EZ-RASSOR

2018 - 2019 Senior Design Project

EZ-RASSOR | Team Members

Ron Marrero	Chris Taliaferro
Sean Rapp	Camilo Lozano
Samuel Lewis	Lucas Gonzalez
Harrison Black	Cameron Taylor
Tiger Sachse	Tyler Duncan

EZ-RASSOR | Background

- The Regolith Advanced Surface Systems Operations Robot (RASSOR) Excavator
- Engineered around “Dust to Thrust”
- Mines regolith on other planets and dumps it at a processing facility
- Controllable via teleoperations or autonomous routines
- Equipped with rudimentary sensors and stereo cameras for situational awareness

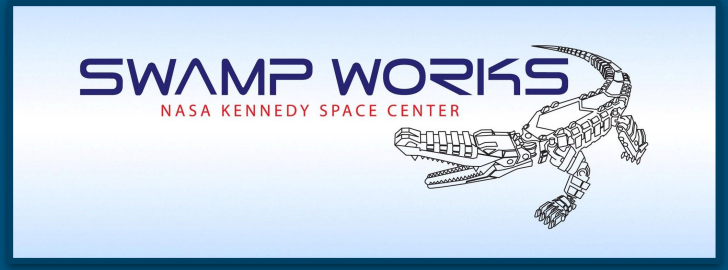
EZ-RASSOR | What is the Problem?

The RASSOR is **too large**, **too complicated**, **and too expensive** to produce and distribute to universities and organizations for research and education.

EZ-RASSOR | Who Will this Help?

The EZ-RASSOR solves this problem by being **easily, cheaply, and quickly distributable** to universities and organizations.

EZ-RASSOR | Introduction



- A combined effort between UCF and NASA SwampWorks
- Main purpose is to recreate the RASSOR using only open source technologies
- Immediately utilized by Kennedy Space Center as a demo unit
- Scoped as a multi-year project by the Florida Space Institute
- Designed for long-term prototyping and fine-tuning of computational requirements by other universities and organizations

EZ-RASSOR | Requirements

The UCF development team was tasked with implementing the following requirements:

- Design a fully functional, reproducible simulation and model
- Create an entire ROS architecture that is distributable and deployable
- Control real hardware with the same architecture
- Control the system autonomously and with a mobile app/gamepad
- Use robot vision to maintain situational awareness
- Monitor the system with a desktop application

EZ-RASSOR | Challenges

- All software has to be open source
- Any simulation has to closely mimic the real world and the real RASSOR
- The EZ-RASSOR’s hardware is underpowered, compared to the full version
- The system must be deployable on hardware that is usually inaccessible
- The team is large and diverse in abilities and strengths

Administrative Work

Sam Lewis

Administration | Team Breakdown

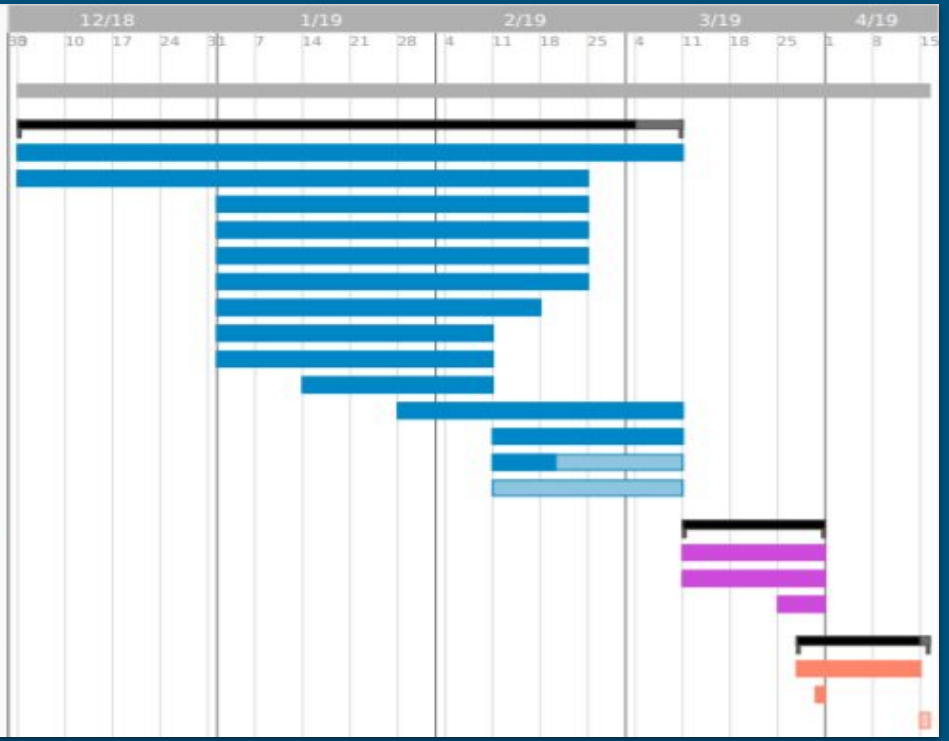
ROS	*Gazebo*	*Interfaces*	*Autonomy*
Harrison Black	**Ron Marrero**	**Chris Taliaferro**	**Cameron Taylor**
Tiger Sachse	Sean Rapp	Camilo Lozano	Tyler Duncan
	Harrison Black	Lucas Gonzalez	Sam Lewis
		Sean Rapp	Chris Taliaferro
			Harrison Black

Administration | Project Budget

Item	***Price***
EZRC v1	$120
EZRC v2	$100
Batteries	$30
Printing Costs	$320
NASA Polos	$300
Total	$870

Administration | Gantt Chart

NASA-E-RASSOR	start	end	94%
Development	**12/03/18**	**03/08/19**	**93%**
Desktop GUI	12/03	03/08	100%
RC Car Full Motor Functions	12/03	02/22	100%
Standard Configuration	01/01	02/22	100%
Installation Scripts	01/01	02/22	100%
Backend Communication with robot	01/01	02/22	100%
Controller Keyboard Support	01/01	02/22	100%
Functional Sim robot	01/01	02/15	100%
Custom Designed Robot	01/01	02/08	100%
Mobile App	01/01	02/08	100%
Functional Gazebo Environments	01/14	02/08	100%
Object Detection	01/28	03/08	100%
Obstacle Avoidance	02/11	03/08	100%
(Stretch Goal) Swarm AI	02/11	03/08	33%
(Stretch Goal) Orbital GPS	02/11	03/08	0%
Testing	**03/11/19**	**03/29/19**	**100%**
Full ROS Node Test	03/11	03/29	100%
Full Auto AI functions Test	03/11	03/29	100%
All Installations Script Test	03/25	03/29	100%
Delivery	**03/27/19**	**04/15/19**	**93%**
Presentation Preparation	03/27	04/14	100%
Stem Day	03/29	03/29	100%
Senior Design Showcase	04/15	04/15	0%



Administration | Work Distribution

	Harrison	Tyler	Sam	Camilo	Lucas	Ron	Tiger	Sean	Chris	Cameron
Simulation	X					X		X		X
Phone Application				X					X	
Swarm Functionality										X
Desktop Dashboard			X		X			X	X	
Robot Vision		X	X						X	X
Basic Motor Functions	X						X			
ROS Architecture	X			X			X			
General Auto Functions	X	X		X						X
EZRC							X			
Scripts / Installation							X			
Documentation			X		X	X				

Team ROS:
Staying on Topic

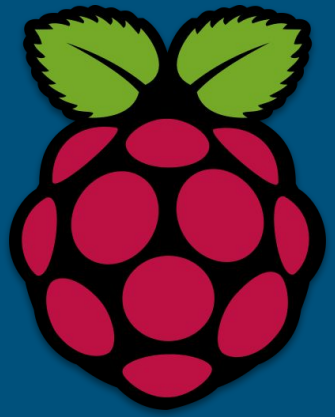
Harrison Black
Tiger Sachse

Team ROS | Requirements

- Interpret movement and autonomous commands from an app or gamepad, then publish these commands to the hardware/simulation/autonomous control
- Switch between manual and autonomous control when appropriate
- Create a modular design that is easily changed and configured
- Maintain a healthy and conventional package structure

Team ROS | Technologies Used

- Ubuntu Xenial 16.04 and Bionic 18.04
- ROS Kinetic and Melodic
- Twist Messaging
- Multiprocessing
- Raspberry Pi
- Python 2



Team ROS | What is ROS?

- The Robot Operating System
  - Not *that* type of operating system
- A framework of nodes (small programs) that communicate
- Nodes can publish to send data to a topic (like a queue or a forum thread)
- Nodes can subscribe to topics to grab data
- Multiple nodes can interact with a single topic
- Indirect communication makes for a safer system
- All nodes are controlled by a ROS master node

Team ROS | What is Twist?

- Twist is comprised of 2 vectors
- Linear velocity vector for straight line movement
- Angular velocity vector for rotational movement
- Each vector has 3 floats corresponding to (x, y, z) coordinates

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Team ROS | Challenges

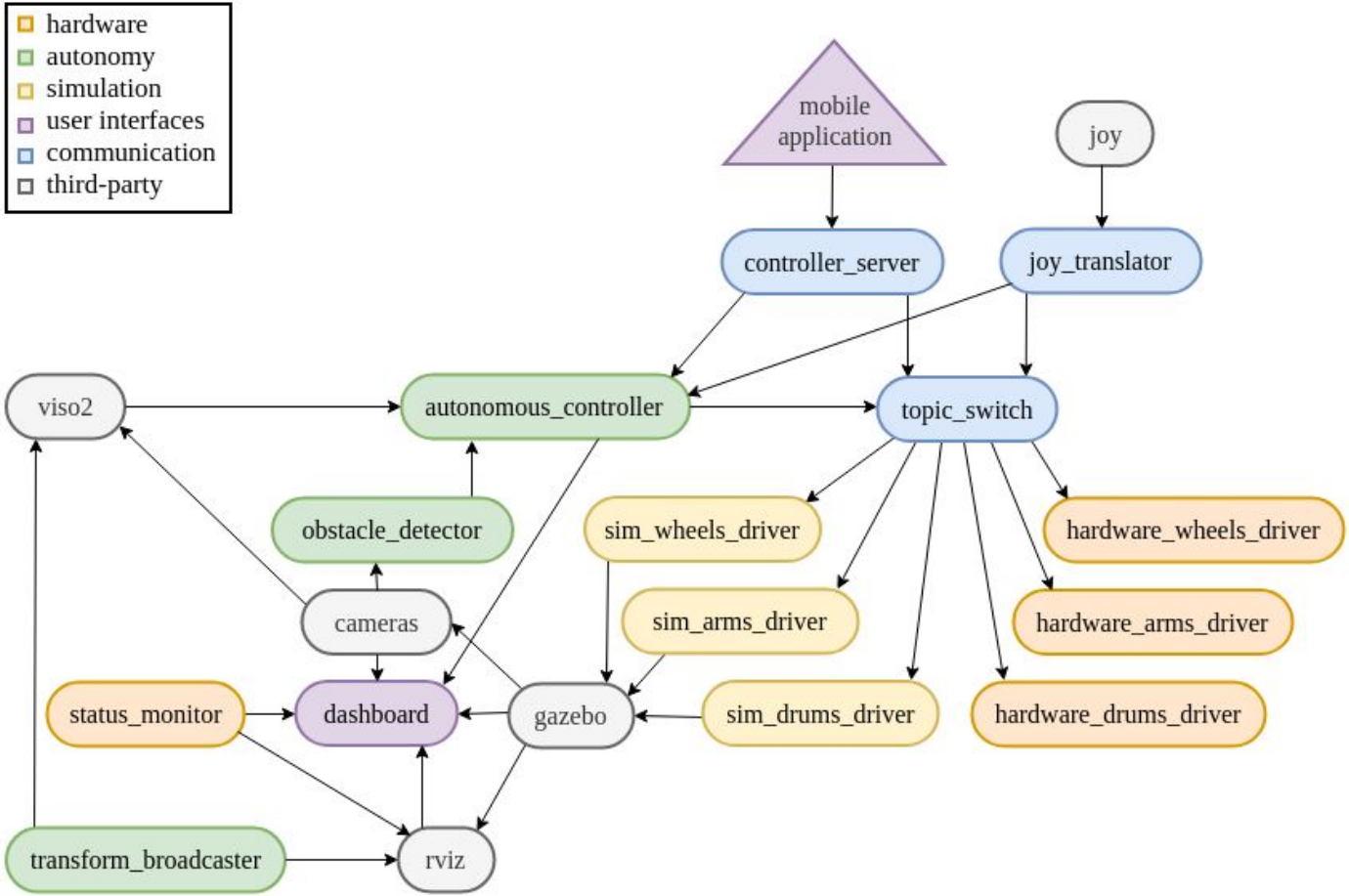
- ROS is parallelized, which complicates the architecture and makes it harder to properly debug
- ROS only works (officially) with Python 2
- The ROS build system (Catkin) is complex
- The system is difficult to test with unit tests, and instead requires integration tests
- What data type do we use?

Team ROS | Bitstring or Twist?

- Three iterations of how we send data around the system
- Command array --> bitstring --> Twist
- Efficiency vs. convention
- It’s 2019, not 1985

Team ROS | Results

- The ROS system is controllable via a mobile application, gamepad, or an autonomous controller
- It can switch between manual control and autonomous control at will
- The system properly instructs the simulation and hardware to move
- It is automatically configured by launch files and can also be manually manipulated with parameters
- All packages are independent and installable


hardware
autonomy
simulation
user interfaces
communication
third-party
mobile application
joy
controller_server
joy_translator
viso2
autonomous_controller
topic_switch
obstacle_detector
sim_wheels_driver
hardware_wheels_driver
cameras
sim_arms_driver
hardware_arms_driver
status_monitor
dashboard
gazebo
sim_drums_driver
hardware_drums_driver
transform_broadcaster
rviz

Team ROS | Integration Testing

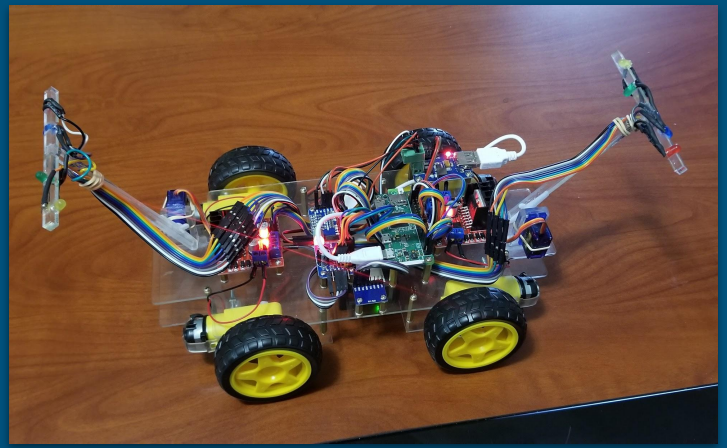
- Both the topic switch and controller server (key portions of the communication system in the ROS graph) are integration-testable
- The tests confirm that the nodes operate nominally under normal conditions

Team ROS | Scripts

- This project includes many scripts that automate annoying/overly complicated processes related to the project
- We include scripts to install the project from scratch, build the project from source, configure a Catkin workspace, wrap a few other ROS operations (like testing) and more
- These scripts make it easier for people with limited technical knowledge of the system to use our software
- Our target audience is made up of students and KSC guests, so this usability is crucial

Team ROS | What is the EZRC?

- An RC car capable of running our main communication ROS graph
- Has arms, simulated drums, and wheels
- Controllable with the mobile app or a gamepad
- Acts as a Wi-Fi hotspot that allows the mobile app to connect directly
- Code removed prior to final release due to completion of real EZ-RASSOR



Team ROS | Why Build an RC Car?

- Demonstrates the team's ROS graph in a real-world environment (complements the simulation)
- Provides additional testing ground for interfaces (mobile app, desktop dashboard)
- Much less expensive to replace if something goes wrong during testing (compared to the EZ-RASSOR)
- An essential backup in case the EZ-RASSOR hardware was delayed beyond the term of this project

Team ROS | Next Steps

- Migrate to ROS 2 and Python 3
- Finalize launch files for entire system
- Write broader, end-to-end integration tests
- Finish single-command installation process

Team Gazebo:
Everything is just a Simulation

Ron Marrero
Sean Rapp

Team Gazebo | Requirements

- Create a 3D simulation model of the EZ-RASSOR
- Create three simulation test environments with accurate physics
  - Lunar environment
  - Obstacle course Earth environment
  - Empty, grey world
- Implement cameras and sensors for the EZ-RASSOR that stream real-time camera feeds
- Control the simulation with the ROS infrastructure

Team Gazebo | Technologies Used

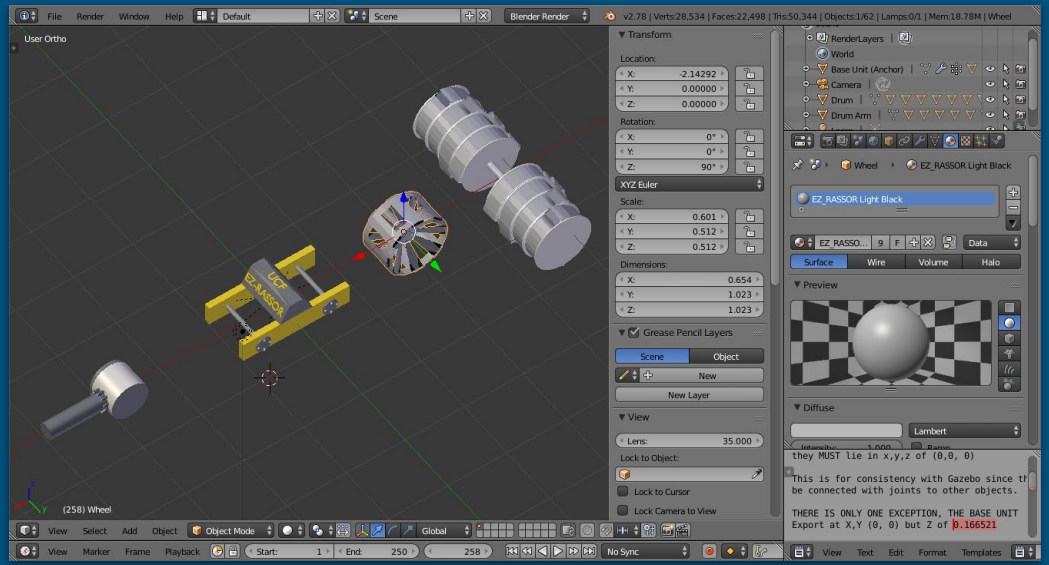
- ROS
- Gazebo
  - Completely open source
  - A fully integrated component of ROS
  - Building with Ubuntu 16.04 and 18.04, Gazebo 7 and 9
  - Server component allows remote viewing of the simulation environment
- Blender
  - Open source
  - 3D modelling tool
  - Creation of visual models for EZ-RASSOR

Team Gazebo | Challenges

- Gazebo (our simulation software) accepts a variety of robot file formats
  - URDF, XACRO, SDF formats are accepted
  - Each file format has its own pros/cons
- Simulated robot movement is complex
  - Physics properties such as inertia and mass must be carefully calibrated to create an accurate simulation
  - Incorrect properties make the robot react in weird ways

Team Gazebo | Making the Model

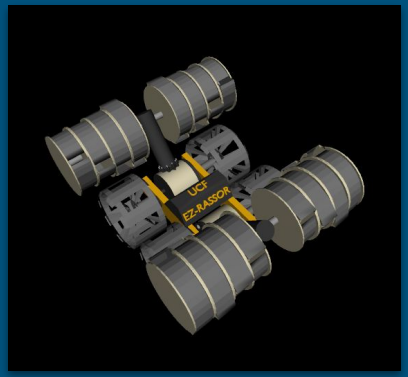
- Drums
  - Consist of 4 blades each
  - 2 drums, 1 attached to each drum arm
- Base Link
  - Connects to drum arms



Team Gazebo | Making the Environments

- Integrating 3D models
  - EZ-RASSOR:
    - Blender
  - Worlds:
    - Custom-designed worlds
    - Digital elevation model
- Environmental properties
  - Acceleration due to gravity
  - Friction with the surface

Team Gazebo | Results



- Three fully functional environments are finished
- The Gazebo environment is fully set up with ROS and is launched directly with launch files
- A custom Blender model has been created that accurately represents the EZ-RASSOR
  - This model is controllable via our ROS graph
- Multiple robots can be spawned in a single simulation and controlled independently

Team Interfaces:
Connecting Everything Together

Chris Taliaferro
Camilo Lozano
Sean Rapp
Lucas Gonzalez

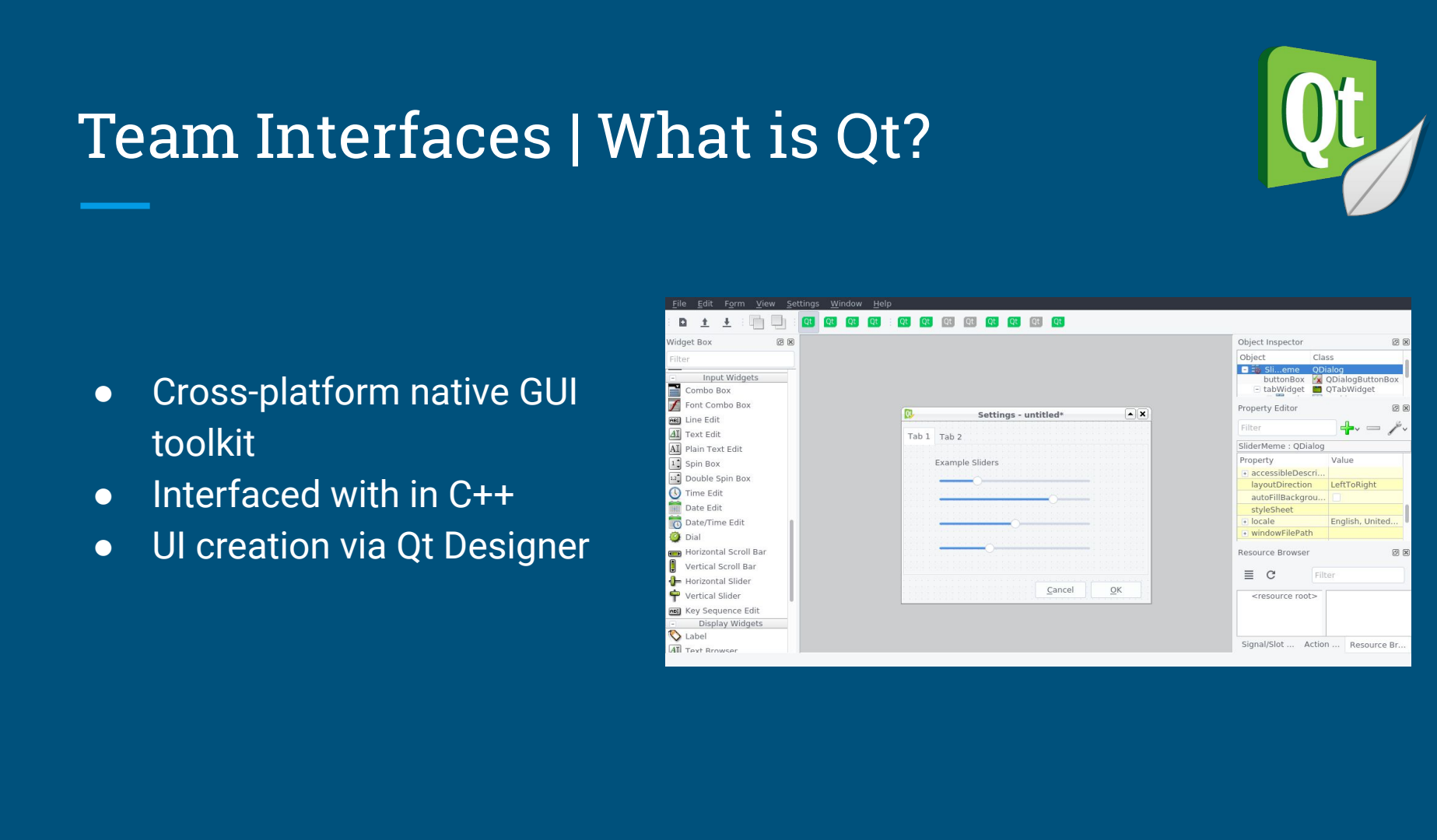
Team Interfaces | Requirements

- Create a mobile application that controls the whole system
  - Control the wheels, arms, and drums
  - Toggle the autonomous routines
  - Connect to multiple different EZ-RASSORs
- Create a desktop application that monitors the whole system
  - Launch individual launch files
  - Show system statistics
  - Show robot pose, camera views, and disparity map

Team Interfaces | Technologies Used

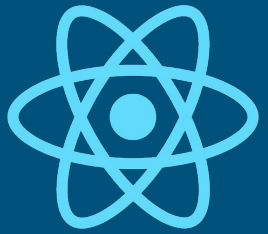
- Qt and Qt Designer
- React Native & Expo CLI
- HTTP Server

Team Interfaces | What is Qt?



- Cross-platform native GUI toolkit
- Interfaced with in C++
- UI creation via Qt Designer

Team Interfaces | What is React Native?



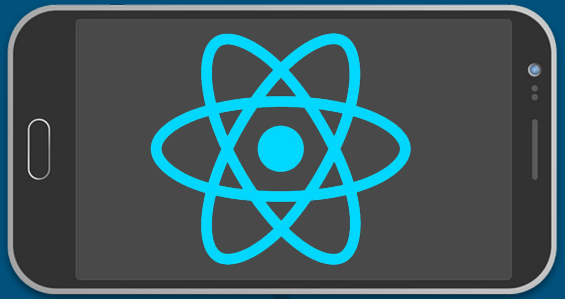
- An open source mobile application development framework
  - Maintained by Facebook and the community
- Based on JavaScript
- Backend web framework
- Enables rapid deployment
- Cross-platform (single code base)
  - Android / iOS / UWP
  - Most popular framework for cross-platform

Team Interfaces | Challenges

- Create an application that is beautiful and performs well
- Multitouch in React Native is finicky
- Network with the EZ-RASSOR without additionally hardware (like a router)
- Qt and ROS integration

Team Interfaces | Mobile App Results

- Fully functional and *beautiful* mobile application
  - Works on both iOS and Android
- Controls the entire system with no router needed
- Supports multiple EZ-RASSORs on a network



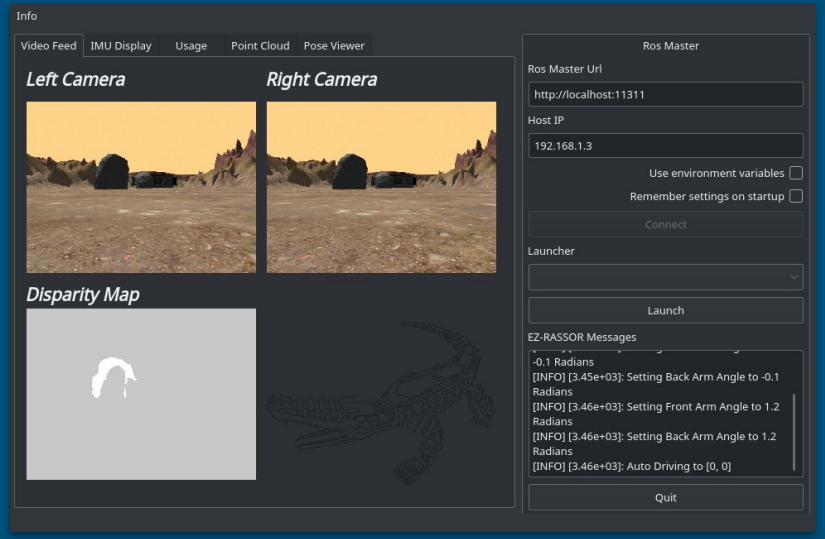
http://

```
handleSubmit(event){

  url = 'http://'+this.state.ip+this.state.endpoint
  console.log(url)

  return fetch(
    url,
    {
      headers: {"Content-Type":"text/plain; charset=utf-8"},
      method: 'POST',
      headers:{
        Accept: 'application/json',
      },
      body: event.toString()
    }
  )
  .catch((error) => {
    alert("Unable to connect to EZ-RASSOR");
    console.log(error);
  });
}
```

```
def do_POST(self):
    """Handle all POST requests."""
    self.send_response(200)
    self.send_header("Content-Type", "application/json")
    self.end_headers()
    content_length = int(self.headers.getheader("content-length", 0))
    instructions = json.loads(
        self.rfile.read(
            content_length,
        ),
    )
    self.wfile.write(json.dumps({"status" : 200}))
    self._publish_instructions(instructions)
```

Team Interfaces | Desktop App Results

- Created a desktop application capable of…
  - launching simulations and ROS network
  - connecting to any ROS master
  - viewing video feeds of left and right cameras
  - viewing disparity map data
  - viewing point cloud data
  - viewing status messages
  - viewing IMU data
  - viewing processing and memory usage
  - viewing the current pose of the robot



Team Interfaces | Next Steps

- Pass arguments to the launcher
- Provide a dropdown list of currently running EZ-RASSORs
- Connect to different online EZ-RASSORs on the fly
- View the autonomously-generated map within a window

Team Autonomy:
Avoiding Obstacles

Cameron Taylor
Tyler Duncan
Sam Lewis

Team Autonomy | Requirements

- Gazebo Setup
  - Setup Viso2_ros, OpenCV, ROS-Navigation, and other packages to work with Gazebo
  - Add and calibrate AI specific model features such as encoders and cameras
- Auto-Functions
  - Auto-Drive, Auto-Dig, Auto-Dump, Auto-Dock, Auto-Self-Right
  - These should function in isolation performing a single task accounting for environmental features
- Obstacle Avoidance
  - Generate disparity maps to detect obstacles in the robot's path
  - Generate warnings when an obstacle is ahead with best action to avoid the obstacle

Team Autonomy | Technologies Used

- ROS
- Viso2
- OpenCV
- Python 2
- ROS Navigation Stack

Team Autonomy | What is Viso2?

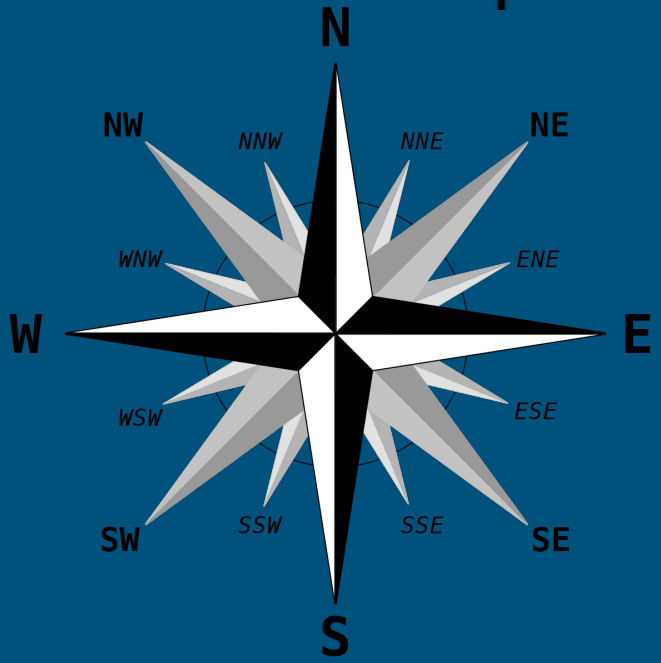
- A pre-built ROS package that handles visual odometry
- Utilizes stereo or monocular camera feeds
- Outputs location and rotation quaternion
- Provides usable and lightweight self localization with increased reliability compared with LSD-SLAM (still somewhat problematic)

Team Autonomy | What is the ROS Nav Stack?

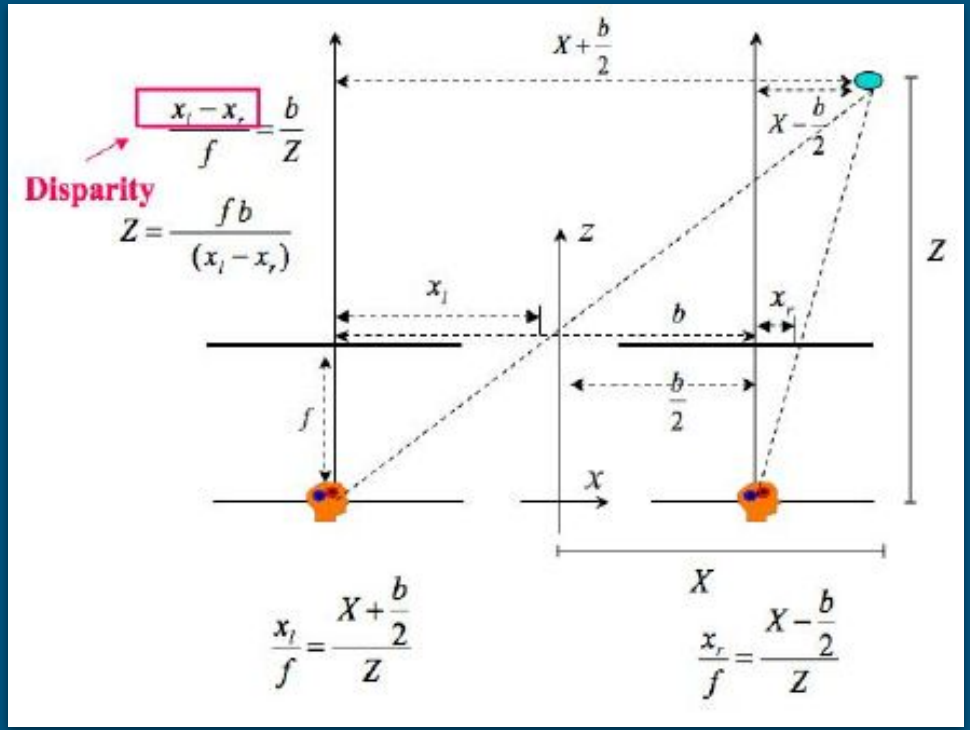
- Powerful super package available for ROS
- Handles navigation, mapping, obstacle avoidance using a variety of possible robotic sensors
- Packages for creating 2d costmaps and calculating paths
- Full stack outputs twist messages for movement but does not consider other joints such as arms and drums
- Can be difficult to implement in conjunction with controller for arms and drums in a fully autonomous system.

Team Autonomy | Navigation

- Calculates the current heading and heading needed to navigate to next target position
- Rotates closest direction in order to align with new target heading
- Recalculates heading when path is interrupted by obstacle
- Realigns from any location assuming correct localization from visual odometry



Team Autonomy | Stereo Vision

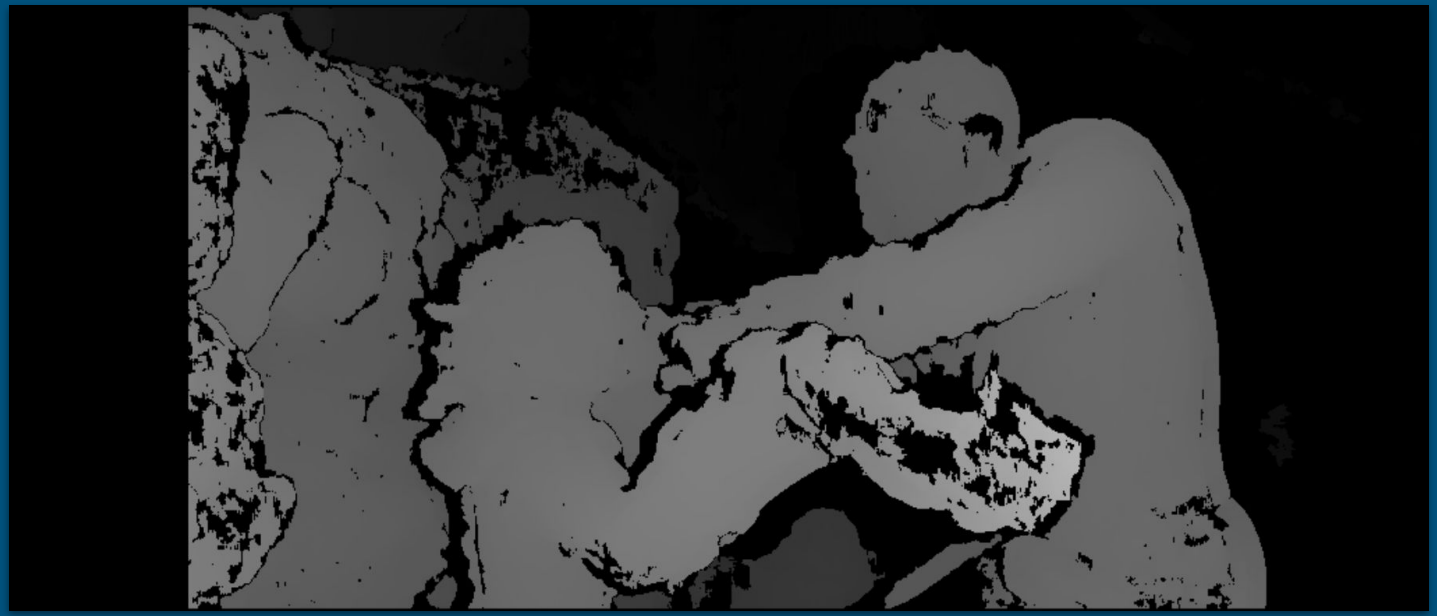


- Use epipolar geometry and stereo matching techniques to calculate disparity and distance to object
- Estimate values for every pixel in image
- Down sampling distance matrix

Team Autonomy | Disparity Map Generation

Stereo Images

Team Autonomy | Disparity Map Generation

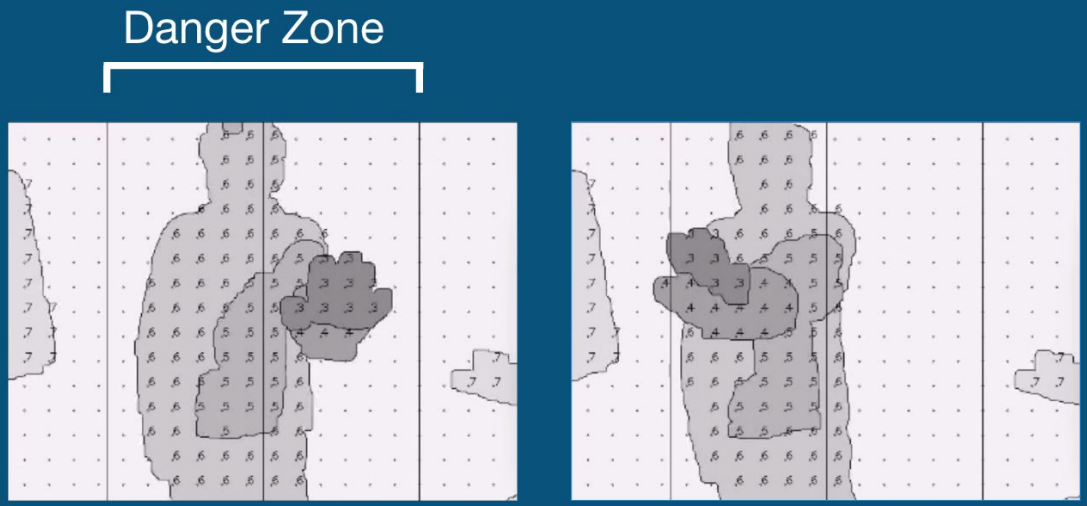


Raw Disparity Map

Team Autonomy | Disparity Map Generation

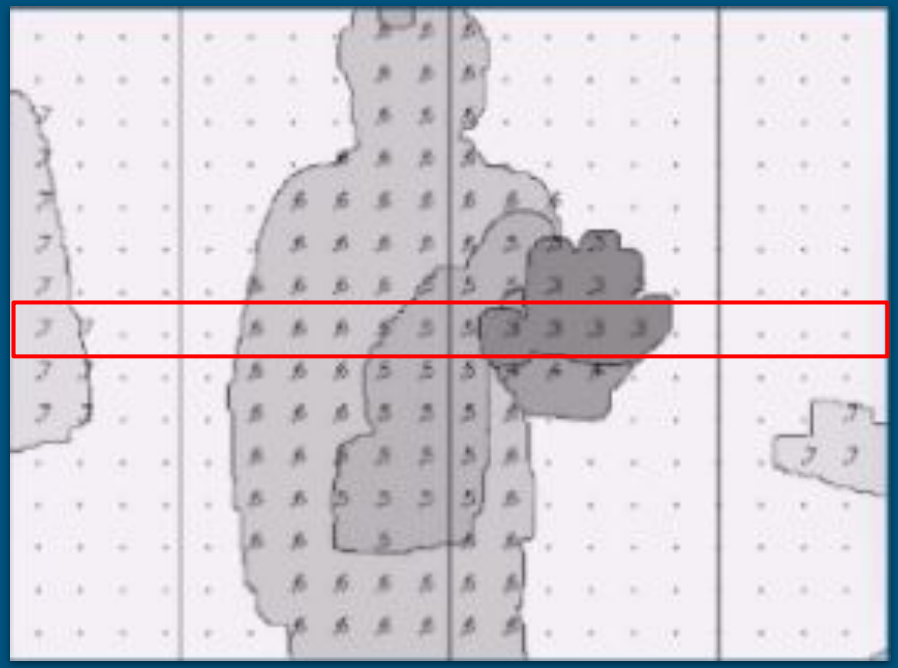
Filter Applied

Team Autonomy | Obstacle Avoidance



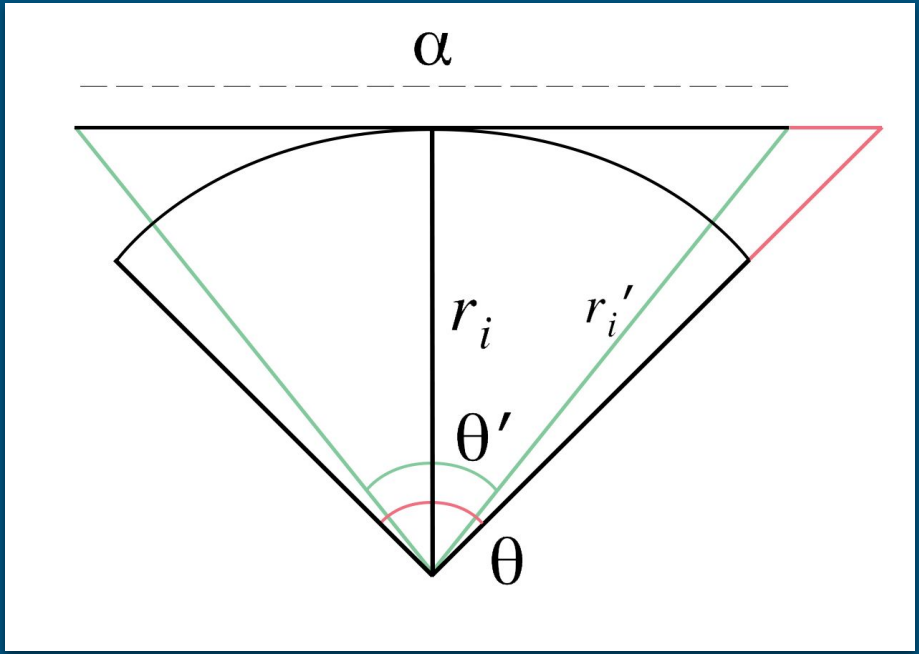
- Divide distance matrix into columns
- Set distance thresholds to assign appropriate maneuvers

Team Autonomy | Depth Map to LaserScan



- After mean pooling and down sampling the resulting image resolution is 80x60
- The field of view of the stereo camera system also happens to be 80 degrees along the horizontal
- Each pixel value along any arbitrary row in the image represents a distance to a point from the robot
- We assign each pixel to a single degree in the artificial laser scan reading
- ISSUE: Since the laser scan assumes data being read in is collected from a device rotating around a central axis all values are then offset

Team Autonomy | Depth Map to LaserScan



- We correct this by applying an counter offset to each pixel value
- The offset amount increases as the values diverge from the center of the reading

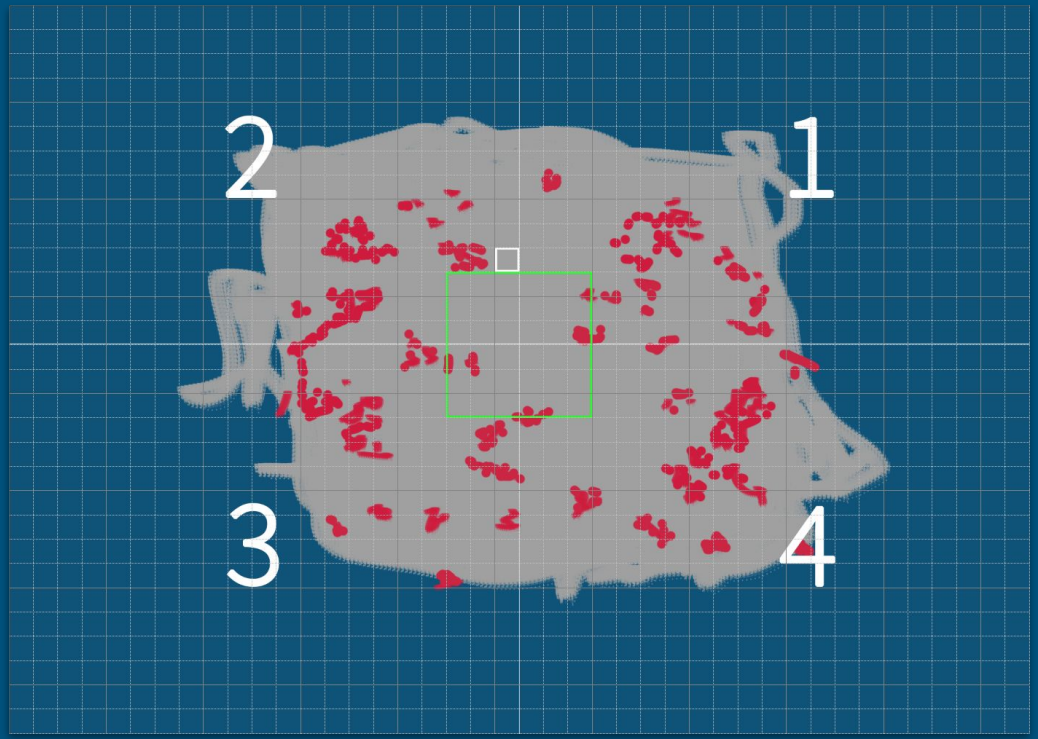
$$\forall\, r_i \in L \mid L \in \mathbb{R}^n, \text{ let } \alpha = \theta' r_i, \text{ and } \theta' = 2tan^{-1}\left(\frac{FOV_{rad}}{2}\right)$$

$$\text{Then, } r_i' = \sqrt{\tfrac{1}{4}\alpha^2 + r_i^2}$$

$$\theta'_{new} = \theta'_{old} - \frac{w_p \pi}{FOV_{rad}\, 180}$$

$$\theta'_{new} = \theta'_{old} + \frac{w_p \pi}{FOV_{rad}\, 180}$$

Team Autonomy | Swarm



- Assign each robot a quadrant
- Assign dig sites based on map features
- Sites must be greater than 100m from processing facility
- Sites closer to processing facility take priority

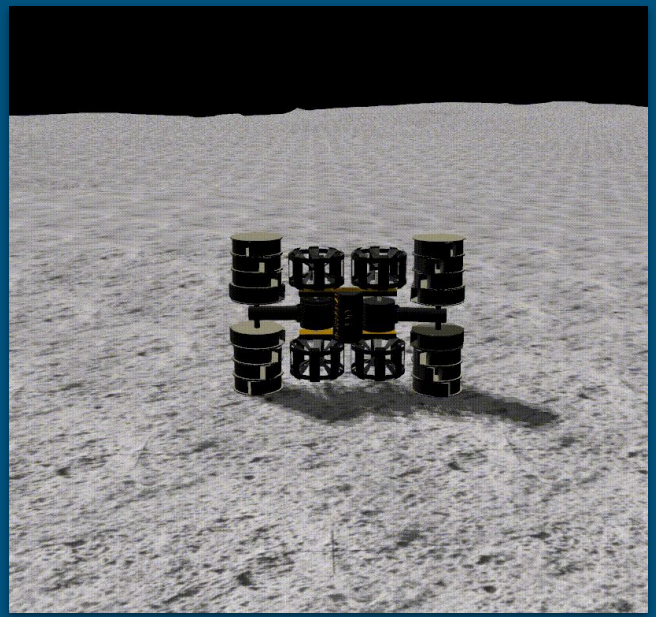
Team Autonomy | Challenges

- Running multiple robots in a simulation causes confusion with the namespace
- Frame rate drops drastically with current execution of disparity map construction
- Testing ground truth distance measurements in simulation using stereo vision
- Utilizing only visual sensor and IMU data to navigate
- Integration of already-existing components such as the ROS navigation stack into our current system

Team Autonomy | Results

- Created ROS architecture that collects environment data and stores it in an object
- Utilized visual odometry to understand global position and rotation within the world
- Implemented obstacle detection through stereo vision and custom laser scan
- Implemented custom navigation and obstacle avoidance movement routines
- Implemented Self-Right and several other utility functions

Team Autonomy | “Self-Right” Function



- Returns to standard position if fallen onto side
- Uses predefined set of movements to insure consistency
- Straightens arms and then raises both to 45 degrees
- Works by taking advantage of the IMU and the difference in width of the drums and wheels

Team Autonomy | “Balance-Arms” Function

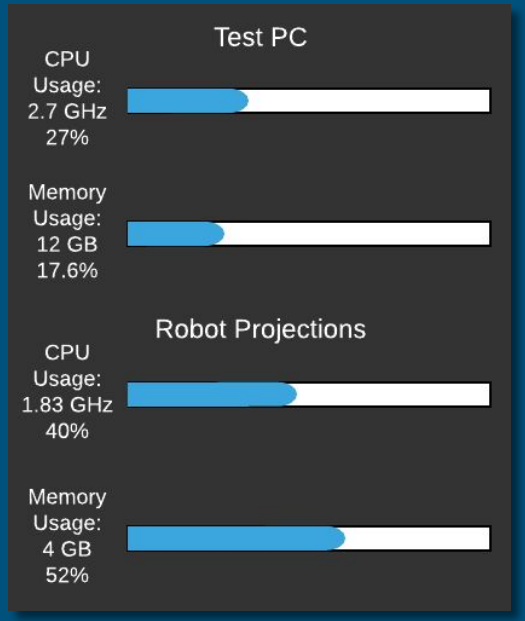
- The key is in the original RASSORs design
  - Allows for its lightweight design
  - Yet be able to excavate large amounts
- Torque in both arms in opposite direction cancels out
- Need to maintain balanced torque to keep digging force stable in low gravity environments
- Created special node to simulate changing force while digging in simulation
- Tested balance arms in auto-dig function using custom node with random influenced data

Team Autonomy | Stretch Results

- Created a fully autonomous digging loop
  - Individual auto functions are linked together to complete an entire dig cycle without assistance
- Able to run multiple robots in a simulation
  - Able to use Auto-Dig and Auto-Drive functions
  - Each robot is able to be controlled by different a controller/gamepad/autonomous controller

Team Autonomy | Performance

- The full system needs an average of 2.9 GHz and 2.112 GB of memory to function smoothly
- The robots have four 1.83 GHz cores and 4 GB of memory
- This means we are at roughly 40% of our available processing power and 52% of our memory
- There is plenty of room for people to add custom autonomous functionality in the future



Team Autonomy | Next Steps

- Finish swarm
  - Add communication between the robots
  - Control all robots via a hive core
- Implement orbital GPS
  - Match the landscape to satellite images to find location globally
- Develop EZ-RACERS
  - Game aimed at interacting with a younger audience
  - Race with NASA robots in a simulation

Create a Minecraft mod?