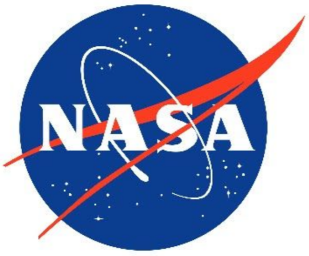


EZ-RASSOR

SWARM AI

Group 28

Spring 2021 - Fall 2021

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

Project Sponsor:

Michael Conroy

Table of Contents

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.

Leveling Algorithm

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.

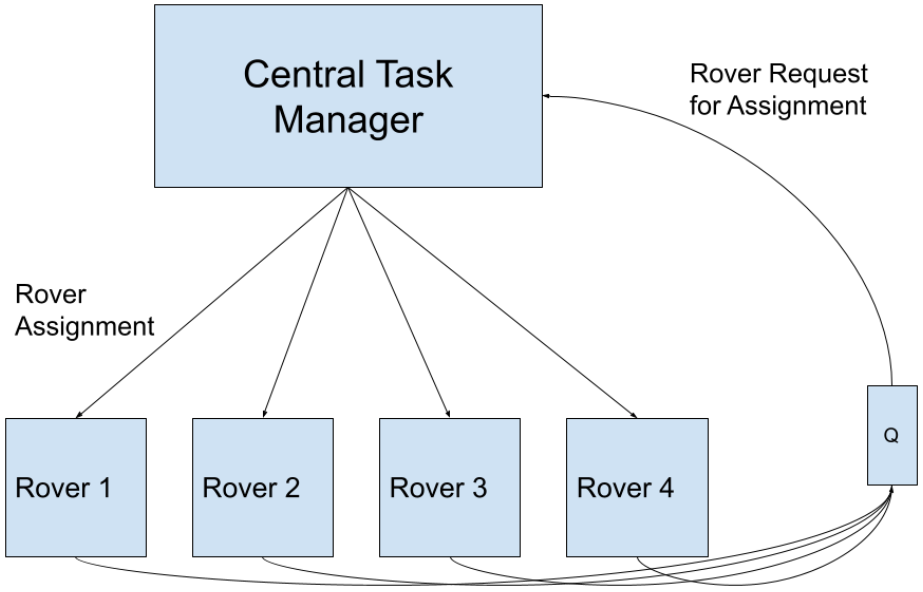
Note that we cannot use dump locations or dig locations which are already part of enough instructions to be considered level; therefore, we might have to delete values from the sorted distance matrix as we iterate through each key. This could result in an iteration of time O(m^2), although very unlikely. 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.

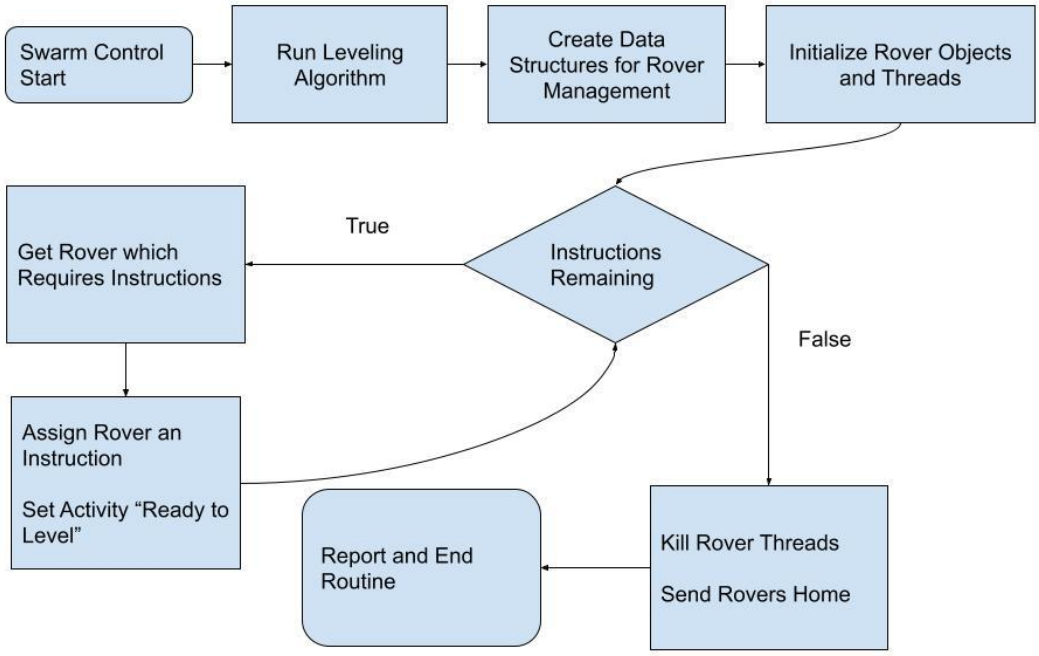
The CTM does not control rovers directly. The CTM instructs rovers what to do. The rovers ask the CTM for instruction or help by placing their ids in a queue which is checked by the CTM.



CTM and rover communication interface

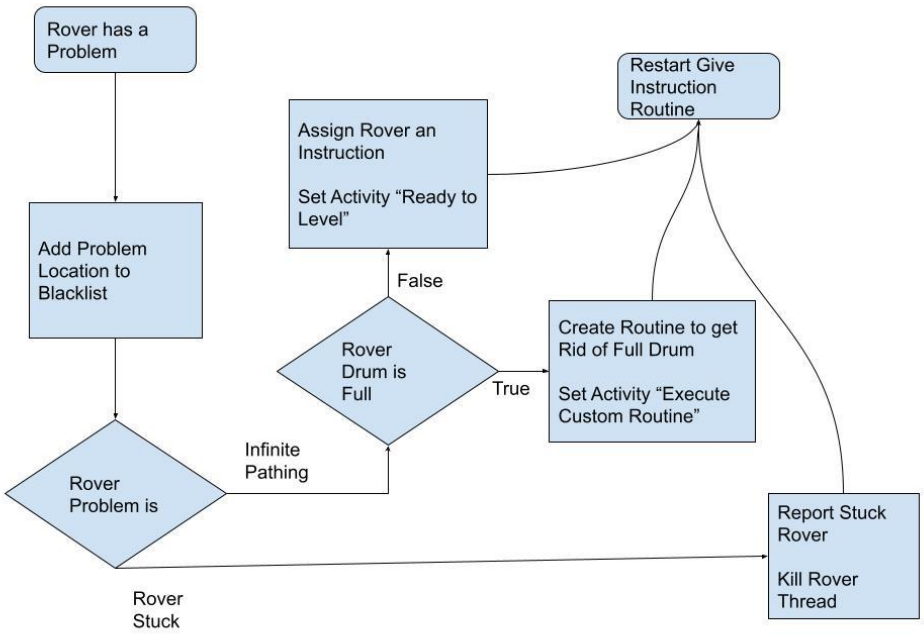
Assignment of instructions is implemented and optimized in the CTM by determining the instruction which contains the dig location closest to the rover's current position and assigning that instruction to the rover. This ensures that the rover's total distance traveled is minimized. The CTM continues 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

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

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

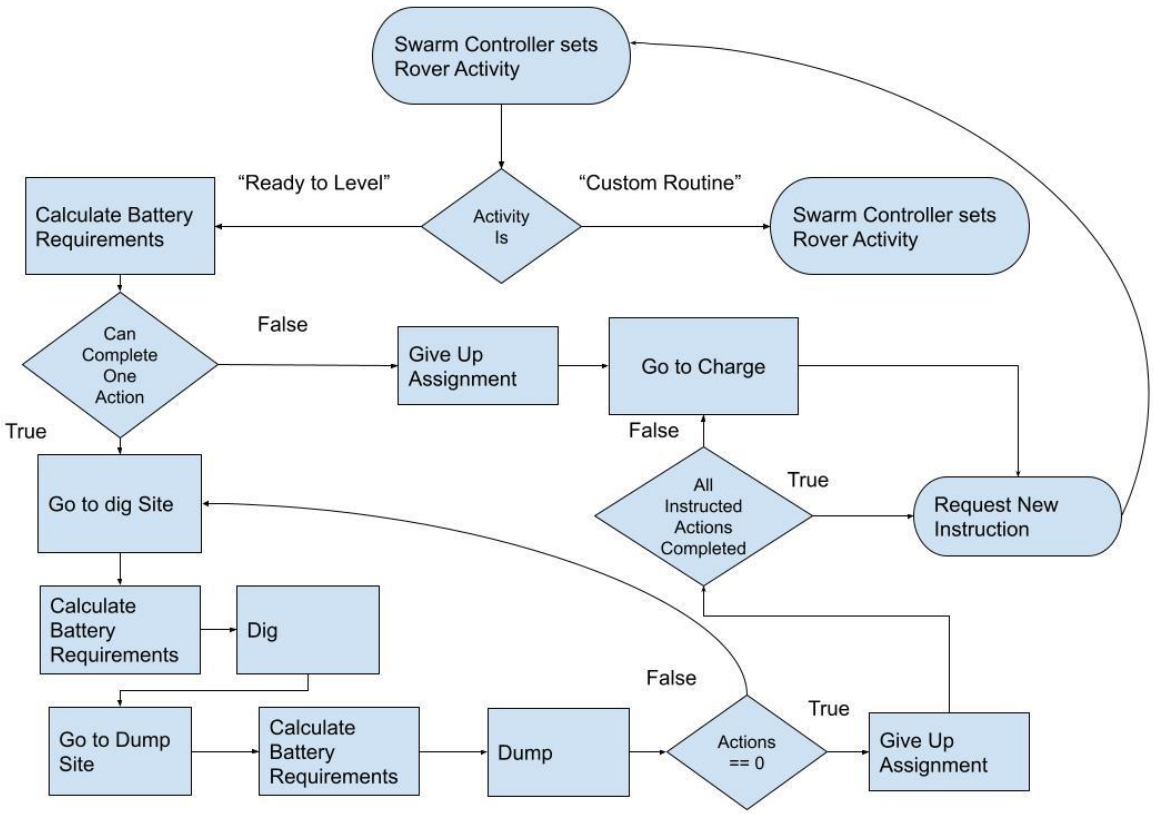
Once a rover is assigned locations to be assigned and its activity is set to “Ready to Level” by the CTM, the rover executes a routine to level at least one of the two assigned locations.

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.

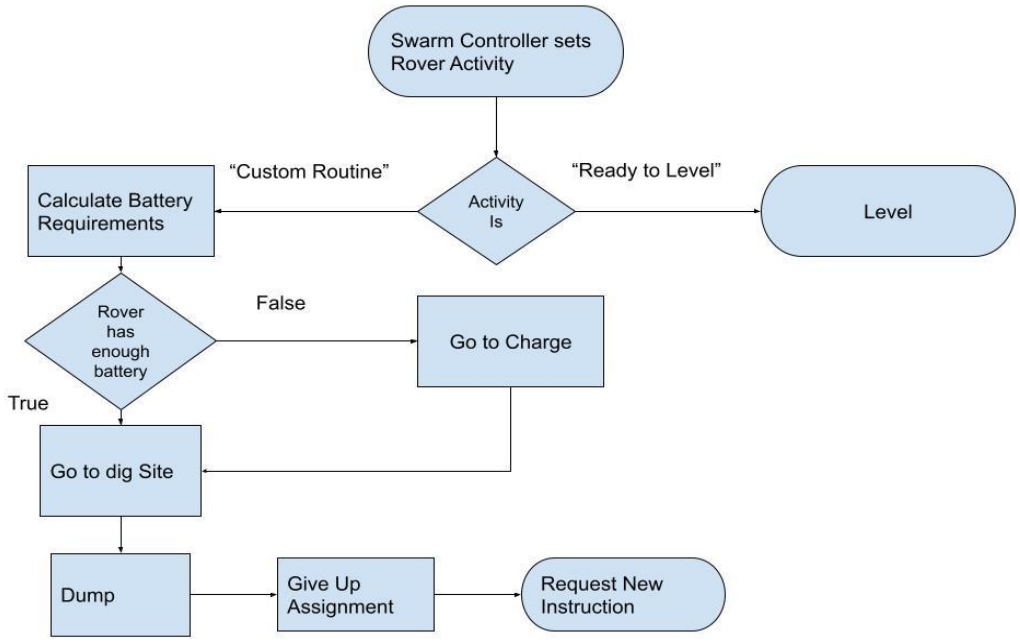
If the rover can complete at least one action, then it begins the leveling routine. The rover begins leveling by driving to the dig location. After the rover arrives at the dig location, the rover checks if more battery has been used to arrive at the dig location then previous location. If more battery 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

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

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.