Search and Sample Return

The Search and Sample return project consists of two phases: a learning exercise and experimentation environment in a Jupyter notebook (Rover_Project_Test_Notebook.ipynb) and a set of scripts which allow the RoverSim to operate in autonomous mode.

The project is available in github at https://github.com/carlosrodriguez6/RoboND-Rover-Project.git

Rubric 1: Writeup

This document as a whole addresses point number 1 in the project rubric, providing an explanation of how each rubric item has been addressed.

Rubric 2: Notebook Analysis

The project notebook builds on the earlier exercises to build a 'perception' pipeline that allows the rover to create a map of real world coordinates as it travels the simulator's 3D environment.

The notebook was initially run without modifications using sample images and data included with the original git project. Then it was modified to allow for color selection of obstacles and rock samples. Lastly a "training" run was made in the simulator and images from this run were used as inputs for predefined and modified functions.

Rubric 2.1 Image processing functions and rock and obstacle identification

This section addresses part 1 of the Notebook analysis rubric.

Imports and setup

The first two or three cells of the notebook simply shows how to load an image into `matplotlib.image` and render it to screen using `plt.imshow()`, and thus I won't go into detail here, except to mention that in order to make the calibration cell useful I had to add the `%matplotlib notebook` directive right after the matplotlib import statements.

Calibration images

The next code cell is provided to show calibration images. These allow to:

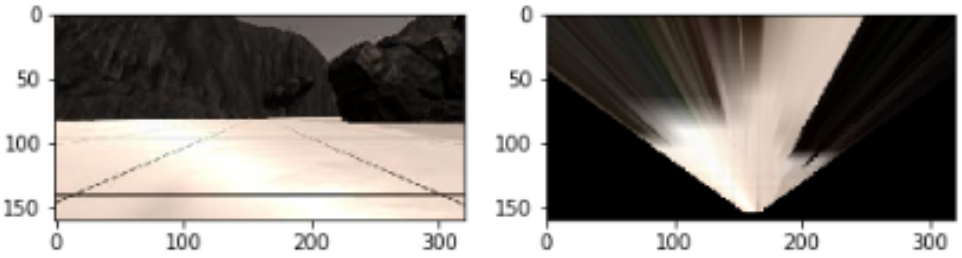
- Define the source and destination frames of a perspective transform, and
- Determine the color thresholds that define pixels belonging to a rock.

Perspective Transform

For the initial iteration the values obtained by evaluating the x-y positions of the corners of the sample grid square were consistent with the prepopulated values in the perspective transform cell, so those were left unmodified.

The following image shows the pre-packed sample grid image against the warped version:

Fig. 1 Sample grid image before and after perspective transform.



Color Thresholds

The Color Thresholding cell provides a function with lower RGB bound to find terrain and output the results of the operation to a binary image. For the initial iteration, the default values for the RGB threshold appeared adequate and were left unmodified.

Fig 2. “warped” image before and after threshold operation to identify terrain, sample data

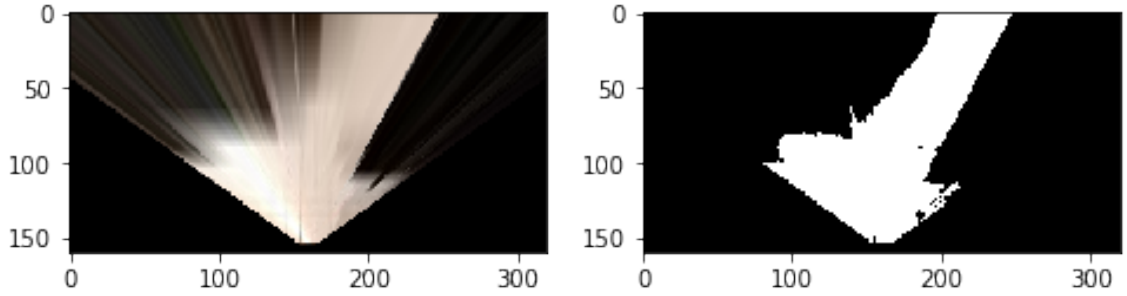
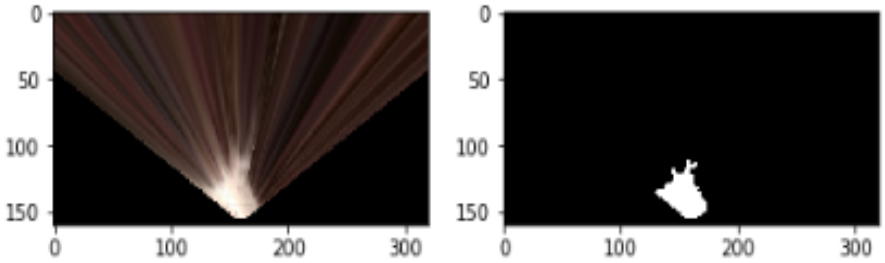
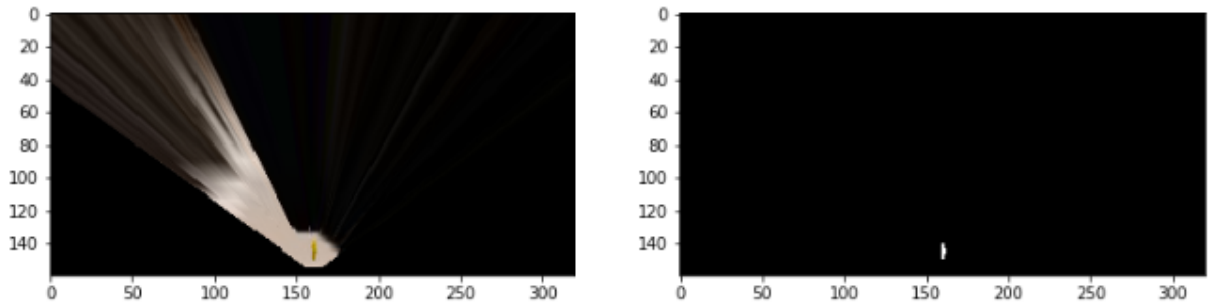


Fig 3. Using a randomized sample from training run



For rocks, I decided to go with the recommendation in the walkthrough and simply used the find_rocks() function, which is very specific in that it selects for *above-threshold* values in the Red and Green channels but selects for *below-threshold* values in the Blue. I would have preferred to come up with a more general threshold function that took upper and lower thresholds but found that this gave me too many false positives when looking for terrain, so I decided to have the two different functions for the sake of simplicity.

Fig. 4 The warped image before and after running through find_rocks()



For obstacle color selection I followed the recommendation in the notebook and simply selected the opposite of the 'ground' pixels:

```
threshold_obstacles = np.absolute(np.float32(threshed) - 1)
```

Coordinate Transforms

The cell for coordinate transformations did not require any changes. This cell defines a conversion to a rover centric coordinate system for the pixel positions of the threshold image, as well as a function to convert these positions to polar coordinates centered on the rover. I took the liberty of adding a text cell to the notebook explaining the transform from "image" space to the prescribed rover-centric space. The following figure shows the warp and threshold operations for terrain, as well as a final conversion to coordinates centered on the rover, overlaid with the average of the polar angles from the rover to all the pixels in that terrain sample:

Fig 5. Coordinate transforms culminating on polar cords centered on the rover, using sample data.

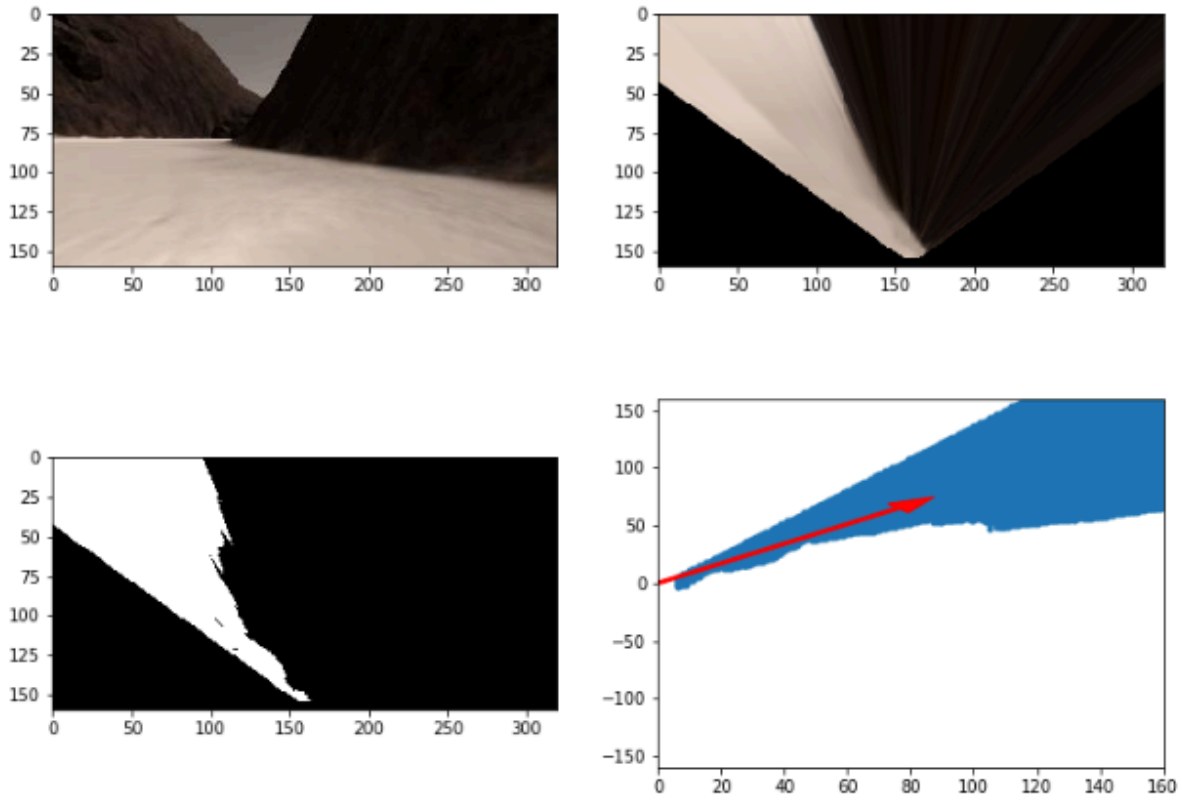
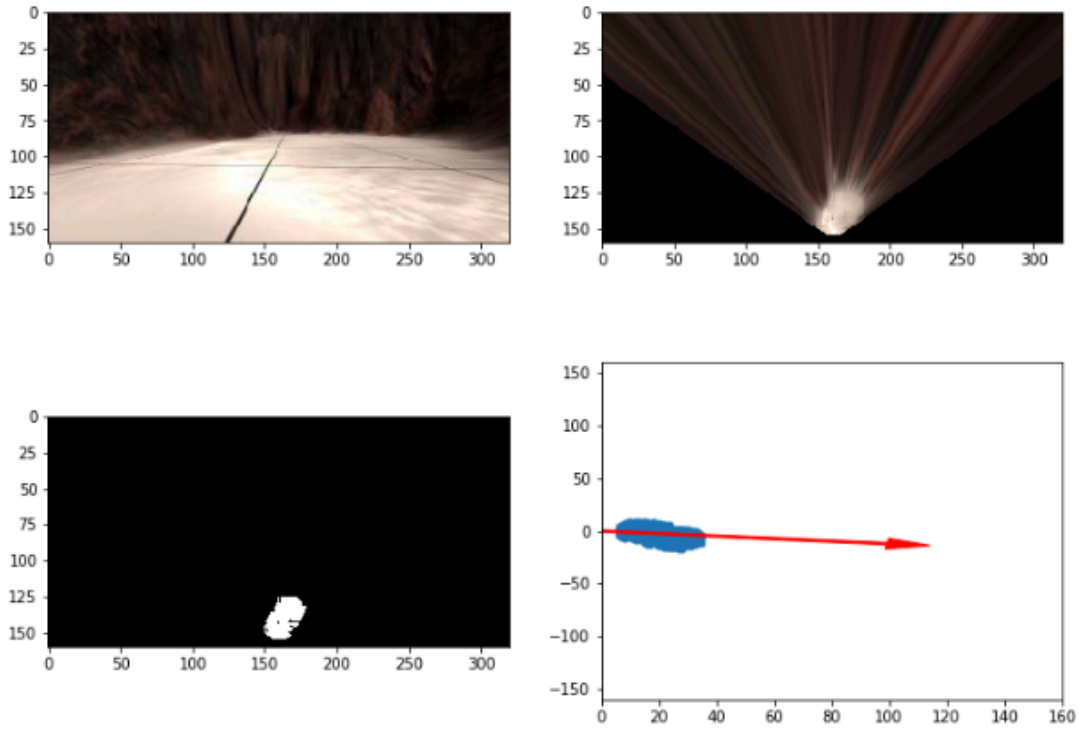


Fig 6. Coordinate transforms culminating on polar cords centered on the rover, using training data



Rubric 2.2: The process_image loop

This section addresses part 2 of the Notebook analysis rubric.

The next cell sets up the data structures needed to complete the second part of the notebook assignment, which uses the .csv log and the images captured during the sample and training runs to repeatedly call the process_image() function. At the end of these iterations, a reasonable map of the terrain traversed during the runs should be achieved, including any rocks found along the way, as well as a rough map of the obstacle features. Since the only change made to this cell was changing the file paths for the log files and the IMG directory I won't go into detail here.

The process_image() function is meant to leverage the previously defined functions in the notebook to perform a perception or mapping step for each image captured by the rover's camera. In addition to filling in the necessary code to fulfill the requirements of the assignment, I added some functionality to make it easier to visualize what was going on in each step. In the following summary all line numbers refer to the process_image notebook cell.

Firstly, I increased the size of the mosaic image for each frame (line 85):

```
output_image = np.zeros((img.shape[0] + data.worldmap.shape[0], img.shape[1]*3, 3))
```

I wanted to show the original, warped and thresholded terrain image on the first row so I wrote a helper function to convert the binary image to RGB (binary_to_RGB in the notebook) so I could add it to the mosaic, then I modified the output image statements:

```
# Put the original image in the upper left hand corner
output_image[0:img.shape[0], 0:img.shape[1]] = img

# Add the warped image next to it
output_image[0:img.shape[0], img.shape[1]:2*img.shape[1]] = warped
# convert the thresholded image for the terrain to RGB so we can show it in the
mosaic
# and put it in the first row, third column
output_image[0:img.shape[0], 2*img.shape[1]:] = binary_to_RGB(threshed)
```

After applying the color_thresh: and findRocks: functions to the warped image, I stitched the threshold image for the rocks in the second row, next to the ground truth map.

I obtained the obstacle threshold image by simply negating the terrain results, but I decided to not display them separately and instead concentrated on the changes needed to populate the map. Firstly I converted the threshold images to rover coordinates (lines 46 - 51).

```
xpix, ypix = rover_coords(threshed)
xpix_obs,ypix_obs = rover_coords(threshold_obstacles)
# if we actually found any rock-like pixels
if yellow_threshed.any():
        xpix_rock,ypix_rock = rover_coords(yellow_threshed)
```

The rover coordinate arrays were converted to world coordinates (lines 58 - 62).

```
terrain_x_world, terrain_y_world = pix_to_world(xpix, ypix, xpos, ypos, yaw, 200, 10)
obs_x_world, obs_y_world = pix_to_world(xpix_obs, ypix_obs, xpos, ypos, yaw, 200, 10)

if yellow_threshed.any():
   rock_x_world, rock_y_world = pix_to_world(xpix_rock,ypix_rock,xpos,ypos,yaw, 200, 10)
```

The worldmap array was set up so that the red channel would show obstacles, the blue channel terrain and the green channel the rocks. However I found it made more sense to just make the rocks white wherever they were found (lines 76 – 79)

```
        # update the map with rocks if any
        if yellow_threshed.any():
              # these are heavy rocks which exist in ALL chanels.
              data.worldmap[rock_y_world, rock_x_world, :] = 255
```

After running the "driver" cell and the video rendering cell, the results for the run using the canned data can be found in output/test_mapping.mp. The results for the run using the data from the training run can be found in output/training_run_mapping.mp4.

The following figures show screenshots of the final frame for both runs.

Fig 7. Using example data.

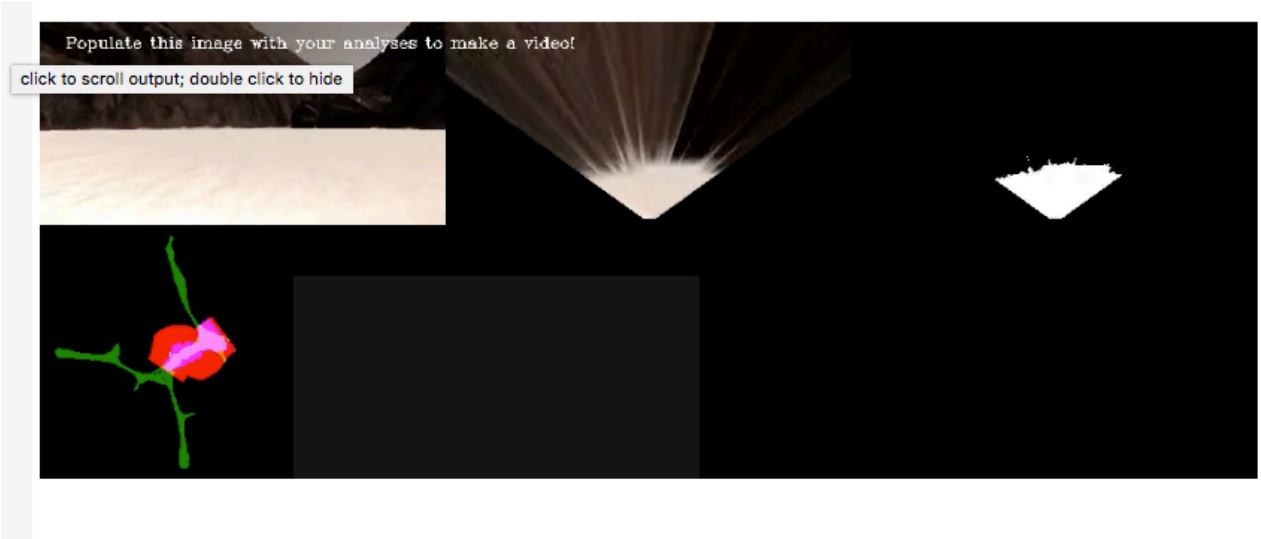
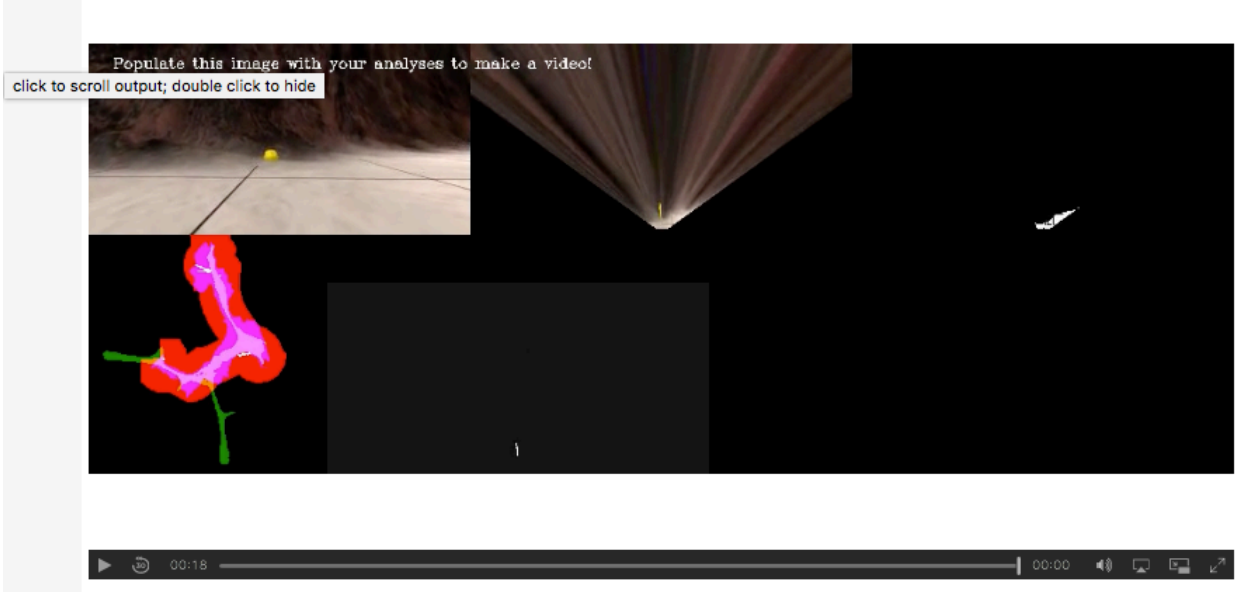


Fig 8. Using data from manual training run.



Rubric 3: Autonomous navigation and mapping.

Rubric 3.1: Modifications to perception_step() and decision_step().

This section addresses part 1 of the Autonomous Navigation and Mapping rubric.

The perception_step() pipeline.

Although not explicitly stated, perception_step() is essentially a pipeline that receives images from the rover's front camera and, with the help of other telemetry values, eventually derives navigable terrain, rock sample and obstacle coordinates in various coordinate systems suitable to make decisions about the rover controls and mapping the rover's environment. It receives what the rover "sees" and other telemetry as part of a Rover object reference that gets passed to it. In turn, it performs its work and updates the rover object's coordinate, image, mapping and telemetry values.

The perception_step() pipeline calls several methods that are essentially identical to the ones defined in Rubric 2.

- A perspective transform to go from the camera pixels to a top down view of the same.
- Several color threshold operations to identify terrain, rock sample and obstacle pixels and produce binary arrays of 'true' pixel coordinates.
- A transform to go from the top down image coordinate space to one centered on the rover. This is applied to the threshold arrays to obtain the coordinates of terrain, obstacles and rock samples relative to the rover.
- A transform to go from the rover-centric pixel coordinates to world coordinates. These are used to update the rover's real world map of terrain, obstacles and samples.

- A polar coordinate transform to go from rover-centric coordinate space to angles and distances centered on the rover. The angles are necessary because we can only send steering angle and throttle commands to the rover, and in order to decide what these should be we need to consider the terrain, obstacles and sample positions in terms of their relative angles to the rover.

The perception_step() function and some of the functions it calls were modified as follows:

- A *find_rocks()* function was added to act as a thresholder for the specific case of the yellow rocks. It is idiosyncratic in that it selects for red and green values *above* a threshold but blue values *below* a threshold.
- The perspect_transform() function was modified to return a warped image and a *mask* in the shape of the bounds of the warped image

The perception_step function was filled out by:

- Retrieving the "camera" image from the Rover object that is passed to the function.
- Defining source and destination points for the perspective_transform() function to calculate the appropriate warping transform to use on the image, and applying that function to the rover's image. The values are obtained in the same manner as they were for the Notebook section: the source rectangle corners are the pixel positions of a grid square located directly in front of the rover and exactly centered on the rover's field of view. The destination rectangle is derived from the dimensions of the image itself, with the assumption that each pixel is .1 m and allowing for the image to be actually slightly ahead of the rover's center.
- Applying the color_thresh and find_rocks functions to the warped image to obtain the corresponding binary images. For the sake of simplicity, the obstacles binary image is defined as the inverse of the terrain image, multiplied by the mask obtained from the perspective transform. This multiplication is necessary because, without it, every pixel in the warped image not within the rover's field of view would be considered an obstacle.
- Updating the rover's vision_image with the obstacle, rock sample and terrain values. Each of the binary images is assigned to the red, green and blue channel respectively. This composite is the image that is shown on the RoverSim's UI right above the Rover's "camera" image.
- Converting the threshold images to rover-centric coordinates, and then converting these to world coordinates so that they can be inserted into the Rover's worldmap. While doing this, we give "terrain" positions a higher value in the blue channel than we do "obstacle" positions in the red channel, so that, when they overlap, the create_output_images function can break the tie. This is set up by default in the project.
- Converting rover-centric "terrain" positions to polar coordinates, which is the simplest way to present to decision_step() a metric that it can use to decide upon steering commands.
- Updating the rover with the terrain polar coordinates.

Decision_step() functions sufficiently well with no further modifications, given the improvements made to perception_step() to more finely tune the discovery of terrain and the accuracy of the mapping. It sets up a simple decision tree that can be summarized as follows:

1. If moving forward is possible, hit the throttle, steering angle is the mean of the terrain pixels ahead.

2. If moving forward is not possible*, stop, turn in place until moving forward is possible, then proceed as above.

This decision tree suffices to accomplish the minimal requirements of the project (find at least one rock, map at least 40% of the environment with at least 60% fidelity). However, several improvements that were considered, but not implemented in the interest of time include:

- Detecting when the pitch or yaw of the rover exceeded certain thresholds, and ignoring those frames for mapping purposes. For example, when the rover stops abruptly, it tends to lower its front and the perspective view becomes invalid.
- Using a more sophisticated criterion to decide the steering angle, such as limiting the angles array on the rover to those corresponding to distances above a threshold. Or detecting when the rover appears to be in a straightway inside a canyon by the maxima and minima of the terrain angles array, and accelerating.
- Finding a way of "closing" areas of the map and marking them done.

Rubric 3.2: Running drive_rover.py against the simulator.

This section addresses part 2 of the Autonomous Navigation and Mapping rubric.

Running the provided project files with `python drive_rover.py` achieves an average of 12 FPS when running the Rover Simulator at 1024x768 on "Good" graphics quality. This was on a Mid 2010 MacBook Pro with a 2.4 GHz Intel Core 2 Duo CPU with 16GB of RAM and a moderately fast SSD.

Several runs were made to ensure that at least 40% coverage was achieved with 60%> fidelity and one or more rocks were found. This established the suitability of the existing code for submission.

* This is a harder question to answer than it sounds, as the rover sometimes got stuck in rocks and could not detect that it couldn't go forward either because it thought there was enough terrain in front of it, or because it couldn't detect that it was stopped (velocity kept fluctuating erratically as the rover collided with and sometimes *through* the rock)

Fig. 9 Screenshot from successful run.