The Mesh Tools Package
Introducing Annotated 3D Triangle Maps in ROS

Sebastian Pütz*, Thomas Wiemann, Joachim Hertzberg

Osnabrück University, Knowledge Based Systems in collaboration with the German Research Center for Artificial Intelligence, Plan-based Robot Control, Osnabrück, Germany

Abstract

Triangle mesh maps for robotic applications are becoming increasingly popular, but were not yet effectively supported in the Robot Operating System (ROS). We introduce the *Mesh Tools* package consisting of message definitions, RViz plugins and tools, as well as a persistence layer to store such maps. These tools make annotated triangle maps available in ROS and allow to publish, edit and inspect such maps within the existing ROS software stack. The persistence layer efficiently loads and stores large mesh maps. The proposed plugins and tools enable the visualization and validation of the whole layered map and its properties to allow fluid interaction. We demonstrate the seamless integration of our tools in two application areas as a proof-of-concept: Labeling of triangle clusters for semantic mapping and robot navigation on triangle meshes in rough terrain outdoor environments by integrating our tools into an existing navigation stack.

Keywords: ROS, Triangle Meshes, RViz, 3D Mesh Navigation

*corresponding author

Email addresses: `spuetz@uos.de` (Sebastian Pütz), `twiemann@uos.de` (Thomas Wiemann), `joachim.hertzberg@uos.de` (Joachim Hertzberg)

URL: `kbs.informatik.uos.de/people/spuetz/` (Sebastian Pütz), `kbs.informatik.uos.de/people/twiemann/` (Thomas Wiemann), `kbs.informatik.uos.de/people/jhertzberg/` (Joachim Hertzberg)

The DFKI Niedersachsen Lab (DFKI NI) is sponsored by the Ministry of Science and Culture of Lower Saxony and the VolkswagenStiftung.

1. 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 rarely seen, as they have several drawbacks like missing topology between points, presence of noise and just sample the scanned surfaces. These drawbacks can be overcome by using the raw sensor data to derive other representations like octrees that can be used as robotic maps. Such octrees - although easy to compute and relatively memory efficient - deliver only a discrete, voxel-based representations. For compact and yet accurate representations, polygonal meshes are commonly used as piece-wise planar approximations. They are default structures in computer graphics, computational geometry and other areas, support fast rendering, collision detection and provide topology via graph-like representations like half edge meshes. With the introduction of efficient polygonalization algorithms like Kinect Fusion, our own Las Vegas Reconstruction Toolkit 2 (LVR2[2]) [1], or large scale real-time CPU based methods [2] the automatic generation of such maps from point cloud data (PCD) has become feasible for robotic applications.

Since ROS [3] is used in research and education, by companies in production and research, and in private hobby projects, more and more modular systems and methods build on top of the existing software stack, which consists of thousands of software packages that solve robotics and automation challenges. It is currently the de-facto standard framework for development of robotic software and provides a complete infrastructure for different applications including mapping, path planning and visualization. In recent years, novel map representations and approaches were designed and developed along with corresponding ROS packages, but most of them focus on 2D scenarios.

[2]https://github.com/uos/lvr2

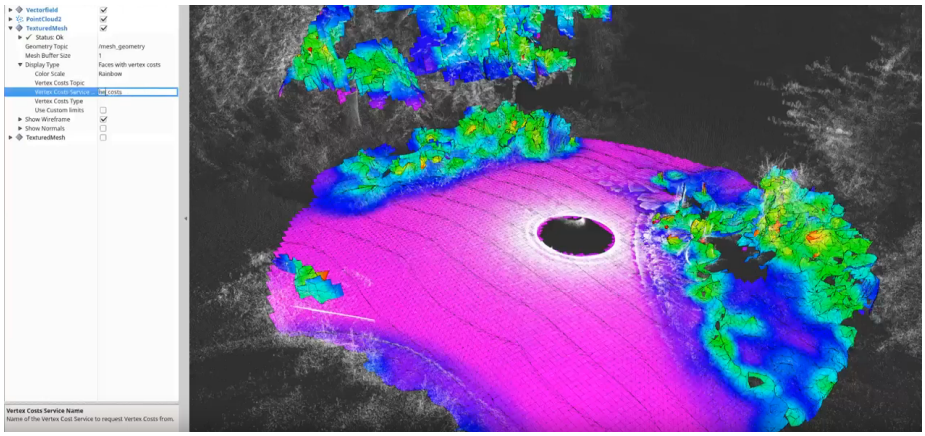


Figure 1: RViz with loaded plugin and 3D mesh map.

In this paper, we add the missing ROS infrastructure to make 3D triangle meshes available as additional 3D environment representation within ROS. We call this package the *Mesh Tools*[3], which is freely available under 3-Clause BSD license. The *Mesh Tools* supports different layers for vertex and triangle attributes to allow a modular layered view of annotations. These layers can be used to represent semantic annotations as well as cost metrics for robotic applications. Our infrastructure includes the definition of new ROS messages to encode annotated 3D meshes, plugins for RViz to interactively render and label meshes, and a new file format to efficiently store and load such meshes. An example of using RViz to visualize a 3D point cloud together with an annotated 3D triangle mesh is shown in Fig. 1.

As outlined in Fig. 2, these *Mesh Tools* consists of interdependent packages and tools for different purposes, e.g., to visualize and transform meshes or to make semantic annotations. Basis for all tools are the *mesh_msgs* message definitions that allow to use such maps in ROS. To store meshes encoded in this format, we additionally offer a persistence layer to save and load annotated 3D meshes in HDF5 files [4]. This format overcomes the ROS message size limits

[3] `https://github.com/uos/mesh_tools`

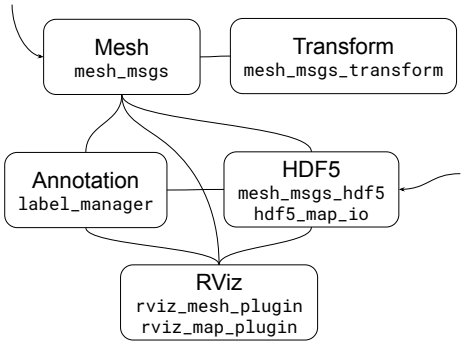


Figure 2: ROS Mesh Tools Overview: The bundle consists of the shown interacting packages to display, transform, annotate, load and store meshes and their attributes. The `mesh_msgs` and the `mesh_msgs_hdf5` and `hdf5_map_io` can be seen as data source, whereas the other packages are for interaction or visualization.

and delivers a compact representation that is able to store large meshes, cost layers, metrics, and annotations in a single file.

The focus of this paper is to present the infrastructure that is necessary to bring annotated 3D meshes into robotic applications. Rather than elaborating on the available algorithms for surface reconstruction and mesh based navigation, we focus on the message definitions and tools that are needed to ground these representations in ROS as a complement to the established 2D tools. This is indented as a first step to open up the possibility to develop novel and innovative algorithms for localization, path planning, and semantic mapping using a continuous 3D environment model. This paper is an extended version of the paper published in [5], adding additional information on the implementation details of the single packages and a comparison of two mesh-based navigation methods using our new toolset.

2. Related Work

To model free and occupied space of arbitrary environments in full 3D, octree-based representations are currently state of the art. In the Robot Operating System (ROS) [3], the well known Octomap package delivers an implementation of such trees to represent full 3D environments [6]. These maps can be used to generate several 2D navigation maps depending on the robot's geometry to allow safe navigation in cluttered environments [7] or for applications in grasping [8, 9] and is also integrated in the well known *MoveIt* package

in ROS [10]. It has also been used to integrate semantic information [11, 12]. However, these representations only deliver a – possibly semantically annotated – discrete geometric world model. The resolution of the representation here always depends on the level of discretization. Higher resolute grid or voxel maps allow more precise navigation, but come at the cost of a significant increase in memory usage and traversal time. To tackle this issue, the *SkiMap* approach [13] 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 the 3D space. Each tree node stores a list of sorted and linked skip lists, which can be accessed in a binary search fashion. Similar the other approaches, *SkiMap* is integrated with ROS nodes and provides a ready to use ROS interface.

For continuous 3D maps, polygonal representations in the form of triangle meshes are an apt candidate as a more general representation. Such maps have been used for decades in the field of computer graphics to represent 3D environments efficiently. Modern graphics pipelines enable an efficient rendering of such maps on GPUs in real time. However, using such maps in an integrated manner is rarely seen in robotic applications, although applications that rely on such maps for path planning [14, 15, 16] are currently being developed. In previous work, we have shown that the generation of high quality maps from 3D point clouds acquired from 3D laser scanners is feasible [1, 17]. This previous work also showed that polygonal representations can significantly reduce the memory required to store an accurate model of the environment, compared to the input point clouds or voxel maps. Using mesh optimization techniques, it was possible to reduce the memory requirements to a few megabytes in comparison to several hundred megabytes when using voxel maps as reported in [6].

For typical flat environments, *Costmap 2D* has proven to be a reliable representation for robot navigation in 2D [18]. 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 [19], which allow to encode more detailed information than just occupancy in multiple layers. The processing of costmap data is separated into different semantic layers, each one representing one type of obstacle or constraint. This structure allows for example to store static environments, preferred zones and dynamic obstacles as well as an obstacle inflation in different map layers. Each layer has its own semantic meaning and can be computed or loaded at run-time and reconfigured on demand. These layers are later combined to one master costmap. In analogy to these ideas, our approach includes a layered mesh map, which loads meshes that support several semantic layers as plugins using the ROS `pluginlib`.

A straight-forward extension of 2D cost maps are so-called 2.5D grid maps, which encode local height with respect to a given reference level. The *Grid Map* library overcomes the *Costmap 2D* limitations by storing values in a stack of float matrices. The *Grid Map* package comes along with visualization tools for RViz and message definitions and allow easy conversion to OpenCV images [20].

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 and other metrics. In addition to these benefits the *Grid Map* library uses a 2D ring buffer to efficiently move data in an azimuthal centric fashion around the robot to reduce the memory footprint when the robot moves. The main drawback of such grid-based maps is that they are unable to model multi-level environments such as multi-story buildings, stairwells, underpasses, bridges, or overhanging structures. Such information gets lost if 3D is projected onto 2D. In our previous work we showed how to integrate semantic information and 3D geometry in navigational planning when using 2D occupancy grid maps [21]. At that time, there were no development tools for 3D maps available in ROS and the only way to use 3D environment data was to somehow transform the data back into 2D to make use of the existing 2D software stack. This example stresses the necessity to support and simplify the development of 3D approaches in the ROS framework. Our tools close this gap and enable the usage of annotated 3D meshes offline and online during

robot operation in combination with the already existing ROS tools. The *Mesh Tools* supports the ROS coordinate frames and tree transformations of the ROS transform library *tf* [22].

Making annotations and adding extra information to a 3D map is often done by using interactive markers [23] or adding an extra layer. These markings are not part of the internal representation and add further information to the underlying map format. This fact limits their usability as an integral part of the map, especially with regards to more general solutions in the context of semantic mapping. 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.

To efficiently support and to expedite all these relevant research topics concerning 3D mapping and related topics, it is essential to have an integrated set of tools that supports the development of novel algorithms efficiently. Such tools should allow to inspect the current world view together with the incoming sensor data and the perceived robot state. This in turn allows newly developed methods to be debugged and inspected in a common view. Integrating such tools into a well-established and widely used framework such as ROS, allows to build such methods based on existing and well tested software, while at the same time providing the missing infrastructure to seamlessly integrate future developments. As stated above, future development of robotics will heavily rely on 3D environment representations. It is crucial that tools for 3D meshes reach the level of integration that is already available for 2D map applications. In the remainder of this paper, we present our proposal for such a package, explain the message definitions, data structures and RViz plugins for map representation and present a navigation application example to demonstrate the desired seamless interaction with already existing ROS packages. In this navigation application example we use *Move Base Flex* [24] 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 [25].

3. Basic 3D Representation and Visualization

In this section, we discuss the general structure and technical foundations of our *Mesh Tools* package. We present the basic data structures and message definitions to represent meshes within ROS, plugins for RViz to render and interact with them and a schema for a storage solution to serialize annotated triangle meshes.

3.1. Package Structure

Our Mesh Tools package consists of several sub-packages as depicted in Fig. 2. 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 reposts 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. It transforms vertices and normals given a *tf* transformation or two frames and corresponding time stamps. This *tf* compatibility allows a seamless interaction of the novel data structures with the existing tools.

The *label_manager* handles the generation and conversion of annotated parts of the mesh maps. Labeled maps generated with this tool can be serialized into HDF5 files [4]. The proposed schema for this container is a direct extension of the point cloud format described in [26]. 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 into the configured HDF5 file.

3.2. Message Definitions

Our ROS messages strictly divide between the geometry of a mesh, associated attributes, material definitions, and texture information. The annotations and textures are linked to the geometry via *Universally Unique Identifiers* (UUIDs) as standardized by the IETF [4]. UUIDs are 128-bit numbers designed to uniquely identify pieces of information. UUIDs and their generation are standardized within the IETF standard RFC-4122[5]. To generate our UUIDs, we use the standard implementation provided by the OSF and convert them into strings that are stored in the corresponding ROS messages.

This message structure can communicate geometries without attributes to RViz and other nodes. Geometrical change, such as adding, removing, repositioning vertices or faces requires to generate a new UUID, since already existing computed attributes might not match to the new geometry anymore. This strict separation allows the design of 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 sent to other nodes in form of attributes of the initial mesh without having to send the geometry information again. As described above, the linking of the respective data is done via the UUIDs of the mesh geometries. These UUIDs are generated once for each mesh geometry and stored within the messages. Summarized, the mesh, its attributes, and its metrics are modularly dispatched together with UUIDs where resources with the same UUID belong together. This seperation helps to save network traffic and computation time as unnecessary double-publishing is avoided. In RViz each cost and metric layer can be selected by the user and visualized independently, together with the corresponding mesh geometry. In the following paragraphs we give an overview on the most important mesh messages:

[4] `https://www.ietf.org/rfc/rfc4122.txt`

[5] `https://www.ietf.org/rfc/rfc4122.txt`

MeshGeometry (Stamped) Defines the geometric structure of a triangle mesh with 1.) an array of 3D vectors, which represents vertices in $\mathbb{R}^3$, 2.) an array of indices, in which three indices each define a triangle by referring to the array of vertices (vertex and index buffer), and 3.) an array of vertex normal vectors.

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 links 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 feature-attributes are designed to ground visual features computed in textures to a 3D position on a surface via a feature descriptor. For

example, SIFT, SURF or other features could be computed for each texture in the map with OpenCV and the resulting feature descriptor vectors as well as their 3D world positions can then serialized with this message. We plan to use these for future applications for camera-based localization in textured meshes.

MeshFeatures List of features for a corresponding mesh.

These `mesh_msgs` build the basis to transfer meshes and their attributes from one node to another. A possible application example could be the automatic generation of navigation maps from 3D PCD. First, a surface reconstruction node could send the computed meshes to RViz to provide a first visualization of the 3D model in RViz. In parallel, terrain analysis nodes can take the same message and compute per-face and per-vertex costs as attributes for the mesh geometry. These attributes can then be included in the visualization after the analysis nodes have finished without sending the geometry again as shown in Fig. 3). A full description of such an application is given in Sec. 5.

3.3. RViz Plugins

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 the rendering of 3D objects. Generally, OGRE follows an object-oriented approach. Each loaded mesh consists of its own index and vertex buffers and associated resources like texture maps. Such additional resources are managed via a global resource management object. The main problem with this design in the context of 3D environment map management is that according to OGRE's documentation of the mesh class, "The `Ogre::Mesh` object represents a discrete model, a set of geometry which is self-contained and is typically fairly small on a world scale." [27]. This object model is sufficient to handle typical 3D objects like robot models, arrows and boxes that are normally rendered in RViz. In the context of map representation, such a design model is problematic, as the meshes can become large in number of vertices and triangles. The reference dataset deployed with our package and used in this paper to demonstrate the

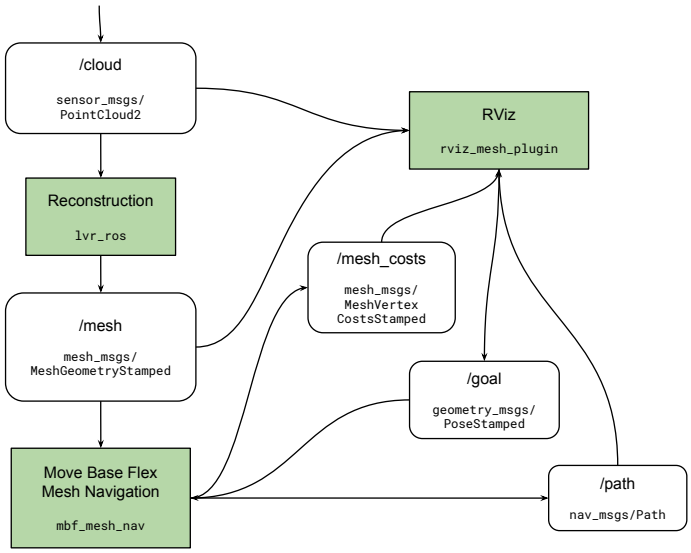


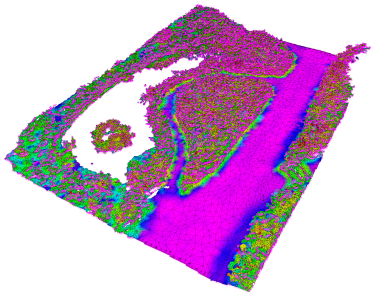
Figure 3: ROS nodes and used messages in the mesh navigation example. Green boxes show the *Reconstruction*, *Move Base Flex Mesh Navigation*, and the *RViz* nodes communicating using messages shown the rounded white boxes.

Figure 4: We recorded a dataset in the Botanical Garden of the Osnabrück University which is used to demonstrate our *Mesh Tools* and `mesh_map` ROS packages. It shows a large pathway connected to a smaller bypass next to a lake.

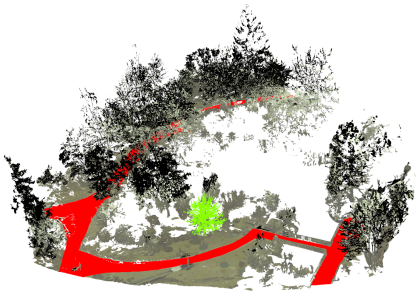
abilities of our tools already consists of 788 850 vertices and 1 436 842 faces but covers only a fraction of the complete dataset presented in [26]. To address the problem of scalability on large meshes, it would be necessary to implement a custom out-of-core object management to segment the large meshes into smaller parts or a level-of-detail approach to allow efficient handling in OGRE. We are currently investigating these possibilities as well as using other frameworks like VTK [28] for visualization of ROS messages. Despite the given limitations of OGRE, we were able to implement the basic requirements within RViz with custom-extensions to visualize 3D triangle maps, including lighting support, textured color materials and wire-frame rendering. The supported modes are integrated into RViz's tree view, and displayed depending on availability.

In addition to this static rendering, we created an interactive pose selection tool to allow setting poses in RViz on the mesh surface. This feature can be used to define goal poses within the 3D geometry and trigger algorithms like path planning as presented in the example in Sec. 5. To support semantic classification and labeling in the meshes, 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 together in cluster of triangles that is labeled with an user-defined attribute string or internal ID. A demonstration of this tool is presented in the added additional screenshots and demonstration videos provided as supplementary material to this paper.

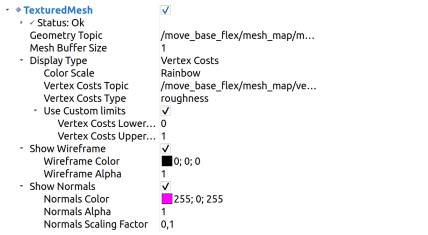
Fig. 5c shows the *TexturedMesh* plugin panel to configure topics and visualization parameters. The *Display Type* can be configured to *Fixed Color*, *Vertex Color*, *Vertex Costs*, or *Texture*. Here, *Vertex Costs* is configured as *Display Type* and in the drop-down menu the *roughness* layer is 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. 5b. It shows an outdoor environment mesh reconstructed from a recorded point cloud from a top down viewpoint, see Fig. 4. A large pathway connected to a smaller bypass next to a lake has been



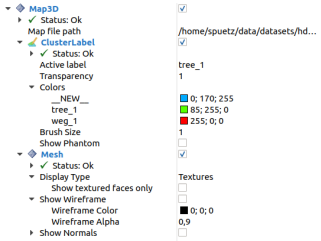
(a) RViz *TexturedMesh* plugin display



(b) RViz *Map3D* plugin display



(c) *TexturedMesh* RViz plugin config panel

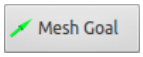


(d) *Map3D* RViz plugin config panel

Figure 5: The *Mesh Tools* RViz plugins *TexturedMesh* and *Map3D* configuration panels.

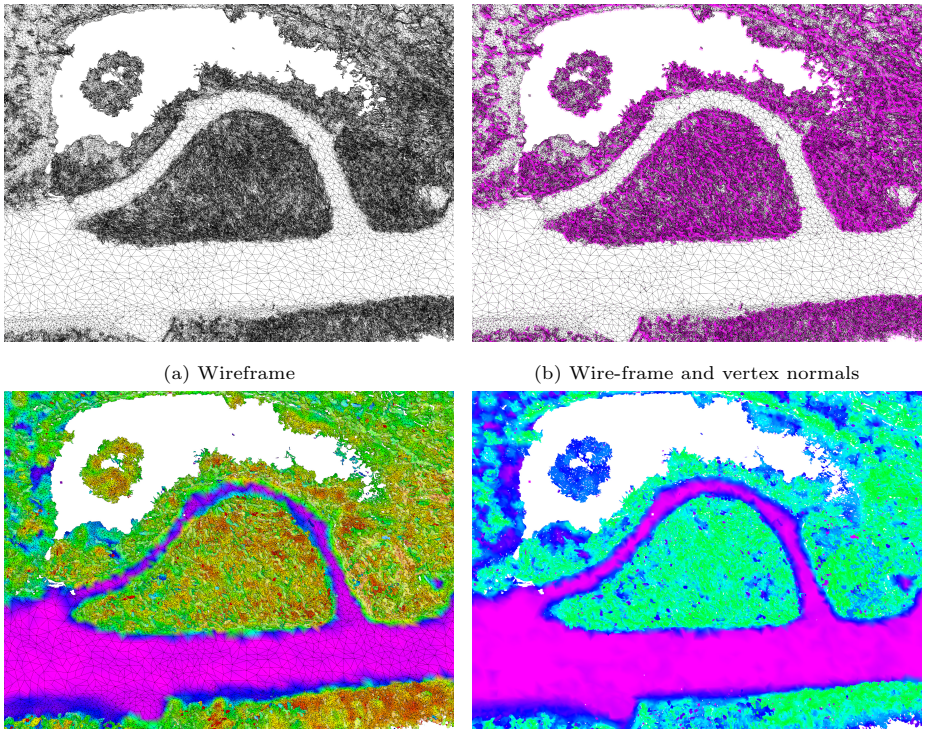
recorded as point cloud and pre-processed. The computed mesh geometry and some analyzed metrics are displayed as layered vertex costs in Fig. 6.

In this example, the *Mesh Tools* can efficiently display the computed metrics on the surface sent to the described *TexturedMesh* RViz plugin using the described *mesh_msgs/MeshVertexCostsStamped*. The functionality is briefly shown in [6].

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

[6] `https://youtu.be/ir4kZif5FS8`



(a) Wireframe

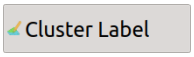
(b) Wire-frame and vertex normals

(c) Roughness Vertex Costs and wire-frame

(d) Height Differences Vertex Costs without wire-frame

Figure 6: *TexturedMesh* plugin demonstrating different rendering modes: Wire-frame, vertex normals and vertex costs. The scene is used for the mesh navigation application example, as shown in the video: `https://youtu.be/qAUWTiqdBM4`

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 an 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 a video in combination with the *TexturedMesh* plugin and a simple path planning scenario using a Dijkstra-based planning approach[7].

The *Map3D* configuration panel is shown in Fig. 5d. The map was loaded from an HDF5 map file that contained a mesh with its attributes and labeled faces as described in Sec. 3.4. 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 (see Fig. 5d). 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. 5d. The *ClusterLabel Panel* allows to manage the clusters and label names as well as the corresponding colors. The usage of this tool is demonstrated in detail in a video[8].

The *Cluster Label* tool can be used to label clusters of faces with a user defined class in real time.

One major issue when using OGRE's native data structures for object pick-

[7] `https://youtu.be/X_TXC9hrAgo`
[8] `https://youtu.be/8n4737D2abM`

ing is a huge time lag in the user experience when selecting triangles. This is mainly due to OGRE's object-based approach for 3D modeling and abstraction from the underlying hardware. Hence, using rendering techniques that combine triangles with object IDs in the framebuffer to allow efficient picking is not possible, as these buffers are not directly accessible within the framework. In OGRE, objects can only be selected via view-based ray tracing. For that, it provides a reference implementation to compute the intersections of view rays with the represented objects. For small numbers of objects this implementation is sufficient, but for meshes with a large number of triangles, rendering slows down massively. Another problem is that after modification, all affected objects have to be rebuilt and sent back to the graphics card. To address these issues, we build a custom implementation that replaces OGRE's default object class *ManualObject*. Instead of resending the whole vertex and index buffer to the graphics card each time a face was marked or labeled, we use a data structure that holds the vertex data of all objects persistently on the graphics card. The creation of such a structure is admittedly more complex, since all data has to be set by hand in the enhanced implementation, but it gives us the ability to create sub meshes of the current loaded mesh without resending vertex data. These sub meshes refer to the same global vertex buffer that was already loaded to the graphics card when the initial 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 the communication overhead significantly and increases rendering performance.

However, the biggest performance boost 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 over all triangles in OpenCL based on the methods presented in [29]. Such a structure allows to check for intersection in logarithmic time with respect to the number of triangles as opposed to the linear run time of OGRE's reference implementation. Additionally, the use of OpenCL allows hardware-accelerated massively parallel intersection computation between the

Table 1: Selection time with our *MeshClusterLabelTool* on different-sized datasets. The selection times for three selection modes: 1) *Single* for selecting singles faces, 2.) *Box* for selecting faces with a spanning rectangle, and 3.) *Radius* for selecting faces in a brush style, here with a radius of 1 m. The selection times are in milliseconds and the bounding box is in meters. The datasets can be viewed and downloaded on `https://github.com/uos/pluto_robot/`.

Dataset	#Vertices	#Triangles	Boundingbox x, y, z			Single	Box	Radius
Botanical Garden	714 760	1 430 188	39.05	49.25	6.67	104.3	101.8	147.3
Stone Quarry	992 879	1 904 178	100.58	100.58	23.94	148.6	146.0	267.4
Physics Campus	719 080	1 617 772	166.02	83.61	26.33	97.1	121.4	134.8
Farmer's Pit	401 036	794 509	122.23	104.57	14.84	82.5	81.8	90.1
Market Garden	1 361 308	2 656 283	174.33	149.61	24.58	168.6	189.0	214.8

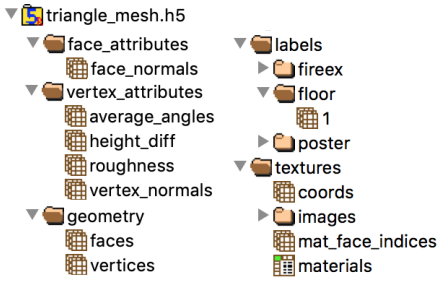


Figure 7: HDF5 file structure to represent textured triangle meshes and labeled clusters.

view rays and the stored geometry, instead of the CPU-based reference that is restricted to a limited number of hardware threads. This BVH-based preordering of the triangles also decreases the cache miss rate, which also drastically improves performance. Using our labeling tool with the brush or rectangle selection, the time for annotation and cluster generation can be reduced to a range of 100 ms to 270 ms for the reference maps provided in our repository, which is sufficient for a seamless user experience. An overview over selection times in different-sized meshes using different selection tool modes is provided in Tab. 1.

3.4. Persistent Storage

Complex environments consist of a large number of structures. When we assign a high number of attributes to vertices and faces and also increase the level of detail and resolution, the data size is growing fast. As a consequence, if we want to store large maps with all attributes for robotic applications, we need

to define a structured persistence layer besides a compact online representation. For that, we use the well known HDF5 format, that is specially tailored to store and organize large datasets. The HDF5 schema for our purpose is sketched in Fig. 7. This structure extends the HDF5 storage layout we initially presented for our reference dataset [26] to support storage of 3D annotated 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. The HDF5 format definition natively supports lossless compression of the stored data using the Deflate (i.e. gzip) and Shuffling [30]. Second, the representation derived from the original input data is directly stored together with it. In general, it is possible to use the ROS message transfer and ROS bags for such purposes, but practical experience shows that large messages are sometimes lost and ROS bags do not efficiently support compression. This special HDF5 file storage is designed to hold the static environment representations which can be accessed as sketched in Fig. 8.

Generally, an HDF5 file is structured into groups that support subgroups and datasets in form of multidimensional arrays. Additionally, each dataset 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 an HDF5 file.

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.

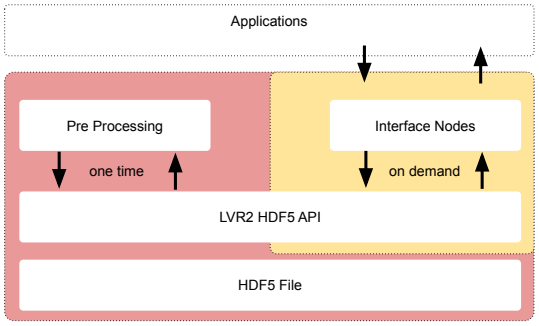


Figure 8: LVR2 persistence storage interface and on demand processing.

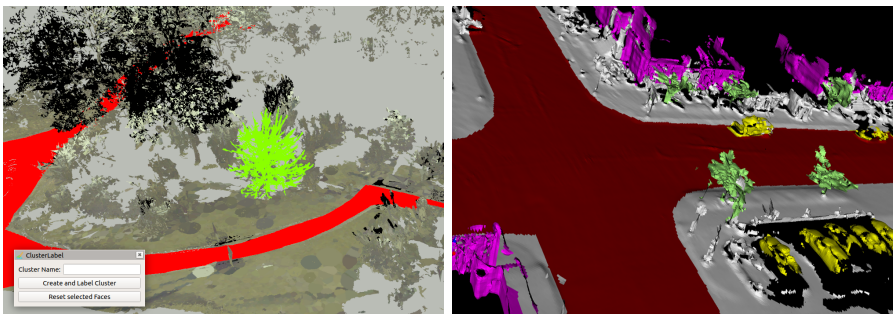


Figure 9: Examples of semantic labeling for a scene in the Botanical Garden and a mesh generated from the Ford Campus dataset [31]. The user can select sets of triangles which are associated with an individual label via an input dialog (left). These labels can be tagged as instances of semantic classes (right).

4. Tools and Data Structures for Annotated Environment Representations

Besides the basic data structures and plugins to handle meshes within ROS, we also provide packages for interactive annotation to generate multi-layer polygon maps for robot navigation, which are described in the following sections.

4.1. Mesh Labeling

The mesh labeling tool can be used to build up annotations for single triangles or groups of triangles. The labeled faces are stored as sets of triangle IDs in the underlying HDF5 file together with the corresponding label string. The ability to create such annotations is essential for a number of real-life applications

like semantic mapping to generate symbolic knowledge about the environment or machine learning to annotate training datasets. An example for semantic labeling is shown in Fig. 9. The left image shows a selected set of triangles (green) that form a tree. After selection, the cluster label tool asks for a string that will be associated with the selected triangle set. Here, it is labeled with an instance name of the class `tree` in a semantic representation (in this case an OWL-DL ontology). The right image in this figure shows a visualization of a semantically annotated 3D mesh from the freely available Ford Campus LiDAR dataset [31]. All instances of a semantic class are rendered with the same color (red for streets, green for trees, yellow for cars and pink for buildings). For machine learning, the same tool could be used to generate training data for supervised machine learning approaches for automatic segmentation based on geometry [32] or textured meshes [33]. Another application is the annotation of dangerous or forbidden areas in mesh-based robot navigation maps as described in the next section.

4.2. Mesh Map

A Mesh Map in our representation consists of a mesh geometry definition and attributes assigned to vertices and triangles. Each element can have multiple attributes to represent semantic layers associated with the geometry or other attributes like costs or constraints. Additionally, the mesh map provides methods and utility functions to access the mesh map and perform certain tasks on the structure, e.g., iterating over the neighborhood of a triangle or vertex, combine the layer costs, and continuously access costs and attributes using barycentric coordinates. The *Mesh Map*, as a central component of our `mesh_navigation` stack, uses the *Mesh Tools* to visualize and interact with stored mesh layers. It is used as map representation for the mesh navigation described in the next section. In analogy to the `costmap_2d` package, which provides layered cost maps for navigation purposes, our mesh map is used as access point to perform robot navigation on 3D meshes. This layered model allows to define sets of layers for different robots with respect to their navigation skills and abilities. Based on

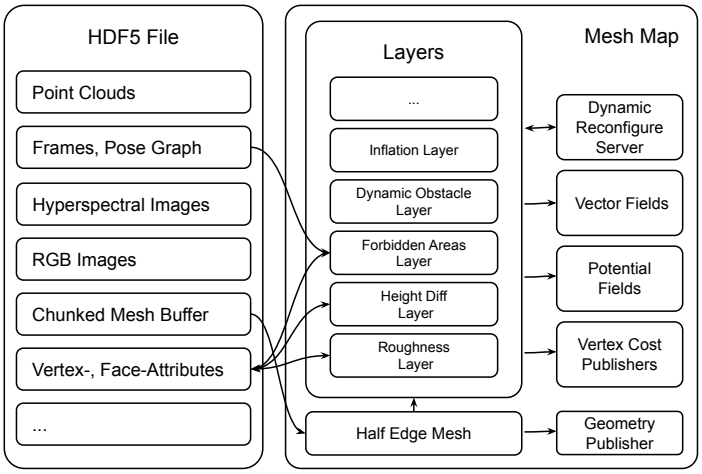


Figure 10: Layered *Mesh Map* architecture. It uses the *Mesh Tools* messages to communicate the mesh geometry, layer vertex costs and vertex colors, e.g. to RViz to visualize metric costs and potential fields, as well as the respective geometry.

these abilities, a set of layers can be specified and configured to model the accessible and traversable parts of the mesh for a specific configuration. This idea allows to reuse the same geometric map for different kinds of robots without the necessity to modify or even recompute the geometric representation.

These layers are loaded as plugins using the ROS `pluginlib` which make the mesh map highly extensible for special requirements. Furthermore, it provides the conversions from the internal LVR2 representation to the corresponding `mesh_msgs`. In the actual implementation, all layers are computed or loaded as LVR2 data types and then converted and published as `mesh_msgs/VertexCostsStamped`, or `mesh_msgs/VertexColorsStamped` beside the `mesh_msgs/MeshGeometryStamped` to visualize and interact with the mesh and its attributes. Fig. 10 gives an overview of the mesh map structure and its HDF5 data access.

First, the mesh geometry, vertices, faces and all distances are loaded. Then the configured mesh plugins are loaded. Some layer plugins will load their vertex attributes on demand or compute these and store them for later usage in the HDF5 file. A dynamic reconfigure server allows to dynamically set and ad-

just the layer parameters. Changing information on any layer will trigger an update function on all subsequent layers, as they may depend on the information from lower layers. For example, the inflation areas in the obstacle inflation layer will be recomputed if obstacles were identified and inserted into one of the underlying layers. This way, we can transfer the idea of obstacle inflation as it is traditionally used in grid maps directly to our mesh maps. Obstacles are inflated by the inscribed radius of the robot by labeling the triangles around an obstacle accordingly. This can be done by expanding the obstacle classification recursively on triangle level, starting from the detected obstacle triangles, until the robot dependent inflation radius is reached. Due to this extension, the robot can be treated as a point in simple path planning, rather than using the robot's footprint. More sophisticated and computational expensive planners, which take the robot footprint into account, may also ignore the inflation layer. The layered mesh map is used by *Move Base Flex* as map representation when performing navigation tasks outdoors. With our label tool we are able to mark certain areas which can be treated as forbidden areas which then can be loaded by an associated layer plugin. This map instance allows a high degree of freedom to integrate extra information into the combined mesh map. The same layer plugins can be loaded with different parameters multiple times in a user configured ordering depending by the robot requirements. Additionally, developers can implement new layer plugins expressing new layers costs, fitted for their needs and the environmental and robot conditions.

The mesh layers define costs for each vertex for the same mesh geometry. In each layer plugin each vertex handle is associated to a vertex cost value. These costs are stored in the plugin's attribute map. The layer vertex costs can be computed following various rules, and thereby express, e.g. roughness, height differences, inflation, or other cost values. The cost values are combined in the order of the layers to a final cost for each vertex. Thus, areas that cannot be accessed can be expressed by a combination of layers. Non-passable areas are named and marked as *lethal*. Each layer plugin determines which vertices are *lethal* and returns the corresponding indices. In the configured plugin order, the

indices are stored layer by layer in a combined set and can be used for the current layer calculation. Furthermore, each layer below can trigger recalculations of the layers above. Fig. 6c shows the roughness layer. Height differences, which were computed on the input mesh geometry, are shown in Fig. 6d. All derived layers can be stored into our HDF5 representation, so the cost computation has to be done only initially when the map is created or if information is updated.

5. Mesh Navigation

In this section we present an exemplary application for the integration of our mesh tools into a real robotic application. Aiming for a fully autonomous robot, operating in environments with rough terrain, we built a processing pipeline starting with no terrain information, consisting of the following steps: Taking high resolution point clouds, triangle mesh reconstruction, terrain analysis, and path planning and execution. The whole pipeline can run online without any user interaction. The navigation meshes are generated directly from PCD and then used on-board the robot to perform path planning on an automatically generated 3D mesh. The aim of this demonstration is to show how our software adds the missing functionality in ROS to realize an integrated use-case that actually relies on the novel structures presented in this paper. It is designed to show how novel algorithms based on annotated triangle meshes can now be used and evaluated directly in ROS. For that, we compare two mesh navigation algorithms directly within the framework.

5.1. Architecture

We implemented a client-server architecture in which the robot sends high resolution PCD to a dedicated server that computes and maintains a compact semantically annotated and query-able environment representation as proposed in [34]. For this proof-of-concept, we assume a stable and fast network connection of the integrated components. If this is not available, all computations can be optionally done directly on-board at possibly lower resultion depending

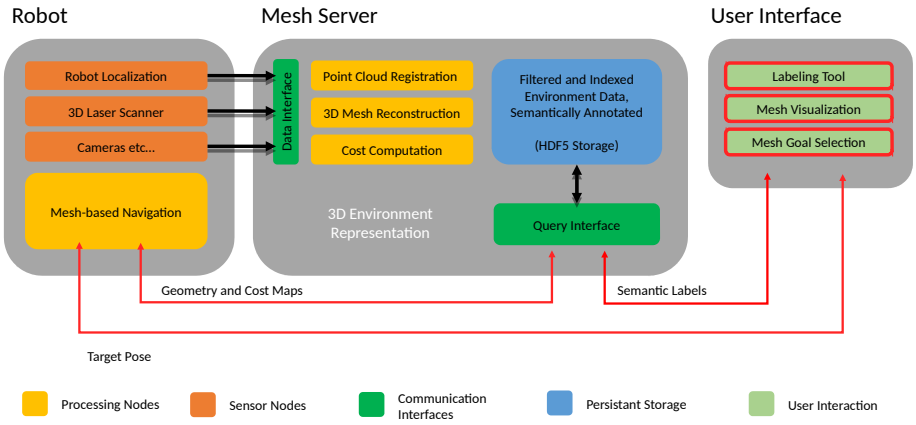


Figure 11: General layout of the processing pipeline for mesh navigation. The MeshTools package provides the graphical user interface to display and interact with the 3D environment representation provided by the mesh server as well as the corresponding message definitions highlighted in red.

on the available computing resources. The high-level architecture for this use-case is sketched in Fig. 11. The robot sends its sensor data via ROS messages (e.g. *PointCloud2* for 3D laser scans) to the Mesh Server. Here, the single point clouds are registered via a 6D SLAM module to a globally consistent georeferenced map. The new data is then converted into a triangle mesh via a reconstruction node that uses LVR2. Based on this mesh geometry, the different cost layers for navigation are computed. All this information is then indexed and integrated into the global environment model that is stored in a HDF5 file on the server. This model can be queried via a query interface that provides the appropriate mesh messages. The map fragments are directly stored in the layered mesh map module. Thereby, we are able to receive a locally stored map from an HDF5 file or from the dedicated mesh server. If manual semantic annotation is needed (which is not part of this use-case), it could be done directly in RViz without the need of exchanging files between the components.

5.2. Implementation in ROS and Provided Packages

In the ROS implementation, the *Mesh Tools* are the basis to visualize the generated maps and states of the implemented algorithms in near real time. The environment map can be easily displayed with the robot and the online streamed sensor data. Now, it is possible to compute a mesh map online from PCD by using the presented data flow.

With Move Base Flex[9] (MBF) we have the possibility to use the same execution logic as for other map representations to perform navigation tasks. Here, we implemented a more complex mapping scenario, in which the robot has to traverse the terrain from its current location to a goal pose selected by the user in RViz using the novel structures. The *Mesh Tools* helped us to visualize the intermediate planning steps and all necessary environment metrics. The MBF mesh navigation server, as well as navigation plugins and the `mesh_map` implementation are provided in the `mesh_navigation` bundle[10]. In this bundle we provide mesh path planning plugins to perform path planning on the mesh representation as described in the next chapter. Further, a motion control plugin following the computed potential field is provided. Based on the loaded layered mesh map the robots is controlled to the goal while avoiding rough terrain obstacles.

A Gazebo simulation setup for our robot Pluto, as well as exemplary HDF5 maps, e.g. a forest with a stone quarry and the Botanical Garden, allow to test the complete mesh navigation pipeline in different environments. For example, the Botanical Garden map file contains the map fragment depicted in Fig. 6 and Fig. 4 of the Botanical Garden at Osnabrück University.

The HDF5 files contain all trafficability layers which can be combined differently depending on the robot's abilities. These combined layers are then used to perform robot navigation on the respective mesh, representing the trafficable surface for a certain robot. In the following example we show how to use

[9] `https://github.com/magazino/move_base_flex`

[10] `https://github.com/uos/mesh_navigation`

the `mesh_navigation` stack together with the `mesh_tool` and our robot *Pluto*, which is modeled in the package bundle `pluto_robot`[11]. The map file, the navigation setup, and the Gazebo simulation setup are provided in the packages `pluto_navigation` and `pluto_gazebo`.

5.3. Potential Field Generation for Navigation

The algorithmic foundations for the computation of the trafficability layer and navigation planning on meshes using Dijkstra's algorithm are described in [16]. The trafficability estimation based on local roughness, height differences and lethal obstacle inflation is combined to a navigation layer. In this demonstration, we will compare two planner plugins for mesh navigation: First, a Dijkstra mesh planner performs path planning by exploiting the triangles' topological connections and their costs, and second, a *Fast Marching Method (FMM)* wave front propagation planner propagating a wave front over the mesh surface. Both FMM wave front and Dijkstra propagation start from the navigation goal and generate a potential field from each accessible position to the goal. The resulting field reflects the distance to the goal. Thereby, it can be seen as potential energy (work, in physics) to bring the robot to the goal assuming a constant mass.

Furthermore, it can be evaluated in a continuous fashion by using barycentric coordinates as formulated in Eq. 3. If the robot is moving on the surface of the mesh its position is located somewhere above the surface. To evaluate metrics, the scalar field and derived direction vectors in a continuously with respect to the surface, and not only at the vertex positions of the mesh, we have to compute a linear combination of the three values corresponding to the triangle vertices. For example, to derive the direction to the goal pose at the current robot position in the map coordinate frame, the origin of the robot base coordinate frame must be projected onto the surface and the coefficients for the linear combination of the three direction vectors must be computed. The projection and the linear

[11] `https://github.com/uos/pluto_robot`

combination of the three direction vectors can be performed simultaneously resulting in a continuous vector field that the robot should follow in order to reach the goal. This interpolation between the three triangle vertices v_1, v_2, and v_3 can be computed using Heidrich's method [35] that allows to compute barycentric coordinates as coefficients α, β, and γ and project any query point $p \in \mathbb{R}^3$ onto the plane of the triangle $\triangle v_1 v_2 v_3$ at the same time in an efficient way. The barycentric coordinates of a point p and $\triangle v_1 v_2 v_3$ are then determined as expressed in Eq. 1. Let $\vec{u} = v_2 - v_1$, $\vec{v} = v_3 - v_1$, $\vec{w} = p - v_1$ and $\vec{n} = \vec{u} \times \vec{v}$.

$$\gamma = \frac{(\vec{u} \times \vec{w}) \cdot \vec{n}}{\vec{n}^2} \qquad \beta = \frac{(\vec{w} \times \vec{v}) \cdot \vec{n}}{\vec{n}^2} \qquad \alpha = 1 - \gamma - \beta \tag{1}$$

The projection of the point p onto the triangle's plane is then given by $p' = \alpha v_1 + \beta v_2 + \gamma v_3$. In particular a point p is located inside the triangle $\triangle v_1 v_2 v_3$ if the following inequations are fulfilled.

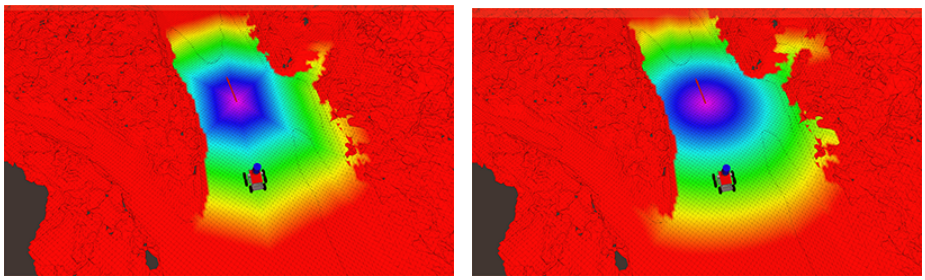
$$0 \leq \alpha \leq 1 \wedge 0 \leq \beta \leq 1 \wedge 0 \leq \gamma \leq 1 \tag{2}$$

Finally, a direction vector $\vec{d}(p) \in \mathbb{R}^3$ towards the goal pose at a point p can be computed as formulated in Eq. 3. The distance value $u(p)$ at p towards the goal can be computed as stated in Eq. 4. The necessary distance and direction vertex attributes are computed during the wave front propagation and stored in associated maps $\vec{m}_{\mathcal{V}}$ and $\vec{m}_d$. A mesh vertex $v \in V$ is mapped to a direction vector towards the goal with $\vec{m}_{\mathcal{V}} : V \to \mathbb{R}^3$. The distance to the goal pose is mapped by $m_d : V \to \mathbb{R}$.

$$\vec{d}(p) = \alpha \cdot \vec{m}_{\mathcal{V}}(v_{h_1}) + \beta \cdot \vec{m}_{\mathcal{V}}(v_{h_2}) + \gamma \cdot \vec{m}_{\mathcal{V}}(v_{h_3}) \tag{3}$$

$$u(p) = \alpha \cdot m_d(v_{h_1}) + \beta \cdot m_d(v_{h_2}) + \gamma \cdot m_d(v_{h_3}) \tag{4}$$

595 The *Fast Marching Method (FMM)* has been introduced by Sethian and Kimmel [36, 37]. The resulting supporting points for the distance scalar field are stored in m_d. The scalar field can be translated into the vector field $\vec{m}_{\mathcal{V}}$ in a post processing or during the FMM update step. The gradient of the computed scalar field leads the robot to the goal and can be seen as an attraction



(a) Dijkstra potential field (b) Wave front potential field

Figure 12: Potential field comparison using the *Textured-Mesh* plugin from the *Mesh Tools* to compare the path planning of (a) Dijkstra's algorithm and (b) a FMM wave front propagation algorithm. The rainbow colors indicate the computed distances to the picked goal (marked by the arrow). Since Dijkstra is constrained to the edges of the mesh, the resulting potential field highly depends on the mesh's structure (a), whereas the wave front propagation produces a potential field reflecting the distances computed over the 2D-manifold (b).

force towards the given goal defined on the surface. Further, $u(p)$ is the scalar potential for the vector field $\vec{d}(p)$. Thereby, the attraction force at a point p towards the goal points in the direction of $\vec{d}(p)$, where p is located inside the triangle $\triangle v_{h_1} v_{h_2} v_{h_3}$ so that the inequation 2 is fulfilled. The computed layers and the scalar fields are expressed as vertex costs and published using the *mesh_msgs/MeshVertexCostsStamped* message. The provided visualization simplifies the development of such methods and algorithms and allows to evaluate differences and to identify problems. Without the new RViz plugin, it would be necessary to export each layer and parameterization in a file format supported by standard software (e.g. Meshlab) and inspect it one by one. With the provided means, the state of the algorithms can be visualized on the fly to find suitable parameter sets for the given environments and robots, as pointed out in the following evaluation.

5.4. Evaluation of Mesh Navigation

The propagation and computation of the potential field can be inspected using the *TexturedMesh* plugin by switching between the different layers directly in *RViz* in the *Vertex Cost Type* drop-down menu. With this tool, we can check

computed values at a given distance by choosing corresponding cost limits in the plugin configuration at run time. In the following, we used it to compare our two path planning plugins (the FMM wave front propagation and the implementation of Dijkstra's algorithm), as shown in Fig. 12. We used the *MeshGoal* plugin to set a user defined navigation goal, which is sent to the execution logic and handed over to MBF and the path planning plugin. This plugin computes the potential field and deducts a path to the goal – if possible. Using our *Mesh Tools*, we were easily able to identify a major issue using Dijkstra's algorithm on the mesh surface. The algorithm is restricted to the edges of the mesh. Depending on the structure of the mesh, this can lead to sub-optimal paths and control commands. The visualization clearly shows that Dijkstra's algorithm is not planning on the 2D-manifold, as it is restricted to the graph structure of the triangle mesh. If the mesh is irregular and detailed enough, Dijkstra's algorithm will result in sufficiently good paths. But, on regular meshes (as produced by Marching Cubes) or sparse meshes with large triangles, the algorithm has to choose between evanescent path length differences on the edges. This favors floating point rounding errors to select one or another path more or less randomly, highly depending on the mesh's structure. Fig 12a shows the resulting scalar field and its problematic diamond shape instead of the desired spherical expansion.

A path planning example in a larger area is shown in Fig. 13. It shows the potential field towards to goal pose in a rainbow color scheme, the computed path, obstacles in red, and the robot model of *Pluto*. The `Mesh Tools` visualization of the presented robot navigation scenario is shown in a supplemental video for this paper[12].

In addition to this exemplary small environment, we also compared our planners in a larger forest environment with a stone quarry in the back as shown in Fig. 14. Again, the *MeshGoal* selection tool was used to select a goal pose on the mesh surface. The robot Pluto is located on a start position

[12] https://youtu.be/qAUWTiqdBM4

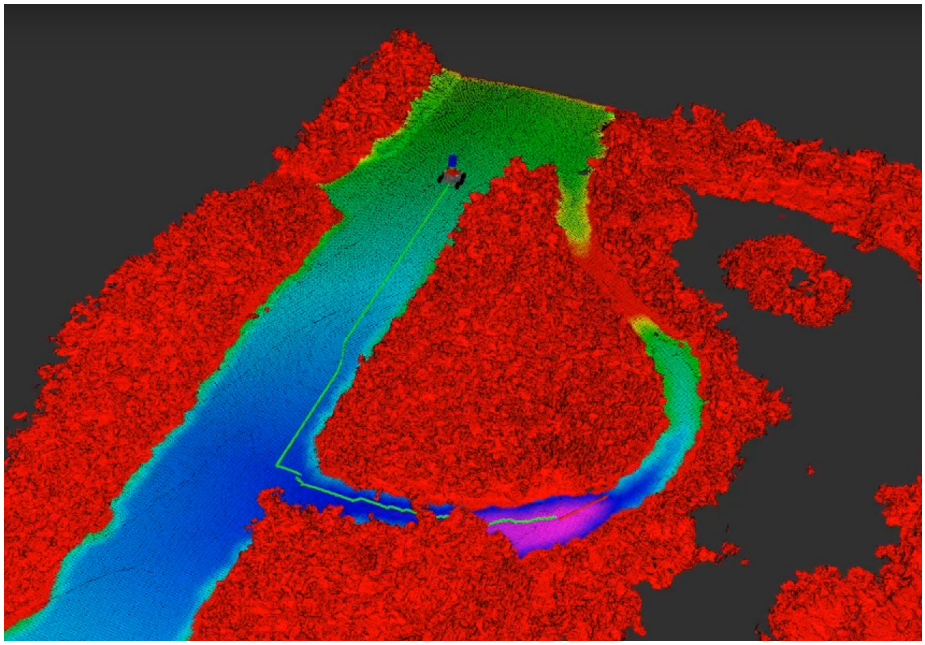


Figure 13: A wire-frame rendering of a mesh and visualization of vertex costs. The wave front propagation planner computed unfolded distance values to the goal, which are then displayed as vertex costs, showing a rainbow color potential field to the goal. Red areas are marked as lethal obstacles.

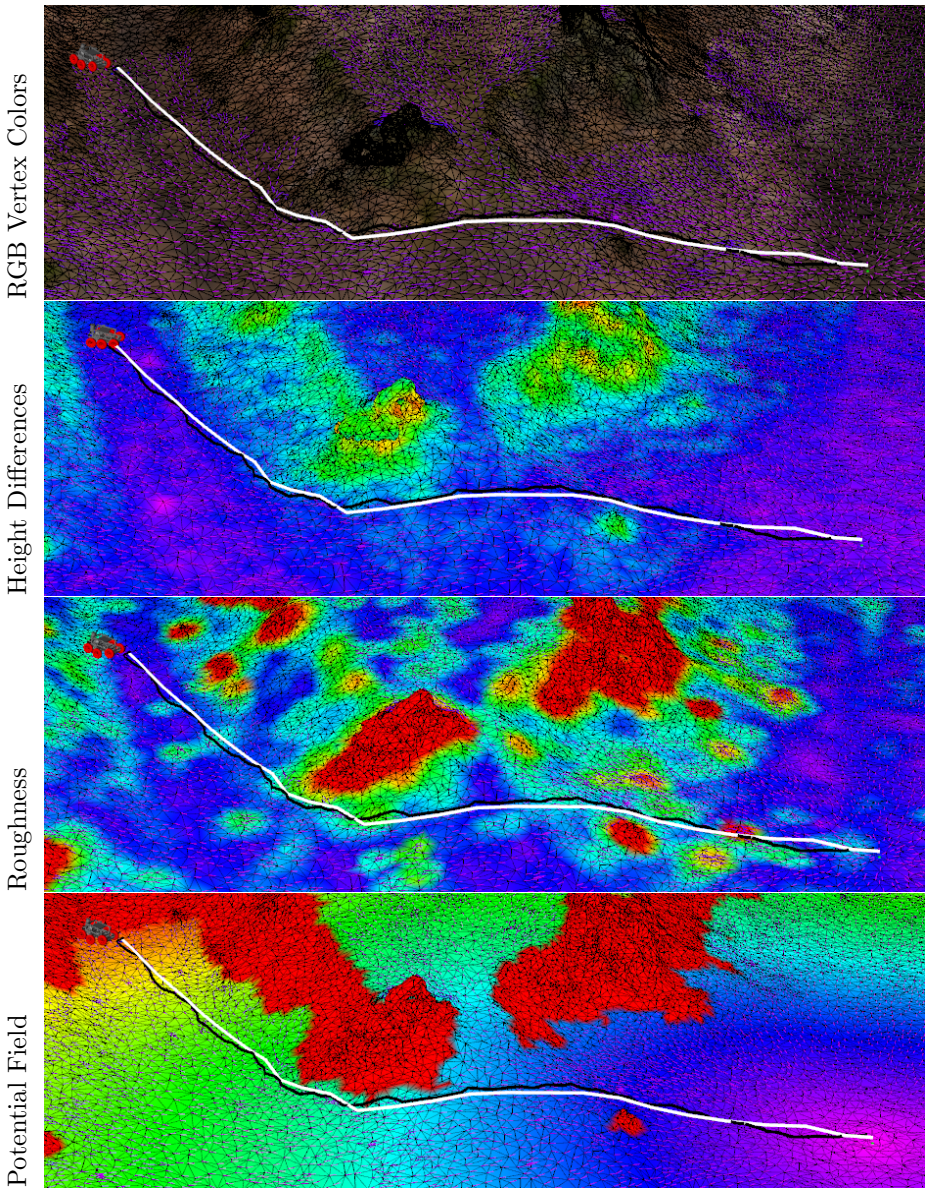


Figure 14: Robot navigation on the Stone Quarry dataset. It shows a fragment of the map visualized in RViz with our Mesh Tools. The roughness and the height differences layer are shown exemplarily together with the resulting continuous FMM potential field towards the goal pose. The white path is computed by the *Wave Front Planner* and the black path is computed by the *Dijkstra Mesh Planner*.

at a height of 1.37 m, whereas the goal is located at a height of −0.25 m with respect to the origin of the model. In the shown scenario, both planners perform path planning accurately while avoiding obstacles. The *Dijkstra Mesh Planner* computes a path length of 17.35 m in 1.051 s, whereas the *Wave Front Planner* computes a path with a length of 16.64 m in 2.074 s. This example shows that our tools helped to develop an infrastructure to compare different mesh-based planning algorithms in a real world application example. It demonstrates how the tools help to visualize the outputs of the different planning algorithms, to set suitable parameters and compare the results of the implemented algorithms. An in-depth evaluation of both algorithms is currently in progress.

6. Summary

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 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 different context like path planning and semantic annotation. The ability to define an arbitrary number of attributes of faces and vertices within the internal representation allows to represent relevant environment information like roughness, forbidden areas, height differences, and obstacle-inflation 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 reuse of the generated maps. The provided infrastructure can be used to develop future algorithms for robotic applications using annotated polygonal environment maps. The *Mesh Tools* are officially released and can be installed from the official ROS sources.

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 our SEMAP infrastructure [34]. The develop-

ment of such tools is ongoing research. In addition, we plan to integrate our Draco-based compression tools [38] into the presented infrastructure. Due to the limitations experienced with OGRE 3D, we are also investigating the use of alternative 3D visualization frameworks.

References

References

[1] T. Wiemann, I. Mitschke, A. Mock, J. Hertzberg, Surface reconstruction from arbitrarily large point clouds, in: 2018 Second IEEE International Conference on Robotic Computing (IRC), 2018, pp. 278–281.

[2] E. Piazza, A. Romanoni, M. Matteucci, Real-time CPU-based large-scale 3d mesh reconstruction.
URL http://arxiv.org/abs/1801.05230

[3] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, ROS: an open-source Robot Operating System, in: ICRA workshop on open source software, Vol. 3, 2009, p. 5.

[4] M. Folk, A. Cheng, K. Yates, HDF5: A file format and I/O library for high performance computing applications, in: Proceedings of supercomputing, Vol. 99, 1999, pp. 5–33.

[5] S. Pütz, T. Wiemann, J. Hertzberg, Tools for visualizing, annotating and storing triangle meshes in ros and rviz, in: 2019 European Conference on Mobile Robots (ECMR), IEEE, 2019.

[6] A. Hornung, K. Wurm, M. Bennewitz, C. Stachniss, W. Burgard, Octomap: An efficient probabilistic 3d mapping framework based on octrees, Autonomous robots 34 (3) (2013) 189–206.

[7] A. Hornung, M. Phillips, E. Jones, M. Bennewitz, M. Likhachev, S. Chitta, Navigation in three-dimensional cluttered environments for mobile manip-

ulation, in: 2012 IEEE International Conference on Robotics and Automation, IEEE, 2012, pp. 423–429.

[8] L. Jiang, J. Smith, A unified framework for grasping and shape acquisition via pretouch sensing, in: 2013 IEEE International Conference on Robotics and Automation, IEEE, 2013, pp. 999–1005.

[9] M. Ciocarlie, K. Hsiao, E. Jones, S. Chitta, R. Rusu, I. Şucan, Towards reliable grasping and manipulation in household environments, in: Experimental Robotics, Springer, 2014, pp. 241–252.

[10] S. Chitta, I. Sucan, S. Cousins, Moveit![ros topics], IEEE Robotics & Automation Magazine 19 (1) (2012) 18–19.

[11] N. Blodow, L. C. Goron, Z. Marton, D. Pangercic, T. Rühr, M. Tenorth, M. Beetz, Autonomous semantic mapping for robots performing everyday manipulation tasks in kitchen environments, in: 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, 2011, pp. 4263–4270.

[12] D. Maturana, P. Chou, M. Uenoyama, S. Scherer, Real-time semantic mapping for autonomous off-road navigation, in: Field and Service Robotics, Springer, 2018, pp. 335–350.

[13] D. De Gregorio, L. Di Stefano, Skimap: An efficient mapping framework for robot navigation, in: 2017 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2017, pp. 2569–2576.

[14] F. Ruetz, E. Hernández, M. Pfeiffer, H. Oleynikova, M. Cox, T. Lowe, P. Borges, Ovpc mesh: 3d free-space representation for local ground vehicle navigation, in: 2019 International Conference on Robotics and Automation (ICRA), IEEE, 2019, pp. 8648–8654.

[15] Y. Hara, M. Tomono, Moving object removal and surface mesh mapping for path planning on 3d terrain, Advanced Robotics 34 (6) (2020) 375–387.

[16] S. Pütz, T. Wiemann, J. Sprickerhof, J. Hertzberg, 3d Navigation Mesh Generation for Path Planning in Uneven Terrain, IFAC-PapersOnLine 49 (15) (2016) 212–217, 9th IFAC Symposium on Intelligent Autonomous Vehicles IAV 2016.

[17] R. Drouilly, P. Rives, B. Morisset, Semantic representation for navigation in large-scale environments, in: 2015 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2015, pp. 1106–1111.

[18] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, K. Konolige, The Office Marathon: Robust navigation in an indoor office environment, in: 2010 IEEE International Conference on Robotics and Automation, 2010, pp. 300–307.

[19] D. V. Lu, D. Hershberger, W. D. Smart, Layered costmaps for context-sensitive navigation, in: Proceedings of the International Conference on Intelligent Robots and Systems (IROS), IEEE, 2014.

[20] P. Fankhauser, M. Hutter, A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation, in: A. Koubaa (Ed.), Robot Operating System (ROS) – The Complete Reference (Volume 1), Springer, 2016, Ch. 5.

[21] H. Deeken, S. Pütz, T. Wiemann, K. Lingemann, J. Hertzberg, Integrating Semantic Information in Navigational Planning, in: Proceedings of the International Symposium on Robotics (ISR), 2014.

[22] T. Foote, tf: The transform library, in: Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on, Open-Source Software workshop, 2013, pp. 1–6. doi:10.1109/TePRA.2013.6556373.

[23] D. Gossow, A. Leeper, D. Hershberger, M. Ciocarlie, Interactive Markers: 3-D User Interfaces for ROS Applications, IEEE Robotics & Automation Magazine 18 (4) (2011) 14–15.

[24] S. Pütz, 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), 2018.
URL https://github.com/magazino/move_base_flex

[25] J. Bohren, S. Cousins, The SMACH high-level executive, IEEE Robotics & Automation Magazine 17 (4) (2010) 18–20.

[26] T. Wiemann, F. Igelbrink, S. Puetz, J. Hertzberg, A File Structure and Reference Data Set for High Resolution Hyperspectral 3d Point Clouds, IFAC Symposium on Intelligent Autonomous Vehicles, 2019.

[27] Ogre: The core objects, https://ogrecave.github.io/ogre/api/latest/_the-_core-_objects.html, accessed: 2020-04-13.

[28] W. J. Schroeder, L. S. Avila, W. Hoffman, Visualizing with vtk: a tutorial, IEEE Computer graphics and applications 20 (5) (2000) 20–27.

[29] M. Hapala, T. Davidovič, I. Wald, V. Havran, P. Slusallek, Efficient stackless bvh traversal for ray tracing, in: Proceedings of the 27th Spring Conference on Computer Graphics, 2011, pp. 7–12.

[30] M. Folk, G. Heber, Q. Koziol, E. Pourmal, D. Robinson, An overview of the hdf5 technology suite and its applications, in: Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases, 2011, pp. 36–47.

[31] G. Pandey, J. R. McBride, R. M. Eustice, Ford campus vision and lidar data set, The International Journal of Robotics Research 30 (13) (2011) 1543–1552.

[32] Z. Xie, K. Xu, L. Liu, Y. Xiong, 3d shape segmentation and labeling via extreme learning machine, in: Computer graphics forum, Vol. 33, Wiley Online Library, 2014, pp. 85–95.

[33] M. Rouhani, F. Lafarge, P. Alliez, Semantic segmentation of 3d textured meshes for urban scene analysis, ISPRS journal of photogrammetry and remote sensing 123 (2017) 124–139.

[34] H. Deeken, T. Wiemann, J. Hertzberg, Grounding semantic maps in spatial databases, Robotics and Autonomous Systems 105 (2018) 146–165.

[35] W. Heidrich, Computing the Barycentric Coordinates of a Projected Point, Journal of Graphics Tools 10 (3) (2005) 9–12. doi:10.1080/2151237X.2005.10129200.

[36] R. Kimmel, J. A. Sethian, Computing geodesic paths on manifolds, Proceedings of the national academy of Sciences 95 (15) (1998) 8431–8435.

[37] J. A. Sethian, A fast marching level set method for monotonically advancing fronts, Proceedings of the National Academy of Sciences 93 (4) (1996) 1591–1595.

[38] T. Wiemann, F. Igelbrink, S. Pütz, M. Kleine Piening, S. Schupp, S. Hinderink, J. Vana, J. Hertzberg, Compressing ROS Sensor and Geometry Messages with Draco, IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR), 2019.