Trajectory Visualisation

Ingmar Ludwig
University of Bremen
Bremen, Germany
iludwig@informatik.uni-bremen.de

***Abstract*—One important functionality of the TraVis simulation is the ability to first show the AUVs trajectory in 3D and second to project color coded AUV status data on the trajectory for different benefits. In this paper the process of creating the three dimensional trajectory and using it for projecting AUV status data is described and examples for the projections and possible uses are given.**

***Index Terms*—unreal engine, unreal, trajectory, travis, visualisation**

I. Introduction

The documentation of the TraVis simulation is divided in several parts. In this part the visualization of the trajectory is explained.

One of the key features of the TraVis simulation software is the ability to show the AUVs trajectory in 3D as shown in figure 1. This visualization enables the user to get an easy overview over the AUVs movement in space especially in regards to the other parts of the simulation like the landscape.

The trajectory is drawn using three-dimensional spheres which collectively form a dotted line from the missions start point to the endpoint. This mechanism is further utilized to make collected data overseeable within the simulation by projecting color coded data on the spheres depending on their position. This can help to make effects visible that might not be visible otherwise or help with resolving errors e.g. by uncovering when an error first occurs.

Currently six different data sets can be projected on the visualization: rotations per minute of the motor, speed, pressure, heading angle (direction of movement), pitch (ascend or descend angle) and roll (rotation around the long axis). Examples for those data projections and possible uses are given in figure 2 to 7.

The visualization is created in the initialization phase of the Level Blueprint and then modified during runtime according to the user inputs (for further information on the initialization phase refer the Lifecycle-paper). In this paper the process of the the creation of the visible trajectory and the data projection is described.

II. Creation of the visualization

The visualization of the trajectory consist of a dotted line which is realized using generated spheres. This is done in the *draw_trajectory* method, which is called in the initialization phase of the level blueprint after the connection to the DataCommunicator object is established through the *StartCommunication* method. The DataCommunicator enables accessing the AUV mission data using so called Waypoints with one waypoint representing one point in time at which the AUV recorded its status (or, in other words, one line in the AUV log file). It is necessary to keep this order of method calling, because the *draw_trajectory* method uses the *DataCommunicators* Waypoints internally.

After the *draw_trajectory* method finished, the mission time needs to be reset through the *reset time to mission start* method, otherwise the simulation will not work correctly.

The *draw_trajectory* method has two parameters: *distance_between_objects*, which determines the distance between the spheres, and *use_every_nth_value*, which determines how much Waypoints are left out between two connected Waypoints while drawing the trajectory. For example if *use_every_nth_value* is 2, the first and third waypoint is connected, then the third and the fifth and so forth. This parameter is introduced for increasing drawing speed with very close waypoints. If for example the maximal distance between two Waypoints is 0.1 meters and the distance between two spheres is 20 meters it is possible to use only every 10th waypoint without risking visible deviations while increasing the drawing speed significantly.

The method itself has two main responsibilities: acquiring and checking the waypoints and connecting the two waypoints with dots.

The waypoints are acquired using the method *get_next_point_from_index* of the *DataCommunicator*. This method returns true if there is a Waypoint in the data with this index and false if not. While this method returns true, *draw_trajectory* takes every nth Waypoint and connects it with spheres as a dotted line.

Since the method might not use every waypoint due to the *use_every_nth_value* parameter, it is possible that some waypoints at the end might be left out. If for example n is 10 and there are 102 values, the last two values are not used. Since there are 500,000+ waypoints in the current context, this is currently not a problem.

Every Waypoint is checked in two ways: First, it is checked weather there are Waypoints with this index at all, using the return value of the *get_next_point_from_index* method. If there are no Waypoints with this index the method finishes (since there are no ”holes” in the waypoints, this means the end of the Waypoints is reached). Secondly, it is checked if both Waypoints have a valid entry. If there was an error while

reading, the entry is x=0, y=0, z=0. In this case the false Waypoint is skipped. Since the position x=0, y=0, z=0 is not in the region of the trajectory, it is no problem to skip these Waypoints.

In the next step, the two positions from the Waypoints are connected with spheres, which happens in its own method *spawn_objects_from_vec_A_vec_B* which connects two positions in unreal coordinates. The procedure in this method is described further below.

Since it is usually the case that the distance between the two positions differs from the distance defined in *distance_between_objects*, it is necessary to consider the distance between the last spawned sphere and the end of the last segment. For example if the distance between the last spawned sphere and the end position in the previous iteration was 10 and the distance defined in *distance_between_objects* is 20, the distance between the start position and the first spawned sphere in the current iteration should be 10. Since it is often the case that segments are shorter than the distance defined in *distance_between_objects*, it is often necessary to add the distance up over several iterations without any spheres before the next sphere is added.

III. Connecting two positions

In this method the actual placement of the spheres takes place, creating a continuous line of equally distanced spheres along the trajectory after it was called for all segments (with one segment being the line between two waypoints).

First it is checked whether the segment should contain a sphere or not. This is done using the distance since the last spawn from the last segment and the length of the current segment: If the sum of both values is longer than the distance between two spheres, at least one sphere is added to this segment. If the sum is smaller, the method terminates and returns this sum.

In case at least one sphere should be spawned, the direction from the beginning of the segment to the end of the segment is determined and, using linear interpolation, spheres are spawned along this direction until the rest length of the segment is shorter than the distance between two spheres.

The first sphere is spawned at *distance_between_spheres* - *distance_since_last_spawn* units. For the rest of the segment all *distance_between_spheres* units a sphere is spawned. Usually this should not happen since a segment is usually a lot shorter then the distance between two spheres. Since at least one time in our test data an error occurred in the data recording process leading to a longer period without data, it is necessary to be able to deal with longer segments.

After each spawn a reference to each spawned sphere is saved in an array for color modification during runtime. Together with the reference the index of the closest Waypoint is stored. This is necessary for accessing the data that the AUV had at the position of each sphere for later setting the colors of the spheres accordingly.

IV. Data Visualization on Trajectory

The Trajectory is initially drawn using the single color red for all spheres. When the user presses a button to project data on the trajectory, the color of each sphere is changed to represent the selected data at the position of the sphere. If the user for example selects the pressure, the color of the spheres are set to represent the pressure at the position of each sphere.

The method is called with one parameter, *visualized_data*, a string containing the name of the data that should be visualized. Local variables are then set accordingly within the method using a switch case. This also means that all maximal values for the different data for the visualization on the trajectory are kept hard coded in this method.

At the beginning of the method the current simulation time needs to be saved for restoring the current simulation status at the end of the method since the simulation time is distorted by the *get_next _waypoint_from_index method*.

Then for every sphere in the trajectory (which are held in one big array) the color is changed. Using the maximal value of the data and the data value at the position of the sphere, the right color for the sphere is determined using the separate method *calculate_color_from_value*. Since the visualization should be similar for all kinds of values and universally applicable, the mapping is kept simple: Using the whole RGB-color-spectrum, the color is determined by using the linear position of the value in regards to 0 and the maximal value, going from blue for the lowest values over green to red for the highest values. If for example the max value is 10 and the current value is 5, the sphere will get the color green. In case the current value is 0, the color would be blue, for 7.5 it would be yellow and for 10 it would be red.

Finally in the *set_color_spawned_object* method the color is set to the sphere. This is done using a dynamic material instance which is created for every sphere and then set as the new material in the method.

V. Appendix

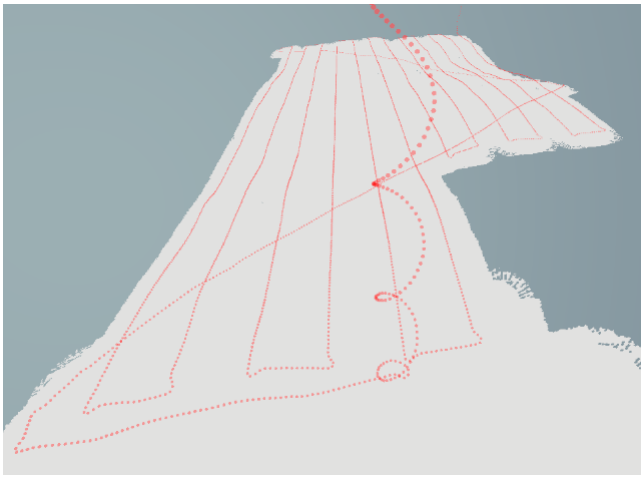

Fig. 1. The AUVs trajectory without projected data as initially shown after simulation startup.

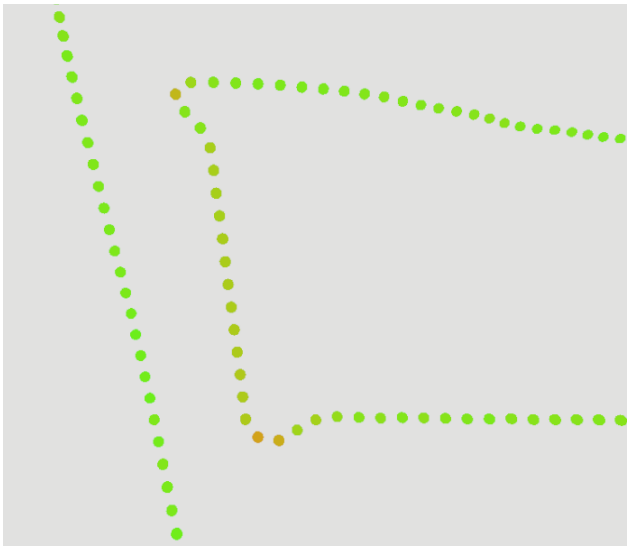

Fig. 2. The AUVs motors speed of rotation in rotations per minute projected on the trajectory. In the picture the effect of cornering on the motor rpm (and through this on the energy consumption) becomes visible: After each turn the AUV needs to speed up again and does this by increasing its motor rpm.

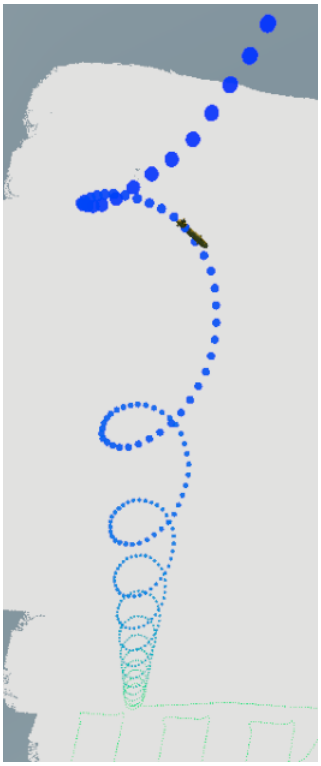

Fig. 3. The pressure on the AUV projected on the Trajectory.The pressure visualization can be used to examine whether the AUV (which is usually using altitude over seafloor for its controls and not depth) is operating within its pressure limits the whole mission.

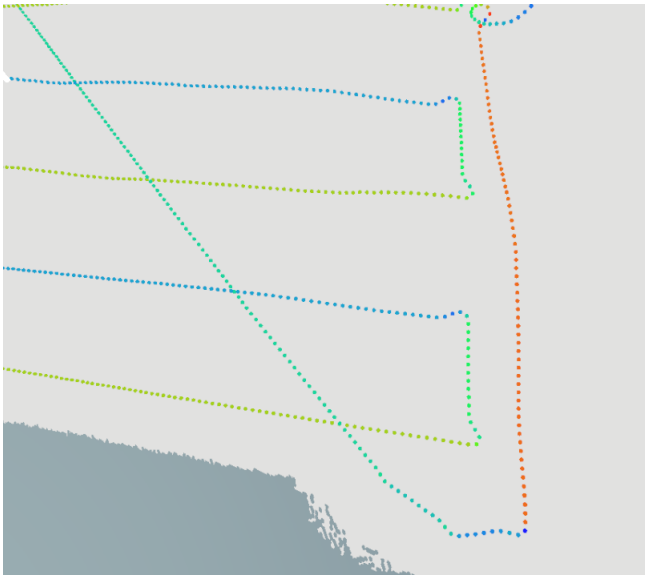

Fig. 4. The heading angle (moving direction of the AUV) projected on the trajectory. Using the projection of the heading angle on the trajectory can help to clarify the trajectories visualization by showing the direction the AUV was moving through a complex three-dimensional path.

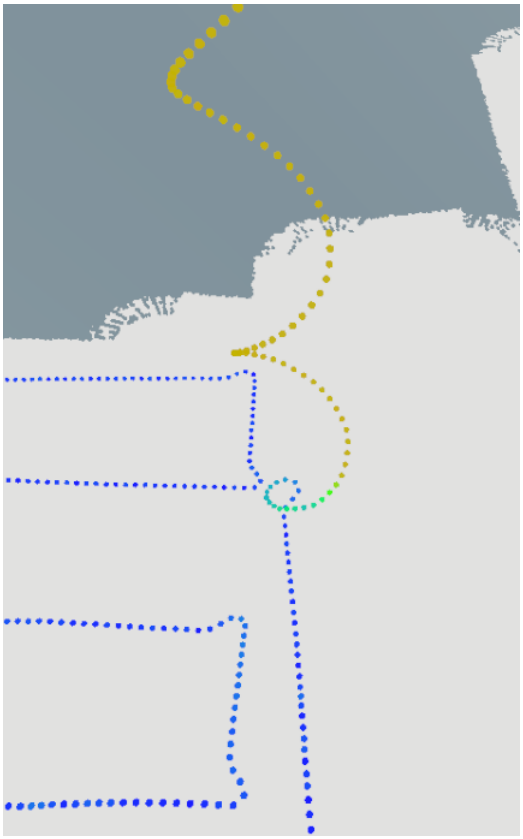

Fig. 5. The AUVs pitch angle projected on the trajectory. The pitch visualization can be utilized to check whether the pitch was correct during the recording of the bathymetry: During recording the AUV should be as horizontal as possible to prevent contortions, only when not recording (e.g. while descending) a large pitch is acceptable.

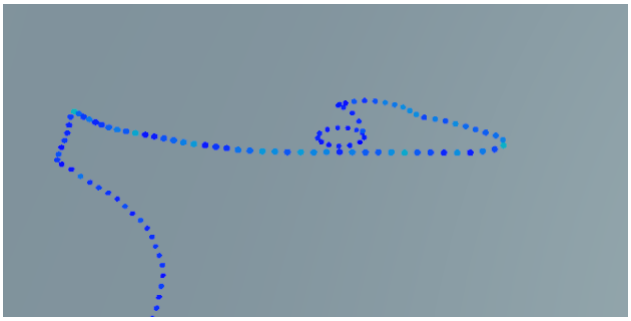

Fig. 6. The AUVs roll angle projected on the trajectory. To ensure proper functionality of the AUVs instruments the AUV is kept upright around the roll axis the whole mission. This means a large roll angle is a strong indicator for severe malfunction. Only at the sea surface some roll might occur due to waves, as shown in the picture.

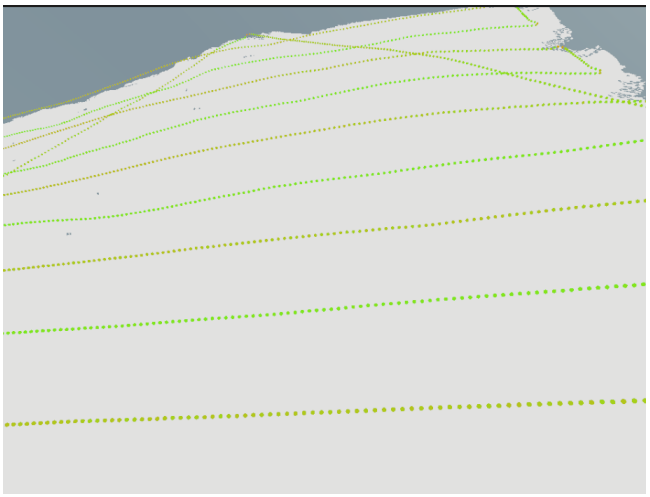

Fig. 7. The AUVs speed projected on the trajectory. The picture shows the effect of ascends on the AUVs speed: The AUV moves back and forth over the ascending seafloor loosing speed in the one direction and gaining speed in the other due to the ascending/descending nature of the seafloor.