

# Tools for Visualizing, Annotating and Storing Triangle Meshes in ROS and RViz

Sebastian Pütz<sup>1</sup>, Thomas Wiemann<sup>1</sup>, Joachim Hertzberg<sup>1,2</sup>

<sup>1</sup>University of Osnabrück, Knowledge Based Systems Group, Osnabrück, Germany

Email: spuetz—twiemann—joachim.hertzberg@uni-osnabrueck.de

<sup>2</sup>German Research Center for Artificial Intelligence, Research Unit Plan-based Robot Control, Osnabrück, Germany

Email: joachim.hertzberg@dfki.de

**Abstract**—Polygonal maps for robotic applications are becoming increasingly popular, but are currently not effectively supported in the Robot Operating System (ROS). In this paper, we introduce the *Mesh Tools* package consisting of message definitions, RViz plugins and a persistence layer to make the benefits of annotated polygonal maps available in ROS. These tools allow to publish, edit and inspect such maps within the existing ROS context. Our persistence layer efficiently loads and stores large mesh maps. We discuss two application areas as a proof-of-concept: Labeling of triangle clusters for semantic mapping and robot navigation on triangle meshes in a typical outdoor environment by integrating our tools into an existing navigation stack.

## I. INTRODUCTION

Recently, the rapid enhancement in 3D sensor technology has led to the development of effective 3D mapping algorithms. Matching 3D point clouds or RGB-D data is the basis for state of the art SLAM algorithms to generate high resolution point clouds of large environments in short time. Using point clouds for robotic purposes other than scan matching is seldom seen, as they have several drawbacks like missing topology between points, presence of noise and no continuous surface representation. These drawbacks can be overcome by using the raw sensor data to derive other representations like octrees that can be used as robot maps. Such octrees - although easy to compute and relatively memory efficient - deliver only a discrete, voxel-based representation. Regarding continuous representations, polygonal meshes are state of the art. With the introduction of efficient polygonalization algorithms like Kinect Fusion or our own Las Vegas Reconstruction Toolkit [14], automatic generation of such maps from point cloud data has become feasible for real life applications.

ROS is currently the de-facto standard framework for development of state of the art robotic software. It is used by a number of universities and companies and provides a complete infrastructure for different applications including mapping, path planning and visualization. In recent years, some new map representations and approaches were designed and developed along with corresponding ROS packages. Most of these state-of-the art solutions focus on 2D scenarios. In this paper, we add the missing ROS infrastructure to make 3D meshes available as 3D environment representations within ROS. This includes the definition of new messages to represent

semantically annotated 3D meshes, plugins for RViz to render such meshes – including textures – and a new file format that supports efficient storage of such meshes. An example for using RViz to visualize a point cloud together with an annotated mesh is shown in Fig. 1.

## II. RELATED WORK

For typical flat environments, the *Costmap 2D* has proven to be a reliable representation for robot navigation in 2D [9]. A *Costmap 2D* stores values as arrays of unsigned characters. Occupancy cells are usually expressed by the respective values for *Occupied*, *Free*, and *Unknown*. Due to the 1-byte restriction per cell, the expressiveness of such maps is limited. These simple occupancy maps have been enhanced by layered cost maps [8], which allow to encode more detailed information than just occupancy.

A straight-forward extension of such maps are so-called 2.5D grid maps, which also encode local height with respect to a given reference level. The *Grid Map* library overcomes the *Costmap 2D* shortcomings by storing values in a stack of matrices of float values. The *Grid Map* package comes along with visualization plugins for RViz and message definitions and allow easy conversion to OpenCV images [3].

These pseudo-3D representations inherit the benefits of 2D occupancy grid maps, e.g. structure and constant access time, but they also allow to express heights, variances, curvature etc. In addition to this benefits the *Grid Map* library uses a 2D ring buffer to efficiently move data in an azimuthal centric fashion around the robot when it moves to reduce the memory footprint. The main drawback of such grid-based maps is that they can not model multi level environments, e.g., multi-story buildings, stairwells, underpasses, bridges, or overhanging structures. Furthermore, information gets lost if 3D is projected onto 2D.

To model environments in full 3D, octree representations are currently state of the art. In ROS, the most popular octree representation is *OctoMap* [7], which also provides means to visualize 3D voxel maps. *OctoMap* represents a map as 3D octree, where voxels (the octree leafs) indicate the corresponding space as occupied, whereas the space for non existing sub-trees or voxels is understood as free space. Octrees have several strengths, e.g., dynamic expansion, level

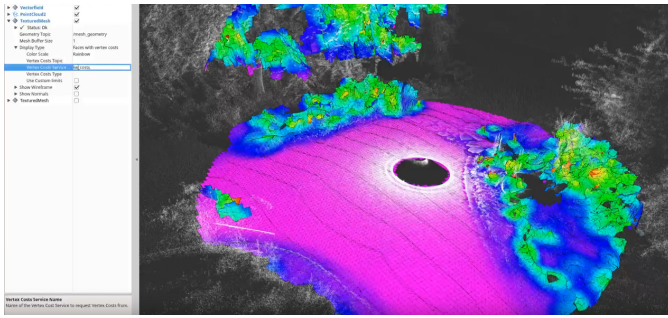


Fig. 1: RViz with loaded plugin and 3D mesh map

of detail support via definition of maximum octree levels, efficient querying and evaluation of regions in terms of free or occupied. However, octrees are discretized and they implicitly do not model a continuous surface.

The *SkiMap* approach [6] tries to optimize the access time of octrees by proposing a skip list approach, which maintains such lists in a three layer tree, where each layer represents one dimension of 3D space. Each tree node stores a list of sorted linked skip lists, which can be accessed in a binary search fashion. As the other approaches, *SkiMap* is integrated with ROS nodes and provides a ready to use ROS interface.

Making annotations and adding extra information to a 3D map is often done by using interactive markers [5] or adding an extra layer. These markers are usually not part of the internal representation, but are added as an additional overlay with added information to the chosen map format. Towards more general solutions in the context of semantic mapping, this fact limits their usability as integral part of the map. For a general semantic mapping approach, these annotations should be part of the map itself. To represent arbitrary geometries with semantic annotations, a 3D spatial representation has to be defined that serves as a basis for grounding object classes and instances geometrically. It should be able to incorporate information from different modalities consistently into the 3D representation and be scalable to represent large environments efficiently. In previous work, we have shown that it is feasible to generate such representations from 3D point cloud data in short time [14]. Additionally, these full 3D polygonal maps allow to generate 2D or 2.5D maps and semantic knowledge can be integrated in to the projection, e.g. for navigational planning [2].

In this paper, we introduce the infrastructure to make the benefits of such maps available within ROS. This infrastructure is freely available in our *Mesh Tools*<sup>1</sup> bundle. The *Mesh Tools* contain interdependent packages and tools to visualize and transform meshes and to make semantic annotations. The tools build on top of our *mesh\_msgs* message definitions for using such maps in ROS and offers a persistence layer to save and load annotated 3D meshes in HDF5 files. This 3D map file layout overcomes the ROS bag size limits and delivers a compact representation that is able to store meshes, cost layers,

metrics, and annotations in a single file.

In the following sections we present the basic structure of this package, explain the basics of the underlying message definitions and give a navigation application example. In this navigation application example we use *Move Base Flex* [10], as a general map independent flexible navigation framework which allows to use the same control sequences and well known task execution control tools like SMACH [1].

### III. MESH TOOLS FOR ROS AND RVIZ

In this section, we discuss the general structure of our *Mesh Tools* software. Details on the respective packages are given in the following sub-sections.

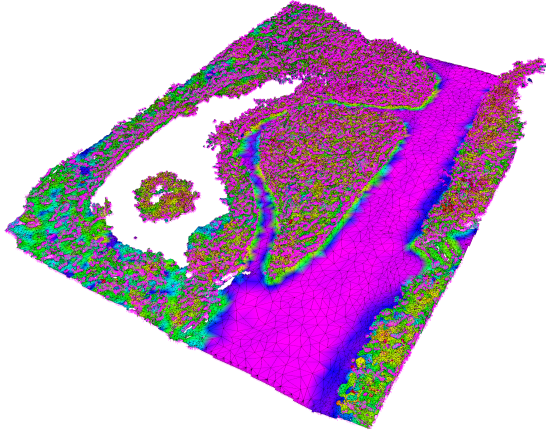
#### A. Package Structure

The messages required to send meshes and corresponding attributes, e.g., textures, vertex costs, face costs, colors or cluster labels from one node to another are defined in the sub-package *mesh\_msgs*. These message definitions are designed to be modular and aim to reduce possible overheads resulting from duplicate definitions or unnecessary re-posts of already sent messages. To transform a mesh from one coordinate frame into another with ROS' internal *tf* system, we implemented the *mesh\_msgs\_transform* package. The *label\_manager* handles the generation and conversion of annotated parts of the mesh maps. Labeled maps generated with this tools can be serialized into HDF5 files [4]. The proposed structure in this container is a direct extension of the point cloud format described in [12]. Besides the general benefits of HDF5 like lazy-loading, compression and large file support, using this general meta-format also allows to store the derived maps together with the initial sensor data. All HDF5-related functionality is implemented in the *mesh\_msgs\_hdf5* package, which reads mesh data and attributes from our HDF5 files and converts them into mesh messages for deployment in ROS. Consequently, it also stores received messages after conversion in the configured HDF5 file.

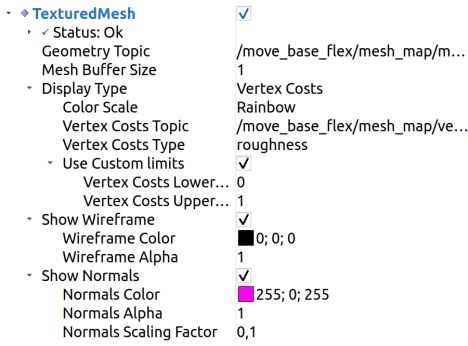
#### B. Message Definitions

The dedicated ROS messages strictly divide between a mesh's geometry, associated attributes, material definitions, and texture information. The annotations and textures are linked to the geometry via unique IDs (UUIDs). This structure enables passing the geometry without attributes to RViz and other nodes. This strict separation allows to design special nodes that can add specific attributes afterwards to support bottom-up approaches for label generation. For example, the user can design a node that computes the trafficability within a certain area based on the received geometry. After computation – which may take some time – the calculated costs can then be send to other nodes in form of attributes to the initial mesh without having to send the geometry information again. Linking of the respective data is done via the UUIDs of the mesh geometries. The layers can then be selected by the user and visualized independently by coloring the corresponding costs via pre-defined color gradients. In the

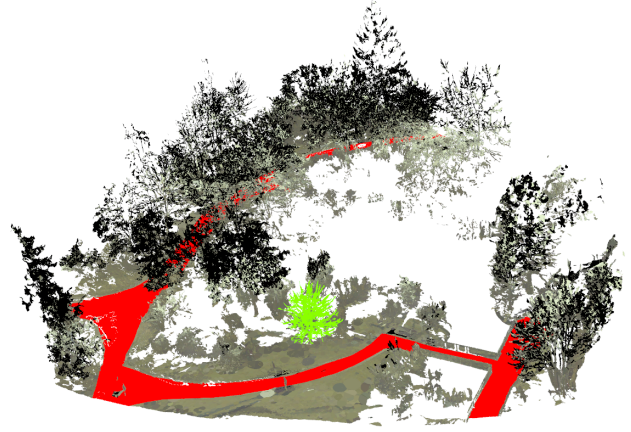
<sup>1</sup>[https://github.com/uos/mesh\\_tools](https://github.com/uos/mesh_tools)



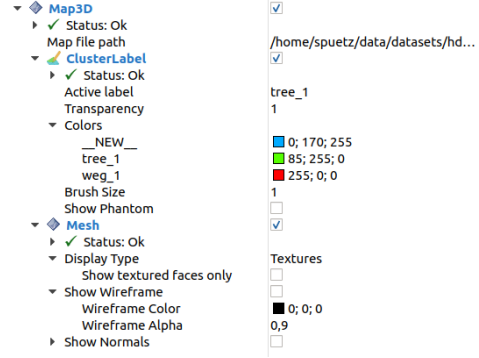
(a) RViz *TexturedMesh* plugin display



(c) *TexturedMesh* RViz plugin configuration panel



(b) RViz *Map3D* plugin display



(d) *Map3D* RViz plugin configuration panel

Fig. 2: The *Mesh Tools* RViz plugins *TexturedMesh* and *Map3D* configuration panels.

following paragraphs we give an overview on most important mesh messages:

**MeshGeometry (Stamped)** Defines the geometric structure of a triangle mesh using an array of vectors, by representing vertices in  $\mathbb{R}^3$ , corresponding normal vectors and an array of indices, where three indices each define a triangle by referring to the array of vertices (vertex and index buffer).

**MeshVertexColors (Stamped)** Colors for the vertices are represented by an array of *std\_msgs/ColorRGBA* and linked to a corresponding mesh by a string representing a UUID.

**MeshVertexCosts (Stamped)** Vertex costs are defined by an array of floats defining cost values for each vertex. A type attribute can be used to associate the costs with a meaning, e.g., roughness, variation, height differences, etc. An UUID refers the cost information to a corresponding mesh.

**MeshClusterLabel** A Cluster label message groups triangles / faces to a set or cluster by referring to their IDs. The cluster is associated to a mesh using the UUID of a corresponding mesh. An optional cluster label can be assigned to provide a semantic label or reference.

**MeshTexture** A texture message refers to an Image of the message type *sensor\_msgs/Image* and is associated with a *mesh\_msgs/MeshMaterial* using an ID.

**MeshMaterial** A material is defined by a Color

(*std\_msgs/ColorRGBA*) and an optional texture ID, referring to a *mesh\_msgs/MeshTexture*.

**MeshVertexTexCoords** Defines a texture coordinate which refers to a pixel of a corresponding image in *mesh\_msgs/MeshTexture*.

**MeshMaterials (Stamped)** Combining materials with texture coordinates and clusters of a corresponding mesh

**MeshFeature** Mesh features attributes are designed to ground images-based features in textures to a 3D position on a surface via a feature descriptor. We plan to use these for future applications for camera-based localization in textured meshes.


**MeshFeatures** List of features for a corresponding mesh.

The plugins for visualization and user-interaction are implemented using RViz's plugin API. Since RViz internally uses OGRE 3D we had to stick to this library for rendering 3D objects. Unfortunately, OGRE 3D's performance is limited for large meshes. However, we were able to implement the basic requirements within RViz with custom-extensions to realize different rendering modes, including lighting support, textured color materials and wire-frame. The supported modes are selectable based on availability in RViz's tree view.

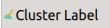
In addition to static rendering, we developed an interactive pose selection tool to allow setting poses in RViz on the mesh

surface. In this paper we used this feature to define goal poses for our path planning example. Semantic information can be added via interactive labeling in the viewer. For this, we implemented a selection tool that can either select single triangles or larger clusters of triangles via a rubber-band selection rectangle. After selection, these triangles can be grouped as a dedicated cluster that is labeled with a user-defined attribute string or in internal ID. To demonstrate the usability of this tool, we added additional screenshot and demonstration videos to the provided supplementary material to this paper.

Fig. 2c shows the *TexturedMesh* plugin panel to configure topics and visualization parameters. The *Display Type* can be configured to *Fixed Color*, *Vertex Color*, *Vertex Costs*, or *Texture*. Here, *Vertex Costs* is configured as *Display Type* and in the drop-down menu the *roughness* layer is selected as *Vertex Costs Type* which results in displaying the roughness vertex costs using the selected color scheme in the configured value limits. Some different configurations and layer cost types are shown in Fig. 2b. It shows a typical outdoor environment mesh reconstructed from a recorded point cloud from a top down viewpoint. In this example the *Mesh Tools* can efficiently display the computed metrics on the surface send to the described *TexturedMesh* RViz plugin using the described *mesh\_msgs/MeshVertexCostsStamped*. The functionality is briefly shown in <sup>2</sup>.

The *MeshGoal Tool*  provides the possibility to select a *geometry\_msgs/PoseStamped* on the surface of the mesh. In analogy to 2D maps like *Costmap 2D*, the possibility to set a goal pose on the 3D surface of the mesh is essential to interact within the recorded environment, e.g., picking a user defined goal pose for robot navigation. The pose setting is performed in two steps: First, by clicking and holding (mouse down event) somewhere on the mesh a ray face intersection is computed and the position of the intersection point is used as the 3D pose's position. Second, the normal vector of the intersecting face is used to define a plane with the support vector. By moving the cursor (mouse down) around the intersection position a arrow is defined. It is oriented in the computed plane and starts on the intersection point pointing to the current cursor position with a fixed length. This vector is then converted into an orientation quaternion with respect to the mesh's frame. The intersection position vector and the orientation quaternion then define a full 6D pose. This is published as *geometry\_msgs/PoseStamped* to a topic which can be modified in the tool properties. The interaction and pose selection is shown in in video in combination with the *TexturedMesh* plugin and a simple path planning scenario using a Dijkstra-based planning approach <sup>3</sup>.

The *Map3D* configuration panel is shown in Fig. 2d. It was loaded from a HDF5 map file that contained a mesh with its attributes and labeled faces as described in Sec. III-C). The plugin works together with the *ClusterLabel Tool* and a

*Mesh Display* and *Cluster Label Display*. The *Mesh Display* displays labeled faces and can be configured in the same way as the *TexturedMesh* plugin. Configuration can be done using the provided input widgets in the blue subgroup *Mesh* of the *Mesh3D* object. The *Mesh* subgroup configures a *ClusterLabel Display*, which displays labeled clusters. The *ClusterLabel Tool*  enables labeling of certain faces using different selection and de-selection methods. The label and its color can be configured as shown in Fig. 2d. The *ClusterLabel Panel* allows to manage the clusters and label names as well as the corresponding colors. Usage of this tool is demonstrated in detail in <sup>4</sup>.

The *Cluster Label* tool can be used to label clusters of faces with a user defined class in real time. One major problem with the previous implementation based on the native OGRE data structures was huge lag in performance when selecting a face using OGREs ray casting implementation. To address this issue, we provide a custom implementation that reduced the selection time from 10 seconds per triangle (OGRE) to 125 ms in our implementation on the presented reference mesh. Instead of re-sending the whole vertex and index buffer to the graphical card each time a face was marked or labeled, we created a data structure which persistently holds all vertex data on the graphical card. Instead of using Ogre's *ManualObject* the plugin now uses a mesh represented in our data structure.. The creation of such an object admittedly is more complex, since all data has to be set by hand in the enhanced implementation. On the other hand, this approach gives us the ability to create sub meshes of the current loaded mesh. These sub meshes refer to the same global vertex buffer that was already loaded to graphics card when the object was created. When the user interacts with the triangle mesh, only the affected index buffers need to be added or updated when faces are marked and added to a cluster. This reduces to communication overhead significantly and increases rendering performance.

The biggest performance boost however was obtained by enhancing the intersection computation between the mouse click position and the loaded mesh. Instead of checking all triangles sequentially, we implemented a bounding volume hierarchy (BVH) tree in OpenCL. This BVH structure speeds up the required ray-casting operations significantly by supporting parallel intersection computations over all faces. Due to this BVH-based pre-sorting of the vertices, we also decrease the cache miss rate, which also drastically improves performance. Using the brush and rectangle selection tool, the time for cluster generation could be dropped to 200 ms to 150 ms, which is sufficient for a seamless user experience.

### C. Persistent Storage

Besides the messages and user interaction and rendering tools described above, we store all meshes and derived clusters into a dedicated persistence layer. Here, we extend the HDF5 storage structure of our reference data set [12] to support

<sup>2</sup><https://youtu.be/ir4kZif5FS8>

<sup>3</sup>[https://youtu.be/X\\_TXC9hrAgo](https://youtu.be/X_TXC9hrAgo)

<sup>4</sup><https://youtu.be/8n4737D2abM>



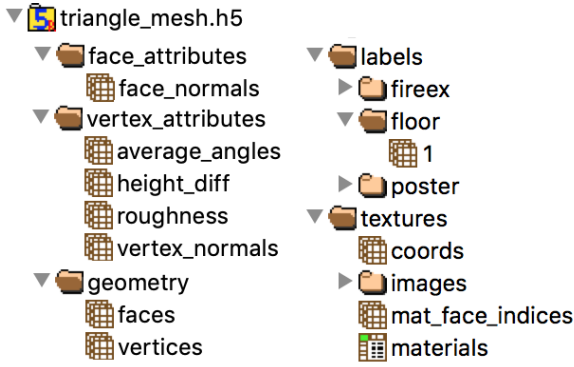


Fig. 3: HDF5 file structure to represent textured triangle meshes and labeled clusters.

storage of 3D meshes and labeled clusters. We decided to use this file format for two main reasons: First, HDF5 provides good data compression and fast access compared to other file formats. Second, the representations derived from the original input data is directly stored together with it. In principle one could use ROS bags for such purposes, but the limited default-size of 1 GB and missing compression are problematic when handling high dimensional large scale data. This HDF5 file storage is designed to hold the static environment representations. Dynamic objects have to be represented within the current ROS instance, although of course these files can be extended when new static world information, e.g., to add new maps.

For this reason, we extended the structure for high dimensional point cloud data presented in [12] to support meshes and labeled clusters. Generally, a HDF5 file is structured into groups that support sub-groups and data sets in form of multidimensional arrays. Additionally, each data set and group can also contain meta-data that describes the content of the stored information. This meta-data is indexed to allow efficient search within a HDF5 file. The structure to store meshes and annotations created with the tools presented in the previous section is shown in Fig. 3.

Within this structure, we distinguish between the mesh geometry, the associated face and vertex attributes, labeled cluster sets and texture definitions. The geometry is represented by vertex and face index arrays. Attributes for vertices are  $n \times m$  dimensional arrays, where  $n$  refers to the number of respective elements (vertices or faces) and  $m$  to the dimensionality of the associated attributes (e.g., 3 for normals and colors, or 1 for “roughness”). The “labels” collection holds a number of sub-collections that contain instances of objects with that label. Each instance is defined by an ID and an array of face indices that refer to the triangles that belong to that instance. The “textures” collection defines texture coordinates for all vertices and material indices for all triangles as well as material descriptions and all texture images in a sub-collection.

#### IV. MESH NAVIGATION APPLICATION EXAMPLE

One major focus using the *Mesh Tools* is motivated by the current development of algorithms for localization and path

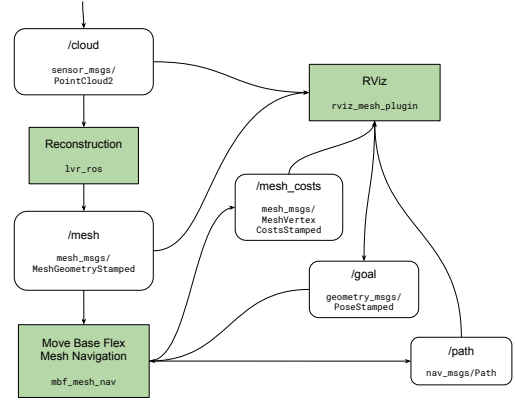
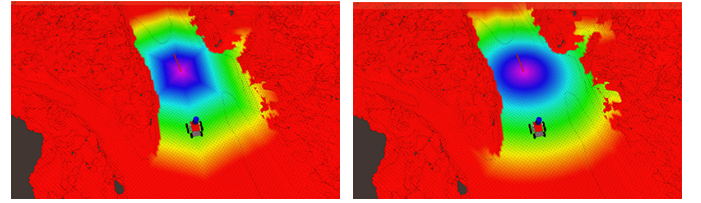


Fig. 4: Data flow in the mesh navigation application. Green boxes show the *Reconstruction*, *Move Base Flex Mesh Navigation*, and the *RViz* nodes communicating using messages shown the rounded white boxes.



(a) Potential field computed by the Dijkstra path planning plugin. (b) Potential field computed by the FMM wave front path planning plugin.

Fig. 5: Potential field comparison using the *TexturedMesh* plugin from *Mesh Tools* to compare Dijkstra path planning against FMM wave front propagation.

planning on annotated triangle meshes and recent work on semantic mapping [2]. In this context, the *Mesh Tools* are the basis to visualize the generated maps and states of the implemented algorithms in near real time. With Move Base Flex<sup>5</sup> (MBF) we have the possibility to use the same execution logic as for other map representations to perform navigation tasks, e.g., integrated in a more complex mapping scenario or simply integrated in scenarios where the robot should traverse the terrain from its current location to a goal pose selected in RViz. A MBF mesh navigation server, as well as navigation plugins and a mesh map implementation are provided in our *mesh\_navigation* bundle<sup>6</sup>. We provide a Gazebo simulation, as well as a HDF5 map file containing the map fragment shown in Fig. 5 and Fig. 6. The HDF5 file provides all trafficability layers which are then combined and used to perform robot navigation on meshes. A combination of these layers can represent the trafficable surface for a certain robot. In the following example we show how to use our *mesh\_navigation* stack together with the *mesh\_tool* with our robot *Pluto*, which

<sup>5</sup>[https://github.com/magazino/move\\_base\\_flex](https://github.com/magazino/move_base_flex)

<sup>6</sup>[https://github.com/uos/mesh\\_navigation](https://github.com/uos/mesh_navigation)

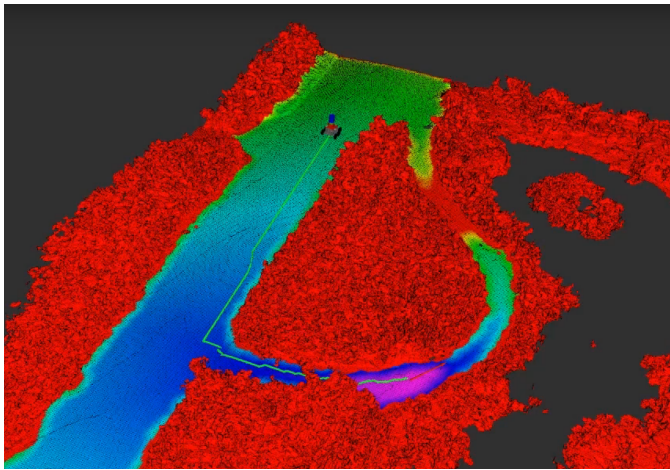


Fig. 6: A mesh with its wire-frame, and vertex costs. The wave front propagation planner computed unfolded distance values to the goal, which are then displayed as vertex costs, showing a rainbow color potential field to the goal. Red areas are marked as lethal obstacles.

is modeled in the package bundle *pluto\_robot*<sup>7</sup>. The map file, navigation setup, and the Gazebo simulation are provided in the packages *pluto\_navigation* and *pluto\_gazebo*. The trafficability layer computation and the navigation planning on meshes is described in [11]. The trafficability estimation based on local roughness, height differences (cf. Fig. 6) and lethal obstacle inflation is combined to a navigation layer. For this demonstration, we implemented two planner plugins: A Dijkstra navigation planner performs path planning by exploiting the triangles' topological connections and their costs, and a Fast Marching Method wave front propagation planner. Here, the wave front or the Dijkstra propagation starts from the navigation goal and generates a potential field from the current robot pose to the goal. The computed layers and potential field are represented as vertex costs and published using *mesh\_msgs/MeshVertexCostsStamped*. The propagation and computation of the potential field can be inspected using the *TexturedMesh* plugin by switching between the different layers in the *Vertex Cost Type* drop-down menu. Furthermore, we can introspect the wave front at a given distance by choosing corresponding cost limits in the plugin configuration. We used the *MeshGoal* plugin to set a user defined navigation goal, which is send to the executive logic and handed over to MBF and the configured path planning plugin. The plugin computes the potential field and deducts a path to the goal –if possible – as sketched in Fig. 4.

## V. CONCLUSION

In this paper we presented data structures, tools and a file format to handle and store 3D polygon meshes in ROS. The main contribution of this tool set is to offer support for textured triangle meshes within the ROS ecosystem as an addition to

existing voxel-based 3D packages. More precisely, these tools can be used to generate annotated 3D maps in the context of semantic mapping. The ability to arbitrary number of attributes to faces and vertices within the internal representation allows represent relevant environment information like roughness for path planning. The provided plugins for RViz allow to visualize the stored information and real time user interaction. The proposed HDF5 file structure allows compact storage of the obtained environment information for re-use of the generated maps. The provided infrastructure can be used to develop future algorithms for robotic applications using annotated polygonal environment maps. Currently, our tool set focuses on the representation of static models. An open problem is the integration of dynamic objects and interaction with semantic mapping frameworks like out SEMAP infrastructure [2]. The development of such tools is ongoing research. In addition, we plan to integrate our Draco-based compression tools [13] into the presented infrastructure.

## REFERENCES

- [1] J. Bohren and S. Cousins. The SMACH high-level executive. *IEEE Robotics & Automation Magazine*, 17(4):18–20, 2010.
- [2] H. Deeken, T. Wiemann, and J. Hertzberg. Grounding semantic maps in spatial databases. *Robotics and Autonomous Systems*, 105:146–165, 2018.
- [3] P. Fankhauser and M. Hutter. A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation. In Anis Koubaa, editor, *Robot Operating System (ROS) – The Complete Reference (Volume 1)*, chapter 5. Springer, 2016.
- [4] M. Folk, A. Cheng, and K. Yates. HDF5: A file format and I/O library for high performance computing applications. In *Proceedings of supercomputing*, volume 99, pages 5–33, 1999.
- [5] D. Gossow, A. Leeper, D. Hershberger, and M. Ciocarlie. Interactive Markers: 3-D User Interfaces for ROS Applications. *IEEE Robotics & Automation Magazine*, 18(4):14–15, December 2011.
- [6] D. De Gregorio and L. Di Stefano. SkiMap: An efficient mapping framework for robot navigation. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2569–2576, May 2017.
- [7] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. OctoMap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, 2013.
- [8] D. V. Lu, D. Hershberger, and W. D. Smart. Layered costmaps for context-sensitive navigation. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2014.
- [9] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige. The Office Marathon: Robust navigation in an indoor office environment. In *2010 IEEE International Conference on Robotics and Automation*, pages 300–307, May 2010.
- [10] S. Pütz and J. Simón, J. S. and Hertzberg. Move Base Flex: A Highly Flexible Navigation Framework for Mobile Robots. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2018.
- [11] S. Pütz, T. Wiemann, J. Sprickerhof, and J. Hertzberg. 3d Navigation Mesh Generation for Path Planning in Uneven Terrain. *IFAC-PapersOnLine*, 49(15):212–217, 2016. 9th IFAC Symposium on Intelligent Autonomous Vehicles IAV 2016.
- [12] T. Wiemann, F. Igelbrink, S. Puetz, and J. Hertzberg. A File Structure and Reference Data Set for High Resolution Hyperspectral 3d Point Clouds. *IFAC Symposium on Intelligent Autonomous Vehicles*, July 2019.
- [13] T. Wiemann, F. Igelbrink, S. Pütz, M. Kleine Piening, S. Schupp, S. Hinderink, J. Vana, and J. Hertzberg. Compressing ROS Sensor and Geometry Messages with Draco. *IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, September 2019.
- [14] T. Wiemann, I. Mitschke, A. Mock, and J. Hertzberg. Surface reconstruction from arbitrarily large point clouds. In *2018 Second IEEE International Conference on Robotic Computing (IRC)*, pages 278–281, Jan 2018.

<sup>7</sup>[https://github.com/uos/pluto\\_robot](https://github.com/uos/pluto_robot)