects

2. Prototype adding support for CAN devices in simulation

   At present, one of the biggest bottlenecks to the wider use of FRC simulation tools is probably the lack of support for CANTalon devices in the WPI Simulation library. This deficiency prevents students from being able to compile their robot code to run in the simulator directly without resorting to programming constructs like “ifdefs” and the “GPMotor” class described above.

   1. It may be possible to develop a wrapper class for CANTalons in the simulation “allwpilib“ source code tree and adapt the build process to generate a custom library that includes support for these devices.
   2. If successful, this prototype could then be offered to WPI as a starting point for integration into their code base that would be made available to all First teams (hopefully, prior to next year's build season)
   3. At a minimum, the library could be made available on the Team159 github website for First teams that wanted to get a head start in experimenting with simulation

1.2 Vision Processing

As alluded to previously, one of this year's programming goals was to develop a vision based auto-targeting system that could be used to adjust the position of the robot chassis, shooter angle and fly-wheel speed to accurately shoot a ball into the “high goal” windows of the towers on the opposing side of the field. The software for this was written and tested on a prototype system but, unfortunately, because of problems encountered in the shooter angle adjuster it was not possible to implement this feature during actual competition.

In addition, the “IMAQ” software libraries that were employed in this effort may not be ideal as a long term solution because of the following reasons:

- Since there is currently no simulation support for this library, a fully operational robot (with camera hardware) is required in order to test and develop image processing and targeting software
- There is a question of whether or not the IMAQ software running on the RoboRio would be fast enough to be adequate for real-time targeting in Teleop mode (it wasn't possible to test this adequately but there is anecdotal evidence that other teams may have rejected this approach for this reason)
- The API doesn't seem to support true object recognition and classification using training examples but is more of an image processing package and requires that the user software analyze features in regions of an image to recognize target objects etc. The IMAQ package does however, provide some tools that make this easier to do (such as particle identification and moment analysis) which might be adequate for locating simple objects but perhaps not ideal as

a way to analyze more complex scenes.

A possible project that could involve and engage both mentors and students during the off-season might be to look into other image processing options. Ideally, the image analysis package chosen should be able to work both in the Gazebo simulator and on actual hardware. Some possible candidates for this worth looking into are:

- OpenCV
    - An open source vision processing API that includes support for both Arm and Linux operating systems
- CUDA (Digits, OpenDNN)
    - Supported only on systems that have NVIDIA hardware (graphics processors)
    - However, NVIDIA also sells a small GPU board (Jetson) for a reasonable price (<$200) that other FRC teams have successfully used for image processing purposes