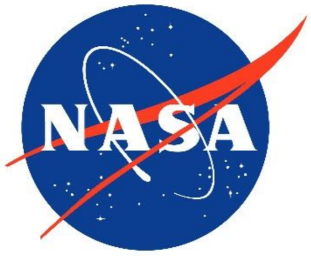


EZ-RASSOR

SWARM AI

Group 28

Spring 2021 - Fall 2021

Team Members:
Richard Malcolm
Stanley Minervini
Camry Artalona
Hung Nguyen
Coy Torreblanca

Project Sponsor:
Michael Conroy

Executive Summary

The overall goal of our project is to level an area on the moon using a SWARM of EZ-RASSOR rovers so that a landing pad may be built. In doing this, we will illustrate that these low-cost rovers are to do high level tasks autonomously. The project will also enable educators to have a 3D printable rover that will aid in the education of students and those who are curious in such technologies for little to no cost. The EZ-RASSOR project is entirely open-sourced and open to the public to learn and contribute too.

Our solutions will be demonstrated in two simulation softwares. The first being Gazebo to simulate the surface of the moon and to simulate the SWARM of EZ-RASSOR rovers. The second being Unity, to illustrate the technicalities of our solution, such as the math involved when leveling the surface of the moon.

Our project will be broken up into three main tasks: calculating the pathing of the rovers, the digging functionality of the rovers, and creating a task scheduling system sophisticated enough to handle multiple rovers in a SWARM doing different tasks.

While we must create efficient solutions to our tasks, we cannot just simulate our solutions in the software Gazebo. Gazebo lacks critical functionality in manipulating the simulation environment. This is a critical issue to our project, since our goal is to manipulate the environment in such a way to level the ground. Because of this, we will implement our solutions in Gazebo and in Unity. In Unity, we will illustrate the functionality of path planning, digging, managing rover health, and handling unexpected problems.

Project Narrative Description

The EZ-RASSOR AI SWARM Team is only one of many groups working on making autonomous robots capable of complex routines on the Moon. Specifically, the AI SWARM Team goal is: leveling an area on the surface of the Moon such that a landing pad could be built for rockets. We will create algorithms which condense the scope of leveling land into pieces accomplishable by individual robots in an efficient, safe manner. Required capabilities include leveling construction sites as specified by a desired topology map, managing rover health and problems that may arise.

Nasa will land the first woman and the next man on the moon in 2024 with project Artemis, but the scope of the Artemis project is much larger than the Apollo missions. Nasa and private contractors, such as Boeing; Toyota; Blue Origin; and SpaceX, will help build the infrastructure required for a long term Moon base from where other space missions will begin, specifically the colonization of Mars. How will a base on the moon help humans expand their reach in space?

With the ability to launch satellites, landers, and other payloads from the moon, space mission costs will be reduced substantially as rockets will require less thrust to reach escape velocity; furthermore, rockets entering the moon will be safer as the Moon does not have an atmosphere. The Artemis project is necessary for the future of human space exploration.

Artemis Logo [1]

Project Artemis will change space exploration forever, however, without a landing pad on the Moon, explosive rocket boosters arriving on the lunar surface will disrupt the moon dust on the landing site. For example, Neil Armstrong reported during the landing sequence of Apollo 11 in 1969 moon dust obscured his visibility. Furthermore, rockets landing on the moon send dust across the lunar surface, possibly for miles. The moon has no atmosphere, no air, and low gravity, which makes plume physics drastically different on the moon than on Earth. Dust, gravel, and rocks may travel very far and perform substantial damage to buildings, astronauts, and orbiting machines. Therefore, without landing pads to separate engine exhaust from the lunar surface, a long term human presence on the moon is impossible. Also, Chirold Epp, project manager for the Autonomous Landing and Hazard Avoidance Technology at NASA's Johnson Space Center in Houston, said lunar landers which land off center more than 12 degrees may not be able to take off again, further illustrating the necessity of landing pads.

Broader Impact

The EZ-RASSOR (EZ Regolith Advanced Surface Systems Operations Robot) is an inexpensive, autonomous, mini version of NASA's RASSOR robot. The project was first created as an educational project to demonstrate to visitors at the Kennedy Space Center the capability of NASA's RASSOR robot; however, the project has evolved to be a relatively low cost and accessible robot for anybody to use for any project. Projects which are available to the public often have hidden impacts as the designers cannot imagine the ideas and contributions which the public will bring. However, EZ-RASSOR is designed to be able to move across light-moderate terrain, collect and deposit regolith, execute routines, autonomous navigation, and communicate with other EZ-RASSOR rovers.

Specifically, the impact of EZ-RASSOR SWARM AI will advance the EZ-RASSOR project capabilities since more complex routines may be completed by multiple robots, including a non EZ-RASSOR mothership. Specifically, this team will add the capabilities necessary for EZ-RASSOR robots to survey land and dig/dump regolith in an efficient, intelligent manner to create berms on the surface of the moon, and afterwards, a future team can place and connect NASA tiles in a coordinated fashion to construct a landing pad for rockets to land on the Moon. Because current NASA autonomous robots do not meet their requirements we hope EZ-RASSOR robots will one day coordinate on the Moon and construct landing pads on the south pole for project Artemis bases.

On Earth, we hope that EZ-RASSOR robots may be used to educate future scientists. Furthermore, we hope that EZ-RASSOR robots may be a cheaper alternative for certain construction on Earth infrastructure projects as well. Overall, the project will be as useful as the open source community makes it.

Project Members:

Title:	email/discord
Camry Artalona:	cartalona@knights.ucf.edu / @mintchocolate#4528
Stanley Minervini:	stanley.minervini@knights.ucf.edu / @linked_list#3752
Hung Nguyen:	hungnguyen@knights.ucf.edu / @hungjn#0550
Coy Torreblanca:	coydiego@knights.ucf.edu / @coydiego
Richard Malcolm:	richardmalcolm@knights.ucf.edu / @tehflamex#0742

Project Sponsor:

Micheal Conroy	mike.conroy@ucf.edu / MikeConroy#2293

Team Motivations

Coy Torreblanca

I believe that this decade is the most exciting for space exploration since the 1960s. With only two megahertz of processing power, the Apollo Guidance Computer helped win the space race when Niel Armstrong and Buzz Aldrin landed on the moon in 1969. However in the 2020s, most personal computers have four gigahertz of processing power, one example of the massive advantages of today's technology. If organizations today can assemble the enormous talent required, we may further explore the cosmos and achieve the space superiority promised in 1969. Luckily, space research has never been better funded. SpaceX sent astronauts to the International Space Station and later reused the rocket boosters. Blue Origin is building a new lunar lander and Toyota a lunar surface vehicle. Today, a new space age is emerging. Personally, I'm super grateful that I, with five other team members, may do our small part in this exciting time. I hope our generation can experience the excitement, wonder, and unity which enveloped the world when Apollo 11 landed on the moon.

Stanley Minervini

“Space: The Final Frontier”. One question humans continue to debate is the existence of life in outer space. One reason we struggle to get out into space and explore such topics is our limited ability to land on other planets or moons. A little bit of background about me, I was born and raised in Florida. All of my life, I’d watch the rockets launch from Cape Canaveral. I’ve always been interested and curious about space. Furthermore, my personal interest in this project draws from the fact that I have the ability to make a change. I want to be part of the team that allows for more rockets to land on the moon in more sustainable ways. The ability to further the progression of the human race into space is an opportunity I couldn’t pass up. Additionally, the level of software engineering and development behind this project is a learning experience like no other. Oftentimes, coming into a large project can be daunting and scary for me. This project gives me the opportunity to develop my skills of learning code bases and new technologies faster. This project entails plenty of code and ideas to build upon, and I am beyond excited to be a part of this project.

Richard Malcolm

I was born and raised in Apopka Florida, and joined the magnet program at Apopka Highschool to learn engineering. In this engineering program I learned about general engineering methods and the different aspects of engineering. Through those classes I began to think about my future in the stem field and joined more classes that were engineering related. In these classes we worked with things such as vex robotics, bread and circuit boards as well as arduino. There I also learned more about hardware and how it works on a basic level, from things to logic gates to things like the binary and hexadecimal.

At the same time I performed in multiple band classes which were wind ensemble and jazz band. Unlike other general education classes I very much enjoyed band and engineering related classes. In jazz we were able to perform at many different venues which eventually led to me joining a small time jazz band which played at a couple different gigs to earn some money. I also got to perform at different venues during my four years performing in marching band. I ultimately ended up dropping band and musical related activities soon after I entered university due to a lack of time and a lack of motivation to continue

playing my instrument. However I do look back on those years particularly fondly as band and engineering classes made my favorite and most memorable memories from high school.

From there I was introduced to computer science and programming and found a passion for coding and web development. This led me to attend UCF for a Computer Science bachelors to improve my knowledge in the field of computer science and widen my prospects for my future career.

My personal interest in the swarm AI project is to learn and develop AI, I think it’s a really interesting subject because it has a very wide scope of uses such as in video games or in machine learning or in robots. I also find this project to be rather interesting because of its scope, not only is the end goal to use the things we develop on the moon, but there are many different teams from different universities working on different parts of the rassor project, so we are simply a part of a big project which I find interesting.

In addition, I also am interested in the robot vision aspect of this project, because the rassor rover has a camera on it that can be used to help do things like pathing, distance detection as well as other metrics that and detection systems that we can develop or work with. I also am really excited about becoming a Nasa collaborator because it's quite a hefty addition to my resume.

Camry Artalona

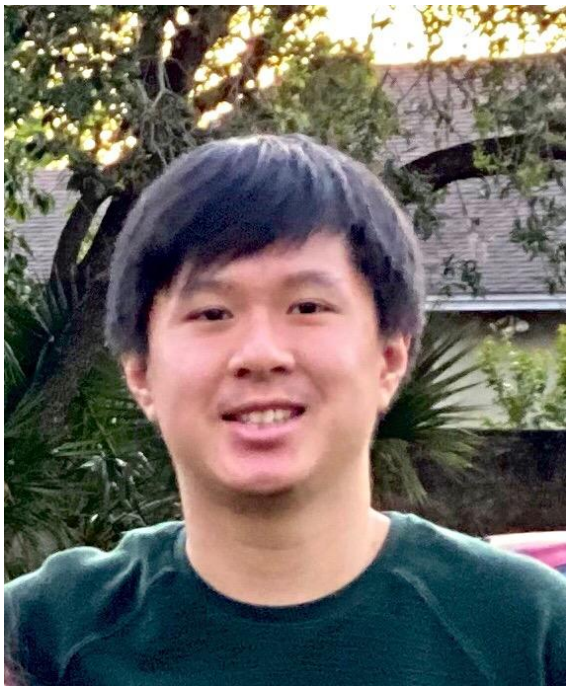
My personal interest in space draws from a greater perspective of benefiting the world. Going back to the moon to create lunar bases would be the first step in celestial travel to other worlds. This is our future. I've always been interested and inspired by the stars and our galaxy. Watching the movie Interstellar always made me wonder how far we can actually get in developing travel to other worlds beyond our own. This is the first step into making that a reality. Being part of the EZ-RASSOR team is an amazing opportunity that will be filled with many challenges, seen and unseen. I am beyond excited to work with an extremely motivated group of individuals, along with the experienced engineers at NASA and the Florida Space Institute.

My freshman year at the University of Central Florida, I had an interview with the contractor Jacobs for a software engineering intern position located at Kennedy Space Center. Arriving at Kenedy, going through the security checkpoints, and seeing all the engineers go about their business felt like a movie. It was surreal. Even right before my interview, I sat right in the middle of the office waiting for my interview, watching the engineers and scientists run around me. To ease my nervousness before my interview, I was able to look at a "countdown to launch" clock located on one of their walls located just above the interviewer door. Just

before my interview, a senior software engineer came up to me to talk about what it was like working at Kennedy and to wish me luck. Even though I did not get the position, it left a long lasting impression about the eliteness and opportunity of NASA at Kennedy Space Center.

The goal of Swarm AI is for a swarm of low-cost rovers to locate suitable land to build a landing pad, excavate that land, and then build landing pads on the moon for rockets to land and take off on. We have the added benefit of reaching out to experienced engineers at NASA and Florida Space Institute to aid us in reaching our goal. We can also contact other teams that are working on different parts of the project that may overlap with ours for help and ideas towards what we are working on.

Hung Nguyen



Growing up born and raised in Orlando, Florida my entire life, it is inevitable that my interest in space has grown into the passion that it is now. Being right next to the Kennedy Space Center and being able to watch historic launches up close and in person, I am honored and excited to be working on this project.

My personal interest with the swarm AI project really ties into the end goal and purpose of this project, developing a rover that will be eventually used on the moon and also creating the rover in such a way that it can be easily reproduced by anyone with the open source software and 3-D printed parts. Eventually, everyone should have access and should be able to recreate the EZ-RASSOR robot and even modify it into their own versions using the foundational source code and design that was made by me. That's why the core concept of this project being free and open source really hits home for me.

Additionally, learning and building new skills about everything necessary for the project to function, like artificial intelligence which in turn leads to a deeper understanding of python, ROS for robotics programming, NASA development,

standards and styles, robotics, and other software related knowledge is something I look forward to learning in this project because I would in turn become a better developer. I envision that working on this project will lead to dealing with a lot of lower level software and systems, this is also a skill that I wish to hone.

Lastly, being able to look back at this massive project proudly because of its impact, success, challenges and rewards is also one of the biggest points of interest and motivation for me. Taking part in such an important task and contributing to space related efforts, and being a part of something much bigger than me is an honor and I’m excited to get started.

Legal, Ethical and Privacy Issues

Our team's mission is to specifically enable a SWARM of EZ-RASSOR rovers to level a geographical area lucrative enough to build a landing pad. This is going to be created and tested in a simulation software over many iterations before any live code is pushed for real rovers on the Moon to implement these solutions. That being said, there may be unseen errors when translating our codebase from the simulation to the actual rovers, simply because the simulation isn't real life. Further down the road, as more rovers are implemented in building infrastructure, it becomes even more apparent to build safe and secure software.

The EZ-RASSOR project in its entirety is open-sourced and free to use by anyone and the general public. The philosophy of this is meant to enable anyone to learn how an actual rover works and give them the ability to do their own research. Open source is a cornerstone of the EZ-RASSOR project; however, all of our collaborators are not open source; therefore, adding friction and uncertainty to the project. NASA has promised us resources which we need to complete this project, but these promised resources have been blocked or delayed in the past by NASA lawyers because of their concerns. Specifically, NASA lawyers, according to our sponsor, are worried that FSI could be a direct competitor to NASA, using the resources they provide to make money in the private sector; however, our sponsor has guaranteed our team that FSI is a purely non-profit, educational institution. Therefore, we see this roadblock as potentially inhibiting but completely unnecessary.

On the basis of privacy, our team has found no privacy concerns when working with the EZ-RASSOR software. This is entirely an open-sourced project, therefore there should not be any privacy concerns for the project. If any due arise, they will be handled by contacting our sponsor, NASA, and the Florida Space Institute.

The Problem

Landing rockets on the moon is a difficult task on its own. Getting to the moon is an incredible feat on its own, but isn't the only challenge we must overcome. Many seen and unseen problems arise from rockets launching and landing on the lunar surface. One of the less obvious problems is once a rocket launches or lands, if not on a landing pad, the rocket will eject regolith at high velocities into lunar orbit.

Regolith *being kicked up by an Astronaut [2]*

Upon landing, a certain amount of regolith is displaced by the rocket and it's weight. This ejects the regolith up and will create an atmosphere with regolith inside of it. As more rockets, or rovers, land on the moon, the atmosphere may become full of the regolith. At first glance, dust doesn't seem like it should be able to harm a rocket. But the regolith will be ejected at a high velocity. At an extremely high velocity, even the smallest particle of dust can be a problem for rockets, rovers, and sensitive space equipment.

Regolith by nature is coarse and abrasive. So it is naturally damaging to surfaces through friction. A lot of the regolith can become ultra-fine particles and can be difficult to clean off astronaut suits. This itself can cause more problems such as becoming a danger to accidentally breathe in, causing damage to astronauts' cardiovascular systems.

The lunar surface of regolith can become ionized and charged by radiation, cosmic rays, and solar winds. This may be one of the biggest dangers when it comes into contact with equipment with electronics.

If enough regolith is launched into the orbit of the moon, it can create an environment too hostile for any modern rocket to land. Thus, preventing any future missions to the moon.

On another note, a bigger problem that arises from doing our tasks is having multiple rovers doing different tasks at the same time. We want to find an efficient, yet elegant solution where rovers will not interfere with another rover's task. A solution to this will most likely involve the central task manager scheduling the tasks in a way such that the rovers do not interfere with one another. For example, if one rover is currently digging in a position, another rover cannot go to that exact location and start digging at the same time as the other rover. This will cause them to collide and possibly become damaged.

This is also a problem when rovers are pathing to the same place and their paths cross. Since we will have one location to where rovers will path to excavate and build the landing pad. We need to figure out a way for the central task manager to schedule the rovers to path in a way such that the rover will not collide. The rovers have ample opportunity to collide in every part of the pathing process. They will be most susceptible to collide with another rover on the path back from the dig site to the recharge station.

Goal of the Project

The goal of the project would be to autonomously be able to level land on the moon to help create an effective solution to prevent regolith from being kicked up by landing and launching rockets. The solution must be autonomous and efficient. This way, an astronaut does not have to spend time controlling the SWARM to complete all tasks associated with building the landing pad.

Stretch Goals

The stretch goals of the project depend on the amount of time we are able to complete the tasks in. The ability to meet the stretch goals are also dependent on the feasibility of implementation as well. This all depends on the software we are required to use and its limitations that will potentially hinder us from showing our solution in an elegant manner.

The first stretch goal being integration of our team's solution with the other FSI EZ-RASSOR ARM team. This stretch goal has not been met due to the mutual understanding and prioritization of both teams respective solutions, however strides towards integration were made. The second stretch goal is simulating our solution in Unity. Although our requirements only had us demonstrate our solution through Gazebo, we were able to have a working Unity simulation of the solution. One slight drawback is that the Unity simulation requires Gazebo to run due to necessary camera dependencies, this is another stretch goal that we have not met.

Difficulty of Our Goals and Interests

Determining the viability of land and resources will be one of our biggest struggles. Firstly, we need to determine whether or not we will control where landing pads will be, or if the robots will report back to us with information on optimal places to build landing pads. This issue will directly relate to the functionality of the task manager and our task 1. Either we must have “scout” rover’s go out and determine if the land is suitable. If this becomes the case, then we must identify an optimal and efficient solution for rovers determining the eligibility of the land. There is a possibility of not having to do this. In this case, we would be given the topology map of the moon area we are surveying and then from the map, determine if we can build a landing pad and in what area. This would still involve a solution to determine the eligibility of the land, but we would not require a rover to go and “scout”.

Secondly, it’s not entirely easy to determine the viability of a landing pad due to the nature of the variable structures around the moon. There are many different situations to consider, such as having very tall mountains, or very steep valleys. Thus it will require a creative solution and very in depth planning to consider the possibilities that we face as a team. Again, to find a robust solution we must consider many possible issues that we may face when trying to find suitable land.

Another issue that we came upon is adding code to an existing EZ-RASSOR code base. It is incredibly likely that we will need to change some of the source code in order to handle the situations we will encounter. Furthermore, the way in which we utilize the current code may be different from how it was intended, and so changes to existing code always create new problems and challenges.

A major issue that has risen from current research is the ability to manipulate the environment in the Gazebo simulation. Our project involves changing the environment in the simulation via EZ-RASSOR rovers. This is the key fundamental of our project. But in the simulation software, we are unable to edit

the environment in such a way. This is a major problem when it comes to finding, implementing, and testing any solution that we may find.

Another issue or difficulty is having the rovers bump into each other either when performing a task or when moving to a build site. Proper planning must be considered when path-planning between many rovers will be occurring. Also, it becomes more complex when we have more than one SWARM involved in the building process.

An issue we foresee will be changing and upgrading the software involved with this project. Since this is an open source project which many other teams are also working on and have worked on and has a large code base as well. The software and languages being used cannot be upgraded very easily. For example, this project has a lot of python code, but this python is in python2. Upgrading to python3 would have many advantages when it comes to speed and efficiency. But this upgrade would not be up to our group to implement. We would have to seek permission from the maintainers of the project, NASA, and Florida Space Institute. If we were to commit to such a change, it would also be undertaken by every other group as well.

Another concern I have is how to deal with upgrades and changes to the project by NASA and the admins of the project. If they do decide to change a software, for example an upgrade from ROS1 to ROS2, how will this affect the path and time it will take to implement our solution when we are working with the current software.

Metrics

1. How many?
    a. Many robots in a swarm (4-5) + a mother ship
    b. Many different Swarms
2. How often?
    a. Autonomous vehicles should be running 24/7 (as long there is infrastructure to be built)
3. How long?
    a. They work as long as they have battery charge and then need to come back to recharge. (Distance to recharge/speed of robot/length of battery remaining)
4. How complex?
    a. Complexity is high. This project will use robotics, computer vision, artificial intelligence, and collaboration with past teams on how to implement solutions to the project
    b. Many dependencies
    c. Many teams working on the same project
5. What values?
    a. Topography matrices (given to use by NASA)
        i. Find hills where rovers need to dig and holes where rovers need to dump.
            1. Needs to have enough regolith to redistribute evenly in area.
            2. Need to level the ground to surface level
    b. Landing pad size is proportional to rocket size
        i. 50 feet by 100 feet
6. What events occur
    a. Sequence of events to create a landing pad
        i. Find the difference between surface map and ideal landing pad
        ii. Create a set of instructions
        iii. Redistribute regolith as described by instructions

Initial Ideas about the Project

Coy's ideas

1. Topographical Map
    a. Data Structure
        i. Numpy arrays (matrices) are a great choice for our maps. Matrices hold values and their positions relative to one another. Furthermore, numpy arrays have built in, efficient operations, such as subtracting two matrices which will be required for this project as explained below.
    b. Points
        i. Longitude and latitude coordinates identify every point on a surface and their relative positions to all other points on the map.
        ii. Each point also has an elevation value indicating the height of the surface at that point's coordinates.
    c. Maps
        i. Current Moon topology map
            1. This map consists of the current topology on the moon.
            2. This map consists of the desired topology of the moon, and has the same points as the "current moon topology map" but different elevations.
    d. Map processing
        i. We would like to find how much regolith needs to be removed or added for each point. We can do this by subtracting the original elevation of each point by the desired elevation at that point, giving us how much dirt needs to be placed or taken at that location.
        ii. Furthermore, we would like to know the distance between points where there is too much regolith and the closest

points that have too little regolith. Simply find a point with too much regolith and check the closest points with too little regolith. With these distance values between appropriate points, any rover at a point knows where the nearest place is to dig/dump regolith.

Stanley's ideas

1. Utilize robots to rotate around area and survey land
2. Collect information for surface map
3. Run data against algorithm to determine viability
    a. Compare surface map to ideal surface for landing pad
    b. Calculate holes vs hills
        i. More dirt needed than accessible : possibly dig dirt from farther away or scratch idea
        ii. More dirt than needed : find a place to put dirt, maybe a dumpsite? (needs to be allocated at beginning of project)
4. Determine where robots are "MOST" needed and position accordingly
    a. This could look like staying away from parts of ground where dirt is already level.
    b. Research farther into efficiency of robots' positions.
5. Start process of filling in holes or digging out hills
    a. Break down hills first to utilize dirt for craters
    b. Then level out the entire field
6. Utilize survey methods, techniques, and built in depth cameras to verify land is level and ready for building
7. Begin the process of building the landing pad.
    a. Make sure that all rovers are charged up and then send out the rovers with material for the landing pad to be built off of.
    b. Place robots towards the center and have them place the bricks from inside to outside the area.
8. Repeat the process from 7 until the landing pads are all placed.

    a. Verify the landing pad is done via surveying methods, techniques, and built in depth cameras to verify the landing pad is complete and correct.

Richard's ideas

1. Finding an "optimal" area to build a landing pad
    a. Survey area
    b. Determine which area is most level
2. Developing an Algorithm to determine which land is flattest
    a. This Algorithm would compare heights and dips in the land
3. Level the area
    a. Fill in holes
    b. Remove hills
4. Construct landing pad

Camry's ideas

1. How many rovers?
   We will have a SWARM of low-cost rovers in a pack of around 30. This is not a static number and is subject to change.
2. How high can the peaks and low the valleys be?
   Ideally, we do not one to select an area of the moon that is super complex with its land. We want to select an area where the Rovers will be as efficient as possible.
3. Do we need to send individual rovers out to find suitable land?
    a. If so, how many?
    b. If so, in what way will we send out rovers?
4. What will be the threshold in determining if part of the land is suitable for the landing pad to be built?
    a. How do we find an optimal solution to this?
5. What language will we use to implement our solutions?
6. Will we only need to simulate our solutions once implemented?

7. In what ways might we need to consider using 3rd party software to show or help implement our solutions?
8. How will decisions by other groups affect progress on our project?
9. How much do we have to depend on other groups for solutions before we can move onto parts of our project?

Hung's ideas

1. Analyzing net elevation of the moon's surface based on surveying data or contour maps to represent maximum and minimum height, which will have to be retrieved based off given or existing technologies and tools from previous navigation teams
    a. Within the autonomous swarm package, utilizing the park ranger node for depth camera generated digital elevation models to calculate topology data
    b. Within the swarm control package, utilizing the get rover status service will allow for precise pinpointing of the rover location, this functionality could be used to self-map the surface of the moon with the rovers
    c. Comparing the pre-existing global digital elevation model and cross analyzing it with locally produced digital elevation models from the depth camera is another means of narrowing down the precise area a rover is located on the surface of the Moon
2. Utilizing the drum buckets as a means of efficient dirt transportation alongside excavation to optimize and simplify the entire landing pad process
    a. Establish a set difference between digging holes and filling in holes, dual modality for bucket usage if necessary
    b. Possibly alter the intended design of the drum buckets (slightly) for multipurpose usage
    c. Determine environmental factors that can lead to failure in excavation
3. Based off existing ROS packages within the EZ-RASSOR like the Swarm Control Package, a feasible means of reapplying the existing

functionalities, nodes and services into an altered modality with additional features that are aimed towards tackling swarm pad building completion

a. Maximizing efficiency among rover collaboration and constructed pathing
b. More sophisticated system for tracking rover status in parallel with current operations
c. Add instructions and functionality for actually building the landing pad

High Level Swarm AI Block Diagram

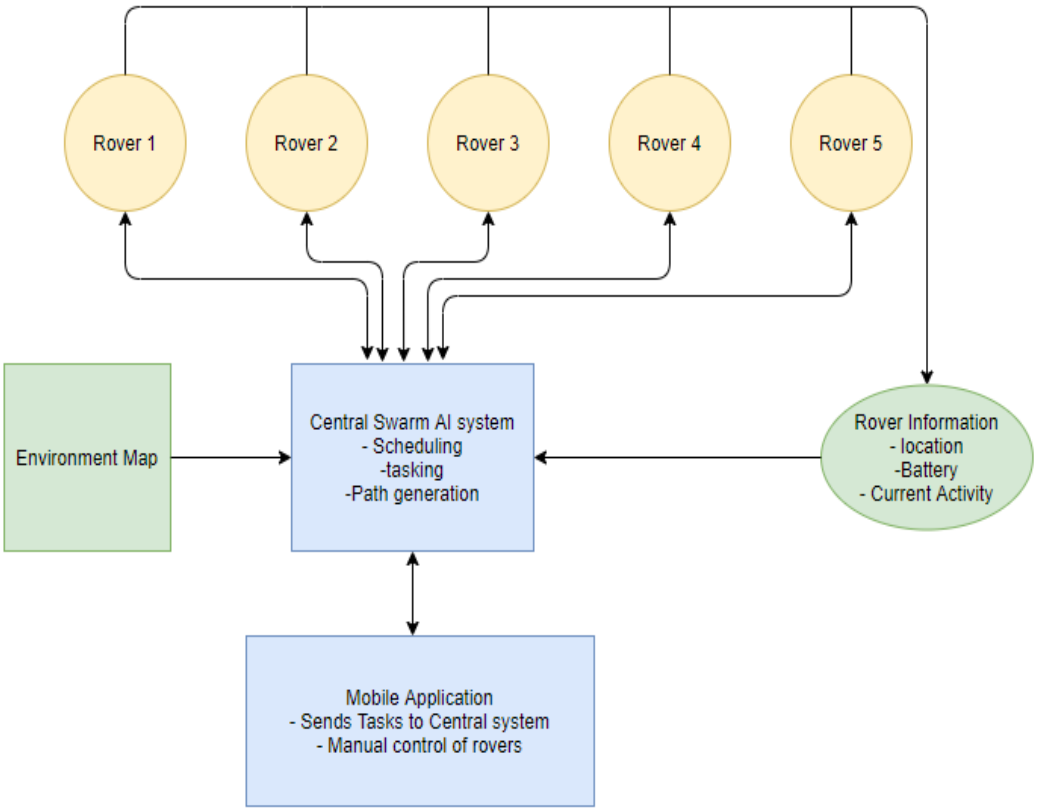


Diagram 1, that illustrates each component and their connections/interfaces.

Diagram 1

Diagram 1 represents a high level representation of how the rover operates under swarm intelligence. The mobile component is used to input commands to the central AI system. This AI system calculates the best path to take for each rover through a number of different path planning algorithms. This then calculates schedules, and tasks for each rover and assigns them according to it's calculations. As the rovers progress and do their tasks, they report back to the central Ai system. The rovers are constantly given commands from the central AI system according to the information that they provide back to the AI system. Furthermore, the central AI system uses the environment map to help guide it's algorithm and path planning calculations.

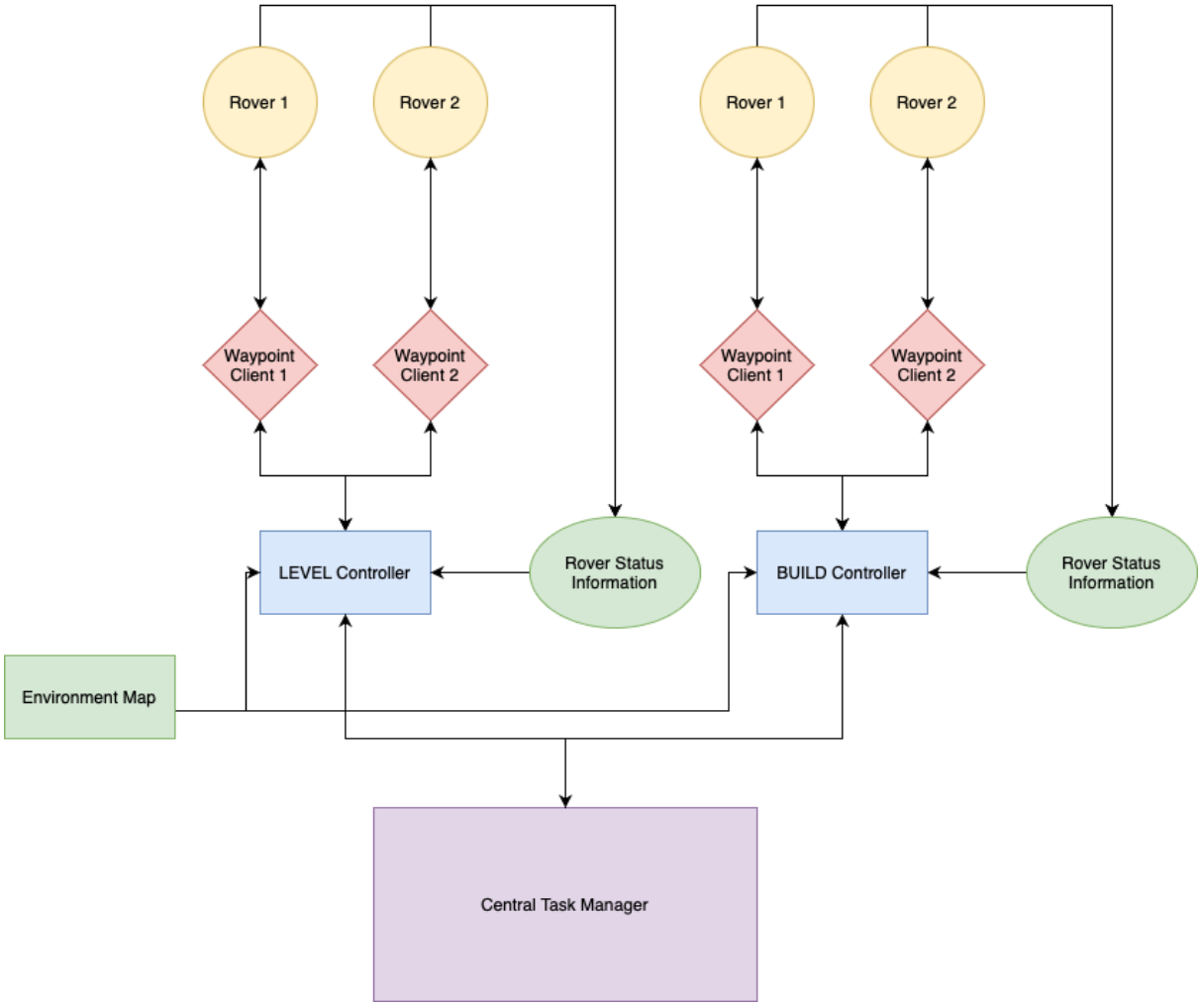


Diagram 2, that illustrates each component and their connections/interfaces.

Diagram 2

Diagram 2 represents our unique solution regarding our UML block diagram. Our central task manager oversees the two controllers, Build, and Level. It works as the hive, or queen bee of the system.

The central task manager helps to use our controllers in an asynchronous manner. The next aspect of our diagram is the connection between waypoint client and controller. Our waypoint client will serve as a communication method between the controller and the rovers. The controllers form a one to many relationship with the rovers, where each rover has their own way point client. As the rovers go about doing their tasks, they report back to the controllers with information regarding their status, such as battery level, location, and task progress.

These controllers then report back to the central task manager, and the cycle repeats. Another notable aspect of our diagram is the environment map feeds into the controllers. This allows for the controllers to essentially navigate the terrain around them, and adapt their tasks appropriately.

Block Status

Orange Blocks: Currently being built by other teams, can be used in simulation with the ROS software.
Green Blocks: Currently being researched.
Blue Blocks: Currently being Researched, and designed.

Diagram Legend

Rover:

1. These are small 3-D printed rovers designed to be used on the moon for surveying, building, and excavation.
2. Software is run on rover hardware

Environment Map:

1. A map of the environment that contains topography details.
2. Stored in mothership. Used to calculate rover requests.

Central Swarm AI system:

1. This system controls the scheduling, tasking, and paths of the rovers in the swarm.
2. In mothership

Rover Information:

1. Metrics related to the rovers, such as, location information, battery levels, and current task status.
2. In rover

Communication

All communication between devices is through Wifi, or a local area network. The communication component of the EZ-RASSOR works on taking in messages from controllers and routing the messages to either the simulation, or the hardware. The communication component receives messages from the controllers. Some examples of controllers are the mobile application, a gamepad, or even an autonomous controller. This could look like reading in button presses from our mobile application, button presses on a gamepad, and also toggling autonomy on and off based on the users commands and desires.

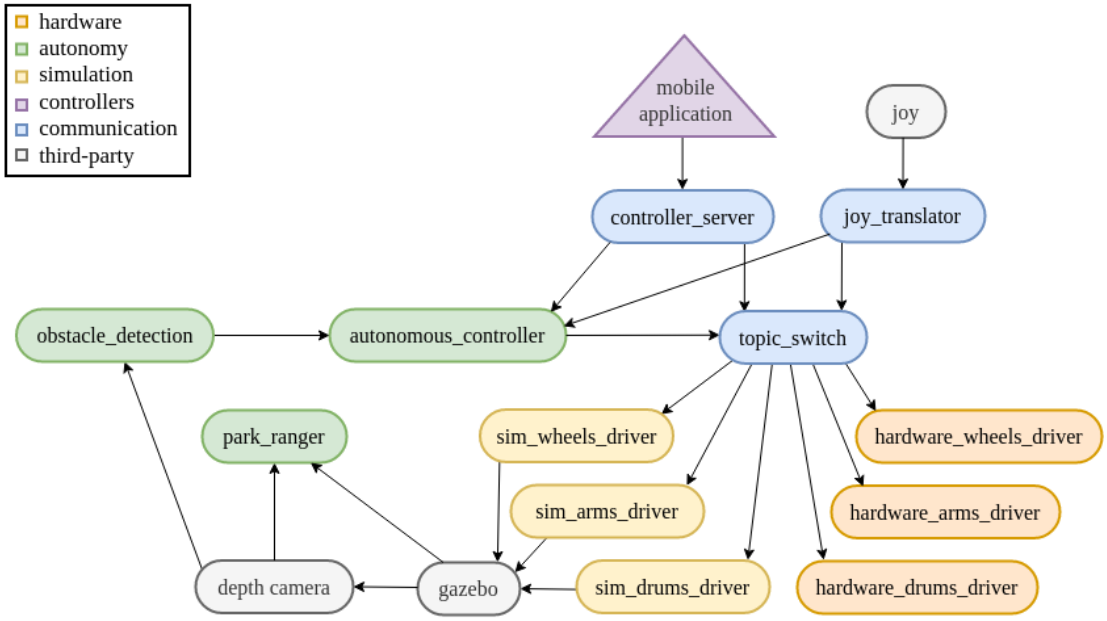
The rovers are all equipped with 802.11ac Wi-Fi. This is also known as Wi-Fi 5 or Gigabit Wi-Fi. It can achieve data rates of up to 1300 Mbps. Obviously the concern that we have regarding Wi-Fi is the distance at which this will be able to communicate with our central task manager. In general, the standard operational range is approximately 170 feet at most. We believe that this will be sufficient for the purposes of our project, but may cause concern if the scope increases.

Research

Following this section is a compilation of the research accomplished by our team. The following subsections help to give a deeper understanding into the projects' architecture and code. Furthermore, it helps to elaborate on different algorithms that were researched and studied.

The research section will contain information that will be used to directly implement our solutions for our various tasks. It is essential for our team to be able to have a full understanding of the software we will be implementing. Our team should also explore creative solutions possibly not thought of by other teams that have worked on or are currently working on the EZ-RASSOR project.

EZ-RASSOR Architecture



EZ RASSOR Current Architecture [3]

The Overall EZ-RASSOR architecture is divided into 5 main components:

Hardware

The hardware component of the EZ-RASSOR includes all packages and nodes that control any mechanical aspects or physical hardware and nodes that monitor the hardware. Specifically, all hardware components are implemented using ROS nodes This correlates to the rover movements in all cardinal directions, digging actions and more.

Communication

The communication component of the EZ-RASSOR receives messages from controllers (this can be either a gamepad, autonomous controller, or the mobile application) and delivers those messages to the hardware and simulation. Communication is mainly in charge of parsing signals from the specified controller based on user commands.

Simulation

The simulation component of the EZ-RASSOR robot is able to simulate and manage an environment and models developed for the rover. The simulation package comes with nodes that translate the hardware messages into parsed information that Gazebo will be able to understand. Overall, it contains all model data and generated world data.

Controllers

Most users will interact with EZ-RASSOR robot through the mobile controller application. The purpose of this program is to allow users to interact with the system in a simple and easy method. The mobile application allows users to control the EZ-RASSOR rover physically or in simulation.

The topic switch is the master controller which receives inputs from all other controllers and executes them.The controller_server and joy_translator take inputs from users and routes them to autonomous routines or directly to the topic switch. Autonomous routines send instructions directly to the topic switch.

Autonomy

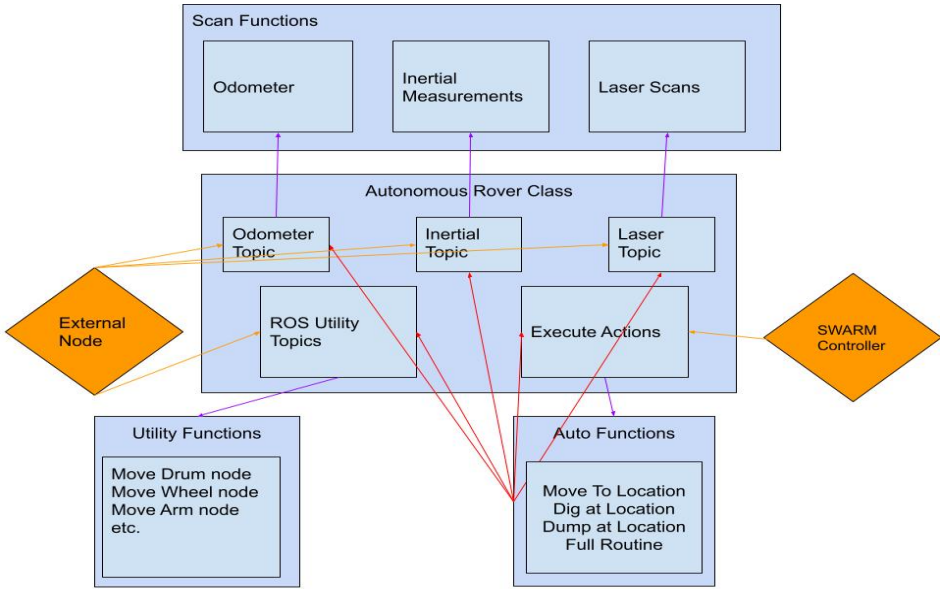
Autonomous EZ-RASSOR Swarm Control provides the tools necessary to control a group of rovers who work together to asynchronously accomplish a specific task. The package includes methods to request a task, such as dig a

hole, from each rover and continuously monitor that rover, possibly interrupting it, as it completes the task. Furthermore, the package provides a controller, central AI controller, which communicates each rover monitor to ensure the completion of the overall goal is in progress.

EZ-RASSOR Autonomy

EZ-RASSOR Autonomy packages allow commands to be sent to rovers to be executed automatically, possibly in sequence or in a loop with other commands. Autonomy packages categorized by whether they control a single rover or a swarm of rovers.

Autonomous Control



Autonomous functions interactions with external methods

Basic Autonomy

The autonomous control package allows individual rovers to perform automatic operations. Any automatic processes performed by a rover, including those called in the swarm control package, are contained within this package. Furthermore, the autonomous control package is a class which encapsulates all of the functions in the file, abstracting a single, autonomous rover. The class is responsible for the control of a single rover, including publishing its status. Therefore, any new additions to the automatic actions of a single rover should be placed in this package.

Currently, single rovers may use a plethora of different sensors. The basic sensors of an EZ-RASSOR robot include laser scans, odometry, and inertial measurements, and they are controlled by sending messages to the appropriate scan topic. Furthermore, individual rovers may be instructed to perform several actions, affecting the real world around them, by sending messages to the correct topics. Rovers are capable of moving their wheels, rotating their digging drums in a dig or dump position, and moving a mechanical arm. The “autonomous_control.py” file in the “ezrassor_autonomous_control” package links the callers of autonomous functions to the utility and automatic functions, whether scan or action functions, which implement the desired functionality. Callers are then directly linked to executors through a one way bus, where callers are the publishers and the rover autonomous package is the consumer. Note that the file’s functions are general and, for sensor functionality, simply subscribe the sensor nodes to the topics which call them. Therefore, the “autonomous_control.py” is like the neural network of the autonomous control packages and does not implement functionality. EZ-RASSOR robots are open source and equipment may be replaced as needed, and the software allows for diverse builds.

Autonomous control includes a “autonomous control loop,” which makes a rover continuously wait for commands from a topic called “autonomous_toggles,” which is just in an integer representing a certain action for the rover to execute. The rover is currently capable of the following actions: dig, dump, travel to,

dock, and full autonomy. However, to build a landing pad, we will require a place and pick up action from the EZ-RASSOR arm team.

The autonomous control package includes a full autonomy function, mostly for demonstrative purposes. The autonomous function will continuously move a rover from digsite to dumpsite, digging and dumping regolith in the correct sites.

AI Objects

AI objects are instantiations of classes which contain the setters and getters for the state of the rover and the state of the environment around the rover. Specifically, the world state class contains the sensory data of the rover, including the rover itself and its environment. Furthermore, the world state class has callback methods to change its members. Methods in the autonomous package will use a world state instantiation to make autonomous decisions.

The ROS Utility class provides methods in the autonomous package with the functionality to command the rover to move any of its components by providing publishers to send messages to the correct mechanical nodes.

Publish Actions Method

The publish actions methods change the world state by publishing the given parameters to the topics which will change the state of the rover, and call the appropriate world state callback methods.

Utility Functions

Utility functions order the rover to perform mechanical actions, such as turn and move, and return or perform calculations with rover status information all at a higher level then the nodes subscribed to the ROSUtilities AI object node. The utility functions are to be used by ezrassor autonomous programs; therefore, the implementation of them is abstracted by sending the appropriate message to

the correct action server, created by ROSUtilities, to which the appropriate methods are listening, waiting to move the rover as requested.

For example, the utility “set_x_arm_angle” functions will be given an angle to set the arms to and will continually move the arms until they are at the given angle. The “self_check” function checks the status of the rover and quits gracefully if it cannot handle any issues with the rover. For example, self check will attempt to charge the rover if its battery is too low and will attempt to flip the rover if it is upside down.

Swarm Autonomy

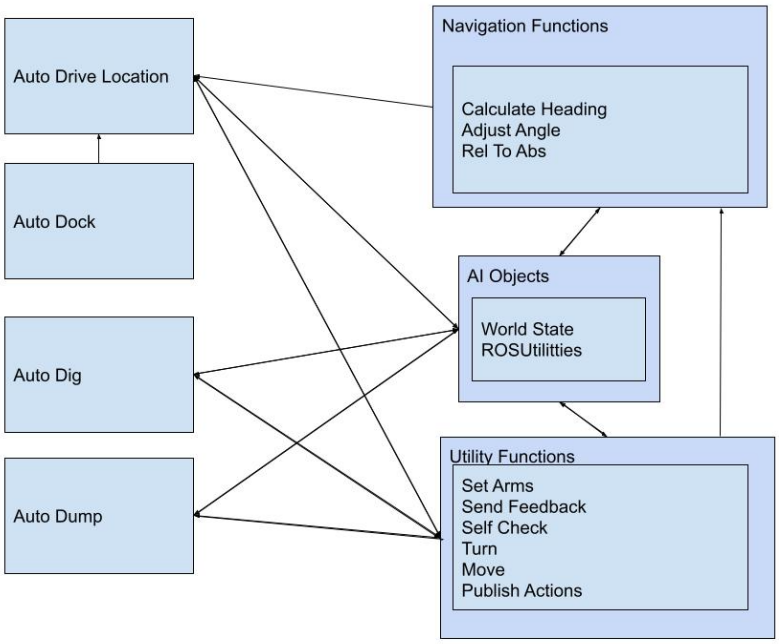
The autonomous control package includes the logic and data to execute swarm control functionality of a single rover. Therefore, the autonomous package is responsible for instantiating the required functionality of an EZ-RASSOR swarm rover, and establishes communication between the swarm controller and rover through the waypoint action server. Note, requests for swarm autonomy execution occur exclusively through waypoint action servers; furthermore, swarm actions require calling back to the autonomous packing for other autonomous rover functionality. Individual rover action executions, called by the attached waypoint, include set rover to charge, rover digging, and go to location using pathing.

The “execute_action” function in the autonomous package, which is only called by a rover’s waypoint action server, may call autonomous functions and instruct the rover to charge, dig, or go to a location while providing feedback to and accepting new requests from the waypoint client. The rover may only call automation functions, from the “auto_functions.py” file in the autonomous package; therefore, the “autonomous_control.py” file does not implement any functionality, rather calls it at the behest of a waypoint. However, the “execute_action” must verify the autonomous action function results and report them back to the waypoint.

Autonomous Functions

Autonomous functions are found in the “auto_functions” file, and they use navigation functions and ros utility functions to perform the functionalities requested by the “execute_action” function in “autonmous_control.py.”

Autonomous functions use utility functions to perform a series of actions automatically. The following is a description of the currently available autonomous functions.



Relationship between navigation, utility, and automation functions

Charging

Charging at the time is implemented by stopping the rover and making it sleep at its current location. This is feasible if the rover has solar panels and the environmental conditions are correct; however, the “charge_battery” function does not message any hardware nodes, such as solar panels, to charge the battery. Therefore, we may conclude that charging is not implemented in the autonomous package, instead pseudo-code is executed.

Auto Drive to Location

The “auto_drive_location” function is responsible for making the rover move from one location to another, if possible, avoiding obstacles along the way. Note, the function may execute without a waypoint server, but we will discuss from the perspective of swarm execution. The “auto_drive_location” function must set up parameters before execution, including setting up communication between rover utilities and rover waypoint for asynchronous rover feedback to the waypoint client.

Furthermore, certain utility functions are called to prepare the rover to move, including raising its arms. Lastly, utility statuses (sensors) are accessed to verify the rover can journey to the specified destination. For example, if the rover cannot achieve the requested journey because of inefficient battery charge, the waypoint is preempted and the routine is aborted. Lastly, before beginning the routine, the rover calculates the correct direction, using the navigation module, of the goal and turns to it.

The routine is executed in a for loop which continues to execute until the rover has reached the goal location or the routine is preempted by the waypoint client or rover. During each iteration of the routine loop, the rover is again instructed to set up its arms and check its battery status, possibly aborting the routine. Again, the rover turns to the desired location, calculating the turn delta using the

navigation module, and moves by the preset increment specified by the utilities module. Finally, publish feedback to the waypoint.

Auto Dig

The “auto_dig” function is responsible for making the rover dig and collect regolith with its arms while providing feedback to the waypoint client, if applicable. The function begins by using the world state and the utility function “self_check” to verify the rover is upright and has battery charge. Then, the rover’s arms are placed down onto the regolith, and the attached drums are spun for the requested direction, changing direction, from forward to backward and vice-versa, every one hundred seconds. Rover feedback is published and logged.

Auto Dock

Use “auto_drive_location” to make the rover move to the “home” coordinates. Save the coordinates of the last location for possible use on the next outing.

Auto Dump

The “auto_dump” function rotates both drums inwards and drives forward for a specified duration of time.

Rover Simulation

Unfortunately, Gazebo currently does not have the necessary models and animations to simulate the production of a landing pad on the moon; however, we have decided to use a simpler simulation interface. Our rovers are autonomous once a construction site has been chosen by an operator; therefore, real time, 3D visual feedback is not required. The simulation for this project will mostly be used to preemptively see how the task will be executed by a multitude of rovers. The simulation is essential for the operator to see how long the project will take, and it will allow the operator to see any potential problems before the rovers begin moving, including the feasibility of the project. For example, an operator might find that a slightly higher landing pad elevation will have enormous runtime costs if the rovers will have to allocate more space or resources for the additional task.

The construction simulation will provide visual assistance for the operators. The simulation will include every rover required to construct the landing pad, representing each rover as a dot, whose colors change depending on the current task. For example, 5 rovers might be found on the simulation map at any one time, and if 2 are in the digging site then they might be the color green; however, if another two are in the dumping site, then they might be the color blue. With the visual aid, any operator can see the expected sequence of events.

Note, with the swarm action server feedback feature, we may simulate the rovers in real time. Since the simulation will require information from the central task manager and each rover after each timestep, we suggest wrapping autonomous functions to acquire rover status information for the simulation after each timestep, and create another action server so the rovers and the task manager may communicate with the simulation without affecting any existing functionality. For example, after each timestep, the new action server will send feedback to the simulation identifying each rover, its current job, position, and any changes it made to the topography of the moon. We could use MatPlotLib to plot the topological graph using a colored pixel which describes the height in that region and plot the rovers after each feedback message.

Autonomous control

This node is the central location for everything related to autonomy with the rovers. This node performs the majority of publishing to all rovers and their clients.

The code for this is demonstrated below.

```
front_arm_topic = rospy.get_param(
    "autonomous_control/front_arm_instructions_topic"
)
back_arm_topic = rospy.get_param(
    "autonomous_control/back_arm_instructions_topic"
)
```

Autonomous Control Node Code [4]

The node itself has plenty more commands, but for simplicity's sake, we've omitted the majority to demonstrate the purpose and actions of the node. You can see that the different parts of the rovers are fetched from the rospy itself and are used to help the autonomous control dish out commands to the rovers in the simulation.

Obstacle detection

The Obstacle detection node is fairly intuitive in its purpose and actions from its name. However the way in which this is performed is fairly complicated and deserves some research into. Below is the code for the node.

```
 #!/usr/bin/env python
import rospy
from ezrassor_autonomous_control import obstacle_detection

max_angle = rospy.get_param("obstacle_detection/max_angle")
max_obstacle_dist =
rospy.get_param("obstacle_detection/max_obstacle_dist")
min_hole_diameter =
rospy.get_param("obstacle_detection/min_hole_diameter")

obstacle_detection.obstacle_detection(
    max_angle, max_obstacle_dist, min_hole_diameter
)
```

Obstacle Detection Node Code [4]

This node utilizes a couple of different parameters from the rospy to help achieve its goal of obstacle detection. We can see, max_angle, max_obstacle_dist, and min_hole_diameter.

These parameters are utilized alongside a depth camera to use laser scans for obstacle detection. Three different techniques are used along with this laser for obstacle detection. The first is known as the Hike Method. This method compares the change in distance between consecutive points in a direction to a to a configurable threshold, which is min_hole_diameter. The next method is the Slope Method, which utilizes the max_obstacle_angle to compare the change in height divided by change in distance between two consecutive points. The

Combined Method uses the minimum distance to an obstacle in each direction, with the combination of the results of Hike and Slope.

Park ranger

From the existing packages that are located within the EZ-RASSOR software code base, Park Ranger is a node from the Autonomous Control Package. The Park Ranger node is responsible for taking information from the depth camera in the form of a point cloud and converting it into data that can be represented as a local digital elevation model.

Although Park Ranger uses the digital elevation model in order to compare and analyze with a pre-existing global DEM to pinpoint the rover's location on the Moon, our group plans to use Park Ranger's produced local DEM as a means of analyzing the topography of the current local position and extrapolating the data to be used for calculations of net elevation necessary for the SWARM rovers to be able to understand what parts of the surface need to be excavated to build a suitable environment for a landing pad.

The data from the local DEM will be sent to some kind of Central Task Manager system that will calculate and create instructions for each rover to level the land.

```
1  #!/usr/bin/env python
2  import rospy
3  from ezrassor_autonomous_control import park_ranger
4
5  real_odometry = rospy.get_param("autonomous_control/enable_real_odometry")
6  world_name = rospy.get_param("park_ranger/world")
7
8  park_ranger.park_ranger(real_odometry, world_name)
```

Park Ranger Node Code [5]

Above is the actual code within the EZ-RASSOR code base from the Autonomous Control Package that utilizes the Park Ranger node to execute its functionality.

Get Rover Status

From the existing packages in the EZ-RASSOR software code base is the Get Rover Status service. The Get Rover Status service is a feature located in the Swarm Control package responsible for relaying critical information about an individual rover to any node that calls this service function.

The exact details about a rover that is returned is first the battery level of a specified rover. The battery levels of a rover is returned in the form of an INT8, which stores whole numbers on a range of –9,223,372,036,854,775,807 to 9,223,372,036,854,775,807, or 19 digits of precision [6].

This kind of precision is necessary for a swarm management system to accurately determine the threshold and limitations that a rover can handle before returning to a home base and charge its batteries.

The second information of a rover that is returned from the status call is its position. The pose data is returned in the form of a point position in free space with three axises and the orientation of the rover in free space in quaternion form. With these two data values, the central swarm management system is able to accurately determine a rover's location.

Get Rover Status is definitely a functionality that our project will need to utilize because when developing a landing pad on the Moon, the battery level of a rover will be important information at all times. It allows for the rover to be able to operate with a given set of instructions for the task while factoring in trips required back to the charging home base and if a task can be completed with the remaining battery.

The position is also a key piece of information that will be necessary because understanding where a rover is located constantly allows for a central swarm management system to efficiently manage all rovers in a swarm to operate with instructions that do not interfere with the operations of other rovers.
The Get Rover Status is capable of being used by any node within the EZ-RASSOR environment, this will be extremely helpful when developing in other packages.

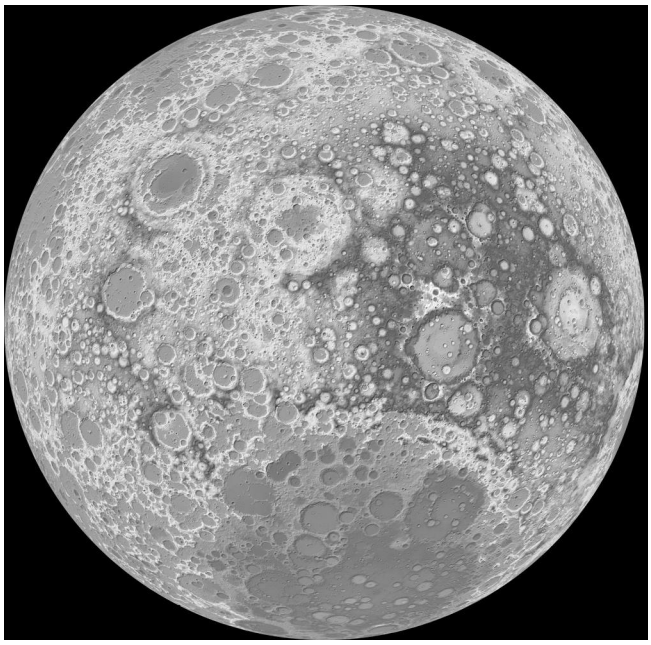
```
from swarm_utils import get_rover_status

# Send status request to rover number 1
rover_status = get_rover_status(1)

# Parse reply message
battery, pose = rover_status.battery, rover_status.pose
```

Get Rover Status Usage Example [7]

Above is an example in code of how the Get Rover Status service would be implemented with the EZ-RASSOR architecture

Topology 2D Simulation

The following is a demo created to show how the change of the topology of the lunar might be simulated. Topology may be demonstrated by a geographic picture where the color of a pixel represents the height of the corresponding location. The areas which have a higher altitude might be lighter and then the lower altitude locations will be darker. For simplicity, we can use a grayscale image of the lunar surface where the closer a pixel is to white, the higher altitude that location has.



A high resolution image for topological analysis [8]

For example, by analyzing the figure we can see that the South Pole has a significantly lower altitude than some areas at the equator of the moon. Furthermore, we can compare two pixel values to see the difference in altitude between them by subtracting their values.

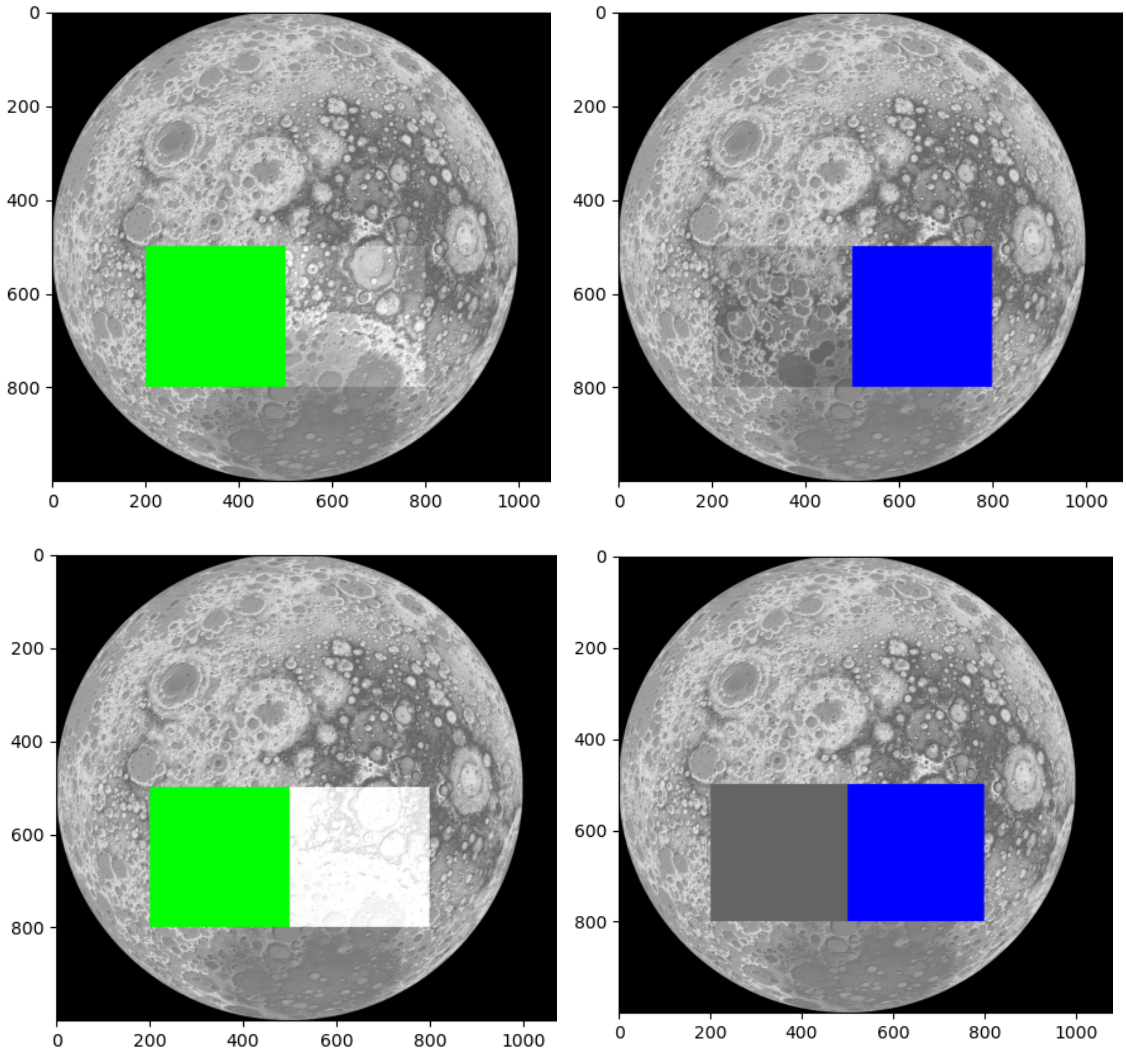
Additionally, we can find the height that each pixel represents by subtracting the highest altitude required from the smallest altitude and dividing the result by the possible pixel values. For example, if the highest point on the moon is 18,046 feet (Mons Huygens), and the lowest point is 13,120 feet then each pixel value

in grayscale would represent 20 feet because there are only 255 possible values in grayscale.

Fortunately, our simulation would just show a subset of the moon; therefore, the range of altitudes would be much smaller. Another solution would be to use RGB rather than grayscale, which has 255^3 number of possible values; therefore, given the range of altitudes on the moon currently each pixel would be .0003 feet or .0035 inches.

To demonstrate the feasibility of we created a naive implementation of the described simulation in MatPlotLib; however, for the sake of ease of implementation, we were liberal with the requirements which are as follows:

1. Rover pathing is not described in the simulation, rather rovers jump from their digging location to their dumping location.
2. The simulation shows the entire moon with grayscale topology.
3. The simulation shows millions of rovers working an impossibly large area for clarity.
4. Rovers which are digging are green. The rovers which are dumping are blue.



Examples of topological changes throughout the simulation

Pixel intensity characterizes land topology, image resolution (amount of pixels) describes distance in space. An extreme example is if we represented the moon with a single pixel, then the pixel would represent a side of the moon or, approximately, the area given of a circle given the radius of the moon divided by the amount of pixels representing the area.

Furthermore, we will want to represent the rover using a couple of pixels. For example, if the rover is a couple of feet wide and long, we would require each pixel to represent approximately a foot, so that the rover could be represented using a few pixels. Therefore, given a 1080x1080 resolution image, we would want to represent a 1080 square foot portion of the moon.

Due to time constraints, only a simple simulation , as described, could be created. Overall, the simulation shows that we may manipulate an image of the moon in such a way that we can represent rovers on the moon changing the lunar surface. Furthermore, the demo demonstrates a plausible method of demonstrating the change of moon topology by changing the values of pixels in a gradient to represent the height in that region.

The code to create an accurate simulation is easy to implement, making it an appealing choice despite Matplotlib being in 2D; however, an external API to receive rover information is required, including rover status and location. Each time a rover moves or performs an action, the simulation will adjust the image to demonstrate the change in rover location, rover status, and the lunar geography, and plot it using MatPlotLib.

Our dependencies include pillow, to load and preprocess the image, numpy, to manipulate the pixels, and Matplotlib.pyplot to show our manipulated image of the moon.

Given a high resolution image of the lunar surface in the same directory as our code, we can load the image, convert it into grayscale for our topology, and convert it back into RGB to represent the rovers, using the following code.

```
 8 # Load the image.
 9 an_image = PIL.Image.open('87_elevation_map.jpg')
10 # Convert image into grayscale for moon topology.
11 grayscale_image = an_image.convert('L')
12 # Convert image back into RGB to show rovers.
13 grayscale_image = grayscale_image.convert('RGB')
```

Load and preprocess image

Using the processed image, we create two numpy arrays that contain the values of each pixel in the image.

```
# Convert image to array of pixels.
dig_array = np.array(grayscale_image)
dump_array = dig_array.copy()
```

Create two arrays for simulation

We create two because we will only have to manipulate half the pixels. This is because, in this simulation, the rovers will only be in two positions and two statuses and we can represent their movements with both images; therefore, we will only have to manipulate the change in topology in real time. In a more accurate simulation, we will only use one image since rover positioning within dig/dump areas will not be defined before processing.

Then for each image, we change either the dumping site or the digging site pixels to blue or green respectively to represent the rovers and their status. Note that a pixel is a tuple of three integers where the first integer represents red, the second green, and the third blue. Therefore, to change the pixels in the desired area, we change the second and third pixel values to the max value (255) in the digging and dumping area respectively.

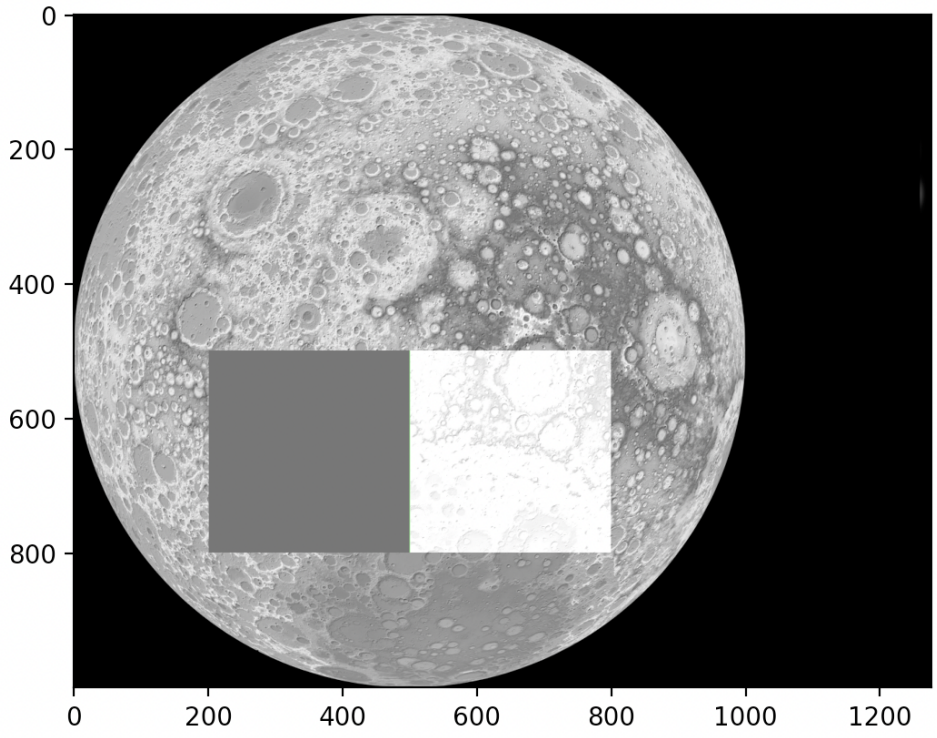
Finally, we may manipulate the images to show the change in topography. The pixels in the digging and dumping area are manipulated by adding to the pixel a value corresponding to the change in topography created by the rover, a parameter we call “step.” Each step is stimulated with an iteration of a while loop which continues until the digging area is the correct altitude. Each iteration alternates which image is manipulated; therefore, we decide which image to use at the beginning of the while loop.

```
    # Change every coordinate in dig/dump area.
    for cord in coordinates:
        pixel = array[cord[0], cord[1]] # Get current pixel to change.
        for i in range(3): # For every value of pixel.
            if (dig): # Choose to add step or subtract step.
                pixel[i] = (pixel[i] - step) if (pixel[i] - step) >= wanted_level[i] else wanted_level[i]
            else:
                pixel[i] = (pixel[i] + step) if (pixel[i] + step) <= 255 else 255
        array[cord[0], cord[1]] = pixel # Save edited pixel back into dig/dump coordinates.
```

Changing the image to reflect topological change

Note that these pixels are grayscale which are represented with each value of the pixel being equal; therefore, to simulate a digging rover we subtract the step from each pixel value, unless the pixel value is equal to the desired altitude, and to simulate a dumping rover we add the step to each pixel value.

Finally, after manipulating the image of the current step, we show the new image, and swap which image to manipulate in the next loop.



Completed desired topological change

The final image of the simulation shows the digging (grey) and dumping (white) coordinates with the colors indicating the topologies. Every pixel in the gray section has the same pixel value; the value equal to “wanted_value” or the desired topology of the landing site. The difference between the original pixel values and the desired pixel values in the digging site was added to the dumping site.

We can calculate the difference which is subtracted from the digging site and added to the dumping site by subtracting the initial pixel values in the digging site from the desired values.

```
diff = []
for cord in dig_coordinates:
    # Subtract the initial pixel value in dig array from the desired pixel value.
    diff.append(dump_array[cord[0]][cord[1]] - wanted_level)
```

Swarm Control

Swarm Control is the second part of our autonomy package within the project. The ‘ezrassor_swarm_control’ package provides a system that manages all of the individual rovers. The package has the responsibility of tasking, deployment, and guidance of each rover.

Path Planning

Firstly, in order to understand how the Swarm Control package calculates paths for rovers, we must understand the algorithm's implementation, and it’s reasoning. Originally these algorithms were not understood by our group, so it took a while to research and understand how the previous team utilized these algorithms.

Why A* search algorithm?

Other algorithms were probably explored by the other group, so let’s consider some popular ones. Dijkstra's Algorithm is certainly one of the first that comes to mind for path planning. Dijkstra’s Algorithm is famous for finding the shortest path, but it’s not known for doing this quickly. The time complexity of Dijksta’s Algorithm is $O(V^2)$, where V represents the size of the input vertices. This is a common, but naive way of implementing Dijkstra’s Algorithm. Another way of doing so to improve the time complexity is to use a min-priority queue, so that you can achieve O(V+ElogV), where V is the number of vertices, and E is the number of edges. We can see that this can become pretty large over long distances and massive graphs.

The next one that comes to mind is a Greedy Best-First Search. Greedy Best-First Search utilizes techniques that allow it to operate faster than Dijksta's Algorithm. One of the key parts to keep in mind of Greedy Best-First Search is that it will not always return the most optimal path, and in its worst cases, it can be pretty bad.

These disadvantages lead us to our A* search algorithm.

A* Search Algorithm

A* is guaranteed to find the shortest path like Dijkstra's Algorithm, and similar to the Greedy Best-First Search, it utilizes a heuristic function to get closer to the target or goal. The implementation of A* keeps in memory a tree of paths which are extended until the problem is solved, but paths which are too costly are dropped along the way. One way to think about this algorithm is by thinking of it like an informed Dijkstra's Algorithm. The A* Search Algorithm starts from a specific node, and attempts to find the path to a given goal node with the smallest cost. This cost can be estimated by a couple factors such as distance travelled, or time taken. The equation to represent this can be shown as below:

$$f(n) = g(n) + h(n)$$

Where n is the next node on the path, g(n) is the cost of the path from the given start node to n, and h(n) is the heuristic function that estimates the cost from the current node to the given target or goal node. The idea is to calculate the local f(n) values of nodes, and visit the neighboring nodes with the lowest f value. One of the key factors of this algorithm is the heuristic function. The way that the heuristic function was implemented in "path_planner.py" is shown below.

Given point p with Cartesian coordinates, (p1,p2), and point q with (q1,q2). The distance between these two points is given by:

$$d(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}.$$

```
f_scores[neighbor] = tent_g + self.euclidean(neighbor, goal)
```

The code for this type of algorithm is incredibly simple, and helps "path_planner.py" to be written cleaner.

This is where f_scores stores the f_scores of all neighbors to the current node. Tent_g represents the g_score of the current node, and self.euclidean is the heuristic method in which we calculate the distance between neighbor and goal.

```
def euclidean(self, a, b):
  a_x, a_y = int(round(a.x)), int(round(a.y))
  b_x, b_y = int(round(b.x)), int(round(b.y))

  return np.sqrt((b.x - a.x) ** 2 + (b.y - a.y) ** 2 + (self.map[b_y,   b_x]
-self.map[a_y, a_x]) ** 2)
```

The euclidean function was a custom implementation, to calculate the distance between 2 ROS points.

The algorithm takes the node with the lowest cost, usually by popping a priority queue or min heap, and calculating the cost function of each of its neighbors, adding the neighbors back into the data structure. The algorithm is finished when the lowest cost node in the min heap is the target node. Each node points to the previous node; therefore, when the lowest cost node is found, we use backtracking to find the path of the node.

Understanding this algorithm helps us to elucidate some portions of the code in the existing Swarm Control package. Furthermore, it allows us to understand the decision making behind using our next topic, the MAPP Algorithm.

Nodes

Currently, the EZ-RASSOR Swarm Control acts as two ROS nodes, which act as the intelligence unit for the swarm of rovers.

swarm_control

Node 1 is the swarm control node and all of this code is defined in the “swarm_control.py” file, and the swarm_control node . This node acts by constantly getting the status of all of the rovers. Some information provided is status levels like the battery, rover positioning, and the distance to the dig site, or charge site. Empty dig sites, or sparsely populated dig sites are prioritized. Furthermore, on low battery, robots are directed to go back to the charging stations. Another thing that this node does is path planning. By utilizing the A* path planning algorithm, the node is able to determine the shortest path to a given point, and task the rover to follow the calculated path.

waypoint_client

The second node on swarm control is the waypoint client. This operates as a simple action client server. This is the node that the swarm control node uses to communicate with the rovers and receive information from. After path planning, the node sends the paths to each waypoint client that each rover has, and then the rovers will execute their tasks.

How the nodes work together

The swarm control package allows for the rovers to operate and carry out tasks as a swarm of rovers. Through the A* path planning algorithm, the rovers are able to calculate efficient ways to travel from one point to another. We can see the code for the A* search algorithm in the “path_planner.py” file. The function “find_path” demonstrates all of the code in python. The A* algorithm loops through lists of neighbors that have been unvisited and are neighbors to the

current node. The code uses self.euclidean of the current node and the neighbor nodes for the heuristic function, and uses the g value to calculate the f value. The function constantly checks it's current path to the path it is taking to check for the shortest and most optimal path.

In our "swarm_control.py" file, we can see how the SwarmController utilizes this path planning to control it's rovers. Our SwarmController class is in control of the swarm of rovers, by constantly getting updates about the rovers, and adjusting paths as necessary. It does this by passing paths to each rover's waypoint client. The waypoint client will respond with information such as battery levels, positions, or tasks.

Multi Agent Path Planning (MAPP)

Unfortunately, the A* algorithm is a pathing algorithm for individual rovers, and a valid location, as described by "path_planner.py" is one that avoids steep mountains and deep craters, not other rovers; therefore, the current implementation of the A* algorithm does not guarantee the avoidance of other rovers [9]. A* algorithm has been extended to prevent collisions with the LRA* algorithm, Local Repair A*, which can significantly reduce runtime *O(mn),* where *m* is the number of graph nodes and *n* is the number of active units, by recalculating the A* algorithm every time a collision occurs. However, the LRA* may generate cycles between units and may have severe bottlenecks, which leads to inconsistency and a longer, possibly never ending, runtime.

A potential solution for a robust, decentralized pathing algorithm is the Multi Agent Path Planning algorithm, MAPP. An advantage of the MAPP algorithm is that it gives guarantees concerning runtime and solution quality compared to the traditional algorithms, FAR and WHCA*. However, MAPP has a low polynomial upper bound worst case runtime and memory usage, more expensive than traditional approaches. Specifically, MAPP has a $O(m^2n^2)$runtime and $O(m^2n)$ space complexity, which is expensive but that is the cost of a decentralized approach with consistent runtimes and no cycles or deadlocks because units of lower priority do not force units of higher priority to advance.

The **Slidable** Instance Class

MAPP is able to reuse preprocessed paths for each node and the possible repositioning between any three adjacent points in a weighted, undirected graph to avoid collisions. A collision is when a node attempts to travel to a space and it is not *blank*, meaning there is another node in that location. In the event of a detected collision, MAPP uses an alternate path to provide a blank location for the node with higher priority to continue its path. Note that alternate paths are solved with non-diagonal movements; however, diagonal movements are equivalent to 2 movements in the cardinal directions, i.e. north-east is north and east.

An *instance* is an undirected, unweighted graph representation of a map, and a non-empty set of mobile units *U*. All units are equal in speed and size. Furthermore, each unit has a starting point and target point pair (s_u, t_u). The starting and target points of all units are unique. The goal of MAPP is to navigate all units from their starting position, s_u, to their target position, t_u while avoiding all fixed and mobile obstacles. Each unit occupies one node in the graph at one time. Each node may move to an unoccupied neighbor node, and they move together at the same time.

A sub-class of instances is *slidable instances* which are solved instances. A mobile unit, u is slidable if and only if a path, $\pi(u) = (l^u_0, l^u_1, ..., l^u_{|\pi(u)|})$, of nodes exists where $l^u_0 = s_u$, $l^u_{|\pi(u)|} = t_u$, such that all the conditions are met:

Definition 1

1. Alternate connectivity: For every three consecutive nodes in a path, l^u_{i-1}, l^u_i, l^u_{i+1} on $\pi(u)$, *except the last trip ending with* t_u. Another alternative path, Ω^u_i exists between l^u_{i-1} *and* l^u_{i+1} which does not go through l^u_{i-1}.
2. Initial Blank: the initial state, l^u_1, is blank
3. Target isolation: No target interferes with π or Ω paths of other units. The following hold for each target nodes, t_u*:*
   a. $\forall_v in\ U \setminus \{u\}$: $t_u\ not\ in\ \pi(v)$
   b. $\forall_v in\ U,\ \forall_i in\ \{1, \ldots, |\pi(v)| - 1)$: $t_u\ not\ in\ \Omega^v_i$

An instance belongs to the class slidable if and only if all units u in U are slidable

In other words, the MAPP algorithm can find a solution for an instance, or graph with multiple rovers each with starting and ending points, if that instance is slidable. An instance is slidable if every rover is slidable. A rover is slidable if and only if it meets 3 conditions:

Definition 1

1. Alternate connectivity: For every three adjacent steps (except the last three) in a calculated path, $step_1$, $step_2$, $step_3$, another, alternative path exists which begins with $step_1$ and ends with $step_3$ but does not include $step_2$.
2. Initial Blank: The node after the initial location (the first step) is empty
3. Target isolation: No node has a target which lies in the path or alternative path of another node.

These three conditions may be verified for an instance in polynomial time. If only a subset of U is slidable, then MAPP may solve their paths as a smaller instance. It's possible for MAPP to solve the original instance given that non-slidiable units are set as active but with lower priorities than slidable units. This implementation is only described later when explicitly stated.

Basic **MAPP**

Algorithm 1

1. To compute the MAPP algorithm for all mobile units, *u in U,* do:
    1. Compute π*(u),* the optimal path from the origin of u to the target of u, using modified A*. For every collision compute alternative paths, $\Omega(u)$.
    2. If *u is slidable,* mark u as slidable
    3. *A* is the set of *slidable* units
    4. While *A* is not empty:
        a. Do progression step
        b. Do reposition step

Note that π(u) and alternative paths are preprocessed and saved in memory; furthermore, rover paths are static, i.e. rovers need to move from one location (digging) to another (dumping) and back again. Pathing within a digging/dumping area might be done separately. Therefore, the most expensive MAPP calculations, $\forall_u$ in *U*: π(u), only need to be calculated once for the entire simulation. π(u) paths are fixed throughout the entire simulation.

Slidable units are uniquely classified into two subsets: *S* of *solved units* and *A* of *active units.* Initially all units are in the active set, and as each one reaches its target, it is moved to the solved set. Units in the solved set will no longer move and will not interfere with any other active units for the rest of the simulation; therefore, solved units are no longer considered.

Definition 2

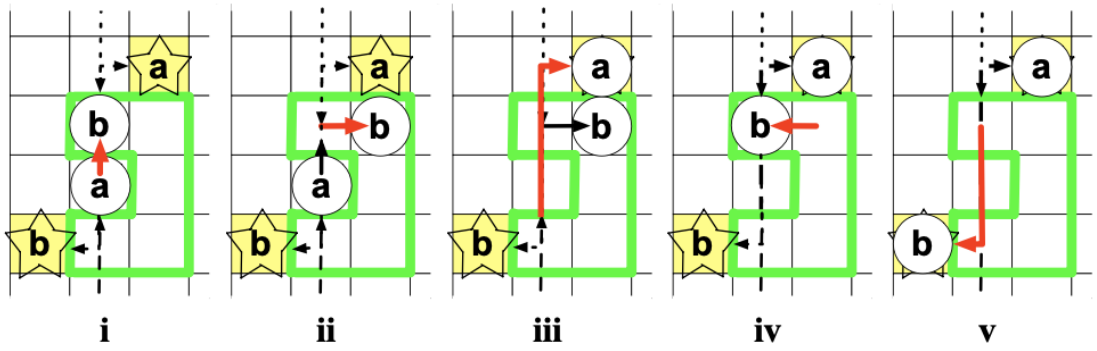
The *advancing condition* of an active unit, u, is satisfied if and only if its current position, *pos(u),* belongs to the path π(u) and the next location in the path is blank.

Definition 3

A *state*, like a snapshot, of the instance is *well positioned* if and only if all active units have their advanced condition satisfied.

Algorithm 2 describes two different steps: progression step and reposition step. Each progression step shrinks the active set by at least one rover by bringing an active rover(s) to its target and moving it from the active set to the solved set. Each progression step has the capability of breaking the advanced condition of one or more active units. The repositioning step makes sure that the advance condition is satisfied for every active unit before the next progression step is taken. In summary, a progression step moves at least one rover to its destination, moving any other rovers in the way to an alternative path. Then, the repositioning step moves any rovers on alternative paths back to their normal paths.

Algorithm 1 Example



Example of basic MAPP algorithm [9]

An example of basic MAPP is as follows using the previous figure for reference. The figure has two active units, *a* and *b*, drawn as circles with their targets, with the same name, drawn as stars. Note, unit *a* has higher priority than unit *b*; therefore, *b* will be moved to an alternative path if it blocks the path of *a*. Figure i-iii is the first progression step, and it begins with *a* having an unsatisfied advancing condition because *b* is blocking the next step in *a*'s path, shown in red. In response, as shown in figure ii, *a*'s next step is cleared, made blank, by sliding *b* down an alternative path, shown in black; in this case, *b* is slid to the right.

Now, as demonstrated by figure ii, the advancing condition of *a* is clear, and the progression step may complete without incident by moving *a* to its target by following the steps in $\pi(a)$. Now, the progression step is finished as *a* has reached its target. Now, *a* is no longer considered for the rest of the algorithm. Figure iv shows the reposition step where *b* is shifted back to its original path by reversing the steps used to move it to an alternative path. After the repositioning step is finished, the advancing condition of *b* is satisfied, the global state is well positioned, and we may proceed to the next progression step. Figure v shows the final progression step. The $\pi(b)$ path is unblocked; therefore, we may shift *b* through every step in $\pi(a)$until it reaches its target. Now, both units have found their targets, and the MAPP algorithm may terminate without a reposition step.

Path Computation

For each problem instance, we compute each path,$\pi(u)$, individually. The paths $\pi(u)$ are static throughout the solving process; therefore, they can be preprocessed; furthermore, as mentioned before, rover pathing between different activity regions will be static throughout the construction process, so we can preprocess all paths in between activity regions for each rover before the construction process. Within construction zones, the instances will be more dynamic since the target locations will change.

Paths are computed using a the A* algorithm; however, the A* algorithm is modified slightly to compute alternative paths, $\Omega(u)$, simultaneously; thus,

ensuring paths satisfy alternative connectivity (definition 1.1). The modifications to A* are as follows:

1. When adding a new node, *x',* a neighbor, *x'',* is added to the path, the priority queue, if and only if there is an alternative path between *x* and *x''* that does not include *x'*.
2. *x'* might have to be expanded three times, once for every potential parent of *x'*, to give every neighbor, *x''*, a change to be added to the priority queue

Moon Topography

The first task that needs to be handled for being able to build a landing pad on the Moon is having a deeper understanding of the surface of the Moon. Well known for its varying features of craters, mountains, valleys and flat lands, the Moon is naturally a terrain that proves to be uninhabitable and unsuitable for most human needs and resources. It is our duty to terraform the land in such a way that it satisfies the requirements of basic development for man made structures.

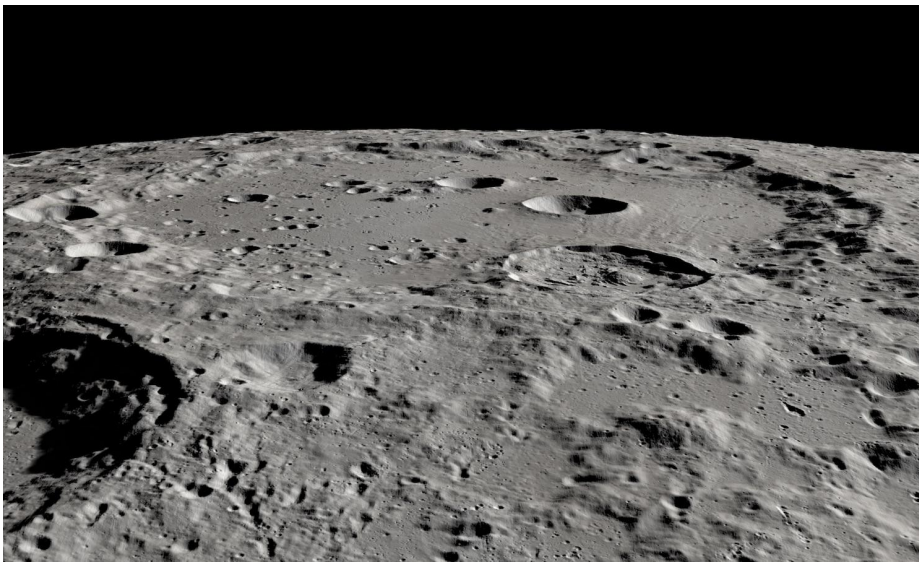
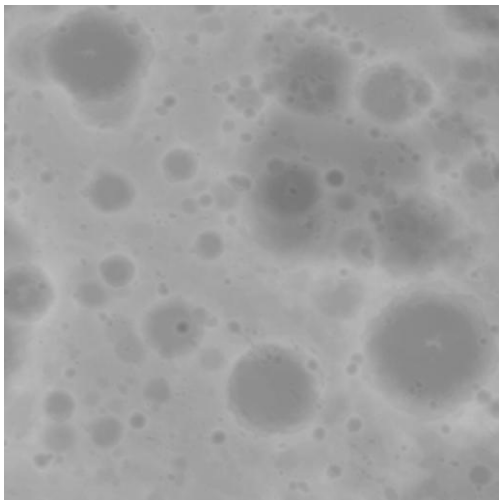


Image of the Moon's surface by NASA (Clavius Crater) [10]

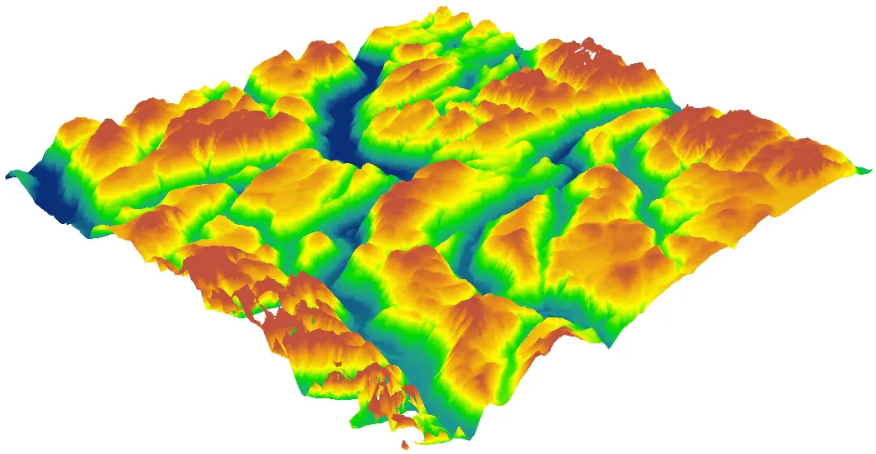
Understanding the complexity of the Moon's surface, it is obvious that terraformation of the land through excavation will be necessary. The EZ-RASSOR rovers will be responsible for this excavation in an optimal and efficient manner. However, the rovers will need information and instructions on what to excavate. This requires knowledge and data of the topography of the surface of the Moon. With this data, the EZ-RASSOR rovers can be given routines on what area of the surface necessitates digging mounds or hole filling to create an optimal leveled surface capable of landing pad development.

The EZ-RASSOR rover comes with built-in depth cameras that allow for top down photography of the ground. This was a feature developed by prior UCF EZ-RASSOR SWARM teams. With these images, it is possible to use both the local and global surveying pictures and transform them into DEM (Digital Elevation Models). These are models and graphs that depict the elevational features of the Moon.

NASA Top Down Lunar View [11]



NASA Digital Elevation Model Image [12]



Colored Topography Digital Elevation Model Example [13]

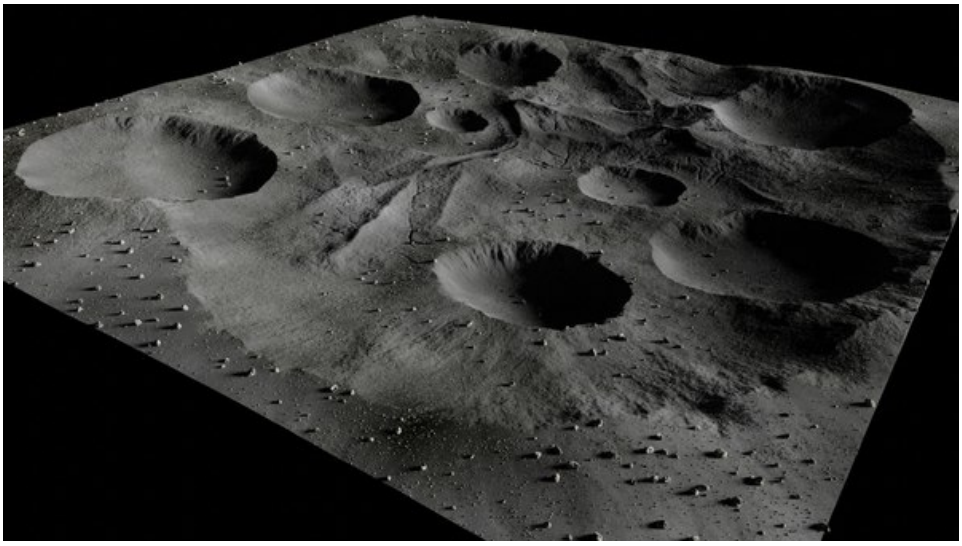
The ideal plan is to convert the given Digital Elevation Models and graphs into some form of topology data to represent the surface of the moon. This data can then be transferred and sent to the EZ-RASSOR rovers to determine how the process of excavation will be completed.

As you can see from the images depicted above, these are real images taken from the EZ-RASSOR depth cameras of the moon's surface. They are DEM processed to portray the elevation density of areas in the photo based on dark shades of highlighting. Once produced in this form, it will be easier to translate this into understandable data for the EZ-RASSOR rovers to interpret and schedule excavation routines.

Surface Maps

Another task that will aid the testing of the EZ-RASSOR rovers is realistic mapped terrain for simulation purposes. Because the project is still under development, in order to test the functionalities and software of the rover, things need to be done in a simulation that closely resembles the actual environment of interest, the Moon. Fine detailed and thorough generated maps that mimic the surface of the Moon are critical to the development of the EZ-RASSOR robot. Previous groups have developed a few maps that help to show the surface of

the moon. Unfortunately for us, we will need to develop more detailed maps, as well as more rigorous maps.



3D Lunar Blender Model Example [14]

Blender is one of the software technologies used for this project that will assist in being able to replicate the terrain and features of the Moon. Blender is an open-source 3-D computer graphics software toolset that excels in creating 3D models and maps. With this tool, highly sophisticated models can be made and allow for testing of features such as object detection, rover path traversal, and overall SWARM management.

The maps made in blender will then be rendered and simulated on Gazebo, a simulation software that will help us mimic the environment of the Moon with the maps produced. Currently we understand that there are heavy limitations to Gazebo such as the inability to manipulate the map environment, so being able to simulate digging or regolith movement in general will require some creativity or a workaround.

Surveying

The RASSOR rovers come equipped with a depth camera which allows the rover to survey the area and gather data and metrics which ultimately allow it to create a digital elevation model. This functionality was implemented by a Gold team (2020-2021). The digital elevation model will allow us to determine whether or not the immediate area is an ideal area to begin the excavation and leveling process.

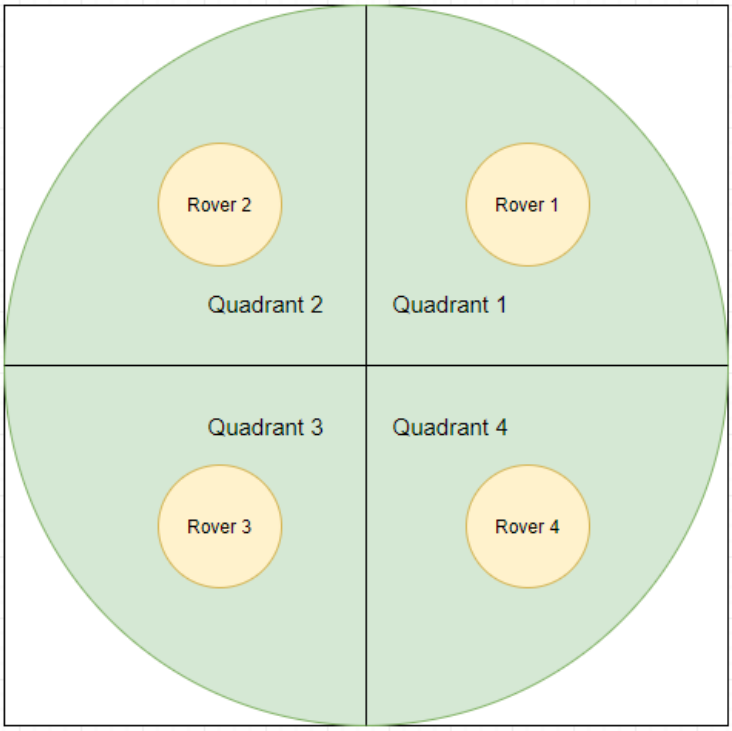
Surveying will also be utilized in the rover correction routine. This will enable the rovers to be used in a search and rescue fashion if a rover was lost. This could be done through robot vision and the camera would be utilized to detect a rover and then send its location to the central task manager for retrieval.

Surveying will also be used to determine if a landing pad was completed properly. If any part of the landing pad was not built to standard, this can be detected via surveying and that information would be sent to the central task manager and a landing pad correction would be run.

Initially we considered taking the elevation delta, the mean elevation based upon all the hills and craters, and comparing that delta to the deltas of adjacent areas to determine which area is relatively the flattest, and then begin the leveling process in the area with the smallest delta. This would allow ideally us to have the smallest hills to level and the smallest craters to fill, but simply taking a delta wouldn't account for other factors, such as having abnormally large craters or hills, or a lack of movable regolith to fill in craters. Other factors include distance from the rover’s “home” and other resources such as landing pad bricks and charging stations.

Pathing

Our plan is to utilize the MAPP algorithm to get our rovers to the digsite. As an initial thought, the swarm pathing we planned to implement when leveling an area or building the landing pad would assign each rover to a quadrant of the circle. Utilizing this approach can help us to achieve faster runtimes with the Swarm Controller. Because the robots are sectioned off, there is less concern of different rovers crossing paths. Furthermore, by dividing the digsite into separate sections, we can track the progress of each rover with their current task in a more manageable fashion.

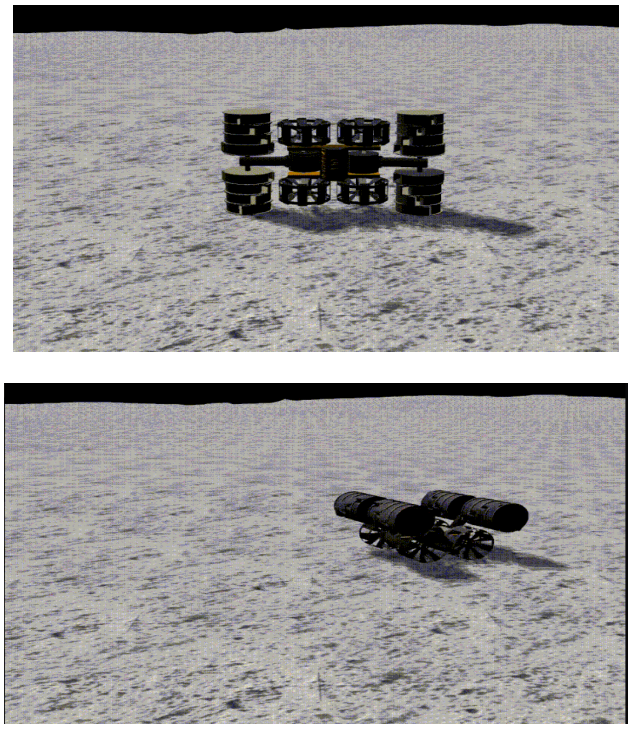


Pathing diagram

Another idea that we've considered is the idea of using the rovers in a sweeping fashion. We could path the rovers into one single side in separate positions, and have them sweep over the digsite slowly.

Climbing Hills and obstacles

The RASSOR rover is currently capable of climbing hills with an incline of (insert) and climbing hills or obstacles with an incline of (insert) when carrying a regolith payload. However the arms and drums themselves can also be used to clamber over obstacles such as boulders. In the event the pathing is incorrect, the rover would be able to right itself using the arms, or simply get over the obstacle using the aforementioned method. This would allow the rover to right itself, then continue on its pre planned path rather than failing to continue on its path.



Rover self-righting [15]

The pictures above show the rover correcting itself via its digging drums and arms.

Digging

“The Rassor rover has rotating drums on its front and back that have slots to gather and deposit regolith. These drums rotate in opposite directions to give each side enough traction to dig on a lunar surface. The counter rotation of each drum allows the rover to dig without moving itself which is especially innovative due to its decreased weight because of the lower gravity on the moon [16]. ” The slots on the drums have shallow scoops that slowly shave the regolith down rather than shoveling it in large scoops. The drums are also mounted on arms which can also be utilized to move the rover over things like boulders or larger obstacles.

Two rovers the EZ-RASSOR(left) and the RASSOR(right) [17]

Once the ideal area for a landing pad is determined, the rover begins its digging routine, digging in the way described above. Currently the way in which rovers will level mountains or areas of higher elevation is simply by digging them down, but there may be an additional attachment added that would allow it to drag or push regolith.

Digging routine

The rovers dig via the two rotating drums. Our initial idea is to simply just take regolith from hills or nearby areas and deposit it in craters to level out the ground.

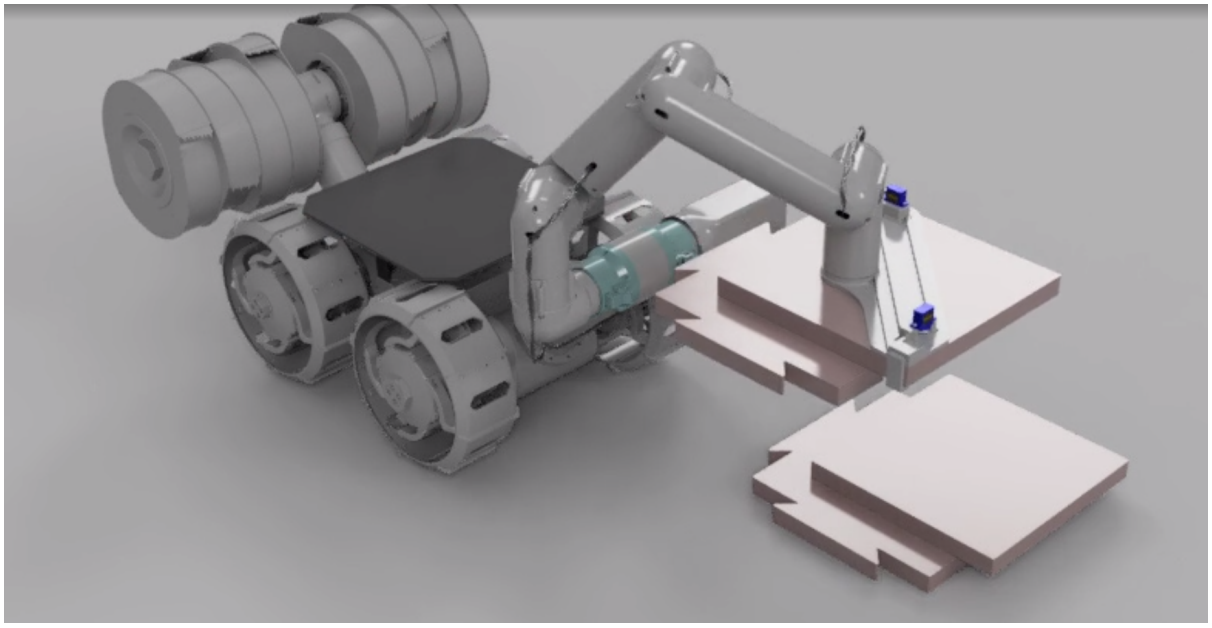
This approach, while simple, may be very time and energy consuming, this is because the rovers would need to constantly drive back and forth between a hill and a crater to begin to level out the land. Each rover would be delegated to its own sector thus making this method time to completion be very dependent on each rover's sector which could potentially cause some rovers to be very behind on its task, delaying the construction of the landing pad.

Instead, a potentially faster approach would be to use an entire swarm to level each sector then have the next swarm work on placing the tiles in the completed sector and to continue that approach in a pipelined fashion. This would be very hard to implement, because the pathing would have to take the other rovers into account as well as their planned paths. As such this is a stretch goal for our project.

Building the landing pad

With the addition of the rover arm which is currently still under development by a different team working on the RASSOR project, the rover will have the ability to place landing pad tiles. Our initial plan for this task would be to delegate a section of the landing pad to each rover and simply have each rover complete its task.

There are a few things to take into account when doing this, how many bricks the rover can carry, the distance to the supply of bricks, and the pathing between the landing pad and the supply of bricks. Ideally each rover would have its own supply of tiles that is placed near its section of the landing pad so it could drive back and forth between the supply of bricks and its delegated section of the landing pad. This would avoid interference between rovers leading to simpler pathing and less time and energy spent on the computation of its path.



Rover with arm [15]

The Central Task Manager

The Central Task manager is the overseer of the rovers. It will communicate to the swarm controllers which will then communicate to the swarm its tasks. The central task manager will also be able to check in on each rover and have information about their locations and status' such as battery level and their current task. This will enable it to determine if it needs to change the rover's current task. For example, if the rover is low on battery, it would tell the rover to return to its "home" to recharge. In this case it could tell a different rover to take over its task. The central task manager will also be able to run a correction routine to attempt to correct any rover that has made any mistakes.

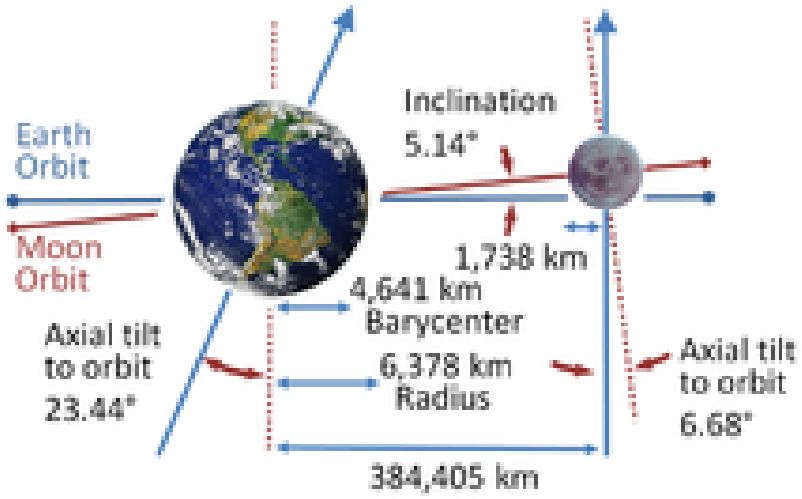
Additionally, the central task manager would be able to create a pipelined workflow for the rovers to increase their efficiency. As an example, the central task manager would be able to assign a rover or multiple rovers to begin placing tiles in an area of the landing pad that has already been leveled while other sections of the landing pad are still being leveled.

The aforementioned functionality would be very difficult and complex and as such, hard to implement so it is currently being considered as a stretch goal.This central task manager will also be able to run a correction routine to attempt to correct any rover that has made any mistakes.

Lunar Atmospheric Conditions

With the interactivity of lunar ground and regolith already existing as a major component to the project's development, an examination of the atmospheric conditions are needed as external variables that have an equally important impact towards building landing pads on the Moon.

Unlike the Earth, the Moon has no seasons. This is a result of a completely different rotational axis tilt on the Moon. The Earth is tilted such that the northern hemisphere and southern hemisphere experience differing quantities of sunlight exposure, causing fluctuations in temperature. The Moon's tilt is miniscule, so insignificant that there is no noticeable "season" presence.

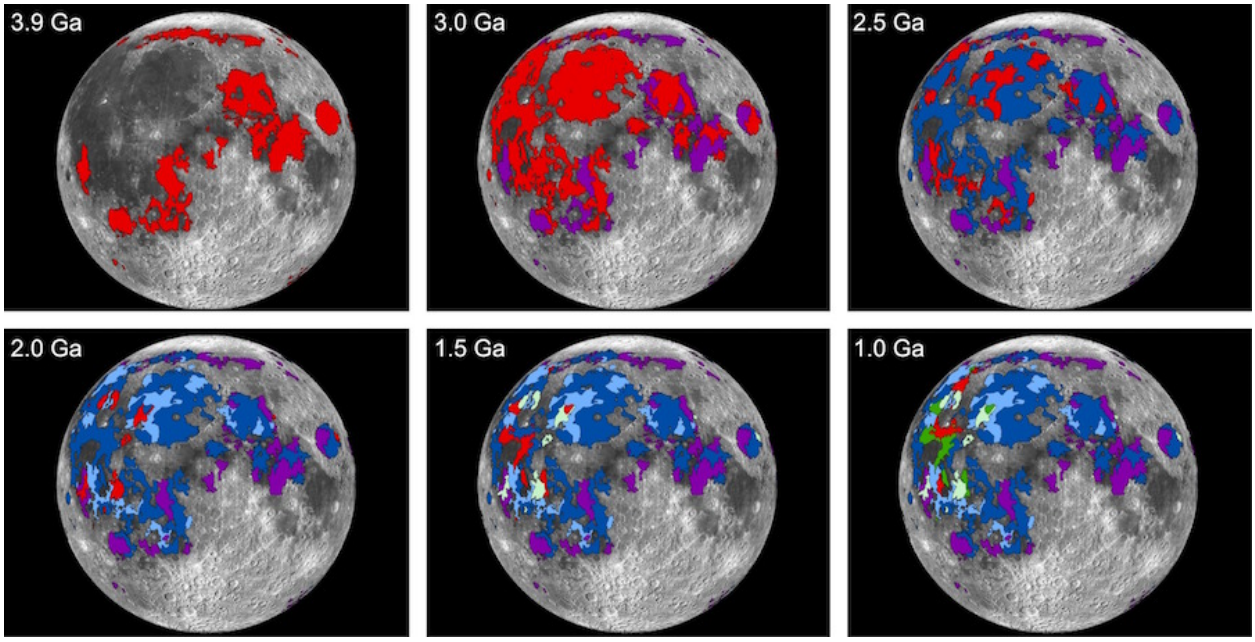


Earth's Rotational Axis Tilt vs. Moon [18]

As a result of the alignment and orientation of placement and rotation of the Moon, there are extreme fluctuations in temperature. This is also caused by the tiny and insignificant protection that the atmosphere provides for the Moon. It is recorded that the average maximum temperature that the surface of the Moon can reach is 260 degrees Fahrenheit. In contrast, the average minimum temperature that the surface of the Moon can reach is -280 degrees Fahrenheit

[19]. These kinds of conditions make it very difficult for any kind of controlled activity on the Moon.

The atmosphere of the Moon is extremely thin in comparison to Earth's atmosphere. Despite a large tri-directional circumference, the Moon's atmosphere only a tiny fraction of the amount of gas molecules present compared to the Earth. The composition of these gases present are abnormal, for instance sodium and potassium [20].



Time Sequence of the Moon's Atmospheric Evidence [21]

It is the EZ-RASSORs responsibility to withstand and mitigate the atmospheric conditions present so that it is possible and even capable to complete the task of building a landing pad on the Moon. Some ideas and solutions that can be utilized towards this problem are an understanding of how the atmosphere will interact with the EZ-RASSOR rover both on a physical level and a chemical level. This can mean testing the material of the EZ-RASSOR rover so that it is durable and able to withstand the atmosphere, but also the landing pad block materials used to build the entire landing zone.

A look into the extent of accuracy and interactivity of the simulation environment to simulate the atmospheric conditions will be very helpful in these tests. The

current software being used is Gazebo. More research must be done on the capabilities of this software to understand what we can represent and account for from the Moon to ensure the rover's success. The most critical factors will be needed to include in simulation are the atmosphere's effect on the Moon's surface and regolith. This could be winds that push and drift lunar regolith into different areas and more.

Moon Regolith

With the Moon's surface being the environment of interest and primary location of work, Moon regolith or "Moon dust" is an important area of interest for us. The EZ-RASSORS will be responsible for the handling of regolith, whether it be digging the regolith with the bucket drums or transporting them to dump in a specified location. For this reason, a deep understanding of the material's attributes and how it behaves is crucial.

Regolith is essentially a layer of loose deposits on top of solid rock, similar to a desert environment, except the sizes of grain can vary and the top layer is not very deep. This top layer includes broken rocks, dust and other similar materials.

Apollo 16 Astronaut Collecting Lunar Regolith [22]

Regolith makes up almost the entire surface of the Moon. The type of regolith developed on the Moon comes from ancient meteor impacts and broken pieces or particles. Often related to the material on Earth, Moon regolith is different from Earth soil because it contains no organic content.

More into the specifics of the actual composition of lunar regolith, it is composed of silicon dioxide glass formed from meter collisions. Because the Moon contains no known water, the material is extraordinarily dry and brittle, this takes any kind of hydrogen dioxide details and information known out of the window [20].

In the case of the EZ-RASSOR project, complete interactivity with the lunar regolith is required to be able to landscape and transform an area to being a landing pad suitable area.

The drum buckets need to transport and dig lunar regolith in an optimal and efficient manner. This is important because if any kind of material goes unaccounted for, major problems could arise. Several to list are lunar dust causing movement and dropping causing other nearby regolith or structures to be moved or affected. Improper care of lunar regolith can also lead to some being distributed into the atmosphere and possibly into orbit. This contradicts the entire project and mission of being able to successfully land rockets onto the Moon in a suitable environment.

NASA RASSOR rover excavating [23]

Specific things to take into account with working with the Moon regolith is transporting and excavating the regolith such that a particular section of the Moon's surface is leveled and viable for development of a landing pad. This will be done with accurate calculation and measurements of exact quantities of material and regolith necessary to move, dig and transport to other locations that would end in a very close net elevation level of altitude. Because the numbers aspect of quantifying and calculating the regolith is critical, taking in factor external implicit side effects will be necessary. This includes the Moon regolith's chemical reaction to certain materials, the physical properties exerted by lunar regolith that could have a negative impact on excavation, or possibly even create positive impacts. It is clear that regolith is such an important topic for research in this project that requires full attention and knowledge in order to successfully create a landing pad on the Moon.

Potential Issues

From doing current research with previous teams and ongoing teams, there are various problems that we believe that we may encounter when striving towards our goal of successful completion of our tasks. These issues arise from the nature of the project itself. The EZ-RASSOR project is an open source project. This project has been worked on by multiple groups from multiple Universities. This project is also currently being worked on by multiple groups at multiple Universities currently; that being said, issues can and will arise when our tasks specifically involve the decisions made by other groups.

The first foreseeable issue would be conflicts that arise when a specific task is dependent on a task of another group. The tasks that we will be completing will have a lot of dependencies on the work already done by other people. If there is an issue with the already developed code, we can request a change be made to the existing code, but do not have the authority to change it ourselves. The process of discovering the error, and requesting it be corrected will cost a lot of time when it comes to the completion of our tasks. Proper planning is needed in order to finish our tasks in a timely fashion.

The second issue would deal with the limitations of the simulation software we are using for this project Gazebo.

- A subproblem of this issue is the issue of switching to another simulation software. Again, since this is already a well-established open sourced project; we do not have the authority to change the major architecture of the project. We can request to change it and explain our reasoning. But like anything, this will take a good amount of time to change and implement.
- A main part of our planned project is the excavation of the land. Since we will primarily be working with the simulation software Gazebo, we will simulate almost all of our solutions on Gazebo. The excavation of the land is crucial and directly related and the primary step before the actual building of the landing pad by laying out the bricks. From current research, the simulation software Gazebo lacks mesh manipulation,

meaning we may not be able to make our EZ-RASSOR rovers dig in the simulation software. If so, this would not be the end of our project. We can still create effective, and optimal solutions without the simulation. But again, we would much rather have the simulation software have the functionally so that we can test as much as we can on our end.

Another problem that may arise is the issue of the version of python. Currently, the project is utilizing python2 instead of python3. The process of upgrading the whole project from python2 to python3 would take a large amount of time and effort. It would also have to be approved by NASA, Florida Space Institute, SWAMPWORKS, and the moderators of the project in order to start the process of conversion.

- But if the project as a whole were to switch to python3. This would present us with a number of advantages instead of using python2.
- This challenge if taken would not just be up to us to implement. Since it is an open source project, we would first have to get permission from our sponsor. And then we would have our group

Budget and Financing

Because this is purely a software project, where the dependencies have no costs, we don't need to buy anything. Notice, the hardware components of EZ-RASSOR are being built by other groups. All 3D printing and major parts will be printed by other teams working on the project and NASA. If any additional costs are to occur, they will be financed by the Florida Space grant given to NASA and the Florida Space Institute.

Expenses

Items	**Costs**
Software Components	$0
Total	**$0.00**

Tasks

Task 1 - Image Processing

- Take a topographical image of the moon
- Gray scale the image, all pixel values between (0-255)
- Zero scale the image, such that all pixel values have had the mean subtracted from them. This allows for us to designate all hills as positive values, and all holes as negative values.
- Pixel Expand the image such that the pixel values are distributed into a more realistic setting. This is because of the scale of the images.

Task 2 - Leveling Land

- Implemented using a level controller, and unique central AI system, also known as swarm intelligence.
- Loop
    - If the quadrant is level, exit loop.
        - Verify the land is level via surveying and complete using built in depth camera and information returned from the rovers.
    - Dig down mountains and place dirt in designated dumping site
    - Fill holes with dirt dug down from the dumping site, or from other holes.

Task 3 - Central Task Manager

- Manages the swarm of rovers to accomplish the tasks above.
- Manages calculations involved:
    - Calculates paths for individual rovers
    - Calculates when individual rovers must recharge

        - Tracks progress of each rover in finishing their tasks
        - Allocates rovers when issues arise

Task 4 - Simulation

- Simulate the solution of autonomous swarm leveling in Gazebo
- Develop a Unity simulation of the solution
    - Terrain manipulation
    - Map scaling
    - Import and fix model of EZ-RASSOR rover
    - Parse msg files and create proper logic
    - Create connection between ROS running on Ubuntu and Unity running on windows
    - Create necessary subscribers in for each rover in the Unity scene, so that each component can get the proper messages from ROS

Another note about the tasks

It has become more apparent that the tasks will be broken down into two main groups. The first group being the original four tasks that were already planned. The second being the illustration of the solutions in a software called Unity. The main goal of the second section of tasks is to illustrate our solutions visually. This is due to our simulation software gazebo being extremely limited in showing the implementation of our solutions. For example, when running a simulation in gazebo, we are unable to have the rovers dig, move dirt, and place entities such as the landing pad tiles. So, in order to show our solutions we will be using another 3rd party software, but also implementing them in Gazebo to the best of our ability.

Tasks Gazebo

Gazebo lacks the ability to give us terrain manipulation. This is one of the reasons why we are using Unity. Therefore, Gazebo will show the entirety of our

solution without the terrain being changed over the duration of our simulation. Unity however is a stretch goal of ours, but will be able to show terrain manipulation.

Task Breakdowns

Our ideas are split into two different software applications. One is Gazebo, and the other is Unity. Our ideas in Gazebo are relatively simple, we wish to implement the Swarm Task Manager, as well as the MAPP algorithm. We think that this will not take very long, and can be worked along simultaneously with the Unity software. Our next software task will require all of us to go through and implement custom algorithms to implement our tasks 2-3.

(Task, Person)	Stanley	Richard	Hung	Camry	Coy
Task 1	X		X		X
Task 2	X		X		X
Task 3	X		X		X
Task 4		X		X	

Task 1 Breakdown

Reading in the image in python. The simplest way to do this, while following along with the practices of the groups before us, is to use the OpenCV library. The idea is to read this image with a gray scale. This looks as follows:

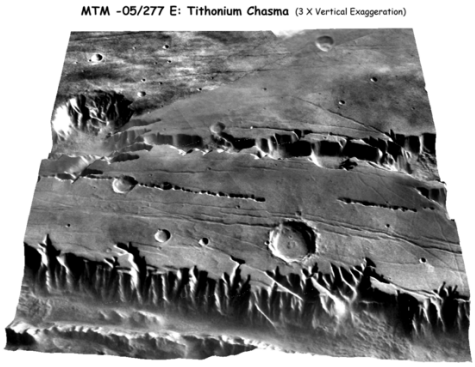
```
    moon_img = cv2.imread("moon_img_1.png", 0)
```

```
    # convert image to float32 and extract shape tuple
    img = moon_img.astype(np.float32)
    img_height, img_width = img.shape
```

This will now return an img array with all the values gray scaled.

The next step in our Task 1 is to zero scale all of the pixel values, and this is as simple as taking the mean of the area, and subtracting all the values by the mean.

The final step in our image processing is to use pixel expansion. OpenCV makes this incredibly easy to do with the resize function. Something like multiply the img_shape and img_height by the appropriate scale values, and resizing with these respective values is all that's needed for the OpenCV resize function.



DEM Topography Example [24]

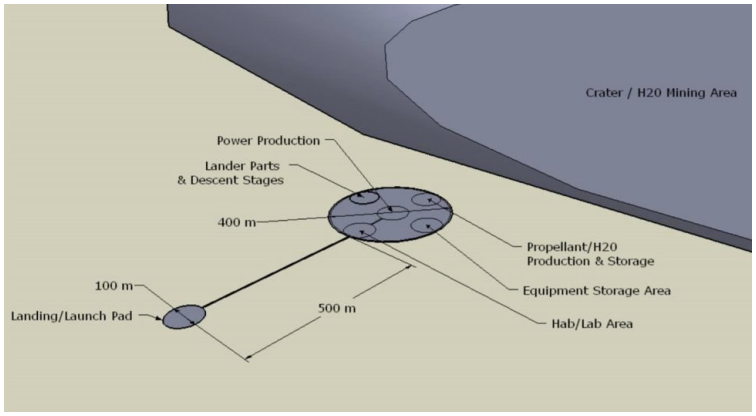
Task 2 Breakdown

Level Land

1. Once the destination is known and provided by the previous task, the swarm of rovers need to travel to designated digging points.

    a. This would be accomplished via pathing algorithms already implemented in the EZ-RASSOR codebase.
        i. We will need to implement a MAPP algorithm to help the rovers work in a swarm like manner.
    b. The pathing will detect and avoid obstacles or hazards and areas which are too steep to drive over, or too deep to exit.
2. Begin the process of digging down peaks, and filling valleys via swarm intelligence and rover arms.
    a. An algorithm will be implemented to determine how rovers will approach leveling out the regolith in a time and energy efficient manner.
    b. Central task manager runs the algorithm and commands the swarm(s) appropriately to accomplish the goals.
3. Verify land is flat via surveying methods and built in depth cameras.

NASA RASSOR Excavating [25]



Possible Lunar Launch Pad Design [26]

Task 3 Breakdown

Task 3: Central Task Manager

This task manager acts as the hive/queen bee to the rover swarms.

1. Oversee project management and routine completion
    a. Assigns rovers to a tasks (swarms) before and during building routine
2. The CTM has the same architecture as the CTM in the current EZ-RASSOR autonomy package
    a. The rovers communicate to the Central Swarm AI via waypoint clients. However, the CTM uses different classes and algorithms to manipulate individual rovers.
3. CTM controls pathing for individual rovers simultaneously by moduleralizing multi-agent pathing planning algorithms
    a. MAPP algorithms are expensive; however, they can be reused. By modularizing the project space into smaller regions processing time is reduced significantly.
4. CTM regions have categories which control what actions individual rovers perform once they reach their destination, i.e. in a digging region, rovers will dig once they reach their target.
5. CTM regions change the target regions where rovers perform their actions. Once enough rover actions have been performed on a target, the target must be moved by the region controller.

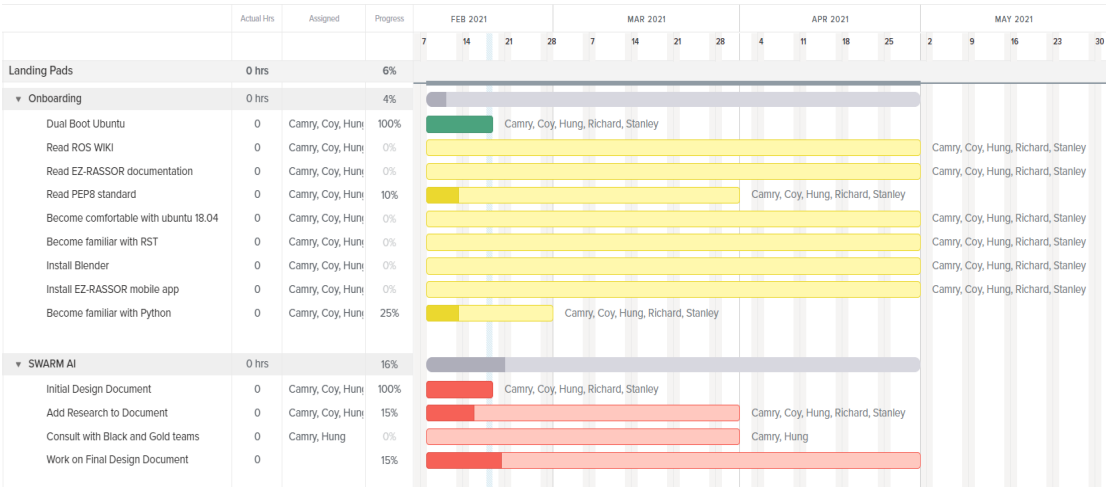
Task 4 Breakdown

Task 4: Simulation

This task will show the simulation running on windows in Unity.

1. Establish a connection between ROS running on an Ubuntu VM and Unity running on Windows
    a. This will require us to configure both our catkin workspace for ROS on the Ubuntu VM AND our Unity project settings.
2. Import Rover model from gazebo simulation
    a. We will need to convert our rover model which is in a xacro format, to a URDF format.
    b. We will also need to clean up this conversion, as the model did not convert properly and was needed to be built from the ground up again.
3. Create rover logic
    a. Once we have created the subscribers to get the proper messages from ROS. We must parse that data and create the necessary logic. For example, when we get an instruction to dig, we must move the arms down and rotate the drums AND change the terrain at standard increments.
4. Create terrain
    a. We must create a terrain gameobject that is editable and scaled to the scale of the gazebo simulation.
    b. We must also reset the terrain after a runthrough of the simulation since Unity does not automatically change the terrain heights back to the original heights
5. Finalize game scene
    a. This is mainly for finalizing cosmetic effects of the game scene
    b. The lighting should be corrected to more accurately reflect a moon setting lighting
    c. We must change the skybox in the unity scene to a space scene.

Gantt Chart for Senior Design 1

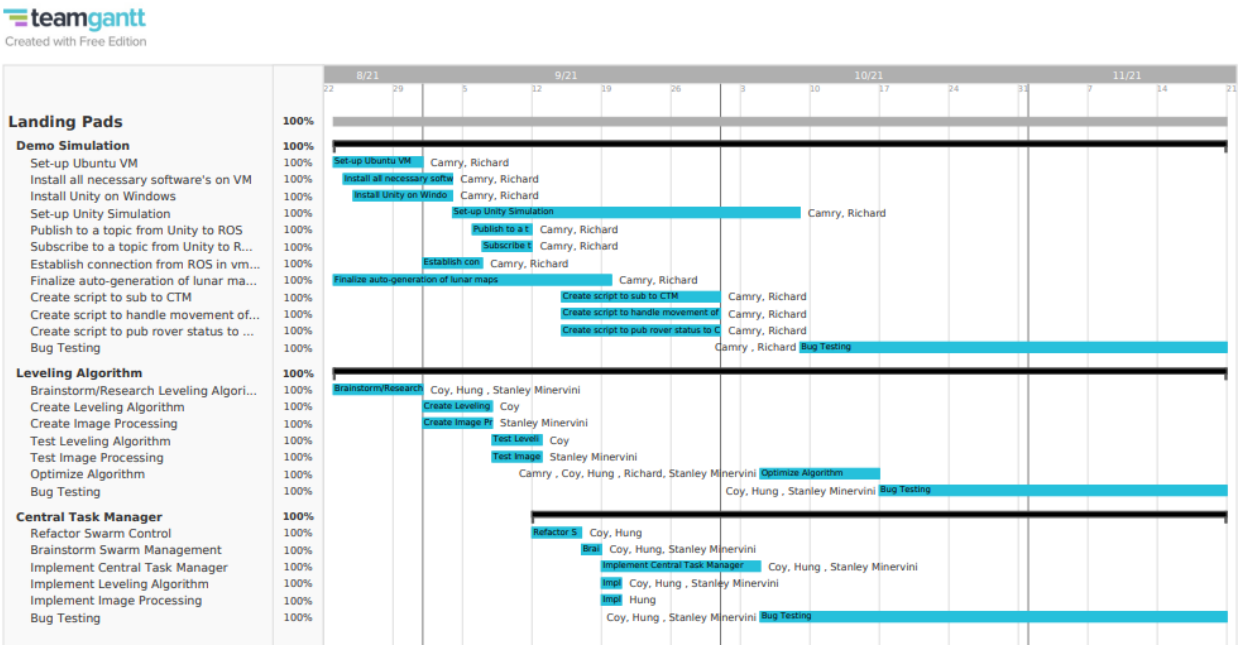


This is the initial prototype for our Gantt Chart that we included in our Senior Design 1 semester.

Gantt Timeline

1. February 8 - April 30: EZ-RASSOR Onboarding
    a. February 8 - February 18: Setting up Ubuntu Dual Boot environment
    b. February 8 - April 30: Research ROS
    c. February 8 - April 30: Read EZ-RASSOR documentation
    d. February 8 - March 31: Read PEP8 Standard
    e. February 8 - April 30: Gain proficiency with Ubuntu environment
    f. February 8 - April 30: Become familiar with RST
    g. February 8 - April 30: Install Blender
    h. February 8 - April 30: Install EZ-RASSOR Mobile Application
    i. February 8 - February 28: Gain proficiency with Python
2. February 8 - March 31: SWARM AI
    a. February 8 - February 18: Initial Project Proposal Document
    b. February 8 - March 31: Contribute research to documentation
    c. February 8 - March 31: Communicate with current/previous teams

Gantt Chart for Senior Design 2



This is our Gantt Chart that we included in our Senior Design 2 semester.

Gantt Timeline

1. August 23 - November 20: Demo Simulation
2. August 23 - November 20: Leveling Algorithm
3. September 12 - November 20: Central Task Manager

Software

ROS - Robot Operating System

ROS logo [27]

ROS, Robot Operating System, is an open source, operating system general enough to control complex robots consisting of many mechanical components. The purpose of ROS is to simplify the robot programming process so anybody can do it with a community and resources to support them. For example, ROS has a package manager, allowing anybody to write, publish, and distribute ROS software from around the world.

ROS provides the functionality any user can expect from an operating system. ROS controls low-level devices and provides an interface for programmers to control them. ROS centralizes APIs for controlling individual robot components, so that the programmer may control each component easily. Furthermore, ROS abstracts a robot into each of its components, reducing the complexity of programming a robot. ROS provides methods for components to communicate, allowing the robot components to work together to accomplish a task. Communication between components may be synchronous or asynchronous, instant or delayed. Specifically, at runtime, a ROS controlled robot is a “graph” of components, called the ROS Computational Graph, which may communicate directly (peer-to-peer) to each other.

ROS is the method in which every programmer in the EZ-RASSOR project interacts with the robot; therefore, it is a crucial keystone of any EZ-RASSOR project. For programmers, the ROS operating system is the lowest level of

programming. Overall, ROS documentation in the EX-RASSOR project is lacking; however, it is crucial for future teams to understand which parts of ROS are used in the project. The following describes the ROS features used by the EZ-RASSOR SWARM Landing Pad team.

ROS melodic Morenia [28]

Names

ROS Names are the fundamental method of identifying ROS programs, including modules and packages. This project will consist of programs with names which are grouped together with a name and communicate internally and externally to other programs using name addressing.

Graph Resource Names

Graph Resource Names is a hierarchical naming structure used to identify all resources in a ROS. Names are critical for managing nodes (machine components) to work together and accomplish a task. Furthermore, the Graph Resource Name structure provides encapsulation and scope between packages in a graph or package. Therefore, names are required for communicating with individual nodes and entire graphs of nodes.

A Name may contain sub names with which it shares resources. For example, the name “/stanford/robot/name” refers to a machine component encapsulated by the name robot encapsulated by the name stanford. Each name is separated by a slash. Each Name is a namespace which can create and access resources. Furthermore Names may access resources within or above their own namespace. Using the example from before, the Name “name” is in the namespace “robot” and can access resources from both the “robot” and “stanford” namespace. Any Names used by resources are resolved relatively, so Names are searched from within the current namespace and then searched above the current namespace.

Package Resource Names

Package Resource Names are condensed Names which refer to resources in a 3rd party ROS package. ROS has the capability of locating a file in a package; therefore, Package Resource Names are just the name of a package followed by a forward slash and the name of the desired resource. For example, the Name “automation/Controller” refers to the Controller node within the Automation package.

Nodes

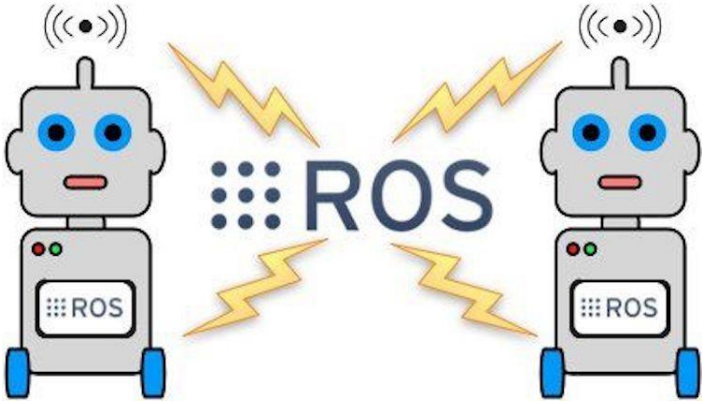
A Node is a process that performs some computation. Nodes are defined incredibly generally because they are the foundation of any ROS project. Nodes are in charge of the most minute details of a system and actions of a robot because they are the lowest form of control in a ROS Computational Graph. This organizational paradigm has many benefits, including isolating faults to a few nodes and reducing the complexity of the entire graph. In fact, ROS programmers need to abstract each machine component in a robot into Nodes and then define their behaviors and communication patterns. Nodes are combined together into a ROS Computational Graph. Nodes communicate to each other within the graph using Topics, Services, or a Parameter Server.

Nodes each have a Graph Resource Name which uniquely identifies them to the rest of the symple. Note that the Graph Resource Name includes every namespace above the node name. Furthermore, Nodes have a Node Type which are defined as the Package Name and the Node Name.

Topics

Topics are communication buses with Names over which Nodes send Messages. Topics do not know which nodes they are communicating with, rather they publish messages for any node to subscribe to. Nodes which subscribe to a Topic will be streamed a Message whenever that Topic publishes it. Topics have defined Types which they must adhere to. Nodes may publish Messages on Topics of a matching Type. These restrictions separate the production and consumption of Messages and reduce code complexity.

Messages



Ross communication illustration [29]

A message is a data structure which contains fields with types. These types may be standard primitives (integer, float, string) or arrays of those types. Furthermore, abstract, nested data structures are supported as fields. Describing the data structures of individual messages are “msg files,” which are stored in a subdirectory in the package.

Services

A Service is another communication method between nodes. Rather than a one to many, one way communication paradigm like Topics, Services provide duplex request and reply protocols between two Nodes. Services are defined by two messages, the request message and the reply message. A Node may offer the Service publicly until another Node sends a request message, then the original Node may send a reply and begin back and forth communication.

Services are defined within a ".srv" file. Like Topics, Services have Types defined as the Name of the package encapsulated the Service, and the name of the Service itself, or the Package Resource Name of the ".srv" file.

Action Servers

Action Servers is a more robust communication protocol than Topics, Services, and Parameters. Action Servers provide similar communication as Services through request, reply protocols between two nodes. However, Action Servers allow requests to interrupt responses, possibly changing the request. Action Servers provide fast communication between nodes which may be preempted and changed at execution time.

Goals

Specifically, Action Servers allow a Action Client to request a "goal" from a server to be completed. A Goal might be a complex execution which will take time to complete. Upon request, an Action Server will complete a goal by executing various subtasks. A Goal is communicated through Messages.

Feedback

Feedback is a method for the server to inform the Action Client about the progress of the requested Goal. While completing the Goal, the Server continuously updates the Action Client with Messages, which might elicit a response from the Action Client requesting a change to the Goal in progress.

Result

A Result is sent from the Action Server to the Action Client upon completion of the Goal. Result is only sent exactly once in a routine. Note Feedback is sent continuously from the Action Server to the Action Client throughout the execution of the Goal. The Result Message provides the necessary information for an Action Client to understand the status of the Goal upon completion. For example, the Goal might be to gather information and the Result would contain this data. The Action Client may use the information in the Result Message to dictate which actions to perform next.

Parameters

A Parameter is a server, shared through an API, of a dictionary or mapping of resources. Nodes may use this server to hold state and store data at run time. However, Parameter Servers are not high performance and cannot deliver information very quickly. Therefore, the Parameter Server is best used for static data which may be viewed globally by any Nodes in the graph. For instant data transfer we use Action Servers.

Parameters may store primitive data types, such as 32-bit integers; booleans; strings; etc., and composite types such as lists and dictionaries, as well as abstract, nested data structures.

Rospy

Rospy is a communication module for ROS nodes written in python, and our primary interface with ROS topics, services, and parameters. Fortunately, in python style, the Rospy API focuses more on speed and ease of implementation rather than efficiency.

All rospy functionality is encapsulated in a single, instantiated ROS node which creates communication classes and establishes connections between them. Therefore, the rospy node instantiate function, "rospy.init_node('my_node_name')," should be called only once.

Rospy will convert ROS messages into python code and vice versa. Message names are translated from "stanford/msg/alerts.msg" to "stanford.msg.alerts," and srv files are converted from "stanford/srv/Bar.srv" to "stanford.srv.Bar." ROS messages in python may be manipulated like any other variable and may be instantiated by using the "std_msgs" object.

To publish messages to a topic, Rospy provides a handler object through instiantiation of the publisher class. The publisher constructor requires a topic name, a message class (type), which must match that of the topic, and a queue size, which holds messages which await publishing. A publisher object may be created like the following, "pub = rosby.Publisher('topic name', std_msgs.msg.String, queue_size)." Publishers must serialize the message and write it in a buffer; therefore, messages must await processing time before being published and any unstable connections might lead to longer dequeuing times or even indefinite wait times. Lastly, messages are published to topics by calling the "publish" method of the publisher object and passing the desired message to publish, for example, "publisher_name.publish(message)."

To subscribe to a topic, a subscriber object is required. The instantiation of a subscriber class into an object attaches a callback function to be called whenever the requested topic publishes a message. The callback method is given a parameter which contains the contents of the published message. It is up to the programmer to decide what to do with the data once it is passed to the callback function. To instantiate a subscriber object, the topic name, type, and

callback function are required arguments. For example, “rospy.Subscriber(“chatter”, String, callback).”

Gazebo

Gazebo logo [30]

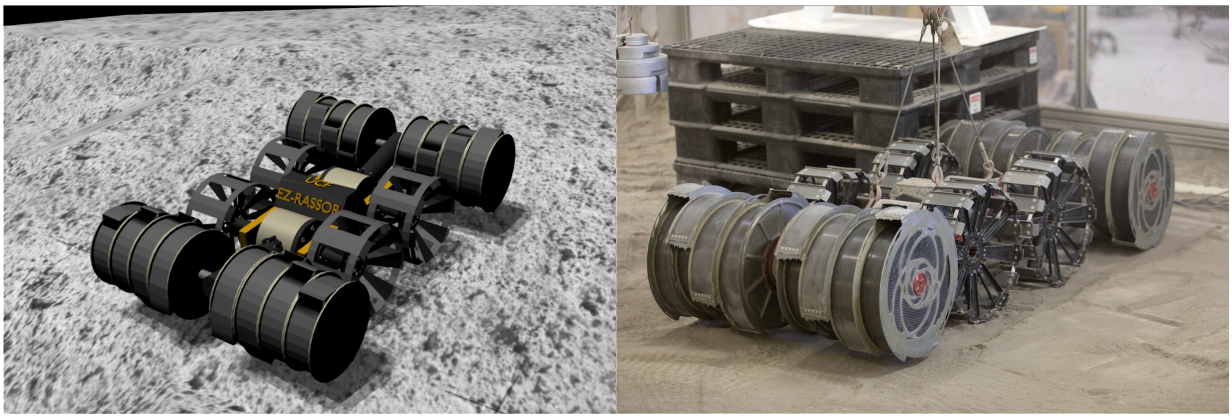
Gazebo is a graphical robot simulation software that has been utilized by the previous teams working on the EZ-RASSOR and RASSOR project. It is a very useful program that allows us to import models and maps from blender. In our case the rassor rover and all of its functionalities such as its wheels, arms, drums and camera are accessible and usable. Gazebo can also use the sensor data from the built in depth camera and any other sensors that are equipped on the RASSOR rover. This enables us to gather data from each sensor as if it was collecting data in real life, so we can use this data to test and help develop our code. For example, we can build a map in blender with various obstacles such as craters or rocks so the camera can see these things, then run the obstacle avoidance functionality and ultimately reroute its path and avoid the obstacle.

EZ-RASSOR controller from mobile application [15]

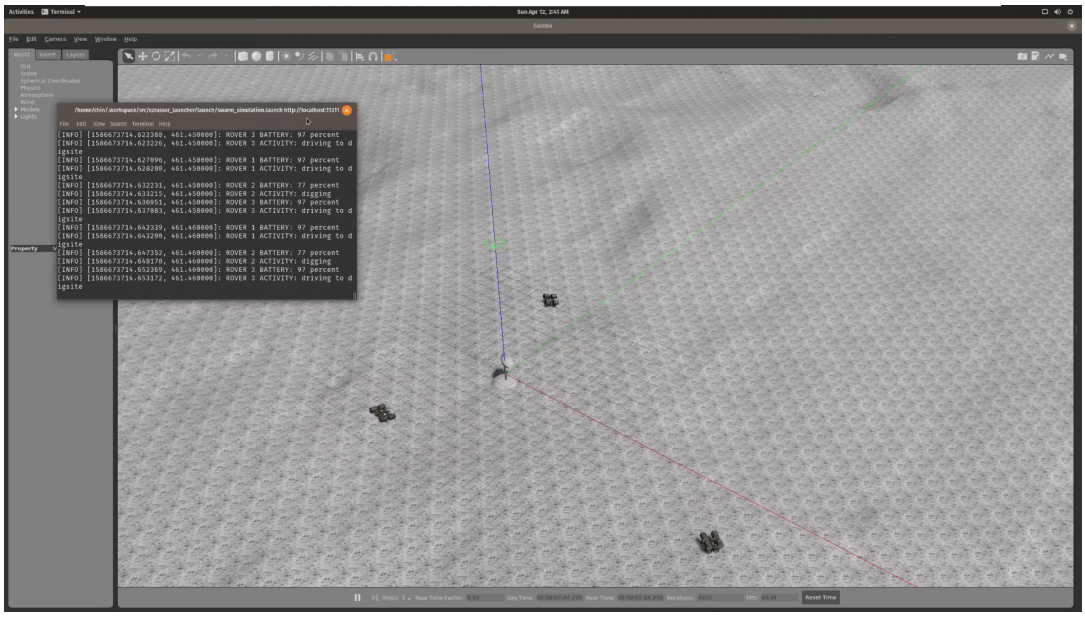
More on Gazebo

The gazebo simulation also allows us to spawn multiple rovers if need be to attempt to simulate a swarm, this will allow us to work on our swarm intelligence and develop our code to allow the rovers to work in tandem.

This is incredibly useful because it allows us to have a visual representation and simulation to see if it reflects what we expect to happen on paper. In addition, testing this with physical rovers currently isn't feasible because we do not have access to the rovers because they are still being built, and even if we did have access to them, testing and maintaining a whole swarm of rovers would be costly and time consuming.



EZ-RASSOR vs. RASSOR [15]



Simulation environment in gazebo [15]

In the first image, you can see the comparison between the simulated rover on the left, and the NASA RASSOR Rover on the right. You can see that the Gazebo simulation gives us a fairly accurate representation of what the RASSOR Rover looks like, but it costs nothing for us.

In the second image, you can see 3 rovers in the Gazebo simulation. They are on their way to different parts of the Gazebo simulated moon world.

Currently the Gazebo simulation paired with ROS has a lunar surface with little to no obstacles or objects to test the surveillance and pathing abilities of the rover, so we will need to update the current environment with new maps built in blender.

Ideally these maps will feature a large variance in land with things such as craters and hills, and regolith that can be interacted with. This will require us to develop new height maps, or Digital Elevation Maps in blender, then import them over to gazebo.

We would also like to implement the ability to place landing pad bricks in this simulation environment to be able to simulate a landing pad creation routine. These kinds of instructions will come from some kind of swarm manager control system that will calculate the necessary steps for construction.

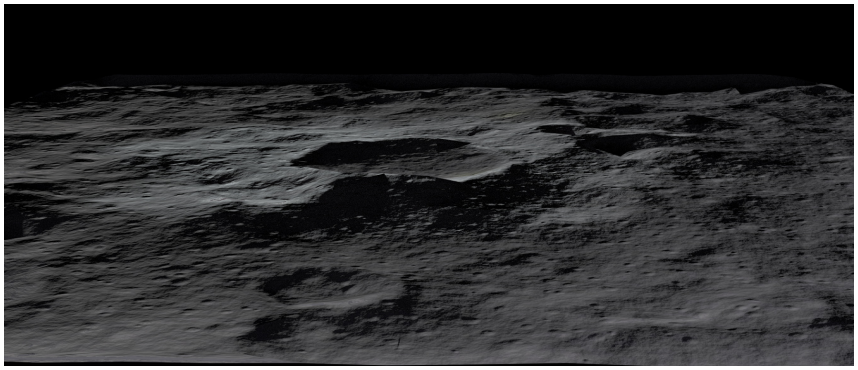
Blender

Blender Logo [31]

Blender is an open source computer graphics software toolset that allows for the creation of various things such as animation, models and interactive applications. In regard to the Gazebo simulation environment, blender will enable us to create maps of the moon based off of pictures provided by NASA. This will be an incredible asset because we will be able to run tests in the gazebo simulation environment to assess the pathing of single rovers and swarms of rovers in an environment that would be very hard to recreate or imitate on earth.

This will allow us to develop and edit our code based on our results and findings in the simulation. In addition blender can create models that can be interacted with, this will allow us to create some sort of regolith that the RASSOR rovers can interact with.

Simulating this is essential because we will be able to simulate digging and excavation of an area in the environment and ultimately leveling an area to allow for the construction of a landing pad. Blender should also allow us to create a landing pad brick object, which can be used to construct the landing pad.



Blender lunar map [32]

Python

Python Logo [33]

Python is the main programming language utilized in the EZ-RASSOR project, it is very simple and easy to pick up which is ideal for this project because many different teams have and will add to this project. Python also has ties to machine learning and artificial intelligence and already has software used to develop and support those fields such as Tensor Flow, Theano and Keras.

For this project, we will be implementing our solutions in Python2. This decision comes from the fact that the project is already in Python2 rather than Python3.

Ubuntu 18.04

Ubuntu Logo [34]

Similar to the aforementioned softwares this Ubuntu 18.04 was used by the previous teams working on the RASSOR project, we currently are unable to upgrade to a more recent version of ubuntu because our simulation software has various errors in newer versions of ubuntu. However Ubuntu is lightweight, fast and efficient.

This is ideal because the operating system must be able to run on weaker hardware such a Raspberry PI because that is what our RASSOR rovers utilize. In addition ubuntu comes with features such as GIT which allows for easy integration and use of git commands to make changes and develop the codebase.

Discord

Discord Logo [35]

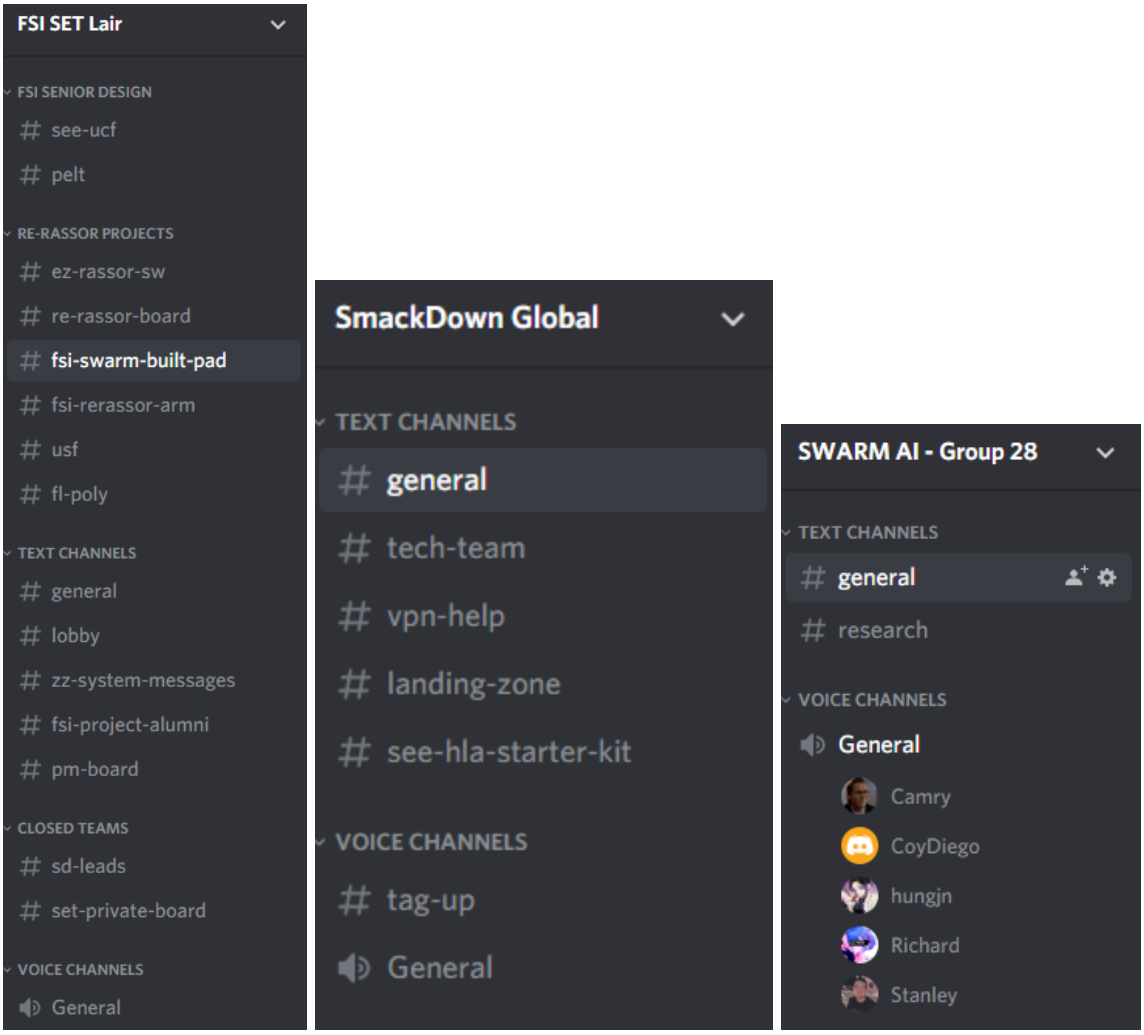
Discord serves as our main means of communication throughout this entire project. We regularly communicated on our own discord server through texts or via phone calls. We used our general channel for communication regarding our collaboration on our documents and power points. We also had another channel called research. In this channel, we regularly discussed some links and articles we had found regarding our project and our own endeavors in robotics research. It was helpful for all of us to check these two channels and understand the tasks and paths that all of us were following.

Additionally, we had voice call meetings at least once a week. In these meetings we prioritized talking about deadlines of our upcoming assignments, this way we could stay on track to meeting our goals. Furthermore, we also used our voice call meetings to discuss progress that we had made over the last week, such as any important observations we had made.

Another thing that we did is that we also utilized our meeting time to ask any questions that any of us had. Originally, there were a lot of misunderstandings we all had about the group, and our group's common knowledge wasn't very common among each group member. So, we all discussed our ideas, and our way going forwards to clarify what it was we were all doing. This helped us to avoid multiple people researching the same topics, or talking about the same things during our meeting times.

Furthermore, another discord server we used was the Florida Space Institute (FSI) server. This server was created by our sponsor, Mike Conroy. The FSI server allowed for us to easily communicate with our sponsor via text and general channels. It also gave us a way to video call with our sponsor. This allowed for clarifications regarding our project and our tasks at hand. He clearly specified what our goals for the project were, and some of the background and nature of the project we were getting into. Another great resource that this channel gave to us was the ability to communicate with other teams also on the server. This really helped us determine where other groups were at, and how the group should proceed.

Below is an image of how the Florida Space Institute discord server and other project related servers. The FSI discord is organized for effective communication. Each team responsible for a component of the EZ-RASSOR project is designated to a specific channel for organization and modular discussions. All channels are also public to all members so that cross communication and collaboration is also possible. This infrastructure provides the most efficient means of teamwork on a remote level.



This is an illustration of the structure of communication using: the Florida Space Institute's official discord (left), Smackdown Globals official discord (center), SWARM AI's official discord (right)

Github

Github Logo [36]

Github serves as the group's way of sharing code with each other. The list of requirements for the EZ-RASSOR project makes it clear that our software needs to be modular, standardized, well-documented, and easily reproducible. This is where Github serves as the best option for our source control.

Founded in 2008, Github has become one of the most popular source control services, by offering a cloud based Git service. Github has several features that make it incredibly advantageous for us to use. One of those is the topic of opening pull requests, and merging pull requests. Another cool feature is the repositories that Github has to offer, and each of the repositories cool features.

The EZ-RASSOR project is a multi-faceted project that has many teams operating on it at once. Thus, it is also a very complicated project by this nature. In order to ensure that the project does not reach merge conflicts, we have pull requests in Github. These requests allow for all of the teams operating on this project to submit their code for review.

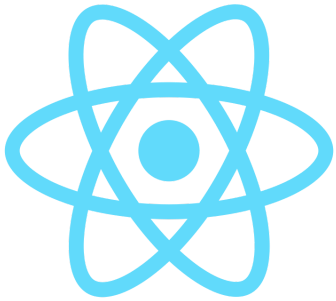
This code is then reviewed by the moderators, and determined whether or not it is sufficient code to add to the repository. By using Github to contain all of our

code in one place, we also meet the need of open source, and easily reproducible. This code is available to anyone in the world due to the availability of our repository.

The topic of well-documented was something that we have a foundation to build upon. The past groups that have worked on this project have developed an onboarding process, and a wiki for our Github repository. This documentation gave us the opportunity to catch onto the project incredibly quickly. While it also gave us a lot of homework to do, it served as an incredibly useful resource provided to us through Github. This gives us the foundation to add our own code, and develop and document our own code alongside the other code in the repository.

Our plan for utilizing Github in an efficient manner is to fork from the main repository and develop code in our own forked repository. The main repository is under control from the Florida Space Institute Organization. Our development branch will be named after the specific task in Jira. This is where we will edit the code we think we need to edit, and also implement our own algorithms and code. Once we have finalized our individual changes, merge our development branch into the mainline branch. Then we will create pull requests for all of the code sections we wish to append from our branch. Our project manager will review all of the code we have written and accept the pull requests in order to merge them into the main repository branch.

React Native



React Native Logo [37]

React native allows you to create mobile apps for IOS or android. It is a framework that allows for easy and streamline development of mobile apps, it uses what are called “ views “ which are the basic building blocks of UI when using react native. These views are a small rectangular element that can be used for several purposes such as displaying text, displaying an image or video, as well as doing things such as being a button used to respond to user input. Many elements in react native are some sort of view and some views can even be a container for other views. React native of course allows you to use many different types of elements to fulfil the needs of your mobile app.

React core four

There are four main concepts used when developing in react, these concepts are components, jsx, props and state. Components are akin to blueprints, these blueprints can contain things like text boxes or display something. For example the code snippet below renders a text element that says “ Regolith! “. This function component is exported with Javascripts export default.

```
import React from 'react';
import { Text } from 'react-native';

const moon = () => {
  return (
    <Text>Regolith!</Text>
  );
}

export default moon;
```

[38]

JSX

Jsx is a special syntax that allows you to write elements inside of JavaScript. For example , this is a JavaScript button that compiles into a react button element. But instead of simply making it in react it is made via JSX.

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

[39]

The snippet above becomes this when compiled.

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

[39]

Props

Props is an abbreviation of properties, these are simply ways to customize components in react, for example if you were to have a text box that displayed text, you could do things such as changing its color, font or size. Different types of components have different properties associated with them and the majority of react native's core components can be altered and changed with the use of properties. As an example this code snippet would add a caption to the image component.

```
import React from 'react';
import { Text, View, Image } from 'react-native';

const lunarApp = () => {
  return (
    <View>
      <Image
        source={{uri:
"https://cdn.mos.cms.futurecdn.net/vxS234BygNDuK7dnojmhUF.jpg"}}
        style={{width: 200, height: 200}}
      />
      <Text>A lunar surface, devoid of life, but filled with secrets.</Text>
    </View>
  );
}

export default lunarApp;
```

[40]

State

State, as the name implies, refers to the current state a component is in. The state of a component is essentially data that refers to how a component may change over time based on many different factors. For example, if you had a button that was a state component, upon button press the component would change states because it has been interacted with and the button may change in some way or trigger some kind of action.

Unity

Unity Logo [41]

This section is dedicated to Unity to show proper illustration of our group's solutions. Originally, we had planned on using the simulation software, Gazebo, to illustrate our solution to all of our tasks. We still plan on doing this, but Gazebo has limitations to its software. For one, entities such as a rover in Gazebo are unable to edit the terrain around it. This is due to the maps being static in Gazebo and not allowing for mesh manipulation. This is a huge problem for our project. An entire task is just about the excavation of the land by multiple swarms of rovers. Another problem we face is the inability to place objects in Gazebo when a simulation is running. This is another huge problem since an entire task is dedicated to dumping regolith to level an area. The solution to this

is to use Unity to illustrate our swarms of rovers doing these tasks with our solutions of optimal efficiency

Simulation in unity

Unity will be utilized to simulate our swarm AI, this will be incredibly helpful to represent what we actually did because unity has a plethora of functionality that gazebo lacks. For example we will be able to simulate terrain deformation which will allow us to simulate excavation. We can either do this via mesh manipulation in unity, or by changing the color of the terrain being worked on to represent its height, similar to a topography map.

Meshes in Unity

Meshes are akin to a net of points with lines connecting the points to form triangles. These meshes define the basic shape of an object, which can be things such as a rover, a space rock or even the terrain. These meshes are also composed of a UV map which is the material an object has which will be useful in the case that we'd want to add a lunar surface like material to the terrain or other objects, in addition this can also be used to change the color of the object or terrain when to indicate a change in the height of the surface or object [42]. These meshes also allow us to use simple scripts(?) to change the meshes points based on some sort of action.

In our case this would be lowering an area in some sort of predefined fashion to simulate digging, or doing the opposite and raising the area when the rovers are filling some kind of hole or crater. In the case of digging we would need to lower the Y axis coordinate of a few points in the mesh to dig down the surface, when the vertices are lowered the mesh will stretch down while the other points in the mesh maintain their coordinates. To keep the other points in the mesh in place a function such as the “PullSimilarVertices'' function can be used, this enables us to keep the mesh from breaking when it's being manipulated by finding the

target vertex position from the vertices array, then it finds all of the vertices that share the position of the target vertex, and updates the position of the related vertices, and lastly it redraws the mesh based on the updated vertices position. As aforementioned, we could also do something similar, rather than manipulating the terrain's height, change the terrain color to indicate the rover's action [43].

Surveying in Unity

The depth camera functionality is being worked on by another team, so for now we can simply use the terrain mesh map to determine if the area in question is good to be used as an area to build a landing pad, if possible we could implement a camera that is able to identify if an area if good to build a landing pad. In addition, to ensure that the landing pad is built properly, we can compare it against a predetermined shape of the landing pad to determine if it was built properly. However you can implement a depth camera in unity]

Scripts in unity

Scripts allow us to control objects in unity, in our case this could be things like the rover script will allow us to develop the swarm AI.

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }
```

```
    // Update is called once per frame
    void Update () {

    }
}
```

Anatomy of a script file [44]

Unity uses c# as its programming language , it's an industry standard language which is easier and more straightforward to learn than something like c++. Moreover, scripts use c#, and these scripts also allow you to implement and logic you need into your simulation or game. These scripts make a connection with unity by implementing a class that takes from the built in class called MonoBehaviour.

Monobehaviour is a class from which helps control many different types of behaviours such as different triggers which can be things such as “ onCollision “ or “ onMouseDown “. These triggers will apply some kind of effect on the simulation or game environment whenever their respective action is encountered. For example, onCollision will trigger an event whenever two colliders touch each other. OnMouseDown will trigger some sort of event whenever the mouse is clicked on whichever object the trigger is applied to.

Importing models in Unity

Unity supports importing blender models, and as such we can import the blender models in the existing github codebase to use for our simulation of the swarm AI, however some additions and edits will have to be made to the imported models so they are able to be used properly in unity. We will need to remove the default camera and light source left in the blender model to avoid any rendered scene related components from the blender model, in our case we are just using the model and unity will handle the rest so there is not a need for these extra scene related components. Furthermore we will need to set a proper origin position for each model to avoid any unnecessary clipping of the object into other objects. This will help to keep the simulation looking clean and professional. Additionally, another important change we will need to make to

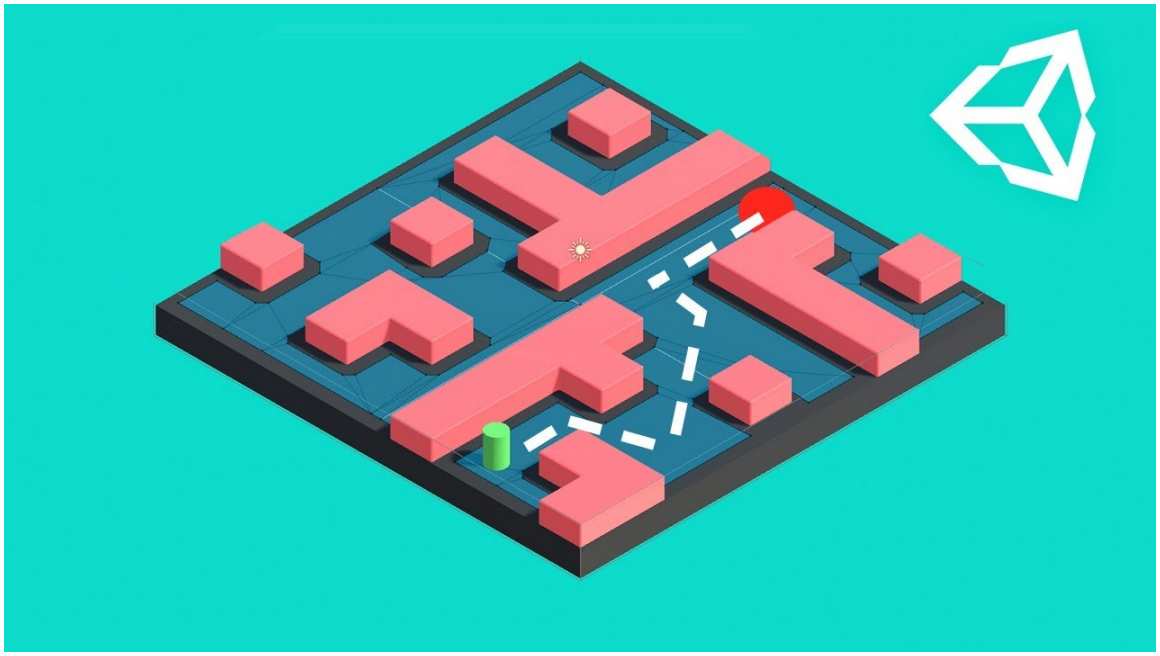
imported models is adjusting the normal vectors position. Normals are vectors attached to the meshes of an object, these normals are used for rendering and how light interacts with an object. These normals should be adjusted to always point outwards so that the object will properly render and not have any issues with any lighting in the simulation. If time permits we could also animate the arm of the rassor rover to improve the simulation of placing landing pad bricks.

AI in unity

General AI are AI of the future, these are meant to be AI that can learn from experience, in a way that mimics what humans can do. This would allow for the AI to adapt to different commands outside of any intended function or purpose it was designed for, and learn new things such as reasoning, logic, planning and communication. This is similar to things like machine learning and strong AI [45].

Strong AI are AI that are similar to narrow AI except they have a more general idea of things instead of concrete details. This allows for the AI to make some sort of choice in how they respond to questions or problems.

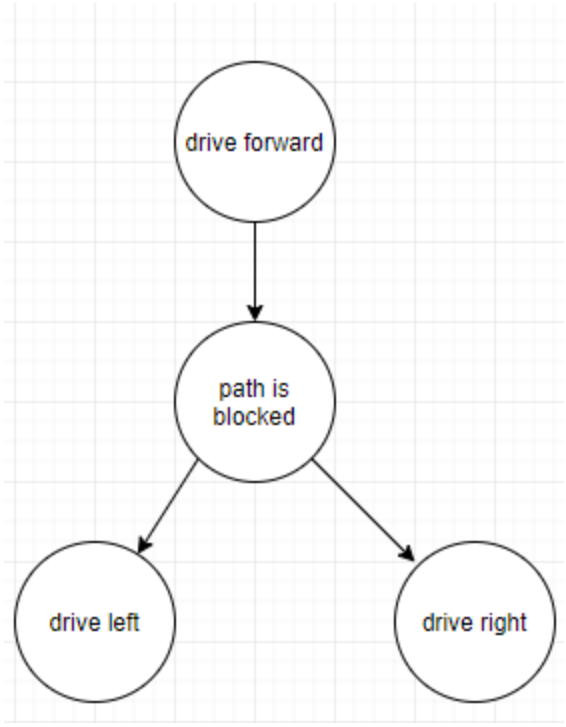
Narrow AI, is a form of AI that is focused on a small or simple task, this kind of AI is similar to something like cortana or just has one specific task. Some examples of this sort of AI are programs like siri and cortana. These programs or AI respond with predetermined responses to the questions you can ask, and if you ask a question or give a command of some sort that isn't something the narrow AI knows how to respond to, it fails to provide an adequate response to satisfy your question or command. Moreover, narrow AI is the type of AI we will be designing and working on for our rassor rovers. In unity this type of AI can be designed with many different features of unity such as the creation of behavioural trees or using navMeshes to control the movement of the rovers. As an example, below is a picture of a nav mesh



[46]

The blue part of the ground is the navmesh and is the area the AIs are allowed to move through , because the AI knows where the navmesh is, it can decide where to go and eventually end up at the red ball, which is the ending point.

Behaviour trees are trees made up of many nodes which represent different behaviours. To iterate through these trees when the AI meets a certain requirement. For example, in the case of our rovers, a simple behaviour tree would be



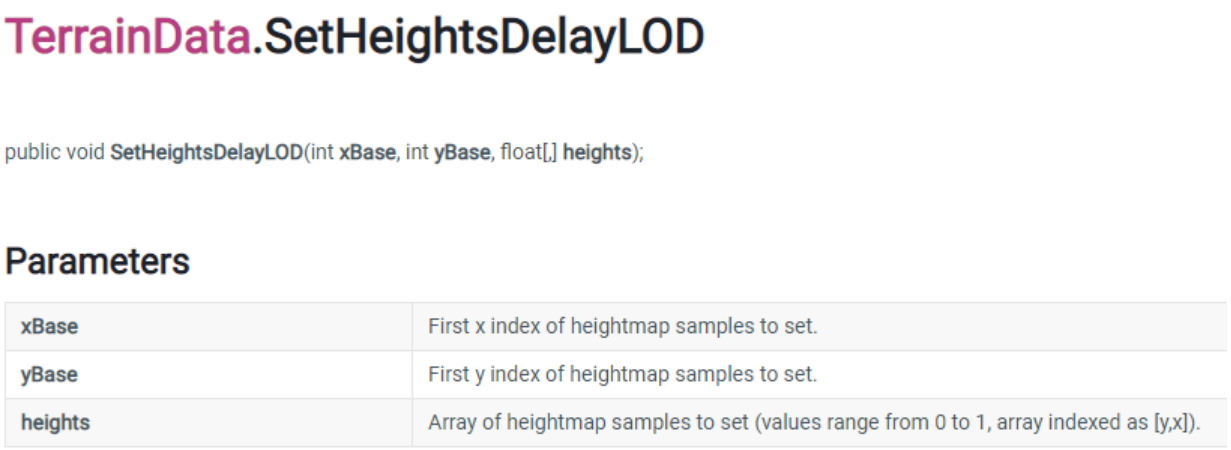
Basic behaviour tree

In this case, if the path is blocked the rover would simply choose to drive left or choose to drive right to avoid whatever is blocking its path. These behaviour trees could be very complex to give the AI a way to get out of many different obstacles it may encounter.

Terrain manipulation at runtime in unity

Unity allows terrain manipulation at runtime, as such, we are able to change the color or terrain itself in the rover simulation, this will allow us to represent the changes in height when digging or depositing regolith to make a surface flat and uniform. There are a few ways to manipulate the terrain height at runtime, one of them being to change the heightmap data , the heightmap is simply a map of the height of terrain. In practice we would use the method

TerrainData.SetHeightsDelayLOD.



TerrainData.SetHeightsDelayLOD

public void SetHeightsDelayLOD(int xBase, int yBase, float[,] heights);

Parameters

xBase	First x index of heightmap samples to set.
yBase	First y index of heightmap samples to set.
heights	Array of heightmap samples to set (values range from 0 to 1, array indexed as [y,x]).

The TerrainData.setHeightsDelayLOD method and parameters [47]

This method sets the heightmap data using a two dimensional array of heightmap samples. The two dimensional array is used to define the area affected. This terrain change will start at xBase and yBase which are the first indexes of the heightmap samples to set. The good thing about the method in comparison to TerrainData.SetHeights is that it's less resource intensive and thus easy to simulate. Utilizing this method along with a script that triggers that method on the rover object location whenever it has reached a specific position designated by ROS or the user. Subsequently the rover would dig or deposit regolith and the terrain would be changed accordingly. Once the terrain modification is complete, in the case that there are other objects such as boulders or rocks, we would call the TerrainData.SyncHeightmap method to update the LOD information for any objects potentially affected by the previous method call.

We could also simulate a change in height of the terrain by using colors in a similar way to how a topographic map works. This can be achieved by using a projector [48]. A projector allows you to project a material of some sort onto any object that intersects its frustum, this being a cone shaped area in the field of view of the projector. In practice, this could be used to alter the color of the terrain when a rover is performs a task on it, for example if it the rover deposits regolith into an area, the projector would change the color of the terrain to be something red to present the increase in the height of the terrain, and if the rover

were to dig up regolith, that area would be colored blue by the projector to represent an area of lowered height.

Ros and Unity

[49] Robot operating system (ROS) is fortunately rather easy to use in conjunction with unity, in our case, all of the tasks given to the rassor rovers by the central task manager and other packages and nodes will all be contained within ros, while unity is simply used to provide a simulation of the tasks provided by the code in ROS. In our case, our simulation would consist of a lunar environment, or more simply a plain environment with various terrain heights for the rovers to work on.

As aforementioned, we can bring the blender model of the rassor rover into unity with relative ease, this model can be improved upon to contain things like collision meshes and additional physical properties if we decide to improve the visuals in the unity simulation. The collision meshes can be utilized to calculate and detect collision with objects in the environment which may be things such as a crater or a rock.

The physical properties that would be relevant to this simulation would be properties such as velocity and weight, but these properties, while important are a non-issue for our portion of the rassor project, however adding these properties into the simulation might be helpful for future teams. These properties can be easily ascertained via a URDF file that can allow us to specify these types of properties, these files can then be imported into unity and be used in the simulation. An example URDF file is below.

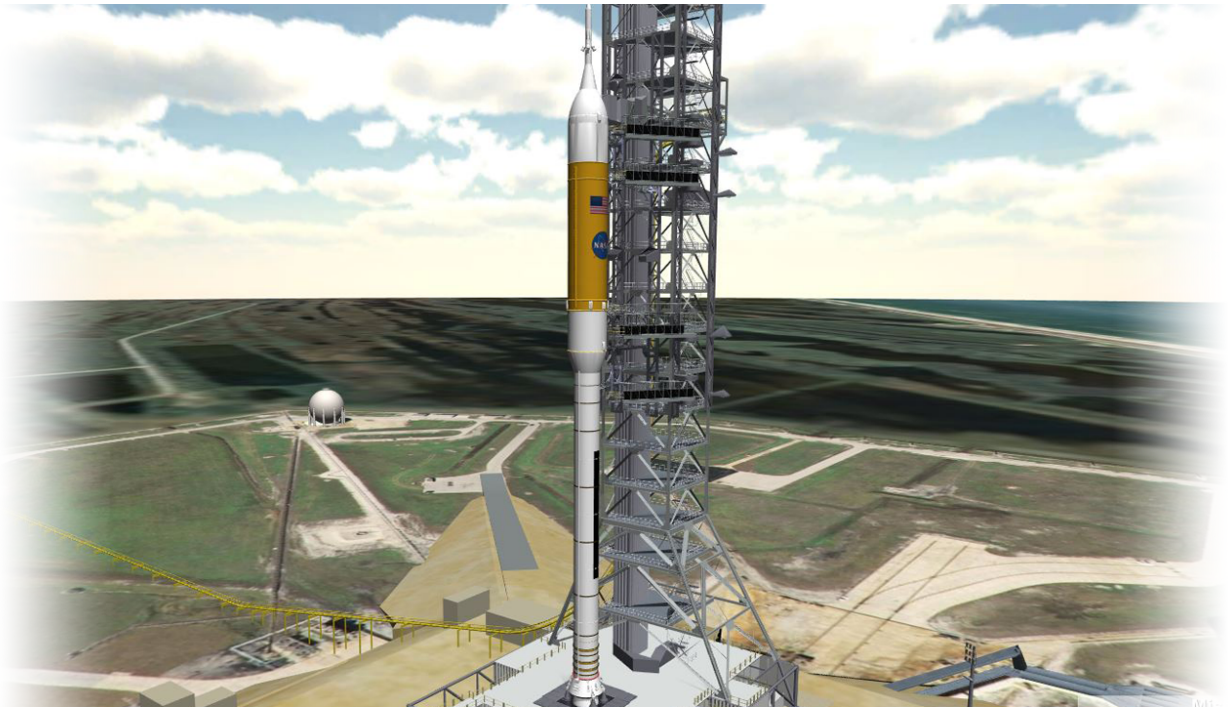
```
<robot name="niryo_one">
  <!-- Links -->
  <link name="world"/>
  <link name="base_link">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://niryo_one_urdf/meshes/collada/base_link.dae"/>
      </geometry>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://niryo_one_urdf/meshes/stl/base_link.stl"/>
      </geometry>
    </collision>
  </link>
  ...

  <!--Joints -->
  <joint name="joint_world" type="fixed">
    <parent link="world"/>
    <child link="base_link"/>
    <origin rpy="0 0 0" xyz="0 0 0.63"/>
  </joint>
  ...

</robot>
```

ROS/Unity Code [50]

When connecting ROS to unity we need to establish a form of communication between them, in this communication unity can pass state information about the rovers while ros will send the tasks delegated to the rovers to unity to perform in the simulation. For this communication to function we need to add a ROS-TCP-connecter package to unity and a ROS package called ROS-TCP-endpoint. These packages allow us to create ROS service requests and responses and they create an endpoint which allows communication between the ROS nodes and the unity simulation. To use these packages we simply have to create a publisher in unity which sends data to ROS over TCP, and in ROS we set up an ROS-TCP-endpoint to receive these messages.

DON (Distributed Observer Network)

The Distributed observer network, known as DON is a virtual environment designed by NASA and ITTS for simulation. The results of the simulation can be accessed by multiple users and multiple locations simultaneously allowing for more streamlined simulation sharing. This simulation creates 3d visualizations by integrating 3D model data from computer aided design programs and NASA simulators with state and meta information to create an environment similar to a 3d game environment. DON also provides access to many different environments for the simulation such as an environment based on the moon.



DON screenshot illustration the simulation [51]

Collaboration in DON

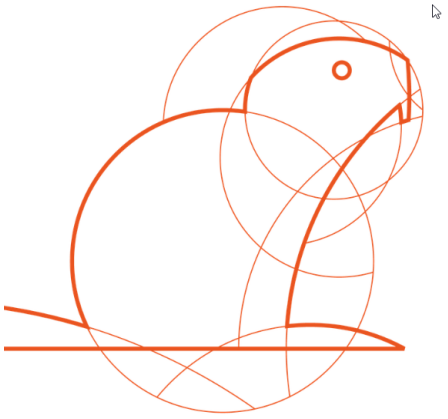
DON also has a streaming interface which enables users to view 3d representations of mission simulations in near real time. Additionally, it has an integrated recording function which allows the simulation to be replayed and re analyzed at any point in time, the plethora of collaboration functions make DON very good for simulation when working with teams. As aforementioned, DON simulations can be accessed by multiple people simultaneously, this allows teams to share information they have gleaned from the simulation and coordinate in an effective manner, it's also compatible with many different operating systems making it a convenient tool for anybody. [51]

Simulation interaction in DON

DON also has the ability to let users move around in the simulation to observe it from any angle, direction or place they want to. This alone is very useful to observe more specific things in a simulation rather than only seeing the big picture. Additionally, users can select a specific object in the simulation and “ snap “ to it to observe that object in the simulation specifically. Users also have full control of the simulation timeline allowing one to access any point in time in the simulation whenever necessary. There is also meta information associated with each simulation which contains information such as positional and state information. This information can of course be accessed from the simulation.

Rationale for Specific Technologies

Why Ubuntu 18.04 LTS?



Ubuntu 18.04 Bionic Beaver Logo [52]

Ubuntu 18.04 is the ideal operating system for the EZ-RASSOR project for multiple reasons. The main advantage stems from the fact that ROS, which is a major component of the project, officially supports Ubuntu LTS (Long Term Support). Because of this main concern, for this project we chose to use Ubuntu 18.04 dual booted.

Ubuntu is a free, secure and fully customizable platform that is better suited towards development. Compared to other options such as Microsoft Windows 10 or MacOS, which will be very expensive to provide for the entire team, Ubuntu seemed like a superior choice. Because of the open-source nature of this operating system, there is a supportive community of developers [53].

Currently, migrating to the newest Ubuntu long term release, 20.04.2, would be preferable; however, the EZ-RASSOR has a dependency problem where moving operating systems would require the project to be ported to python3. While we are currently considering porting from python2, we will not be able to accomplish this project in Ubuntu 20.04.2

Why Not Windows?

Windows 10 Logo [54]

As mentioned before above, ROS is a major factor for determining what kind of development environment will be implemented. Although ROS allows for porting to Windows, this implementation has not fully been explored yet. ROS has full support for Ubuntu and other Unix based platforms [55].

Windows is more of a general purpose style operating system, making it obsolete in comparison to a development oriented and open-source platform such as Ubuntu. Windows also doesn’t come stock with things such as git integration making it cumbersome to get set up and running.

Why Not MacOS?

MacOS Logo [56]

The Mac Operating System is also another powerful platform that was considered in the decision of a development environment. Overall the consensus swayed toward Ubuntu for its direct Linux style and ease of access.

The Mac development environment also comes with many restrictions due to a limited range of application access and support.

The price of establishing Mac OS environments for the entire team was also a major deterrent. In comparison to Ubuntu, which is free and open-source, the choice was simple.

Dual Boot vs Virtual Machine

Windows 10 Ubuntu Dual Boot [57]

The decision to dual boot was specifically related to the experience that previous teams had gained in attempting to learn about this project. Loading Ubuntu onto a virtual machine on a windows system is quite easy to do and was definitely the most preferred option as it required less work. But this was strongly advised against by the Spring 2021 black team. In their experience, they had used a lot of their time to try to configure the virtual machine to run the simulation software Gazebo, ROS, and blender. After struggling to configure the virtual machine, the Spring black team decided to all go with a bare-metal build of Ubuntu 18.04.

Furthermore, virtual machines tend to not perform as well as an operating regular operating system which would hamper things such as simulation software and rendering in programs such as blender. In addition they may generally be slower overall resulting in a less fluid and, possibly, more frustrating experience when developing code for our EZ-RASSOR project.

Why ROS?

ROS, the Robot Operating System, is the foundation of the existing EZ-RASSOR project and cannot be replaced. However, ROS fulfills several of the project needs. The project is required to be open source and easily accessible to users and contributors.

One of the goals of the ROS project is to make ROS software easily accessible across the internet. In fact, the graph architecture implemented by ROS is easily extendible, perfect for open source projects. Furthermore, ROS requires that projects are modularized by enforcing the node and graph architecture on the project. Therefore, our ROS packages are guaranteed to be extendable and modularized, a requirement for any robust software package. Overall, ROS guarantees safe software practices at the foundation of our project.

Python2 vs Python3

Python Logo [58]

The decision to go with python2 vs python3 relies heavily on the nature of the project. The EZ-RASSOR project is open sourced and has been worked on by multiple groups and people before. Along with the group's of students from multiple universities, it is maintained and utilized by NASA's Swamp Works. The codebase's python sections are written in python2. If we were to change to python3, the change would be expensive and cost a lot of time in just the conversion of the already existing code base. In addition to the costly amount of time it would take to convert to python3, this group does not have the power to convert the whole EZ-RASSOR project to python3. Since the project is currently being worked on by multiple groups, the project has two moderderators who maintain the code base. At most, our group would be able to suggest a change to occur. But this effort would have to be done by a multitude of groups not just including UCF's senior design groups currently working on EZ-RASSOR.

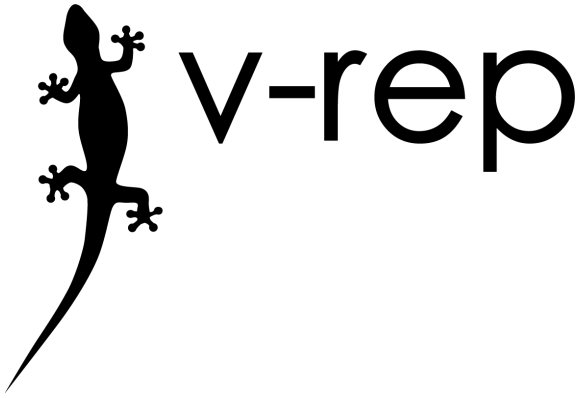
But the change to python3 would present the EZ-RASSOR project a magnitude of benefits as the update to newer versions of code usually result in an increase in efficiency and an increase in efficacy of the project. Migrating to python3 would make the EZ-RASSOR project more accessible to contributors and users; therefore, migration is a high priority. Furthermore, migrating to python3 will allow the EZ-RASSOR project to use new technologies and to update the technologies currently in use.

Why Gazebo?

Gazebo Logo [59]

There are several platforms for robotic simulation. The one we are using for this project is Gazebo. Gazebo was released in 2003, and has been in constant development. Gazebo gives us the opportunity to utilize simulation software. This means that we will have the opportunity to test our code, software, and algorithms on a simulated robot. This is an incredible advantage over the testing on a physical robot of hardware, because it reduces costs, and reduces the chances of errors on the physical robot. Teams from previous senior design classes have already implemented both lower level, and higher level functionality for this robot. So we are presented with a unique opportunity to use these functions to implement really high level code and ideas. Another benefit to using Gazebo software is due to the ability of Gazebo to let us test on 100's of robots, instead of just one. This will allow us to test the algorithms and software that we develop on a very difficult, multiple rover system.

Why not V-REP?



V-REP Logo [60]

V-REP stands for the Virtual Robotics Experimental Platform. It is an incredibly popular robotics simulation platform, and certainly a rival to Gazebo. V-REP has some incredibly built in physics engines, and a really nice selection of robots already made for testing. Furthermore, V-REP is capable of mesh manipulation, so digging is a possibility on an engine like this. However, V-REP is not open source. Under the GNU GPL, general public license, the software can be used for free for educational institutes. In cases outside of the GNU GPL, it requires a commercial license.

This has never been the goal of our EZ-RASSOR project, and therefore does not fit the requirements of our project. We require our EZ-RASSOR to remain open source and available to anyone with an interest in our project.

Methodologies (Project Management System)

Organizational methodologies are one of the most critical parts of a successful project. It is important to utilize the best software development process in order to be as efficient as possible when developing solutions. Our project is an already established open-sourced project which has been worked on by many other groups and Universities. Our goal is to implement a niche solution added onto the EZ-RASSOR codebase.

Due to the unique nature of our project, we will explore many different types of methodologies when it comes to the development of code.

Agile

Agile development processes are some of the most widely used project development methodologies. They offer a management structure that has the ability to adapt and provide high quality solutions that also keep what the customer wants first. Another benefit to the customer would be how critical features are developed quickly through only a short amount of cycles. Agile derives a lot of its components from other methodologies like Scrum and Kanban. Some of the values of Agile development are as follows:

1. Individuals and interactions over processes and tools.
2. Working software over comprehensive documentation.
3. Customer Collaboration over contract negotiation.
4. Responding to change over following a plan.

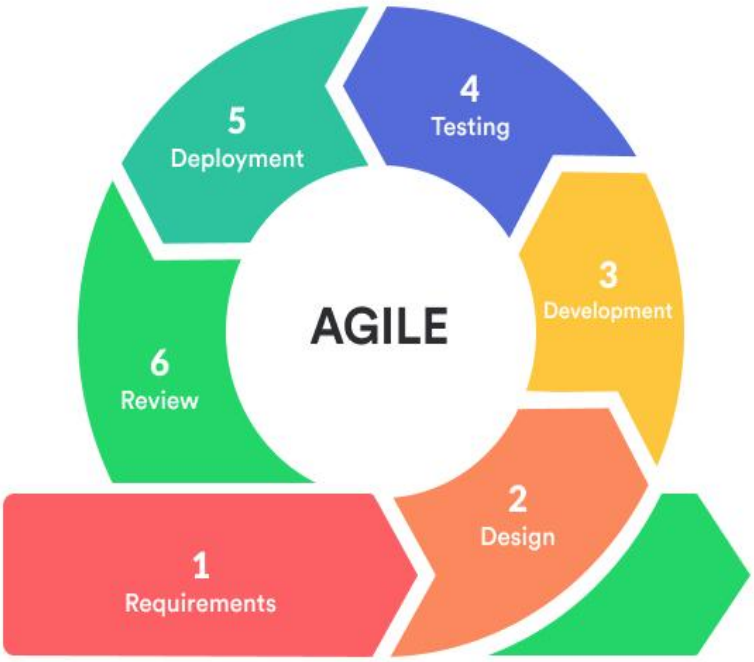


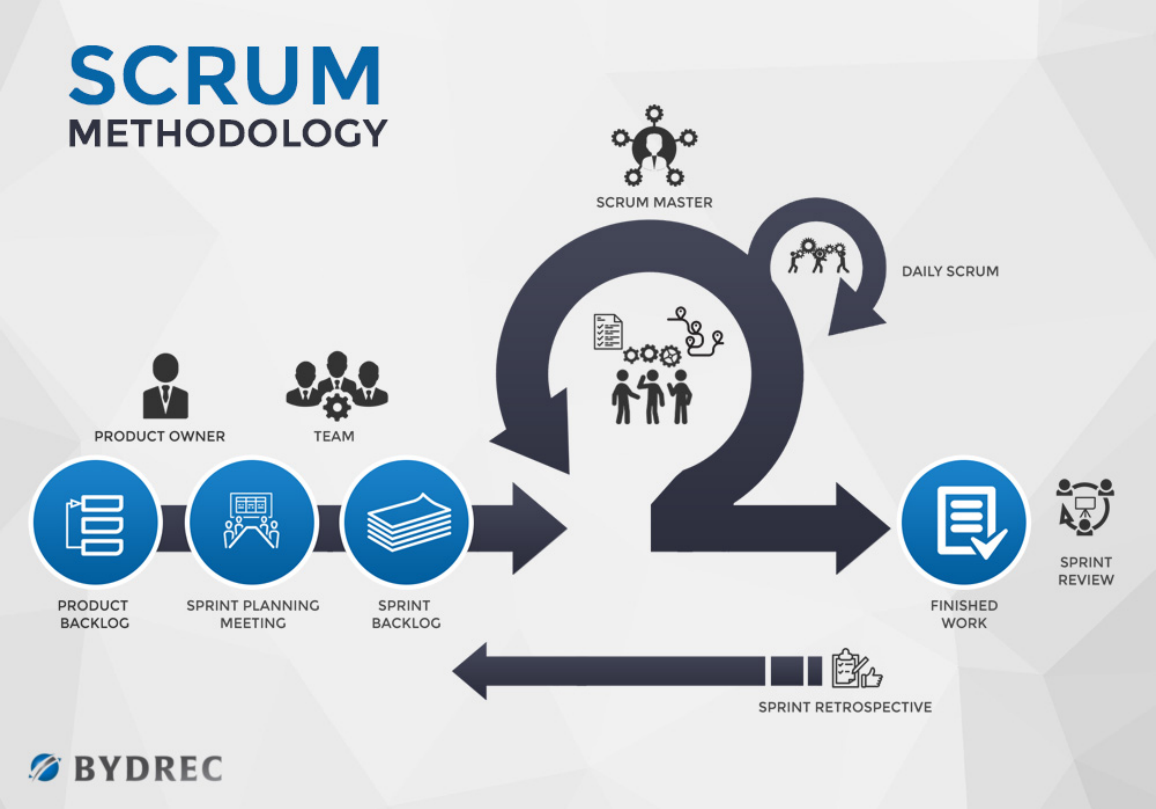
Diagram illustrating the flow of agile development [61]

Agile development is a consideration that our team is considering. The ability to constantly have a working product to show would be incredibly helpful, as we could gain insight into our progress from TA's and our sponsor Michael Conroy.

Scrum

Scrum is a framework that utilizes a similar methodology to agile development. Scrum is designed for teams of 10 or fewer members, and is certainly very popular. Not only is it used within the software development field, it's also used in research, sales, marketing, and some advanced technologies. The idea is to break the goals of the team down into manageable sections that can be completed in sprints. These sprints are generally around 2 weeks to 1 month. There are usually daily meetings that last for only about 15 minutes at most, called daily scrums. When a sprint is finished, the team usually has two meetings. The first meeting is a sprint review, that helps to show what work was done during the sprint, and to gain feedback on said work. The second meeting

is usually a sprint retrospective meeting, where the team talks about the sprint and how they can improve.

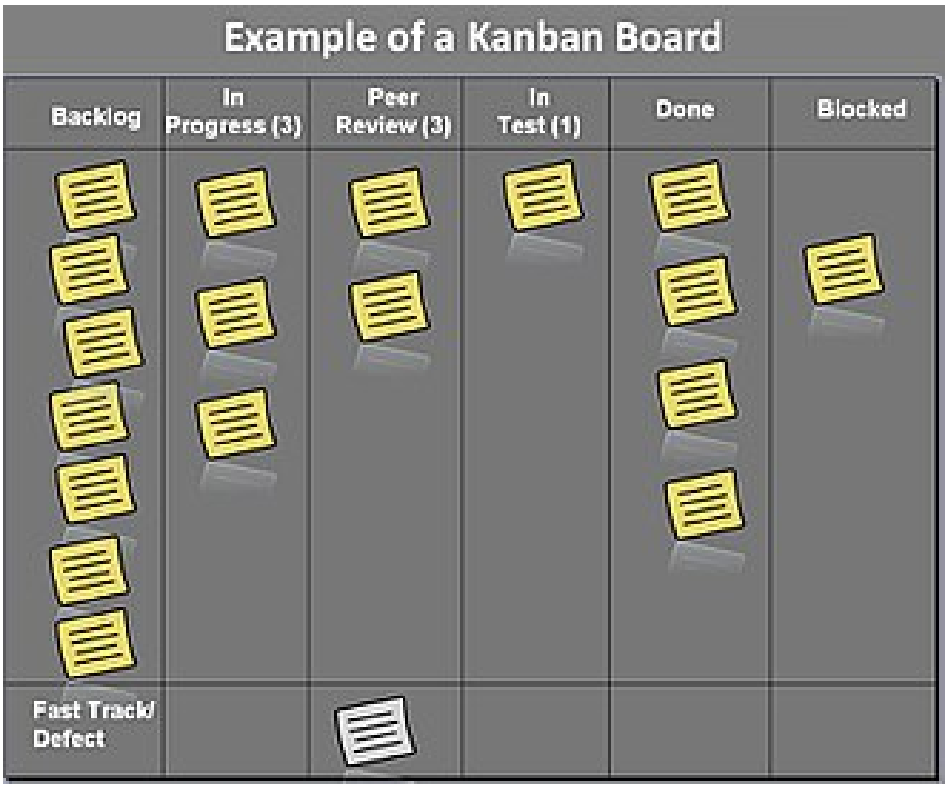


Scrum Image Example [62]

One of the key concepts of Scrum is the idea that the customers of the project will change the purpose or scope of their wants and needs. This does not fit our approach in Senior Design. In Senior Design, we aim to achieve a predicted, and planned approach to our project. We believe that due to this nature of Scrum, it is likely we will go with another approach.

Kanban

Kanban originates from the late 1940's when Toyota implemented a production system referred to as "just-in-time". The concept is to produce based on customer demand, and figure out where the problems are in production and how to change that. Typically Kanban is used in collaboration with other frameworks and methods like Scrum. Kanban is centered around the use of Kanban boards. These boards help to show work item types, such as features, user stories, policies, and swimlanes. An example board is shown below.



Kanban Board Example [63]

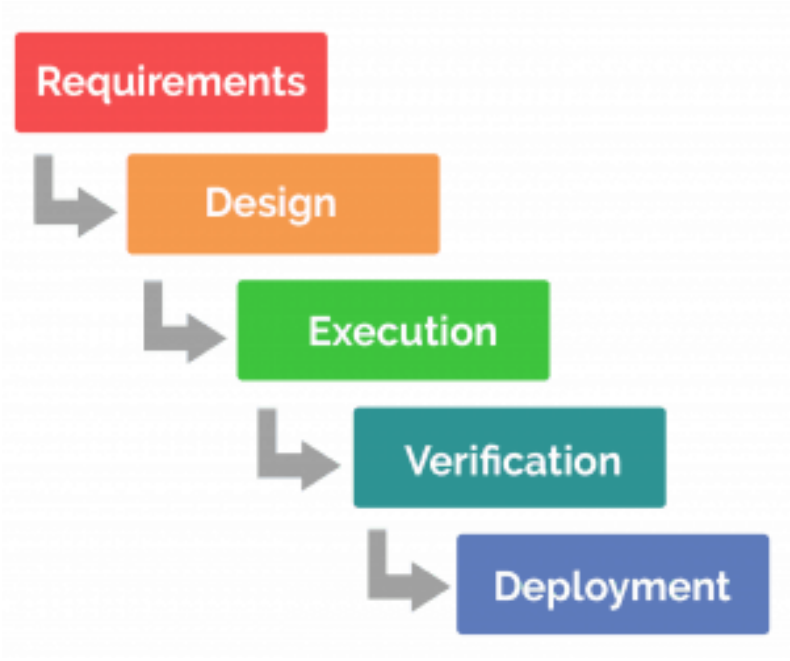
These boards help to demonstrate and visualize the work of the development team. This board is used to manage the workflow and help to identify any bottlenecks in the system. Kanban is an interesting concept to consider, but unfortunately we feel like another one is a bit better for the purposes of our project and our group. That leads us to the following framework, Waterfall.

Waterfall

The Waterfall model methodology is also another widely used project development method that is often compared to its counterpart Agile. Essentially, the main concept of the Waterfall model is a partitioning of the project tasks or activities in a linear chronological sequence of phases. These phases all rely on the previous phase for completion and designated time frames are assigned for finishing set phases.

The phases must be completed in the chronological order established, one phase at a time, either signaling the end of a phase through meeting all established goals or the end of the time period.

Hence the name Waterfall, the development of the project will follow a waterfall like flow down from phase to phase in one direction. There are some variants where it is possible to reiterate on certain phases or move to a previous phase, but it is very uncommon.



Waterfall Method Model [64]

As a result of the unidirectional flow of development with the Waterfall model, it is critical that the set tasks and phases are precise and the chronological order of the phases are correct.

Overall, this means that the planning phase of establishing the Waterfall model must be given a fair amount of time and consideration. Additionally, during the development cycle, a phase cannot fail to reach its goal, as the next phase depends on its prior stage for completion.

Any kind of failure would lead to a complete stop of the project development as a whole. It is important that these kinds of risks are evaluated and weighed when determining what kind of development style to choose.

The Waterfall model is under heavy consideration for the purpose of our project because of the sequential nature of the tasks and goals laid out. Before a landing pad can be built on the surface of the Moon, there must be rovers with instructions to build that pad. Before the Rover can understand how it needs to build that pad, it needs to be given calculated instructions. These calculations must come from some kind of central managing system that receives its data for calculations from topological data retrieved from the rovers.

Every task is predetermined by a predefined task, and the failure of completion for these tasks will impede the state of the preceding task, leading to a halt of the project.

In addition, unlike the Agile development style, the Waterfall model does not require constant refinement of tasks or daily meetings with sponsors and other involved members. Our sponsor for this project has a clear goal in mind of what the project needs to accomplish, and that there is an order in which things must be completed. There is no need for constant meetings of task planning and goal modification.

Testing Plan

Testing in Gazebo

Our initial testing ideas will be around Gazebo. This will include testing the MAPP algorithm inside of the simulation, as well as testing the color switching in regards to progress being made for leveling the land.

- Test Case 1
    - Purpose
        - Pathing Algorithm edge case with no available path
    - Test Description
        - This test will be used to check if there are too many obstacles in the way and if the rovers can't properly get to the site. A map will be generated that will have a big mountain in between the rovers and the digsite.
    - Expected Result
        - The expected result of this test will be to check if our function can determine when there is not a solution to the pathing problem.
- Test Case 2
    - Purpose
        - Pathing Algorithm with no obstacles
    - Test Description
        - The test will administer an environment where there are no obstacles at all. A blender map will be utilized that checks for pathing with no obstacles.
    - Expected Result
        - The expected result is that any path generated will be the most optimal path for all of the rovers. We also expect this

test to run optimally compared to other tests due to their being no interference from obstacles.

- Test Case 3
    - Purpose
        - Pathing Algorithm for lots of obstacles.
    - Test Description
        - This test includes several random obstacles that obstruct the rover's path, these obstacles also force the rover's initial path to be obstructed by other rovers. This will help us determine whether the rover’s path planning is able to adapt to more complex situations
    - Expected Result
        - The expected results for this test is that the rovers are able to correct their course and avoid running into obstacles as well as themselves.
- Test Case 4
    - Purpose
        - Test the Task Manager
    - Test Description
        - We are going to watch the rovers in action and make sure that the process flow for the rovers will be correct and the Task Manager is operating properly.
    - Expected Result
        - The expected result is that the rovers will follow the project's process guide, and switch from leveling to building.

Testing in Unity

Unity provides a game type simulation environment to test and examine the swarm pathing algorithm used. It also allows us to test and examine the excavating and building capabilities of our Swarm AI. Some of the tests here will be similar to Gazebo due to the nature of implementing the pathing within Unity as well as Gazebo.

- Test Case 1

    - Purpose
        - Pathing Algorithm for edge case
    - Test Description
        - This test will be used to check if there is no path to the given site
    - Expected Result
        - The expected result of this test will be to check if our function can determine when there is not a solution to the pathing problem.
- Test Case 2
    - Purpose
        - Pathing
    - Test Description
        - The test we will administer here is one with no obstacles at all.
    - Expected Result
        - The expected result is that any path generated will be the most optimal path for all of the rovers.
- Test Case 3
    - Purpose
        - Pathing check
    - Test Description
        - This test includes several random obstacles that obstruct the rover's path, these obstacles also force the rover's initial path to be obstructed by other rovers. This will help us determine whether the rover’s path planning is able to adapt to more complex situations
    - Expected Result
        - The expected results for this test is that the rovers are able to correct their course and avoid running into obstacles as well as themselves.
- Test Case 4
    - Purpose
        - Survey check
    - Test Description

        - The rovers will be checking for level land and if they can identify if work needs to be done to it.
    - Expected Result
        - Rovers detect a land area.
- Test Case 5
    - Purpose
        - Test Leveling
    - Test Description
        - We will use a lot of high mountains and terrains in this test to establish that our rovers are able to level mountains
    - Expected Result
        - The expected result is a land area that is leveled
- Test Case 6
    - Purpose
        - Test leveling
    - Test Description
        - Our terrain here will be incredibly steep and have plenty of holes for the rovers to navigate
    - Expected Result
        - The expected result is a land area that is level
- Test Case 7
    - Purpose
        - Landing Pad Building
    - Test Description
        - The purpose of this test will be to build a landing pad on a designated area.
    - Expected Result
        - The expected result is a working landing pad in the designated area

Project Milestones

Date	Task	Comments
February 3rd, 2021	Teams were assigned on this day.	This was everyone's “top” choice when it came to picking groups. This implies that our team won’t hesitate when creating optimal solutions to our tasks at hand.
February 5th	We got our discord server up and running with everyone included.	This is important due to discord being our main method of communication. Communication is somewhat strained due to the COVID-19 pandemic
February 9th	Meeting with Michael Conroy	Discussed project background and history, as well as our future and our goals. This meeting established our connection between our group and all resources NASA has to offer. Micheal mentioned he would be the one to point us in the right direction when it came to who to go to for help at NASA and Florida Space Institute.

February 10th	Bootcamp	Had a team bootcamp that helped our team form stronger foundations. It was important that our team was honest with what key values were to be.
February 11th	Met with previous EZ-RASSOR student	Helped us to clarify more about the project, as well as the other groups. Because this project has been worked on by other groups, we have a unique advantage when it comes to what to do and what not to do. The other previous group shared valuable input when it came to what to avoid doing.
February 15	Second meeting with Michael Conroy	Further narrowed down specifics in projects and what the requirements were. Everpreceding meeting with Micheal gives our team valuable insight in how we are going to take this project
February 16	Onboarding completed	We completed the onboarding on this data, this was things like dual booting ubuntu on your machines as well as getting familiar with the existing codebase.

February 18	Project requirements document due	This is an important step. By now we had to solidify what the requirements of the tasks were to be, also checking to see if our requirements and tasks were even viable.
February 19	TA Check In # 1	Our first TA check in was with Zach. He provided good insight and a second outside opinion when it came to the current status of our research and project.
February 24	Third meeting with Micheal Conroy	The third meeting with Micheal Conroy solidified our understanding about ROS and it’s relation to how we will implement our solutions.
March 4	Fourth meeting with Micheal Conroy	The fourth meeting with Micheal Conroy. We had discovered an issue involving Gazebo, our simulation software. He suggested using another third party software to illustrate the parts of our solutions which couldn't be shown in Gazebo.

March 15	Project Status and Doc Review	During this meeting, we presented our idea's and status of the project so far. We presented with Dr. Richard Leinecker. Dr.Leinecker pointed out that using python2 was pretty archaic and suggested we migrate to python3. This was noted in our development.
March 17	Fifth meeting with Micheal Conroy	The fifth meeting with Micheal Conroy. In this meeting, we confirmed that we would use both Gazebo and DON to illustrate our solution's. This way, we can show how we solved our tasks and still use the simulation software Gazebo.
March 25	DON over Unity	We decided to use DON over unity after a meeting with conroy due to its affinity for simulation in lunar environments and usage by past teams.

April 1	TA Check In # 2	In this meeting, we met with Jordan. Our group wanted to meet with both TA's to get a wider breadth when it came to different opinions on our solution to our group.
April 5	Unity over DON	Ultimately decided to use Unity over DON due to the lack of documentation for DON and our surplus of knowledge about unity.
April 21	Sixth meeting with Michael Conroy	The sixth meeting with Micheal Conroy. This was our final meeting with him until we can meet again in the fall semester. We solidified our tasks, solutions and approaches. He also advised us that if we did some extra work during the summer, our senior design two would be amazing.

April 27	Completed final design document	We completed the final design document.
April 28	Final Design Document	Design Document Due! All of our hard work will be put to the test.
April 30	Final Meeting before Summer	This is our final formal meeting as a group for the semester. The goal is to have an idea of what we will try to accomplish during the summer to have a successful fall semester for senior design two.
July 18	Mid-Summer Meet & Greet	This meeting will be conducted as a formal refresher and checkup.
July 30	Group meeting	Establish what work was done, and what needs to be done in order to keep our memories fresh.
August 28	Group meeting	Get the group back together to begin developing software.
September 26	Integrate pathing algorithm	Implement MAPP.

September 27	Group meeting	Discuss how code has been integrated and what needs to be done.
October 4	Begin development in Unity	We should all be fully finished with Gazebo, and working towards writing code in Unity.
November 4	Finish adding code to Unity	Code in unity should be serving as a working prototype
November 11	Meeting with Michael Conroy	Show off prototypes.
November 12	Testing Phase	Begin testing all of the code that we have written thoroughly. The purpose here will be to establish if our code has any bugs.
November 30	Fixing bugs	All bugs and weaknesses should be fully identified and patched.
December 15	Finished Project	All of our code should be integrated into the main code base, and we should be ready for our presentation.

Project Summary

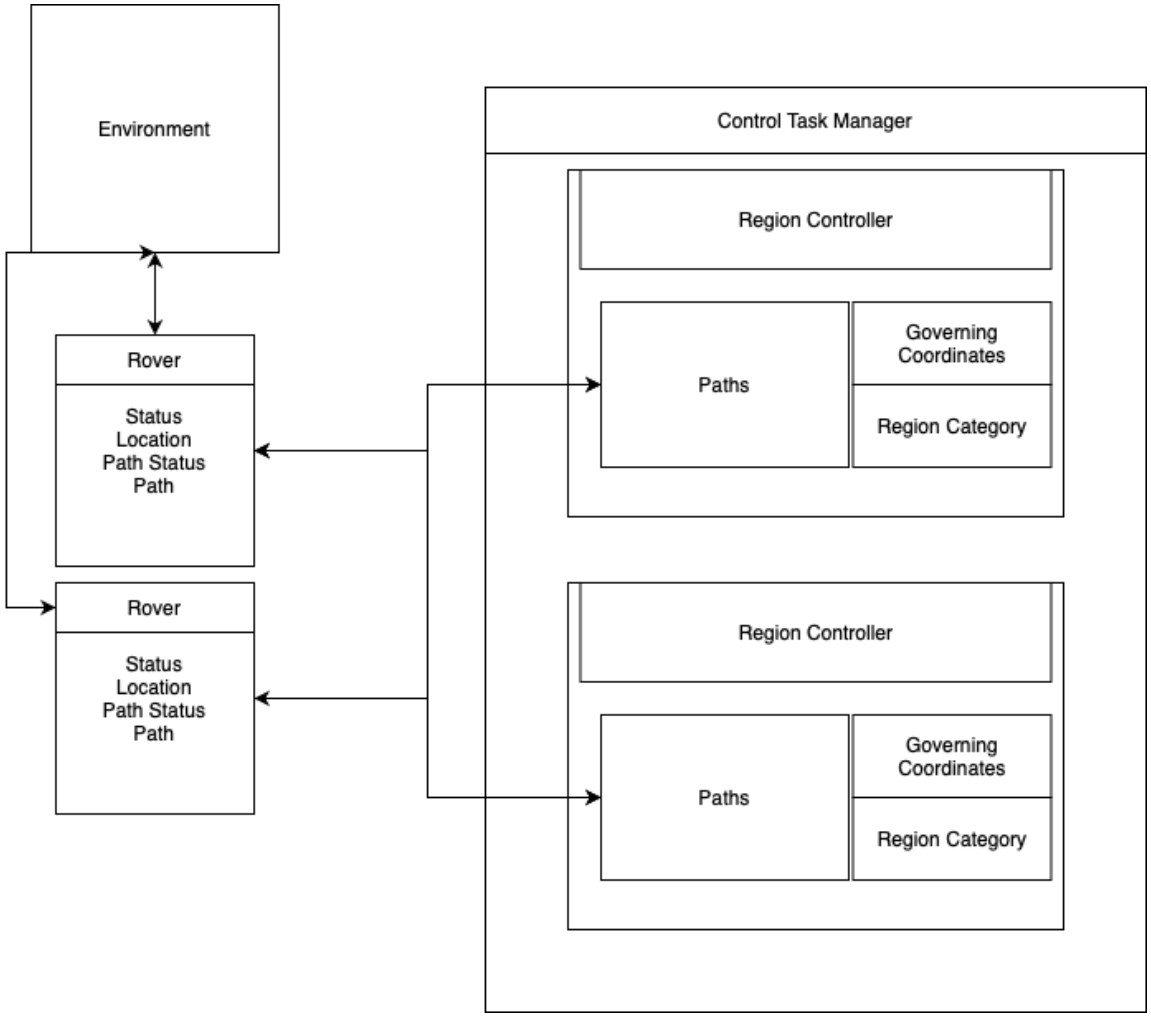
In conclusion, from the days of research and boundless determination of our team we are ready and able to complete this project to the best of our ability. Although this project has been worked on by other teams before, our project is unique. The project is unique in the sense that previous teams projects had some sort of physicality incorporated into the goal of the project, such as implementing the software onto a EZ-RASSOR physical rover.

That being said, we must take extra consideration when testing our solutions. As we start to implement our solutions in Senior Design two, we will surely find examples and edge cases that we may have not thought to incorporate into our solutions, but that is fine. We will have full and thorough testing in both of our simulations.

The great thing about the project is that we have multiple resources and people to reach out to if we are ever stuck and need assistance. Not only do we have our sponsor Micheal Conroy, but we also have the previous black and gold teams, all the other teams from the University of South Florida and Florida Polytechnic University as well. Every person currently and previous who has worked on the project are on the official discord for the Florida Space Institute. Again, having this help will give our team more than we need to succeed and have a successful project.

Central Task Manager

The Central Task Manager, CTM, will designate roles and rover pathing. Objects which control active regions are managed by the CTM. Active regions give paths to rovers depending on their status, reprocessing paths when required. Furthermore, regions updated the target locations where rovers take actions based on the region's geography. The CTM is built using the same autonomy architecture used in the current EZ-RASSOR code base, including waypoints for the CTM and rovers to communicate through.



[65]

Rover Status

Note the rover role will determine its action at its destination, and the action is a simple “autonomy” method call. The completion of the status action requires the rover to change its status. For example, rover status dig captures dirt at the end of its pathing before changing its role to dump.

Furthermore, statuses have default next phases (dig to dump); however, the central task manager may preempt the rover at any time and change its status. The following are the statuses and the default status changes:

1. Dig status picks up dirt at the end of the chosen path. Default swaps to dump.
2. Dump status dumps dirt at the end of the chosen path. Default swaps to dig.
3. Build status uses an arm to place a construction object. Default swaps to stage.
4. Stage status uses an arm to pick up a construction object. Default swaps to build.

Rover status is compared to the current region status to determine the path. If the rover status is different then the region status, then the rover needs to leave the region.

Rover Pathing

The main problem is having to consider specific pathing for each individual EZ-RASSOR rover. The central task manager will have to manage all path's and take into consideration when paths of different EZ-RASSOR rovers cross. In order to avoid collisions, each individual SWARM will have its own method of querying new paths from a centralized task manager which manages every individual EZ-RASSOR rover in the swarm using waypoints and action servers for communication.

Rover pathing is determined by the MAPP, multi agent path planning algorithm, which assigns each rover a preprocessed path given a source location and target location for each rover in a series. Active regions, where pathing targets and sources change frequently, need to be reprocessed; therefore, static paths, such as those in between regions, need to be separated to prevent double processing.

Therefore, regions are managed by an object, controlled by the CTM, which provides paths to rovers and reprocesses the paths when the available targets and sources are sufficiently changed due to the changes in the region environment. Because region objects change because of the environment which is manipulated and viewed by the rover, the rover triggers region objects to change the region paths.

Solution Documentation

Image Processing

- In order to generate efficient instructions for a swarm of rovers to level an area on the Moon, data of the land must be transformed and organized in such a way that it can be used effectively. This is where Image Processing, a program that converts a DEM (Digital Elevation Model) of the surface of the Moon into a 2-Dimensional matrix of elevation values, comes into play. A DEM is a representation of terrain that shows differences in elevation through color shading. There are 3 main steps to image processing: converting the image into a grey-scaled 2-D matrix of floating point values, zero-scaling the values in the matrix such that the mean is shifted to zero, and expanding the number of cells in the matrix so one cell's dimensions are a manageable work unit for the rover to perform actions on. These steps are Image Transformation, Zero-Scaling, and Pixel Expansion, respectively.
- The requirements for our surface area are that the area is an odd, square matrix. For example, in the code we use a 21x21 matrix. A cartesian plane is best represented in the form of an odd matrix because the axes can be represented as the center rows/columns of the matrix, whereas an even matrix is unable to evenly portray the axes.

Image Transformation

- The first step in Image Processing is taking a DEM and actually converting it into a 2-D matrix of values. The color of a pixel in the DEM, which represents a certain elevation height, can be converted into a triplet of RGB values. By leveraging the OpenCV Python library, we are able to accomplish this and grey-scale the conversion, meaning only a single value will represent the color of the pixel.

Zero-Scaling

- To uniformly level a desired area, each cell should be leveled to the height of the mean. This ensures that there is enough regolith to equally distribute among the area. In our case, since the mean is the value of interest for a set leveling point, it is helpful to shift the mean to the value 0. This allows holes to be easily identifiable as negative values and hills to be identified as positive values. To do this, we subtract all the values in the cells of the 2-D matrix by the mean.

Pixel Expansion

- It is critical to ensure that the area a rover digs/dumps (a single cell in the matrix) is a realistically manageable work unit for the rover to perform actions on. With DEMs of the Moon, scaling varies from image to image. One pixel of the image could represent $1000m^2$. By multiplying the number of cells such that a single pixel is replaced by 1000^2 pixels, one pixel in the new matrix is now $1m^2$.

The Leveling Algorithm

The leveling algorithm is a function within the swarm controller package which creates a set of instructions which can be completed by rovers to level the designated area processed by the leveling algorithm.

The leveling function creates instructions which tell a rover to dig in an area where there is too much elevation, and take the regolith and dump it in an area where there is too little elevation. Furthermore, an instruction specifies how much elevation should be distributed between two points. Therefore, the instruction tells a rover how to redistribute elevation between two points to make both points closer to the desired elevation.

For example, if the dig location had regolith above the desired elevation equal to the amount of regolith missing in the dump location below the desired elevation, then the rover could move all of the regolith above the desired elevation in the dig location to the dump location, and both locations would then have an elevation equal to the desired elevation. If both locations cannot be made to equal the desired elevation, then the location with the elevation closer to the desired elevation would become equal to the desired elevation after the instruction is complete.

Input and Output

The leveling function takes in a digital elevation model in the form of a two dimensional matrix. The value of a cell in the matrix indicates the elevation of the point whose location is described by the indices used to access that chell.
The leveling algorithm uses the matrix input to create a set of instructions which would give every cell in the input the same value if those instructions were to be executed.
The leveling function attempts to minimize the amount of battery required by a swarm of rovers to complete the instructions created.

Unfortunately, the leveling algorithm does not calculate paths that the rovers can use, as this would be too intensive. Therefore, the leveling algorithm does not ensure that a rover can reach a destination in an instruction. The paths are created on demand by the path planner module in the swarm control package,

and rovers must use the obstacle detection system in the autonomous package to ensure the rover does not path to perilous locations.
Fortunately, the instructions created by the leveling function may be completed in any order; therefore, the swarm controller may decide to temporarily abandon difficult locations until intermediate locations are more level and easier to traverse.

The Greedy Choice

The leveling algorithm attempts to solve what is thought to be (but not proven to be) an NP Complete problem and provides a solution close to, but not guaranteed to be, the most efficient solution. We attempt to find the most efficient way to level an area in a relatively fast time by using a greedy algorithm where we create one instruction at a time using the best metrics available. The greedy solution will not guarantee the most efficient solution; however, by choosing the correct way to measure the best instruction, we can approximate the best solution closely.

We found that the naive solution is to pair two locations which are closest together. For example, suppose we have two dig locations and two dump locations:

- Dig_location_1
- Dig_location_2
- Dump_location_1
- Dump_location_2

Furthermore, consider the function distance(dig_location, dump_location) which returns the distance between two locations.
Lastly, use the following distances for this example:

- distance(dig_location_1, dump_location_1) = 1
- distance(dig_location_1, dump_location_2) = 2
- distance(dig_location_2, dump_location_1) = 2
- distance(dig_location_2, dump_location_2) = 10

If dig_location_1 and dump_location_1 were to be paired then dig_location_2 and dump_location_2 would be paired and the total required distance to travel would be 11.

However, by pairing dig_location_2 and dump_location_1 and pairing dig_location_1 and dump_location_2, the total required distance to travel would be 3.

Therefore, we decided to create a greedy choice based on which dig location has the largest distance difference between its closest dump location and its next closest dump location.

The Complete Solution

We begin the leveling algorithm by iterating through the given area of k cells and adding the coordinates of each cell to a dig locations or dump locations dictionary.
A cell is a dig location if the elevation at that cell's coordinates is more than desired. A cell is a dump location if the elevation at a cell's coordinates is less than desired.
This portion of the algorithm takes O(k) time.

Then we create what we call a sorted distance matrix to make greedy choices. The sorted distance matrix is a dictionary where the key is a dig location, and the value is a list of tuples which consist of a dump location and the dump location's distance to the key (dig location).
To create the sorted distance matrix, for every dig location we iterate through every dump location, calculating the euclidean distance between the two points, adding the dump location and distance tuple to the sorted distance matrix dictionary.
After iterating through every dump location for a dig location, sort all of the dump locations for a dig location by their distance to the dig location. This allows us to find the closest and next closest dump location to the key (dig location).
If the number of dig locations is n and the number of dump locations is m, then creating the sorted distance matrix takes O(n * (m + log(m))). Unfortunately, creating the sorted distance matrix is slow; however, using extra processing power to create efficient instructions is a worthy investment since we can do that on Earth and send the instructions to the moon.

Lastly, we need to make a greedy choice until all dig locations are paired with sufficient dump locations to level the area. To create an instruction we need to find the dig location with the largest distance difference between its closest dump location and its next closest dump location. Note that when we create an instruction, we designate all of the elevation possible (minimum of the two locations) be moved from dig location to the dump location, so after each instruction is completed, at least one of the two locations will be level. To do this, we iterate through all of the keys of the sorted distance matrix, subtracting the distance between the key's closest dump location and its next closest dump location while keeping track of the largest distance difference found so far.

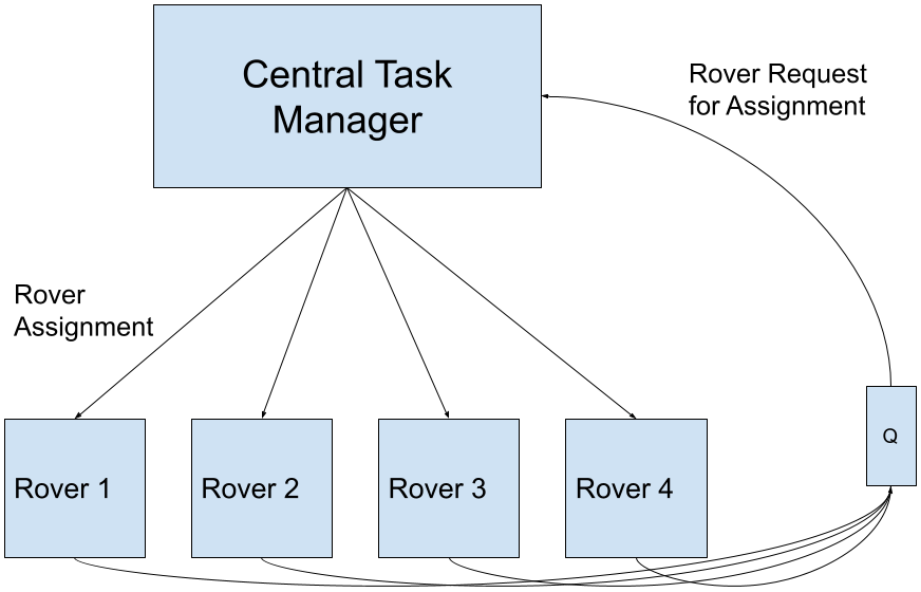
Note that we cannot use dump locations or dig locations which are already part of enough instructions to be considered level; therefore, we might have to delete values from the sorted distance matrix as we iterate through each key. This could result in an iteration of time O(m^2), although very unlikely. We continue picking instructions until all of the locations have been paired enough to be considered level. Every instruction levels at least one location; therefore, we need to create at most O(n + m) instructions. For each instruction we must iterate through the keys, which is O(n) time; therefore, the total big O time of this process is O(n * (n + m)).

The instructions are saved in a dictionary we call sub pairs. The key of the dictionary is the dig location and the value is a list of tuples which contain a dump location and the amount of elevation which must be transported from the dig location to the dump location. This instruction dictionary is passed to the Swarm Controller at run time so that it may instruct rovers how to level the given area.

Central Task Manager

The Central Task Manager (CTM) is an entity responsible for the efficient assignment of instructions to rovers in need of instructions, generated from the leveling algorithm, and handling rovers in need of help. The CTM is implemented as the Swarm Controller class in the swarm_control.py file in the swarm control autonomous package.

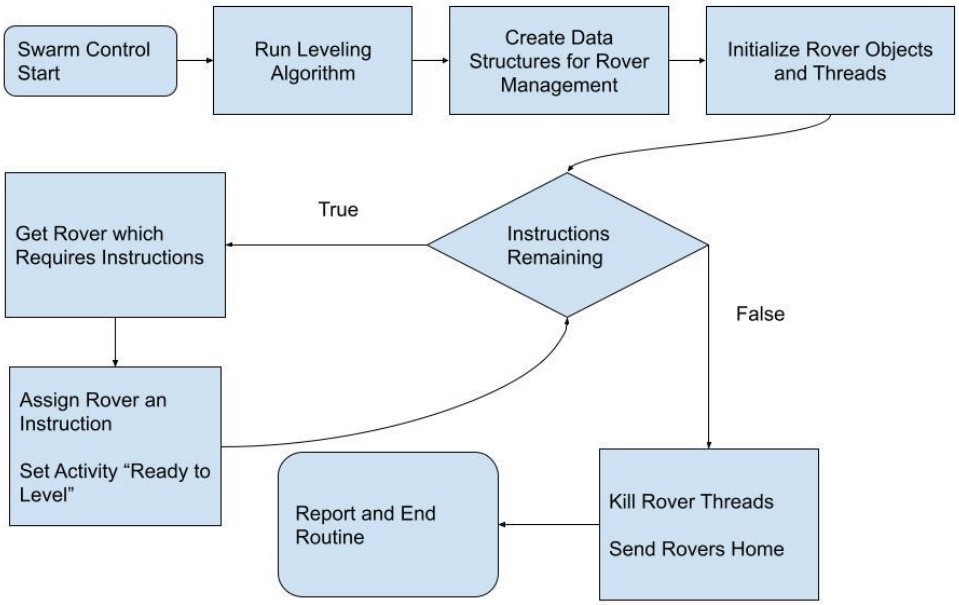
The CTM does not control rovers directly. The CTM instructs rovers what to do. The rovers ask the CTM for instruction or help by placing their ids in a queue which is checked by the CTM.



CTM and rover communication interface [65]

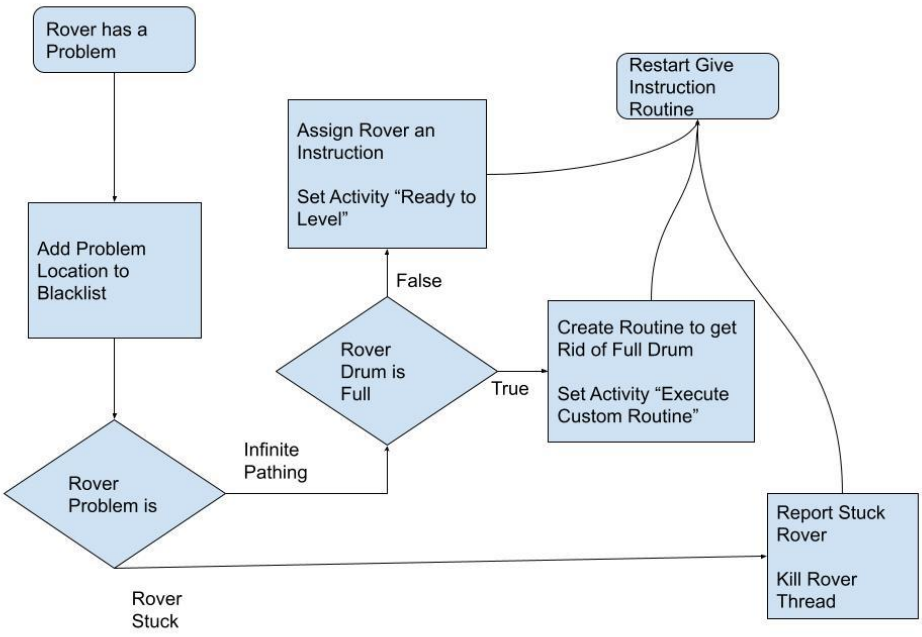
Assignment of instructions is implemented and optimized in the CTM by determining the instruction which contains the dig location closest to the rover's current position and assigning that instruction to the rover. This ensures that the rover's total distance traveled is minimized. The CTM continues giving instructions to rovers who have finished their previous instructions and are ready for a new instruction until all instructions have been executed.

When a rover needs help, it will specify to the CTM the particular problem that the rover is facing, and the CTM will tell the rover how to solve the issue.



CTM providing instructions flowchart [65]

The possible problems a rover can encounter are as follows.
A rover can be stuck in a position where it cannot move. In this case, the CTM will kill the rover thread to save processing power, blacklist the location where the rover is stuck so no rovers go over there, and report to an abstract higher power that the rover is stuck.
Another problem is if a rover cannot reach the desired location. In this case the location the rover is trying to reach is blacklisted so that the problem does not reoccur. Furthermore, if the rover has a full drum, the CTM instructs the rover to go dump the rover drum contents where it dug the regolith; therefore, resetting the instructions.



CTM handling a rover problem flowchart [65]

Rover Logic

The rover has its own logic encapsulated in a rover object which contains all of the code to execute CTM instructions. The rover object is found in the swarm_control.py file, like the swarm controller object.

The rover’s purpose is to constantly be completing instruction sets from the CTM. As the Rover continues on about it’s cycle of completing instruction sets, it constantly reports feedback to the CTM.

After the rover requests the CTM for a new instruction, it awaits to be dictated an activity to execute by the CTM. The activity, called “Ready to Level,” could be to level two assigned locations, or the activity, called “Custom Routine,” could be to execute a custom function which would handle an edge case.

Rover Leveling Logic

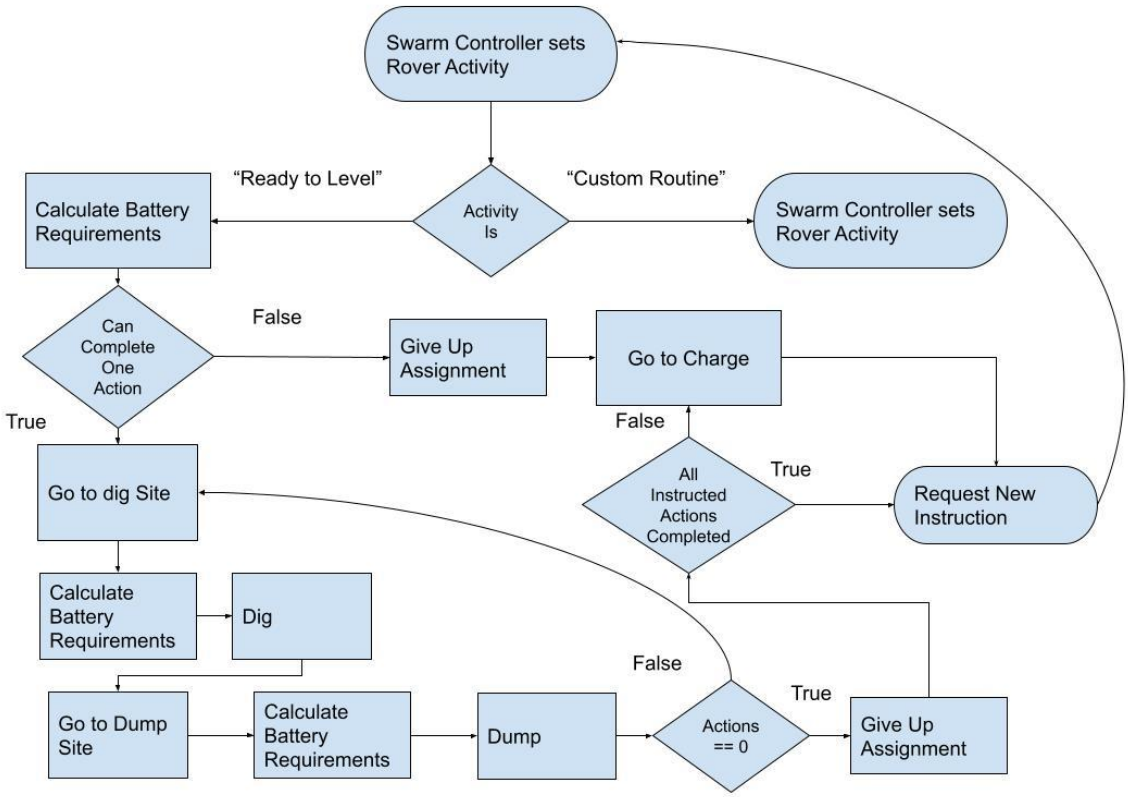
Once a rover is assigned locations to be assigned and its activity is set to “Ready to Level” by the CTM, the rover executes a routine to level at least one of the two assigned locations.

Before the rover begins leveling, it calculates an approximation of how much battery it would take to complete an action and drive to the charging station. An action is defined as the rover driving to the dig location, digging at the location until the drums are full, driving to the dump location, and releasing the regolith in the dump location, thereby distributing regolith between the two assigned locations.

If the rover calculates that it cannot complete a single action, then it gives up its assignment, so the CTM can reassign those locations, the rover goes to the charging station, and after the rover finishes charging, it requests the CTM for another instruction.

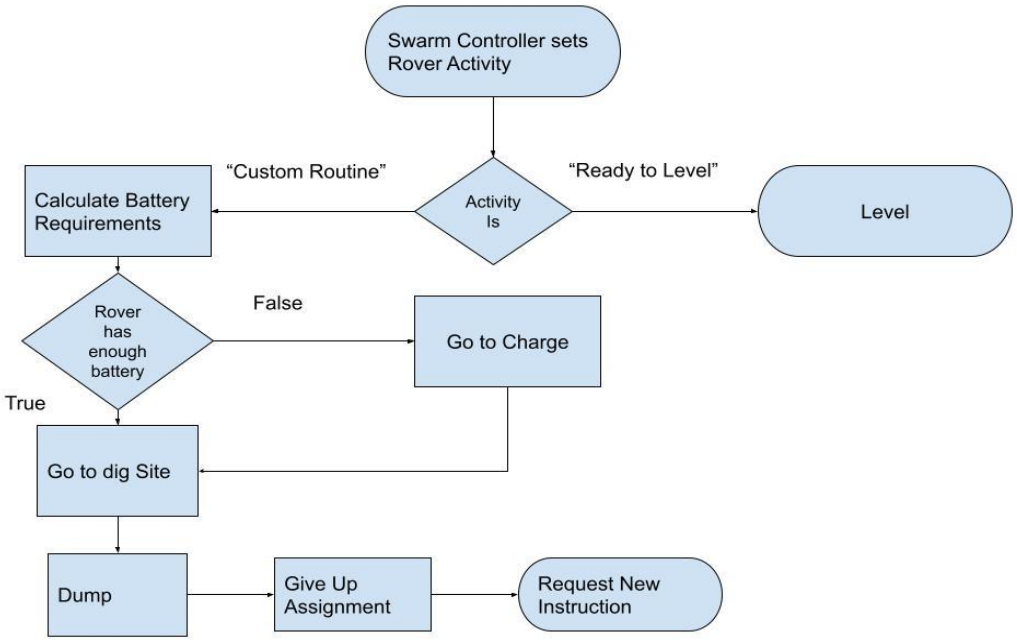
If the rover can complete at least one action, then it begins the leveling routine. The rover begins leveling by driving to the dig location. After the rover arrives at the dig location, the rover checks if more battery has been used to arrive at the dig location then previous location. If more battery was used, then the rover updates its calculation approximations. More battery can be used then expected during pathing because obstacle detection may force the rover to path more than expected. Battery recalculations occur after every pathing is performed by the rover.

If the rover can continue, then it digs at the dig location. Then the rover goes to the dump location, recalculates the battery approximations, and then dumps the previously acquired regolith. This process continues until the rover does not have battery to complete another action or all of the actions specified by the assigned instruction are complete. If the rover was not able to accomplish all actions, it goes and charges.



Rover leveling routine flowchart [65]

If the rover has encountered an infinite pathing problem with a full drum, the following describes the resolution instructed by the CTM. The rover will charge if its battery is below a threshold then the rover will go back to where it dug last and put the regolith back where it found it.



Rover infinite pathing error correction routine [65]

Main Data Structures

The following are the data structures that are used to manage the swarm of rovers.

- dig_locations: A set of tuples that represent the coordinates of all the digsites
- dump_locations: A set of tuples that represent the coordinates of all the dumpsites
- dig_locations_assigned: A set of tuples that represent the coordinates of assigned digsites.
- dump_locations_assigned: A set of tuples that represent the coordinates of assigned dumpsites.
- dig_dump_pairs: An array of tuples where the first value of the tuple is the coordinate to a digsite and the second value of the tuple is an array of tuples where the the first value of the tuple is the coordinate to a dumpsite and the second value of the tuple is the number of dump actions required to fill the dumpsite.
- sub_pair_actions: A hashmap where the key is a tuple where the first value of the tuple is a coordinate which represents a digsite and the second value of the

tuple is a coordinate which represents a dumpsite, and the value of the hashmap is the number of dig/dump actions required between the coordinate pair
- blacklist: A set of tuples that represent the coordinates of areas that should be avoided by rovers
- immobilized_rover_list: A set of tuples that represent the coordinates where rovers are immobilized.

Auto Functions

Auto functions are found in the auto_functions.py file in the autonomous control package. These functions execute common actions for the rover such as go to a nearby location, dig, and dump. The following are descriptions of the auto functions used by our team.

- Auto_dump_land_pad: When the rover dumps using this function, it moves and rotates only the drum behind the rover. Also, the rover changes the direction periodically like auto_dig. The rover sleeps in between changing directions and the rover lowers it's drums slightly when dumping as to prevent the rover performing wheelies. Also, the rover turns to an absolute angle of 90 so that the rover is always facing the same way before it dumps. Lastly, after the rover has dumped, it updates a global variable, indicating that the rover has empty drums.

- Auto_dig_land_pad: When the rover digs using this function, it does mostly the same things as the original auto_dig function. However, the function updates a global variable after digging is completed, indicating that the rover has full drums.

- Auto_drive_location_land_pad: This auto function is very similar to the original; however, when the rover moves with a full drum, it lowers its battery more than usual because the rover is heavier due to the regolith in its drums.

Unity simulation documentation

ROS and Unity connection

- In order to create a bridge between ROS and Unity we utilized the ROS and Unity

integration package and the ROS tcp connector package from the Unity robotics hub github [1]. These packages allowed us to send data back and forth between ROS and Unity in addition to allowing us to import urdf files for the various models in the pre-existing EZ-Rassor github. The ROS and Unity integration package also automatically generates new versions of message files that Unity can read and use to interpret incoming messages from ROS.

Terrain manipulation

- The terrain manipulation we perform in Unity is accomplished via C# scripting. We

utilize the heightmap of the terrain to raise and lower specified points in it. This is done by utilizing either the setHeight or setHeightsDelayLod method to change the terrain height at runtime. The former method can be very computation intensive and thus laggy, so the latter method is recommended.

Rover model creation

- The rover model was recreated in Unity due to some missing or unusable functionality in

the rover game object. Specifically the revolute joints that were being used for things such as the wheels were not able to move and as such, were unusable. To remedy this we created a new rover model in unity utilizing hinge joints, wheel colliders, and rigidbodies to simulate the different joints and movement of the rover. The wheel colliders were originally used for movement but were eventually replaced by object transformation for linear movement and the angular velocity component of rigidbodies for rotating the rover. Additionally, the hinge joints facilitated the movement of the rover arms and drums. A box collider was used as a rudimentary method to implement collision on the rovers.

Topic subscription

- Topic subscription is achieved using C# scripting. These all follow a simple formula,

declare and necessary variables, reference any relevant game objects, subscribe to the topic of interest using the following method from the ROS tcp connector package. “ROSConnection.instance.Subscribe<>()”

Lastly we used that information and applied it to the relevant function of the rover, these functions are things like velocity and arm position.

ROS

In conjunction with Unity on Windows, ROS was run in a Ubuntu 18.04 virtual machine.

In the codebase we added relevant message files in areas specified in the .cs files auto generated by the ROS integration package. These message files were used to define what type of messages were being sent to Unity through the TCP-server.

ROS also performs the logic side of generating all instructions necessary and sends them to the proper rover.

Final Thoughts, Insights Gained, and Some Lessons Learned

We learned how to work together in a group while utilizing the scrum framework. Furthermore, we were able to properly distribute work throughout a group.

We learned how to organize and update sponsor requirements as we approach a solution and further communicate with our sponsor.

Another large insight gained was how to navigate a large, pre-existing codebase and begin to understand how it functions.

Our project became very complex, and we had to be flexible with architectural changes to simplify the code base.

Our project runs multiple threads simultaneously; therefore, we had to learn how to manage a multithreaded, distributed system while preventing race conditions.

We learned how to manage existing dependencies in the codebase and how to decouple them.

We learned how to manage a branch of an open source git repository.

We learned how to control robots using the Robot Operating System.

One of the largest lessons we learned from this project is the necessity to constantly make a little bit of progress. As long as we made some changes or tested some features, we had made progress. Over time, this lesson became more apparent. Sometimes when we had large deadlines approaching, we would go into crunch time, but this was incredibly stressful. Other times with deadlines, when we had made little bits of progress, we were not as stressed, and the quality of our work was noticeably different.

Final Performance Analysis

The project can run successfully, leveling an area on the surface of the Moon with a swarm of EZ-RASSOR 4 (or more) rovers managed by the central task manager which assigns instructions generated from the leveling algorithm. Our solution runs with no known bugs and handles any unexpected errors that might arise. Furthermore, the program is not a heavy simulation and can run 4 rovers on a standard laptop. Technical in-depth documentation of the components of our solution and performance can be found in the documentation folder in our Github repository.

Tasks Completed and Remaining

Completed:

- Image Processing: Image processing transforms a digital elevation model into a 2-dimensional matrix of numbers representative of the elevation.

- Leveling Algorithm: The Leveling Algorithm creates the set of instructions which rovers use to level the area efficiently.

- Central Task Manager: The Central Task Manager assigns rovers instructions and handles rover problems if encountered.

- Unity Simulation: The Unity Simulation simulates our solution, is connected to the ROS topics and publisher/subscriber system, and allows terrain manipulation

Remaining:

- None: All objectives and requirements have been met

Project Terms and Definitions

Terms	Definitions
EZ-RASSOR	EZ-RASSOR is a robot developed to simulate the NASA RASSOR (Regolith Advanced Surface Systems Operations Robot) on a smaller scale. The open-source software for this robot is also called EZ-RASSOR.
RASSOR	RASSOR (Regolith Advanced Surface Systems Operations Robot) is a NASA developed robot rover for lunar exploration and regolith collection.
Regolith	The layer of unconsolidated rocky material covering bedrock. In our case, this concerns the material that covers the surface of the moon.
ROS	Robot Operating System, the current operating system that the EZ-RASSOR rovers are using. This software is open-sourced and free for public use, which aligns with the mission of the EZ-RASSOR rover.
Gazebo	Gazebo is a robotics simulation software that serves as the main software of this project.

References

[1] NASA. “Artemis Program.” *Wikipedia*, Wikimedia Foundation, 24 Apr. 2021, en.wikipedia.org/wiki/Artemis_program.

[2] Allain, Rhett. “The Acceleration of Moon Dust.” Wired, Conde Nast, 7 Mar. 2013, www.wired.com/2013/03/the-acceleration-of-moon-dust/.

[3] Albury, Jordan. (2020). “FlaSpaceInst/EZ-RASSOR.” GitHub, github.com/FlaSpaceInst/EZ-RASSOR/wiki/Architecture.

[4] Albury, Jordan. “FlaSpaceInst/EZ-RASSOR.” GitHub, 2020, github.com/FlaSpaceInst/EZ-RASSOR/wiki/ezrassor_autonomous_control.

[5] Ron, C. “FlaSpaceInst/EZ-RASSOR.” GitHub, 26 Aug. 2020, github.com/FlaSpaceInst/EZ-RASSOR/blob/mainline/packages/autonomy/ezrassor_autonomous_control/nodes/park_ranger.

[6] IBM. “INT8.” IBM, 2020, www.ibm.com/docs/en/informix-servers/14.10/14.10?topic=types-int8.

[7] Ron, C. “FlaSpaceInst/EZ-RASSOR.” GitHub, 2020, github.com/FlaSpaceInst/EZ-RASSOR/wiki/ezrassor_swarm_control.

[8]https://moon.nasa.gov/resources/87/high-resolution-topographic-map-of-the-moon/

[9]MAPP: a Scalable Multi-Agent Path Planning Algorithm with ...https://arxiv.org › pdf

[10] Wasser, Molly. “Clavius Crater on the Moon.” NASA Science Earth's Moon, 2020, moon.nasa.gov/news/155/theres-water-on-the-moon/.

[11] Silva, Daniel. “NASA Moon DEM Top Down.” Github, 2020, github.com/FlaSpaceInst/EZ-RASSOR/blob/mainline/extra_worlds/nasa_moon_dem/materials/textures/AS16-110-18026HR-512x512.jpg.

[12] Silva, Daniel. “NASA Moon DEM.” Github, 2020, github.com/FlaSpaceInst/EZ-RASSOR/blob/mainline/extra_worlds/nasa_moon_dem/materials/textures/nasa_moon_dem.jpg.

[13] Gis Geography. Gis Geography, 2021, gisgeography.com/wp-content/uploads/2016/04/SRTM-300x161.png.

[14] Raffyraffy. “3D Map004 Moon Lunar Landscape in Blender Model.” Turbosquid, 2020, www.turbosquid.com/3d-models/3d-lunar-blender-model-1545577.

[15]FlaSpaceInst. “FlaSpaceInst/EZ-RASSOR.” *GitHub*, github.com/FlaSpaceInst/EZ-RASSOR/

[16] Pazar, Cole & McKeown, Ben & Shanley, Christopher & Coto, Miguel. (2020). Lunar Regolith Sample Excavation Company - A Space Resource Business Plan. 10.13140/RG.2.2.25589.60641/1.

[17]“International Momentum for Space Resources Ramps Up.” Aerospace America, 26 Nov. 2019, aerospaceamerica.aiaa.org/year-in-review/international-momentum-for-space-resources-ramps-up/.

[18] Webb, David. “Orbit of the Moon.” Wikipedia, 2021, en.wikipedia.org/wiki/Orbit_of_the_Moon#/media/File:Earth-Moon.PNG.

[19] Webb, David. “Orbit of the Moon.” Wikipedia, Wikimedia Foundation, 6 Apr. 2021, en.wikipedia.org/wiki/Orbit_of_the_Moon.

[20] Dunbar, Brian. “Is There an Atmosphere on the Moon?” NASA, NASA, 8 June 2013, www.nasa.gov/mission_pages/LADEE/news/lunar-atmosphere.html.

[21] Shepherd, Marshall. “Time Sequence of Lunar Mare -- Lava Plain -- Flows in 0.5 Billion Year Time Increments, with Red Areas in Each Time Step Denoting the Most Recently Erupted Lavas. The Timing of the Eruptions, along with How Much Lava Was Erupted, Helped Scientists Determine That the Moon Once Had an Atmosphere and That the Lunar Atmosphere Was Thickest about 3.5 Billion Years Ago.” Forbes, 2019, www.forbes.com/sites/marshallshepherd/2019/07/18/does-our-moon-have-weather/?sh=2064a41e45b2.

[22] Hsu, Jeremy. “Apollo 16 Astronaut Drags a Rake through Lunar Regolith to Collect Rock Fragments.” Astrobio, 2010, www.astrobio.net/images/galleryimages_images/Gallery_Image_6949.jpg.

[23] NASA. “Prototype-of-NASA-Excavator - RASSOR.” Wikipedia, 2016, fr.wikipedia.org/wiki/Fichier:Prototype-d%27excavateur-de-la-NASA--RASSOR.jpg.

[24] Wikiwand. “3D Rendering of a DEM.” Wikiwand, 2021, www.wikiwand.com/en/Digital_elevation_model.

[25] Schlieder, Sarah. “NASA RASSOR Excavating.” NASA, 2020, www.nasa.gov/rassor-bucket-drum-challenge.

[26] https://www.space.com/nasa-moon-landing-dust-concerns.html

[27] Open Robotics. “Wiki.” *Ros.org*, 25 Mar. 2020, wiki.ros.org/melodic/Installation/Ubuntu.

[28] Tully Foote on April 18, 2018 4:38 PM. “ROS Melodic Morenia Logo and Tshirt Campaign.” *Ros.org*, www.ros.org/news/2018/04/ros-melodic-morenia-logo-and-tshirt-campaign.html.

[29] Francis, Sam. “The Rise of the Robot Operating System.” *Robotics & Automation News*, 16 May 2019, roboticsandautomationnews.com/2019/05/16/the-rise-of-the-robot-operating-system/22485/.

[30]Osrf. “Media.” *Gazebo*, gazebosim.org/media.

[31] Blender. “Blender Logo.” Blender, 2019, www.blender.org/about/logo/.

[32] Andvari. “Blender Lunar Map.” Blenderartists, 2014, blenderartists.org/uploads/default/original/4X/2/0/d/20d1ac9e16586f935e04017a03e1ec6ab1d5d8d9.jpg.

[33] Python. “Python Logo.” Python, 2021, www.python.org/community/logos/.

[34] Ubuntu. “Ubuntu Logo.” Ubuntu, 2019, design.ubuntu.com/brand/ubuntu-logo/.

[35] Barton, Madison. “Discord Logo.” Pininterest, 2021, www.pinterest.com/pin/643311128003585753/.

[36] LogosWorld. “Github Logo.” Logos World, 2021, logos-world.net/wp-content/uploads/2020/11/GitHub-Logo-700x394.png.

[37] React Native. “React Native Logo”, https://www.google.com/url?sa=i&url=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FReact_Native&psig=AOvVaw1xWzPOXIPX6nhgbZkciM3F&ust=1619498776514000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCLi20p2Nm_ACFQAAAAAdAAAAABAP

[38] “Core Components and APIs · React Native.” *React Native*, reactnative.dev/docs/components-and-apis.

[39] “JSX In Depth.” *React*, reactjs.org/docs/jsx-in-depth.html.

[40] “JSX In Depth.” *React*, reactjs.org/docs/jsx-in-depth.html#javascript-expressions-as-props.

[41] Unity. “Unity Logo.” Unity 3D, 2021, unity3d.com/profiles/unity3d/themes/unity/images/pages/branding_trademarks/unity-masterbrand-black.png.

[42] Technologies, U. (n.d.). Meshes. Retrieved April 03, 2021, from https://docs.unity3d.com/Manual/class-Mesh.html

[43] Sean Duffy Oct 9 2019 · Article (35 mins) · Intermediate, & Duffy, S. (n.d.). Runtime mesh manipulation with unity. Retrieved April 03, 2021, from https://www.raywenderlich.com/3169311-runtime-mesh-manipulation-with-unity#toc-anchor-002

[44] Technologies, U. (n.d.). Creating and using scripts. Retrieved April 24, 2021, from https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html

[45] W., Bryan. “How AI in Unity Is Revolutionizing Video Game Design.” *The Ultimate Resource for Video Game Design*, 14 Mar. 2021, www.gamedesigning.org/learn/unity-ai/.

[46] “NavMesh Movement.” *Unity Learn*, learn.unity.com/tutorial/navmesh-movement?uv=2020.2&courseId=5dd851beedbc2a1bf7b72bed&projectId=60645258edbc2a001f5585aa.

[47]Technologies, U. (n.d.). TerrainData.SetHeightsDelayLOD. Retrieved April 21, 2021,from https://docs.unity3d.com/ScriptReference/TerrainData.SetHeightsDelayLOD.html

[48] Technologies, U. (n.d.). Projector. Retrieved April 24, 2021, from https://docs.unity3d.com/Manual/class-Projector.html

[49] Simulating robots with ROS and Unity. (2020, November 18). Retrieved April 17, 2021, from

https://resources.unity.com/unitenow/onlinesessions/simulating-robots-with-ros-and-unity

[50] Robotics simulation in Unity is as easy as 1, 2, 3!「1、2 の 3！」で簡単に出来る、UNITY でのロボティクスシミュレーション. (2021, March 08). Retrieved April 17, 2021, from https://blogs.unity3d.com/2020/11/19/robotics-simulation-in-unity-is-as-easy-as-1-2-3/

[51] Byon, Johnathan. *NASA*, NASA, public.ksc.nasa.gov/Don/.

[52] Medley, Sam. “Ubuntu 18.04 LTS (Bionic Beaver).” Notebook Check, 2018, www.notebookcheck.net/Ubuntu-18-04-LTS-Bionic-Beaver-now-available.301388.0.html.

[53] sil2100. Ubuntu Wiki, 2020, wiki.ubuntu.com/BionicBeaver/ReleaseNotes.

[54] Microsoft. “Microsoft 10 Logo.” Seeklogo, 2021, seeklogo.com/vector-logo/267364/windows-10-icon.

[55] Dattalo, Amanda. “ROS Wiki.” Ros.org, 2018, wiki.ros.org/ROS/Introduction.

[56] Seeklogo. “MacOS Logo.” Seeklogo, 2021, seeklogo.com/vector-logo/86757/mac-os.

[57] Boone, Joseph. “Windows Ubuntu Dual Boot.” Help Desk Geek, 2019, helpdeskgeek.com/wp-content/pictures/2019/09/How-To-Dual-Boot-Ubuntu-Windows-10-Main.png.webp.

[58] Python. “Python Logo.” Python, 2021, www.python.org/community/logos/.

[59] Ruffsl. Gazebo Logo. 2015, en.wikipedia.org/wiki/File:Gazebo_logo.svg.

[60] Beltrame, Giovanni. “V-REP Logo.” Lenkaspace, 2018, lenkaspace.net/tutorials/programming/robotSimulatorsComparison.

[61] Pawar, Pawan. “Insights to Agile Methodologies for Software Development.” Hacker Noon, 14 Dec. 2019, hackernoon.com/a-case-study-type-insight-into-agile-methodologies-for-software-development-cd5932c6.

[62] Scrum Image Example ,https://blog.bydrec.com/a-comprehensive-comparison-between-the-agile-scrum-and-waterfall-methodology

[63] Kanban Board Example, https://en.wikipedia.org/wiki/Kanban_board

[64] Wilson, N. “Waterfall Model.” Datascience, 2017, www.datascience-pm.com/waterfall/.

[65] Made by Senior Design 2 Group 28 Fall 2021