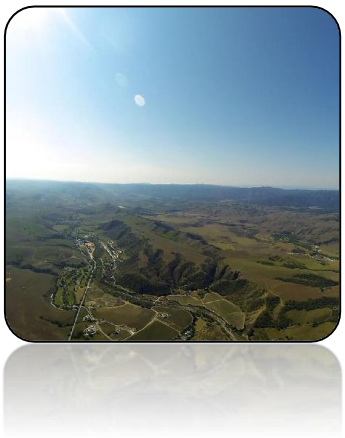
Rover Project Walkthrough

By DigitalCodeFu



The goals / steps of this project are the following:

Training and Calibration – Notebook Analysis

- ✓ Download the simulator and take data in "Training Mode"
- ✓ Test out the functions in the Jupyter Notebook provided
- ✓ Add functions to detect obstacles and samples of interest (golden rocks)
- ✓ Fill in the `process_image()` function with the appropriate image processing steps (perspective transform, color threshold etc.) to get from raw images to a map. The `output_image` you create in this step should demonstrate that your mapping pipeline works.
- ✓ Use `moviepy` to process the images in your saved dataset with the `process_image()` function. Include the video you produce as part of your submission.

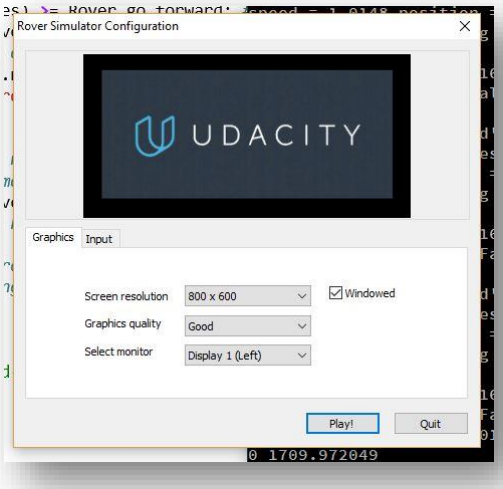
Autonomous Navigation and Mapping

- ✓ Fill in the `perception_step()` function within the `perception.py` script with the appropriate image processing functions to create a map and update `Rover()` data (similar to what you did with `process_image()` in the notebook).
- ✓ Fill in the `decision_step()` function within the `decision.py` script with conditional statements that take into consideration the outputs of the `perception_step()` in deciding how to issue throttle, brake and steering commands.
- ✓ Iterate on your perception and decision function until your rover does a reasonable (need to define metric) job of navigating and mapping.

Training and Calibration – Notebook Analysis

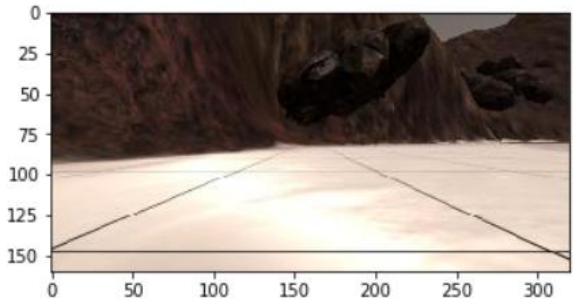
Run the functions provided in the notebook on test images (first with the test data provided, next on data you have recorded). Add/modify functions to allow for color selection of obstacles and rock samples.

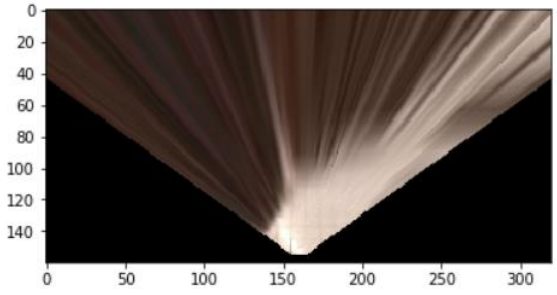
I ran the Rover_Project_Test_Notebook analysis first using the sample data. After recording test data in the simulator and updating the image references, I noticed I had to make small adjustments to the color scales and perspective transform coordinate pixels (depending on the resolution). I ultimately ended up going with 800 x 600, with 'Good' graphics quality, and my main laptop display monitor.



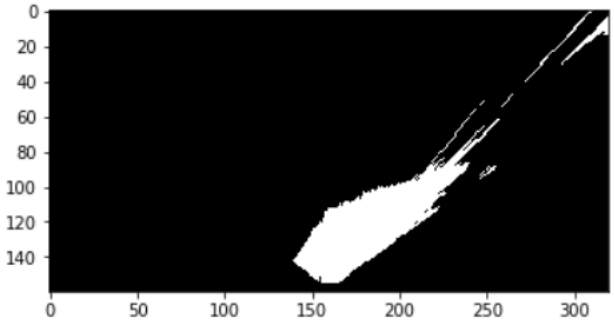
The notebook came with default code from the lessons to map the navigable terrain using a color threshold, perspective transform, coordinate translation and rotation, and coordinate mapping. I used many of the same functions to perform the same steps to identify the pixels from the Rover Cam by category, transform them into a top-down perspective for mapping, convert the image to Rover-centric coordinates (both in x, y and polar formats), and then to rotate and translate those coordinates into world map locations.

A key function provided in the notebook is perspect_transform(img, src, dst). This basically transforms the image pixels from a Rover Cam into a top-down view with the available data. This can be seen from the image on the left (Rover Cam) to the image on the right (perspective transformed).



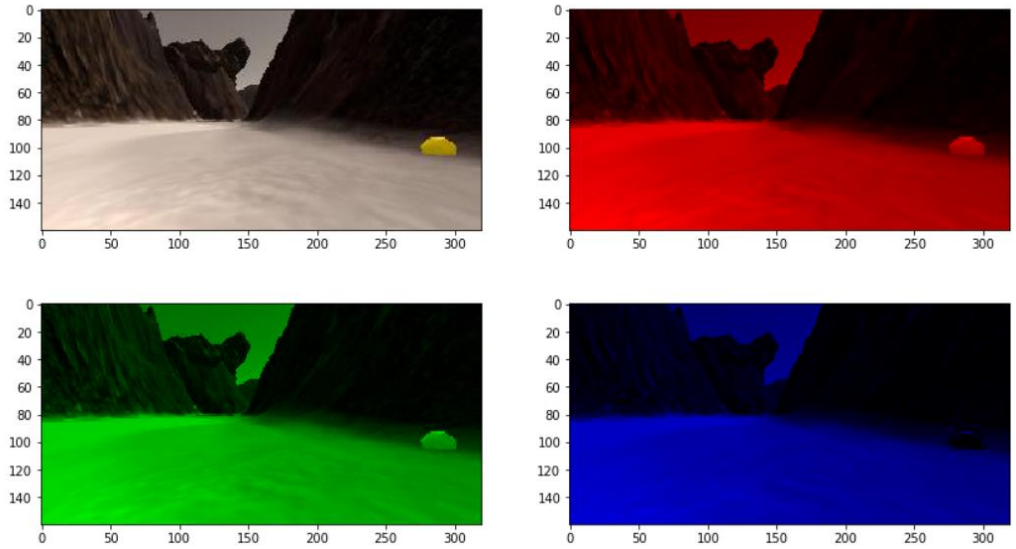


The next step is to apply a color threshold to the top-down view so we can gather meaningful data, such as where the navigable terrain is. The notebook function color_thresh(img, rgb_thresh=(160, 160, 160)) was used as provided, which resulted in a binary image file highlighting the ground. The ground is a light sandy color, so anything brighter than a 160 value on each of the red, green, and blue pixel values would come through as white in the binary image below:



To get the right color value ranges for each object group to identify, I first made single-channel copies (one color band only) of the image. Then by looking at each color band separately, colors values can be easily calculated from specific regions of pixels unique to each object group (rock, navigable terrain, or obstacles).

```
dtype: uint8 shape: (160, 320, 3) min: 0 max: 255
<matplotlib.image.AxesImage at 0x2184e8645f8>
```



The code I added in to extract color value data from specific regions of pixels is below:

```
myimage_targets = np.copy(myimage)

image_targets = {"sand_box": {"Y1": 88, "Y2": 160,  "X1": 0,   "X2": 145},
                 "obst_box": {"Y1": 31,  "Y2": 70,   "X1": 0, "X2": 138},
                 "rock_box": {"Y1": 95, "Y2": 104,  "X1": 282, "X2": 297} }

colors = [ {"channel": red_channel,   "number": 0, "name": "red channel"},
           {"channel": green_channel, "number": 1, "name": "green channel"},
           {"channel": blue_channel,  "number": 2, "name": "blue channel"} ]

for target in image_targets.keys():
    print("\n", target, " -------------")
    source = np.float32([[ image_targets[target]["X1"], image_targets[target]["Y1"] ],
                         [ image_targets[target]["X2"], image_targets[target]["Y1"] ],
                         [ image_targets[target]["X2"], image_targets[target]["Y2"] ],
                         [ image_targets[target]["X1"], image_targets[target]["Y2"] ]])
    cv2.polylines(myimage_targets, np.int32([source]), True, (0, 0, 255), 1)
    for color in colors:
        print( color["name"], "\t",
               "min: ",  color["channel"][ image_targets[target]["Y1"]: image_targets[target]["Y2"],
                         image_targets[target]["X1"]: image_targets[target]["X2"],
                         [color["number"]] ].min(), ", ",
               "max: ",  color["channel"][ image_targets[target]["Y1"]: image_targets[target]["Y2"],
                         image_targets[target]["X1"]: image_targets[target]["X2"],
                         [color["number"]] ].max(), ", ",
               "mean: ", round(
                         color["channel"][ image_targets[target]["Y1"]: image_targets[target]["Y2"],
                         image_targets[target]["X1"]: image_targets[target]["X2"],
                         [color["number"]] ].mean(), 1) )
```

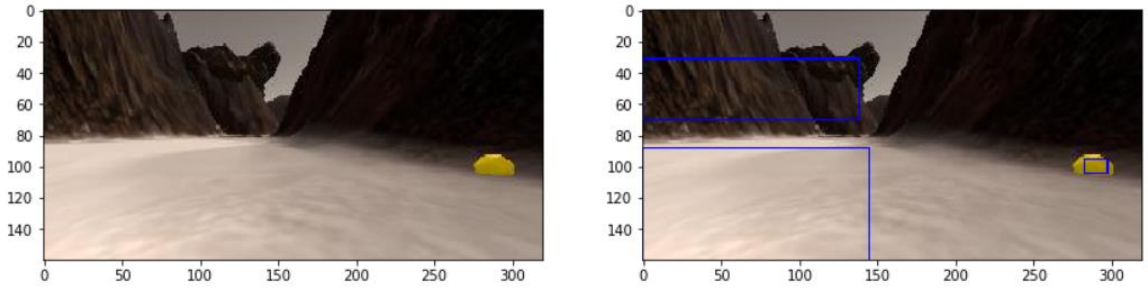
The colors values from each region of pixels are shown below. This is valuable data for use in color filters, which are later used to identify objects by pixel color ranges. The image to the right shows the specific regions of pixels targeted for sampling data.

```
 sand_box  -------------
red channel      min:  171 ,  max:  255 ,  mean:  216.6
green channel    min:  153 ,  max:  238 ,  mean:  196.6
blue channel     min:  141 ,  max:  221 ,  mean:  182.2

 rock_box  -------------
red channel      min:  59 ,  max:  198 ,  mean:  170.8
green channel    min:  27 ,  max:  172 ,  mean:  140.3
blue channel     min:  0 ,  max:  79 ,  mean:  3.6

 obst_box  -------------
red channel      min:  4 ,  max:  104 ,  mean:  40.1
green channel    min:  0 ,  max:  89 ,  mean:  28.0
blue channel     min:  0 ,  max:  82 ,  mean:  21.3

<matplotlib.image.AxesImage at 0x2184e90c8d0>
```



Instead of using the default color_thresh() function, I created a new function color_band() which takes a min or max, or both, to filter the pixels in a specific RGB color band. I set this up to identify the rock samples as precisely as possible, to avoid bad navigation (chasing 'ghost' rock samples) and to not overlap the filters for navigable terrain, obstacles, and rock samples. The design and use of the function is below:

In [7]:
```
# My color band function
def color_band(img, rgb_min=(0, 0, 0), rgb_max=(255, 255, 255)):
    binary_image = np.zeros_like(img[:, :, 0])
    thresh_mask = (img[:, :, 0] > rgb_min[0]) \
                & (img[:, :, 1] > rgb_min[1]) \
                & (img[:, :, 2] > rgb_min[2]) \
                & (img[:, :, 0] < rgb_max[0]) \
                & (img[:, :, 1] < rgb_max[1]) \
                & (img[:, :, 2] < rgb_max[2])
    binary_image[thresh_mask] = 1
    return binary_image

threshed2 = color_band(warped, rgb_min=(160, 160, 160))
plt.imshow(threshed2, cmap='gray')
```

```
# 3) Apply color threshold to identify navigable terrain/obstacles/rock samples
navi_min = (160, 140, 135)
obst_max = (159, 139, 134)
rock_min, rock_max = (130, 100, 0), (230, 180, 40) # (144, 116, 0), (214, 163, 31)
navi_img = color_band(warped, rgb_min=navi_min)
obst_img = color_band(warped, rgb_max=obst_max)
rock_img = color_band(warped, rgb_min=rock_min, rgb_max=rock_max)
```

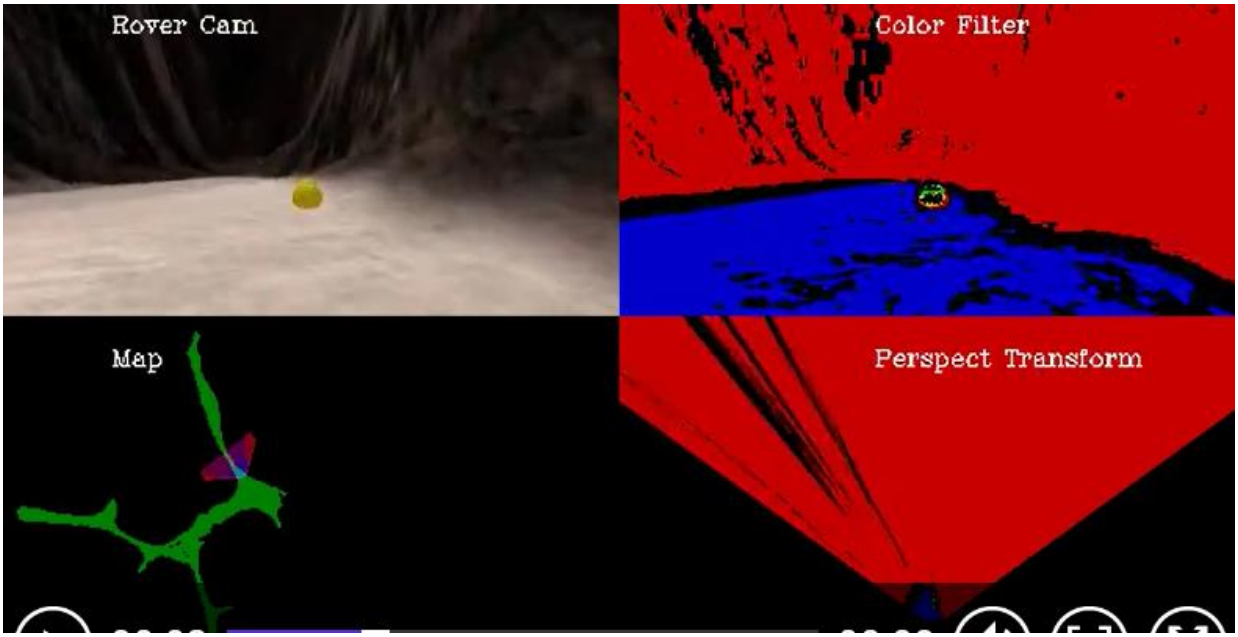
Populate the process_image() function with the appropriate analysis steps to map pixels identifying navigable terrain, obstacles and rock samples into a worldmap. Run process_image() on your test data using the moviepy functions provided to create video output of your result.

The process_image() function takes in raw camera images from the Rover and processes them into a video of four different image streams. The first two steps provided did not need to be modified. First, the book defined the source and destination points of the perspective transformation (the source from the Rover Cam to the destination of map cords). Secondly, it uses the perspect_transform(img, source, destination) function to generate a warped image. The third step involves applying color thresholds, or color bands in this base, so the modification started here. I replaced the navigable terrain color threshold with my color band function, and added code for the obstacles and rock samples.

```
# 3) Apply color threshold to identify navigable terrain/obstacles/rock samples
navi_min = (160, 160, 160)
obst_max = (160, 160, 160)
rock_min, rock_max = (144, 116, 0), (214, 163, 31)
navi_img = color_band(warped, rgb_min=navi_min)
obst_img = color_band(warped, rgb_max=obst_max)
rock_img = color_band(warped, rgb_min=rock_min, rgb_max=rock_max)
```

Then for each type of pixel group, I mapped the image pixel locations using the rover_coords() function to locations on a coordinate plane in which the Rover Cam position is (0, 0), instead of half of the maximum x value and the maximum y value. Then for each group, the rover-centric coordinates are translated into world coordinates and populate the world map. To translate coordinates to the world map, the function takes into account the Rover's x, y location in the map to adjust for the distance from 0 and the Rover's bearing to rotate the map data pixels.

For each image I updated the Databucket.worldmap + 1 for blue if a pixel was navigable terrain, red if it was an obstacles, and green if it was a rock sample. One final step is to add an image HUD at the bottom of the image before its processed into the video. I have the Rover Cam view, world map view, and color-adjusted perspective transform as default. In addition, I added an image which shows the Rover Cam view with the color filter over it in the top right hand corner.



Autonomous Navigation and Mapping

Fill in the perception_step() (at the bottom of the perception.py script) and decision_step() (in decision.py) functions in the autonomous mapping scripts and an explanation is provided in the writeup of how and why these functions were modified as they were.

Robotics is all about perception, decision-making, and actuation. The perception_step() function accepts a Rover object as a parameter and returns the same Rover object after making several updates based on the image data streaming in from the Rover Cam. The images are processed using similar functions such as perspective_transform() to translate Rover Cam data to a top-down view, color_band() to identify object categories from pixel data, and pix_to_world() to translate to world coordinates. These functions are applied to each object category (navigable terrain, obstacles, and rock samples). I added my color_band() function to make the color filtering of pixels much easier to handle, and conditioned the updating of world map data to only update when the Rover was in a relatively stable position (as measured by roll and pitch). This reduces the distortion of data in the perspective transform which translated to the world map data.

```
# 7) Update Rover worldmap (to be displayed on right side of screen)
if ( (Rover.roll < 0.4) or (Rover.roll > 359.6) and
     (Rover.pitch < 0.3) or (Rover.pitch > 359.7) and
     (Rover.vel >= 0) ):
    Rover.worldmap[obst_ypix_world, obst_xpix_world, 0] += 1
    Rover.worldmap[rock_ypix_world, rock_xpix_world, 1] += 1
    Rover.worldmap[navi_ypix_world, navi_xpix_world, 2] += 1
```

Once the perception_step() translates the pixel data to polar coordinates, it updated the Rover.nav_angles and Rover.nav_dists for angle and distance pixel data, respectively. I modified this and expanded it to all three pixel categories, and also updated new Rover categories for rock angles and distances. All four of these elements of pixel data in polar coordinates are very important for the Rover to make decisions, and as such are referred to often in the decision_step() function. In addition to these fields, I also created nav_angles_avg to have the average angle of navigable terrain data ready and processed only once for efficiency, hard_turn to indicate which direction (+/- 15 degrees following the average angle side) to veer to if the Rover needs to avoid an obstacle quickly, rock_angles_avg to store the average rock angle, rock_dists_avg to estimate the average distance between the Rover and the rock sample, and rock_dists_min to estimate the shortest distance between the Rover the rock sample. With this data at the ready, I also added a new field which has a small decision-making element to it based on the visual data. rock_nearby is a boolean flag which determines if a rock sample is visually detected with an average distance value less than 50.0, at which point the Rover adjusts its navigational behavior when chasing a rock sample. This can be adjusted by changing Rover.rock_min_dist.

```
# Update if rock samples are seen close or sample_nearby.
Rover.rock_nearby = ( len(Rover.rock_dists) > 1 and
                      Rover.rock_dists_avg < Rover.rock_min_dist )
```

In addition, I added two processes for enhanced navigational data to be updated in the Rover object, one to avoid incoming collisions and the other to target unmapped terrain. To avoid collisions, Rover.nav_dist_front is updated with the distance values of navigable terrain seen between the angles of -3 degrees and 3 degrees (directly in front of the Rover). If the distance values of navigable terrain narrowly in front of the Rover decreases to a value of 20 or below, Rover.collision_detected is updated to True. This can be adjusted by changing Rover.fwd_obstacle_dist.

```
# Update navigable terrain distance in front of Rover and detect for collisions and impacts.
nav_dist_front = []
for i in range(len(Rover.nav_angles)):
    if  (Rover.nav_angles[i] * 180/np.pi) > -3.0 \
    and (Rover.nav_angles[i] * 180/np.pi) < 3.0:
        nav_dist_front.append(Rover.nav_dists[i])
if len(nav_dist_front) > 0:    # Avoid a zero division error.
    # Average distance in front of navigable terrain.
    Rover.nav_dist_front = sum(nav_dist_front) / float(len(nav_dist_front))
    # Update if a collision risk is detected.
    Rover.collision_detected = Rover.nav_dist_front <= Rover.fwd_obstacle_dist

# Update the average angle of navigable terrain that has not been mapped.
nav_angles_uncharted = []
for i in range(len(Rover.nav_angles)):
    if Rover.worldmap[navi_ypix_world[i], navi_xpix_world[i], 2] < 30:
        nav_angles_uncharted.append(Rover.nav_angles[i])
if len(nav_angles_uncharted) > 0:    # Avoid a zero division error.
    # Convert angle to degrees and clip +/- 15
    nav_angles_uncharted = [i * 180/np.pi for i in nav_angles_uncharted]
    nav_angles_uncharted = [nav_angles_uncharted[i] for i in range(len(nav_angles_uncharted)) \
                            if (nav_angles_uncharted[i] >= -15) and (nav_angles_uncharted[i] <= 15)]
if len(nav_angles_uncharted) > 0:    # Avoid a zero division error.
    # Average distance in front of navigable terrain.
    Rover.nav_angles_uncharted_avg = sum(nav_angles_uncharted) / float(len(nav_angles_uncharted))
    Rover.nav_angles_uncharted_count = len(nav_angles_uncharted)
```

To target unmapped terrain, the perception_step() stores the average and count of angles for unmapped, navigable terrain. If the blue channel (2) color value is less than 30, then it is considered unmapped. This data is used for navigating with a preference for unmapped terrain rather than mapped terrain, causing the Rover to go to unexplored places more often to fill out the world map.

Finally, I included two processes for navigational correction in the perception_step(), one to detect if the Rover has impacted against an obstacle and another to detect if the Rover is already stuck in a position. To detect an impact, I flag the Rover.impact Boolean when both the throttle is greater than 0 (the Rover is accelerating) and the Rover's velocity is less than -0.2 (the Rover is moving backwards).

```
    # Update if an impact is detected.
    Rover.impact = ( Rover.throttle > 0 and Rover.vel < -0.2 )

    # Update last position after Rover.stuck_time seconds and if stuck update Rover.stuck.
    if Rover.time_updated == None:
        Rover.pos_old = tuple(Rover.pos)
        Rover.time_updated = Rover.total_time
    elif (Rover.total_time - Rover.time_updated) > Rover.stuck_time:
        if  round(Rover.pos_old[0], 0) == round(Rover.pos[0], 0) \
        and round(Rover.pos_old[1], 0) == round(Rover.pos[1], 0) \
        and Rover.turningmode is not 'on' and not Rover.picking_up:
            Rover.stuck = True
            if Rover.turningmode == 'off':
                Rover.opposite_direction = (Rover.yaw + 180) % 360
                Rover.turningmode == 'on'
        else:
            Rover.stuck = False
            Rover.turningmode == 'off'
        # Update last position
        Rover.pos_old = tuple(Rover.pos)
        Rover.time_updated = Rover.total_time
```

To detect if the Rover is stuck in a position, I set a timer (checking elapsed time from Rover.time_updated) to update Rover.pos_old and check if the x, y position of the Rover has not changed over that span of time. I chose 7.0 seconds to be assigned to Rover.stuck_time, which is the span of time to check if the position has not changed. The timer does not check whenever the Rover is turning, and 7.0 seconds should be more time than it takes for the Rover to turn around. If the position is the same, then Rover.stuck Boolean is flagged and the Rover.opposite_direction yaw bearing is set. When getting unstuck, the Rover will head to the opposite direction to try to break free.

The decision_step() is the process which controls the Rover's decision making and actuation. In forward mode, I modified the turning and direction by a priority decision to first head toward the average of the rock sample and navigable terrain, and then directly to the rock sample when within near range. This feature prevents the Rover from getting stuck in narrow corridors when attempting to collect rock samples. If there is no rock samples nearby, then the Rover checks if there is an incoming collision and turns as hard as possible if there is one. Otherwise, the Rover will head toward uncharted, navigable terrain first and then toward the average navigable terrain if nothing else. When heading toward either type of navigable terrain, I used a steer_dampener to smooth out the Rover steering. I also triggered a 'stop' state if the Rover detects an impact or if the closest rock sample distance is within pickup range.

```
# Check for nearby rocks, steer toward it.
if len(Rover.rock_angles) > 1:
    if Rover.rock_nearby: # Head directly toward it
        Rover.steer = Rover.rock_angles_avg
    else: # Head between the rock and average navigable terrain directions.
        Rover.steer = (Rover.rock_angles_avg + Rover.nav_angles_avg) / 2.0
# Else If there is an incoming obstacle, steer away harder to avoid.
elif Rover.collision_detected:
    Rover.steer = Rover.hard_turn
# Otherwise head toward navigable terrain pixels searching for uncharted territory.
else:
    if Rover.nav_angles_uncharted_count > 30:
        Rover.steer = (Rover.nav_angles_avg + Rover.nav_angles_uncharted_avg*2) / 3.0 * Rover.steer_dampener
    else:    # Head toward average navigable terrain pixels (clipped for +/- 15 degrees).
        Rover.steer = Rover.nav_angles_avg * Rover.steer_dampener
# If sufficient nav_angles, stop the Rover if it hits an obstacle or can pickup rock sample.
if Rover.rock_dists_min > 0 and Rover.rock_dists_min <= Rover.rock_pickup_range:
    Rover.throttle = 0
    Rover.brake = 0
    Rover.steer = 0
    Rover.mode = 'stop'
if Rover.impact:
    Rover.throttle = 0
    Rover.brake = Rover.brake_set
    Rover.steer = 0
    Rover.mode = 'stop'
    Rover.stuck = True
```

In stop mode, if the Rover is picking something up it just continues to engage the break. If not, then the Rover checks if a rock sample is nearby, and if so, it will either stop to pick up the rock sample or engage forward mode and steer directly toward it. If the navigable terrain angles increase sufficiently, then forward mode is engaged again.

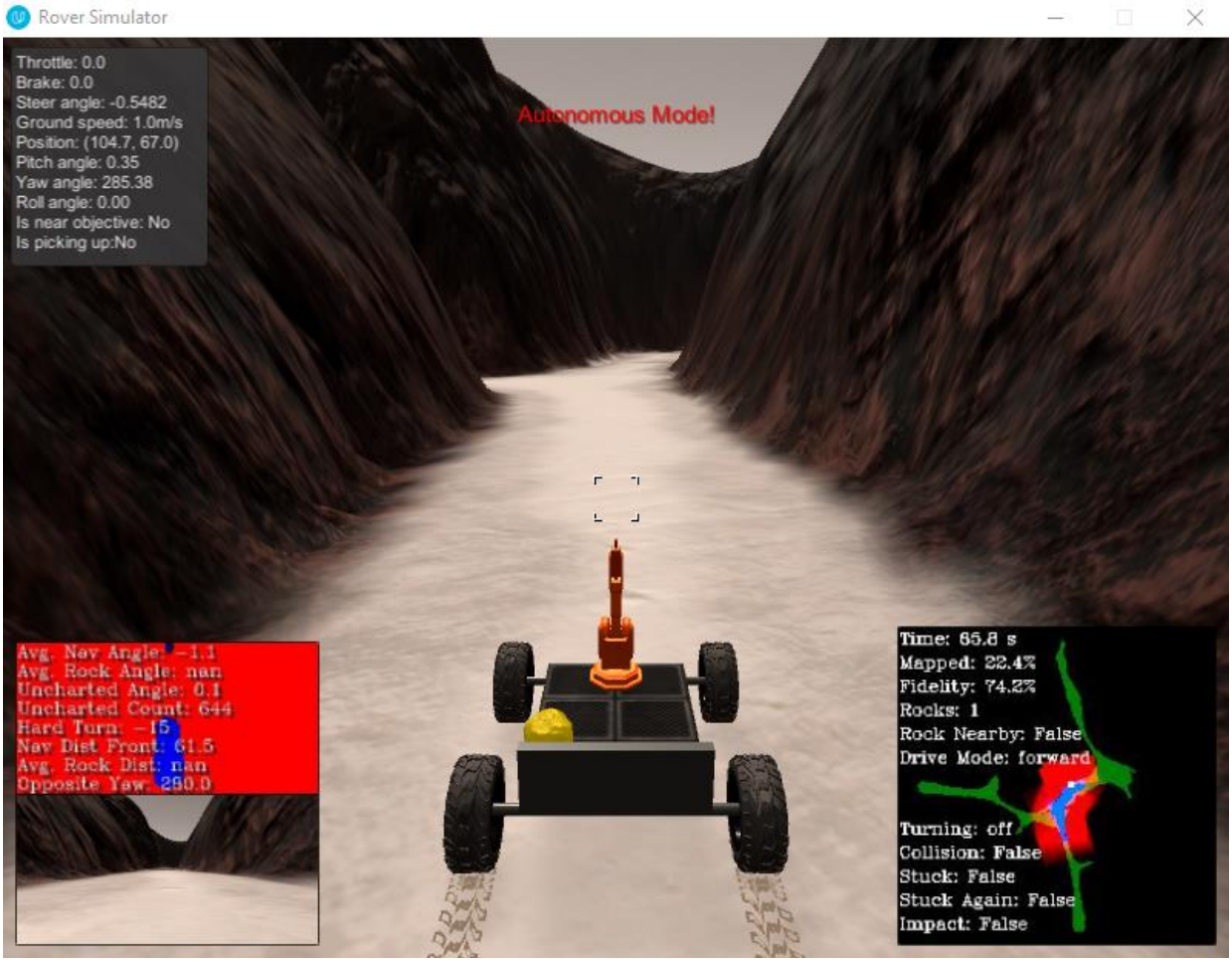
```
# Only execute the turning and forward mode if Rover is not picking up a rock.
if not Rover.send_pickup and not Rover.picking_up:
    # First if the Rover stopped for a visible rock sample, turn to it
    if Rover.rock_nearby:
        Rover.steer = Rover.rock_angles_avg
        # Get closer if the rock sample is not in pickup range.
        if Rover.rock_dists_min > Rover.rock_pickup_range:
            Rover.throttle = Rover.throttle_set
            Rover.mode = 'forward'
            Rover.turningmode = 'off'
        else:    # Brake for pickup.
            Rover.brake = Rover.brake_set
```

Either forward or stop mode, if the Rover.stuck is True, then the Rover will continue turning until it is facing the opposite direction.

```
# If the Rover has not moved and is not in turning mode, turn around 180 degrees.
    #and Rover.turningmode == 'off'
if Rover.stuck and not Rover.picking_up and not Rover.send_pickup:
    # If in the same location, then Rover is stuck and needs to get out.
    Rover.throttle = 0.0
    Rover.brake = 0 # Release the brake to allow turning
    # Go in reverse if turning during the last stuck flag did not work.
    if Rover.stuckagain:
        Rover.throttle = -2.0
    Rover.steer = 15
    if round(Rover.yaw, -1) == round(Rover.opposite_direction, -1):
        Rover.stuck = False
```

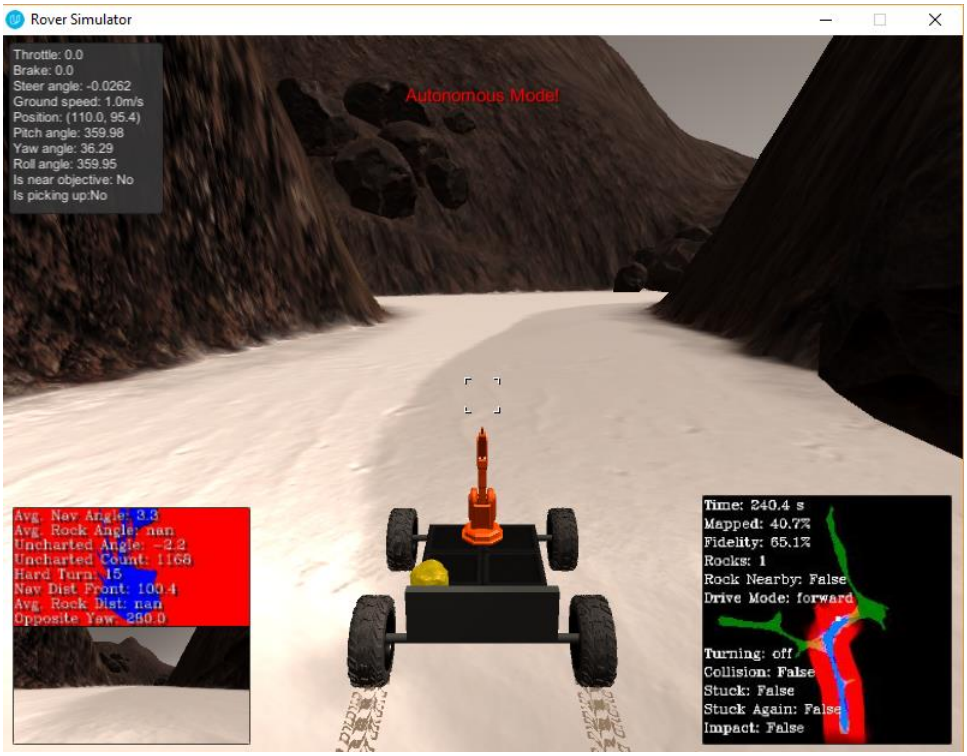
Launching in autonomous mode your rover can navigate and map autonomously. Explain your results and how you might improve them in your writeup.

My results have varied throughout the iterations, but this version meets the project requirements. Overall the results are pretty good, but could be improved further. Calibration with the distances, angle counts, and uncharted angles needs to be performed reviewing the data output through a session. The Rover still will miss an opportunity to turn at a fork in the road that offers uncharted map to explore. Fixing that would be great. The Rover successfully mapped 40.7% of the simulation world at a 65.1% fidelity. However, once the Rover continued and mapped about 83.0%, the fidelity dropped below 60%. I would adjust my color band and map updating tolerances for rover stability to achieve better results.

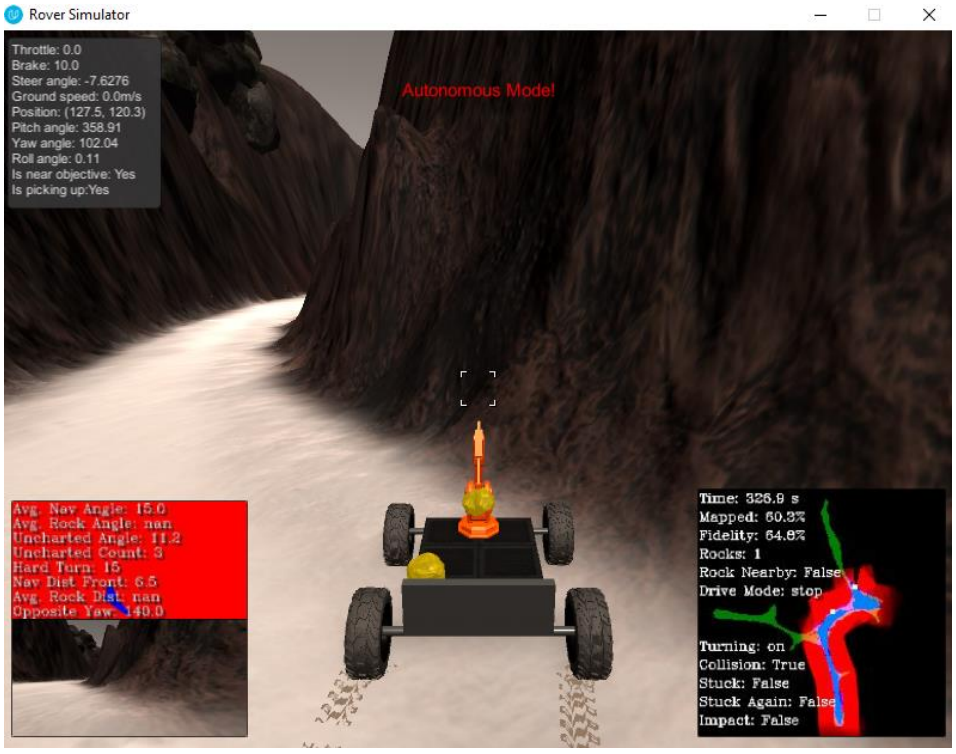


I also modified the supporting_functions.py file to include several additional data elements on the heads-up-display (HUD), such as various statuses, modes, angles, and distances. This makes for much easier adjustment to the logic, because the image files store operational data themselves and this makes for easy review and understanding of the process.

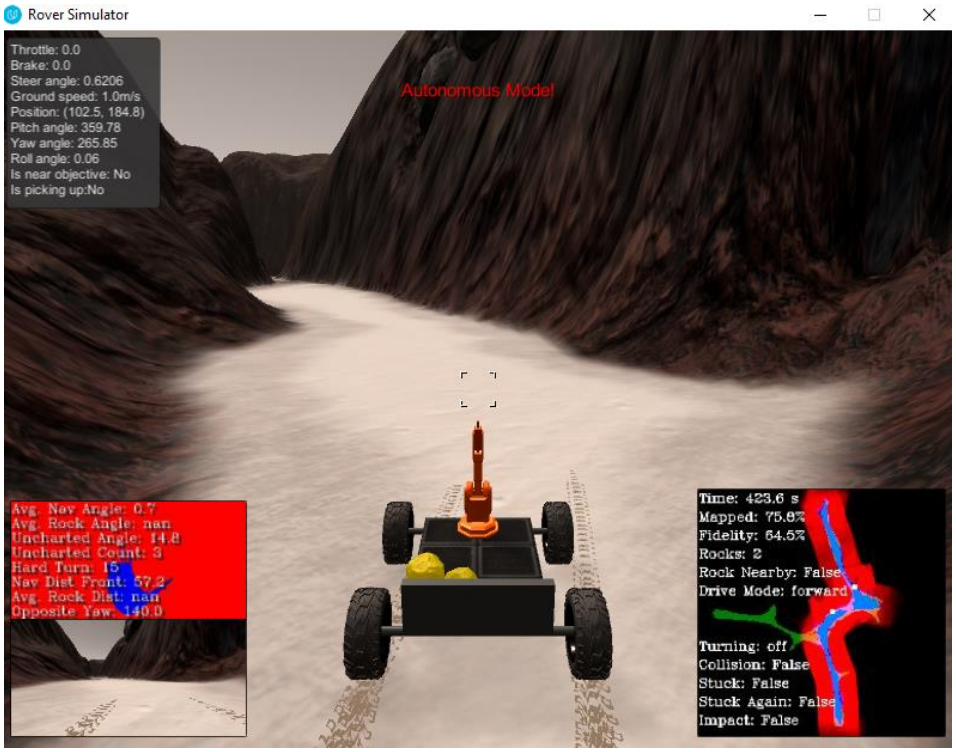
The Rover completed the project requirements and mapped 40.7% at 65.1% fidelity.



After continuing, the Rover picked up two rocks after mapping about 60.3%.



The Rover traversed all the way north and south, mapping 75.9% of the world map with 64.5% fidelity.



Maxed out the Rover mapped 90.9% at only 49.3% fidelity.