Attitude and Ocean Floor Visualization for a 6DOF Underwater Robot

Pierre-Luc Bacon
Supervised by Ioannis Rekleitis, Florian Shkurti

April 27, 2011

Abstract

This paper describes the design and realization of a graphical user interface for visualizing the 6DOF pose of an underwater robot as well as the visual features detected on the ocean floor. This work integrates with an onboard vision-aided inertial navigation system over the ROS meta-operating system. Based on the Visualization Toolkit (VTK), an efficient implementation of a Delaunay triangulation is used for estimating and rendering the ocean floor. The software architecture presented in this paper allowed the development of *navigation* widgets, but also lends itself easily to Heads-Up Display (HUD) setups.

1 Introduction

The problem of estimating the robot pose by a teleoperator is encountered in many different environments, whether it is on land, in the air, in space, or under the water as it is the case in the current project with the AQUA robot. Even though the AQUA robot is designed mostly to operate autonomously, a supervisory control mode is still provided under certain use cases. The name for this type of control arose from research in teleoperation of future missions on the moon [4] and refers to the situation where a teleoperator *supervises* the action of robot executing a sequence of subtasks.

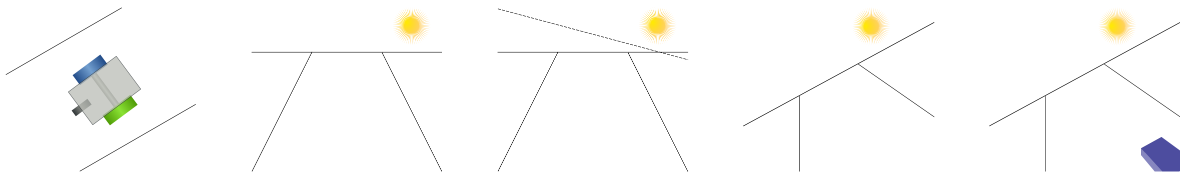
When a teleoperator loses the sense of the robot attitude, rollover or collisions might occur. This problem is even more present in most teleoperated systems where only an egocentric view of the robot is available : that is, a view as recorded by the onboard cameras on the robot. In the rescue mission that followed the collapse of the World Trade Center in 2001, it was reported that 54% of the time spent by teleoperators was attributed to recovery manoeuvres due to rollovers [2]. In the context of the AQUA robot, the loss of situation awareness is also experienced in supervisory control use cases. Indeed, troubled waters or sudden interruption of the data link to the land often cause control problems of the same kind.

Three main viewing modes can be recognized for teleoperated systems [6]:

1. Egocentric view: as if the teleoperator was *sitting* in the robot.
2. Exocentric view: viewing the robot from an outside perspective, as it is the case when controlling a RC plane with a view from the ground.
3. Mixed view: It can be an egocentric view augmented with information about the robot attitude from an exocentric perspective, such as an *artificial horizon* drawn on screen.

In the study [6], different viewing modes were compared with the goal of determining a cognitively efficient manner of representing the attitude information and reducing the risk of rollovers. To this intent, five variants of the three viewing modes presented above were proposed :

Results from this study showed that the egocentric view with gravity reference is the one that resulted in the best response time in terms of manoeuvrability from the teleoperators. However, it remains that



(a) Exocentric view from the top of a differential robot.

(b) Egocentric view.

(c) Egocentric view with artificial horizon

(d) Egocentric view with *gravity reference*

(e) Egocentric view with gravity reference and view of the chassis.

Figure 1: Different viewing modes of a robot for a teleoperator [6]

egocentric views are the most difficult to express because of the lack of environmental reference cues. This situation is often experienced by plane pilot loosing visual information because of difficult atmospheric conditions.

In this paper, an implementation and an architectural software design is presented to allow for the visualization of the robot attitude as well as the estimated ocean floor from a set of visual landmarks. More specifically, an exocentric view augmented with *widgets* informing about the current pose was implemented. However, architectural decisions were made so that different viewing modes could be implemented in the future while allowing the development more sophisticated widgets.

This work integrates with the results from a newly-developed vision-aided inertial navigation system for the AQUA robot at the Center for Intelligent Machines [1] at McGill University. This system mainly originates from the work in [7], and through an Extended Kalman Filter (EKF) allows for the correction of inertial measurements based on static visual features seen from different camera poses. The need for such a system is also motivated by the inherent difficulty to cope with noisy measurements from IMU sensors.

In this project, the requirements were not only to display the estimated robot pose in 3D, but also the static visual features used in the EKF system. Furthermore, the software design had to integrate with the ROS meta-operating system to which the existing code base for the robot is currently being ported.

A C++ object-oriented, STL-compatible and extensible design was developed based on the Visualization Toolkit (VTK) and the QT library. The issue of estimating the terrain from the visual features was addressed by a Delaunay triangulation, provided by the VTK library.

2 The AQUA Robot

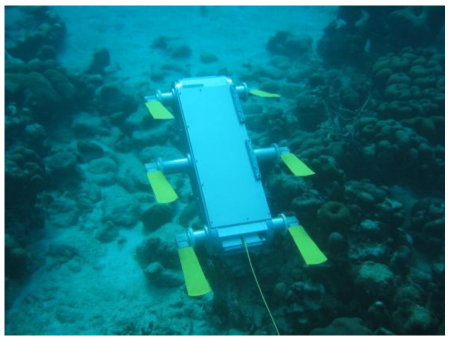


Figure 2: The AQUA robot.

[1]http://www.cim.mcgill.ca/

The AQUA robot is the aquatic evolution of the 6-legged amphibious RHHex robot developed at McGill University. Rather than using traditional propulsion systems, its has six flippers making the robot capable of 6 degrees of freedom under the water.

The AQUA robot is mainly targeted for Site Acquisition and Scene Re-inspection (SASR) applications [5]. A typical usage scenario would involve the supervisory controlled task of collecting sensory information on a remote site. The teleoperator would indicate the location to inspect to the robot, which would then navigate to the goal, collect sensor measurements and return home autonomously. The collected data can be of varying nature such as water contaminant or fish stocks for example.

The AQUA robot has 3 onboard cameras, initial sensors, and 2 onboard processors. One of those processors handles the *Control Stack* : a set of hardware drivers, and gait controllers. The other board supports the *Vision Stack* performing most of the sensory analysis based on the vision sensors. The *control stack* runs a QNX operating system and uses the RobotDevel C++ library, whereas the other one is based on Linux. The *Vision Stack* accesses the *Control Stack* via a custom made UDP protocol. In is important to note that the current architecture is being ported to the ROS meta-operating system and does not reflect precisely the state of the new architecture.

The robot can be either run in supervisory control through fibre optics or gestural control, or in a completely autonomous manner.

3 The Robot Operating System

The Robot Operating System (ROS) [8] was developed at Standford University as part of the STAIR project [2] and the Personal Robots Program [3] by Willow Garage. ROS is not an operating system in the traditional sense of the term but rather a so-called "meta-operating" system. To be more precise, it is made of a set of tools and a communication framework that allows for the rapid development of processing components called *nodes*.

The driving philosophy behind the project is to favor code re-usability and a well-defined separation of concerns. The problem of making hardware drivers easily accessible outside of a given framework is also mentioned as one of the motivation for adopting such an architecture [8]. With the notion of nodes, highly specialized features can be developed more easily by independent teams without having to worry about the final integration into the complete system.

Incidentally, the ROS system is similar in many ways to a micro-kernel architecture. A minimal amount of functionality is put into a core, and supporting services are distributed into independent nodes : configuration, topology querying, logging, etc. Also, the communication between nodes does not occur using procedure call but rather via a custom IPC mechanism providing two main connectors : a stream connector for the TCPROS or UDPROS protocol, and an asynchronous *signaling* XML-RPC connector.

Because of this socket-based communication paradigm, the idea of having a central *master* node through which all communication would take place was rejected. Indeed, ROS rather defines a peer-to-peer topology, eliminating that way any single point of failure.

One motivation for having such distributed architecture is to allow for offboard processing of cpu-intensive tasks that would otherwise not be possible to carry on embedded hardware. Clearly, the decision of avoiding to have a central node rules out the issue of network congestion to that node and routing overhead. Hence, the throughput and response time from a servicing node is also increased by this peer-to-peer topology.

ROS distinguishes between two type of communication : topic-based communication, and service-based communication. The former type of communication is similar to one adopted by the *message queue* type of software architecture, and comply more generally to the *publisher-subscriber* model.

A *topic* is a name that refers to a particular type of information that is being broadcasted by one or more *publisher* nodes. *Subscriber* nodes then register or *subscribe* to the given topic and get notified asynchronously of any new information being broadcasted under it. In general, the publisher-subscriber extensibility stragety aims to assign the task of events notification on the server side (the publisher), rather than leaving the clients

[2]http://stair.stanford.edu/
[3]http://pr.willowgarage.com/wiki/

(subscribers) to discover those new events themselves via polling. It also allows for a greater scalability since an application can be extended more easily by adding new subscriber components without any change in the publisher. Clearly, this approach favors very low coupling, and make the subscribers *anonymous* to the publishers.

Because of the overhead that comes with broadcasting events in the topic-based approach, ROS also defines the notion of *services*. A service is also identified by a name but can only be handled by one servicing node at a time. Furthermore, services make synchronous communication possible between communicating nodes in a way that resembles to HTTP REST architectures. Clearly with this idea, we cannot avoid drawing parallels between *distributed objects* architecture of the kind of CORBA for example [4]. Messages sent from a client node to a servicing node encode the remote service (function) that has to be called along with their parameters. The servicing node then replies back with a serialized response in a standardized format. The roscpp project [5] provides automatic code generation in C++ so that to avoid the burden of handling the low level mechanics that comes with the implementation of the different ROS protocols. As in other *distributed objects* systems, interfaces are defined for the bound objects using the interface definition language (IDL). From this definition is then created what would be recognized as an object request broker (ORB) in other systems. With this component, when a remote call is being made the client code first goes through a *stub* before reaching the ORB component via which the marshalling and unmarshalling process can take place.

Finally, ROS defines the notion of a parameter server that lives on the master node. Even though the communication *per se* does not occur through this node, the master node provides the necessary logic for *naming* and client registration to topics. Thus, the master node can be recognized as the directory service provided by most *distributed object* systems. In addition to that, the master node can also hold configuration parameters available to the nodes.

By this flexible architecture, ROS provides a language-agnostic framework in which existing components from other project can be easily ported. To that account, it already draws from the Player and OpenCV projects for example.

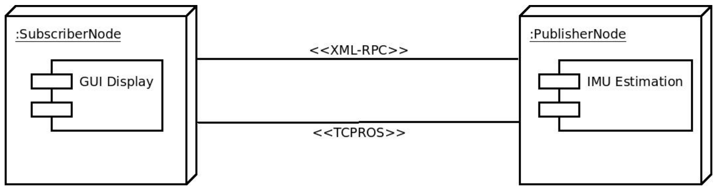


Figure 3: Deployment diagram

Figure 3 depicts the deployment diagram of the system that was developed in this project. It consists of two main ROS nodes : the GUI Display and the IMU Estimation node. The two components are communicating together over the pair of protocols formed by XML-RPC and TCPROS. The details of implementation for those protocols were being hidden by the roscpp library. During development, the two nodes were running on the same physical machine and the IMU Estimation node was actually only replaying pre-recorded data. However, the final deployment might take place on different machines, separated by hundreds of kilometres without any change in the code.

4 The Visualization Toolkit

The Visualization Toolkit (VTK) is an object oriented C++ framework for 3D visualization complemented with Tcl/Tk bindings [9]. Over the years, bindings for other languages were developed independently in the community (Java or Python for example).

[4] http://www.omg.org/
[5] http://www.ros.org/wiki/roscpp

VTK proposes a graphical model in nine fundamental principles mapping to an equivalent object-oriented representation. This model comprises of :

1. A *renderer master* that hides the device-dependent rendering code and facilitate window management under different environments.
2. A *render window* in which a *renderer* draws the scene (the resulting rendered image).
3. A *renderer* component acting as a coordinator between the light, camera and actor components of a scene.
4. One or more *lights* components which illuminate the components of a scene
5. On or more *actors* representing the graphical objects of a scene
6. *Property* components containing the rendering attributes of an actor.
7. *Mapper* components holding the relationship between an actor to its corresponding geometric representation in terms of geometrical features such as cells, lines or triangles.
8. Geometrical *transforms* defined by a 4x4 transformation matrix containing the 3x3 rotation matrix and the translation vector in the fourth column.

VTK relies on a pipe and filter architecture for its data processing model. This data-flow paradigm favors low coupling between the constituting *filters* of a processing *pipeline*. Furthermore, an open-ended set of processing pipelines can be created from those basic building blocks, and thus can handle unforeseen modifications more easily than other designs.

Every processing pipeline is made of the specials filters that are the *source* and *sink* filters. A source filter receives no incoming data from any downward filter but rather produces the data by itself for the upstream ones. Similarly, a sink filter receives a data set but is not connected to any other filter on its *source* port.

An example of such pipeline is the one in figure 7.

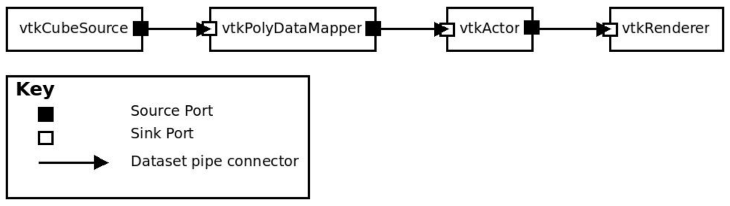


Figure 4: Example of a data processing pipeline in VTK

VTK enforces a proper inter-filter compatibility through static typing in C++. Therefore, it eliminates possible misconnections from the source port of a filter to the sink port of another. A connection between two filters is established with the `SetInput()` and `GetOuput()` methods of a filter, and through which type checking can occur.

Data processing pipelines in VTK can either be *event* or *demand-driven*. In the latter case, a filter that needs to *pump* a larger dataset will trigger automatically the execution of the pipeline. Each filters of the pipeline will then be recursively updated in a two-phase procedure. In the first one (the *update*) the modification times that every filter maintains is compared at the source and sink ports of two filters. If the upstream component has a modification time older than the downstream one, a flag will be set in order to recompute the data in the next phase which essentially consists of the recursive execution of those filters.

In the event driven model, this two-phase procedure still holds but is triggered asynchronously by an event. To that account, VTK defines an object model where every object inherits from an abstract `vtkObject`. This one contains, among other features, the necessary code for acting either as a *subject* or an *observer* of the observer pattern. That way, every filters has the ability to register observer components and notify them asynchronously.

Finally, at the time where VTK was introduced, the need for a better resource management was required due to the state of the C++ language at that time. More precisely, a reference-counting approach was taken for memory deallocation with the `vtkSmartPointer` class. In the upcoming C++0x, the `std::shared_ptr` class template essentially provides the same RAII functionality.

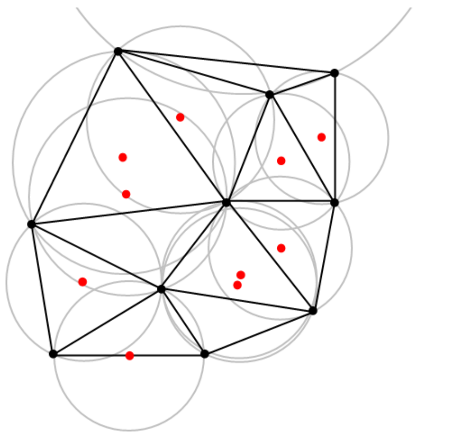
5 Quaternions

The quaternions is a class of the hypercomplex numbers [10] that can represent rotations by an angle around a unit vector. Such representation is preferred to Euler angles represented by three parameters (sometimes called *roll*, *pitch* and *yaw*) but suffer from Gimbal locks : the difficulty in representing certain rotations uniquely.

Rotations through quaternions can be defined in terms of $\langle x, y, z, w \rangle$ and under a well-defined algebra for quaternion multiplication and inversion. However, in the context of [7], a different formulation was used, namely the one recognized as the *JPL* formulation [1]. Under this convention, quaternions are transformed to 3 $\times$ 3 rotation matrices in the following way :

$$M = \begin{bmatrix} q_4^2 + q_1^2 - 1/2 & q_1q_2 + q_4q_3 & q_1q_3 - q_4q_2 \\ q_2q_1 - q_4q_3 & q_4^2 + q_2^2 - 1/2 & q_2q_3 + q_4q_1 \\ q_3q_1 + q_4q_2 & q_3q_2 - q_4q_1 & q_4^2 + q_3^2 - 1/2 \end{bmatrix}$$

6 Delaunay Triangulation


(a) Delaunay triangulation and circumcircles of the triangles.

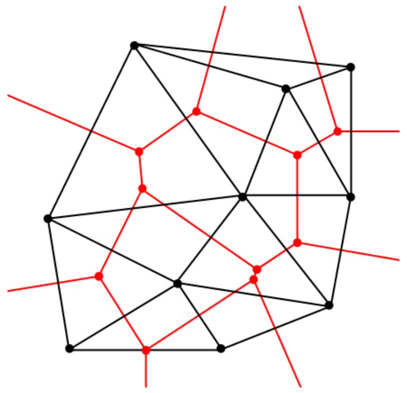

(b) The Delaunay triangulation is the dual of the Voronoi graph (show in red)

Figure 5: Delaunay triangulation of a set of points. Figures obtained under GNU Free Documentation License.

The Delaunay triangulation was named after the Russian mathematician Boris Nikolaevich Delone and was formulated in 1934. From a set of points P, it is defined in such a way the triangulation does not have any circumscribed circle in which a point of P lies. Therefore, the Delaunay triangulation is a planar subdivision with triangular faces and vertices as points.

It can also be seen as the dual graph of the Voronoi graph of the set of points P. The Voronoi diagram is a subdivision of the plane into n regions corresponding to the points of P. This subdivision has the property that for any point p (called *site*) of P, the cell that contains it also contains every other sites of P for which p is closer than any other sites.

Even though the triangulation only takes place in the x-y plane, a given vertex of the triangulation can simply be elevated to the z value of the corresponding point. This approach can be used for computing the terrain approximation of a set of points. The resulting piecewise linear surface is called a *polyhedral terrain* [3].

7 Architecture

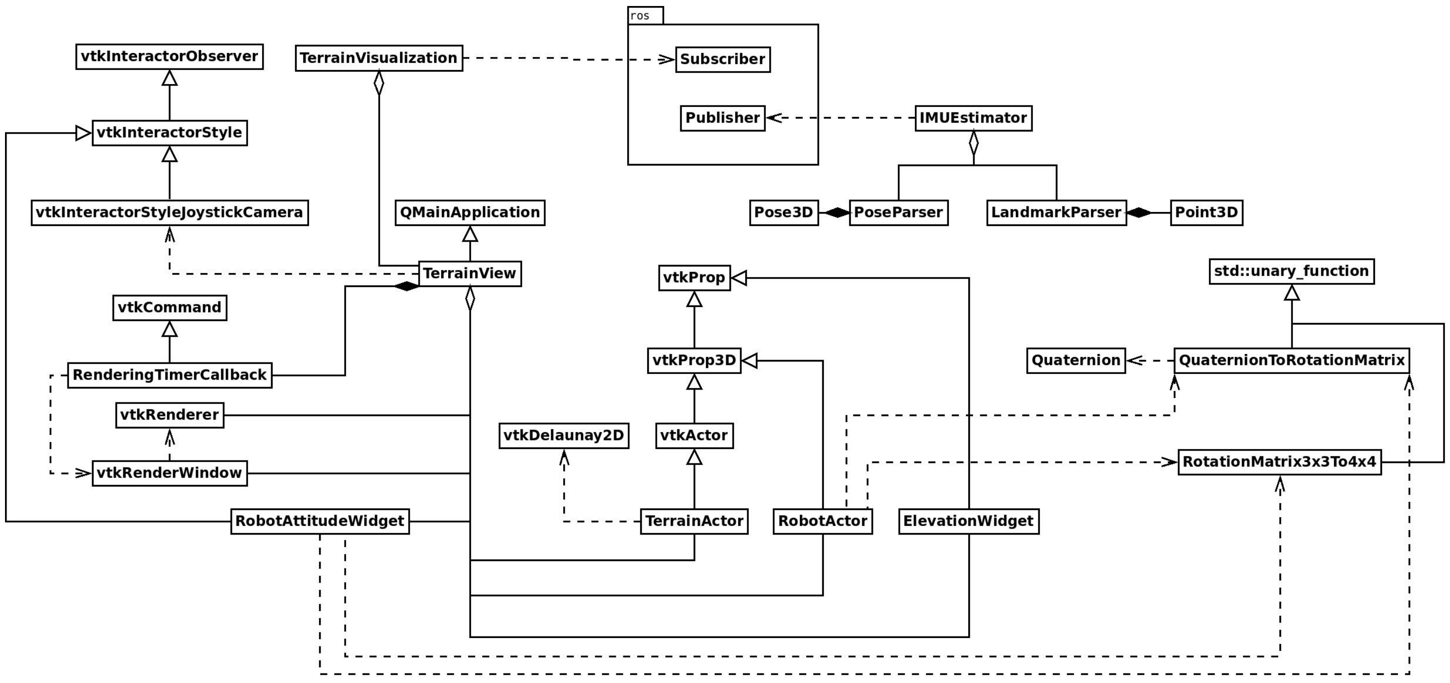


Figure 6: Class diagram depicting the software architecture for this project.

7.1 Continuous Rendering

Reading the VTK manual does not give much information about those use cases where rendering must occur in a continuous fashion. In fact, most examples refer to the canonical examples in the field of medical visualization, requiring only one rendering pass.

In this project, pose and landmarks information was expected to be received at a rate of approximately 10Hz in an asynchronous manner. Clearly, the processing pipeline would have to be refreshed more than once during the lifetime of the application. Intuitively, it would be expected that under the pipe and filter model adopted by VTK, *pushing* data into a source component would suffice to trigger the processing components upstream. However, only a call to the `Render()` method of the sink component can achieve the asynchronous refresh of the whole pipeline. Indeed, doing so will cascade the `Render()` method call to all the filters recursively.

Clearly, one would again be tempted to call the `Render` method directly on the `vtkRenderWindow` within a callback from the ROS subsystem. Here again, an exeption must be made because the underlying OpenGL library disallows concurrent access to the rendering process from an external thread of execution (the one in which `vtkRenderWindow` runs in).

The only solution that comes to that problem is through the use of *timer* object running in the same execution thread and that will call `Render()` at a constant rate. This is achived through the `vtkRenderWindowInteractor` associated with an instance of a `vtkRenderWindow` object. As all other `vtkObject`, the `vtkRenderWindowInteractor` inherits from an implementation of the *observer* pattern. Through the `CreateRepeatingTimer()` method, the user then creates a timer managed within the `vtkRenderWindowInteractor`. At a fixed interval, notification on the list of observers for the pre-defined `vtkCommand::TimerEvent` then occurs.

The `RenderingTimerCallback` (defined as a private anonymous class) subclasses `vtkCommand` so that it can subscribe to the `vtkCommand::TimerEvent` event on `vtkRenderWindowInteractor`. Subclasses of `vtkCommand` must then implement the `void Execute(vtkObject*, unsigned long, void *)` virtual method.

In the current implementation, the timer was set to fire every 100ms – which should be enough considering the expected data reception at 10Hz. In order to avoid unnecessary computation, but also concurrent update on the *filters*, a mutex was defined over the rendering process.

Note that VTK already provides an optimization strategy on all its `vtkProp` based on the use of timestamps, managed by the component itself. With this idea, a component is not redrawn unless its current timestamps is expired. The use of a render lock complements this mechanism most notably in the case of the Delaunay triangulation. Upon receiving a set of landmarks, we avoid recomputing a partial triangulation and force the rendering to take place after all points have been added.

7.2 Transformations

Since the quaternion formulation used in this project differs from the one provided by VTK, some additional classes had to be designed in order to carry the necessary computation.

Due to its nature, it was chosen to make the `QuaternionToRotationMatrix` class a subclass of `std::unary_function`. By this design decision, we favor interoperability with the STL library. For example, `std::transform` could be used together with `QuaternionToRotationMatrix` to transform a `std::vector` of quaternions and produce another *vector* of matrices as output in only one statement.

When subclassing from `std::unary_function`, the `operator()` function must be implemented in order to preserve the expected *functor* semantics. In `QuaternionToRotationMatrix`, `operator()` receives a quaternion as an argument and returns a 3x3 matrix. To better comply with an object-oriented view of the problem, a `Quaternion` class was defined to hold the defining $\langle x, y, z, w \rangle$ variables. Furthermore, an `invert()` function and `leftMultiply()` (together with the operator overload on `operator*()`) were also added.

In order to facilitate matrix manipulation, the *Ublas* library from the *Boost* project was used. Because of our use of roscpp and its existing dependency on Boost, this choice did not have the undesirable effect of bringing a new depency.

That way, `QuaternionToRotationMatrix` was made to accept a `boost::numeric::ublas::vector<double>` and produce a `boost::numeric::ublas::matrix<double>`. Built-in functions such as `outer_prod`, `operator*()`, `operator+()`, `operator-()` or `trans` (transpose) can then be used over those objects.

The resulting 3x3 rotation matrix after the quaternion conversion is not directly compatible with VTK. Indeed, only VTK 4x4 matrices are accepted by the `vtkTransform` class. Therefore, the 3x3 matrix is simply converted to a 4x4 matrix by the `RotationMatrix3x3To4x4` functor object, and setting no translation into the fourth column.

The sequence of transformations to be applied on an actor is carried through the `vtkTransform` class. This one contains a 4x4 matrix initially set to the identity matrix. After instantiation, the multiplication semantics must be manually set to *post-multiply* with a call to `PostMultiply()` and thus changing the default *pre-multiply* semantics. Since VTK defines its own matrix classes, we must also transform the Ublas matrix to its corresponding `vtkMatrix4x4` counterpart. In this order, `Translate(double[3])`, `Concatenate(vtkMatrix4x4*)`, and `Translate(double[3])` again are finally called on the `vtkTransform` object : effectively translating the robot back to the origin, applying the rotation, and translating the robot again to its original position before the rotation. At every step, the transformation is concatenated to the initial identity matrix by multiplication from the right inside `vtkTransform`.

The resulting transform can be then specified to a `vtkProp3D` with the `SetUserTransform()` method. After having called `Modified()` on that *prop*, the rendering timer will trigger `Render()` on it in the next cycle.

7.3 Widgets

Subclassing from the `vtkProp` class has been used in order to encapsulate the logic for the three main widgets developed in this project : `ElevationWidget`, `TerrainActor`, `RobotActor`, and `RobotAttitudeWidget`.

7.3.1 RobotActor

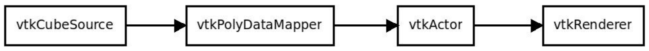
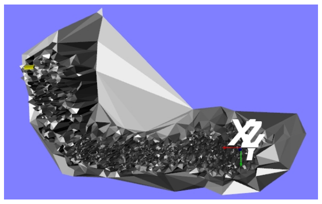


Figure 7: Connector and component view of the processing pipeline for the RobotActor class

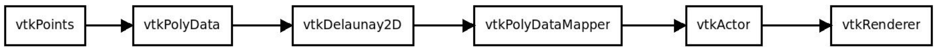
The `RobotActor` class is a simplified representation of the robot in the 3D space. At the time of writing, a `vtkCubeSource` was used for this modelling (see figure 7). However, it is important to note that due to the subclassing strategy adopted in this design, a robot model based on a 3D input data file could as well be used. Indeed, it would suffice to use one of the subclasses of the `vtkImporter` class (such as `vtkXMLPolyDataReader`) instead of the `vtkCubeSource` and keep the same data processing pipeline upstream, piping data into a `vtkPolyDataMapper` and a `vtkRenderer`.

In order to facilitate refinement of the `RobotActor`, the `vtkAssembly` class was used. This one allows for the grouping of `vtkProp3D` objects so that they can be treated as one entity, effectively implementing a *composite* design pattern. That way, when the resulting composite `assembly` objects gets transformed (rotation, scaling, translation), all its parts are recursively modified by the same transformation. Notice that VTK also provides a `vtkPropAssembly`, but this class only provides grouping of `vtkProp` objects without performing any transformation on its *leaves*. This choice of using a `vtkAssembly` was motivated by the vision of a more informative widget, displaying additional navigational controls around the model. It its current state, `RobotActor` does not push this idea further, even though its design permits it. Nonetheless, `vtkAssembly` made the task of sublassing from `vtkActor` easier, implementing all its virtual render methods simply by forwarding them to the corresponding render method of the `vtkAssembly`.

7.3.2 TerrainActor



(a) Top view of a complete Delaunay triangulation.

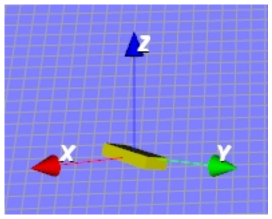


(b) Connector and component view of the processing pipeline for the Delaunay triangulation of the landmarks.

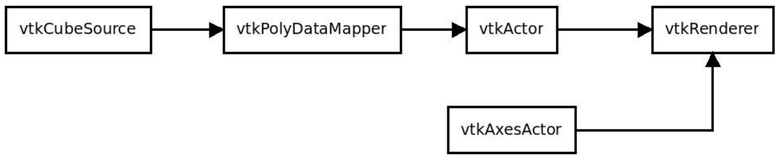
The necessary logic for computing the Delaunay triangulation over the set of landmarks was chosen to be contained in a dedicated `vtkActor` mainly to facilitate future changes in the implementation of the triangulation approach.

The portion of the pipeline upstream remains similar to the other widgets. However, it differs from its `vtkPoints` source in which are inserted those landmarks received over ROS. This class is only provided by VTK to hold 3D points and facilitate their access afterwards. This set of points is then refined in the next component with `vtkPolyData` in order to get organized as a more coherent geometric structure on which the notions of *cells*, *lines* and *triangles* are defined. The `vtkDelaunay2D` filter then reconstructs the triangulation in the x-y plane based on this incoming data and produces a polygonal dataset on its source port.

7.4 RobotAttitudeWidget



(c) The RobotAttitudeWidget is rendered in a fixed viewport in the lower left corner of the screen.



(d) Connector and component view of the processing pipeline for the RobotAttitudeWidget.

This widget was meant to provide a precise visual feedback on the current robot pose. A `vtkAxesActor` was used together with a `vtkCubeSource` to create a widget that stays in the lower left corner of the display. In order to make the widget independent from the camera pose in the main render window, a separate `vtkRenderer` had to be used.

Furthermore, the position of the new renderer must be specified through the `SetViewport()` method on the `vtkRenderer` class. Since two renderers are then used at that point, the `SetLayer()` method must also be used to ensure that the new widget appears transparent on top of the other.

7.5 Parsing

Throughout this project, it appeared of prime importance that we could easily replay the robot trajectory in the virtual 3D environment from a file. That way, previous on-field experiments could be analyzed later in the lab using this tool.

The dataset for this project was available in two different files : `poses.txt` and `landmarks.txt`. Both were space-delimited files with each line corresponding to a different sample. The pose information was formatted in the following way under 31 columns :

1-4 The 4-tuple $\langle x, y, z, w\rangle$ corresponding the quaternion that represents the rotation from the global frame to the IMU frame.

5-7 The bias of the gyroscope

8-10 The velocity of the IMU frame in global coordinates (meter/sec)

11-13 The bias of the accelerometer

14-16 The position of the IMU frame in global coordinates (meter)

17 Timestamp

18-20 The acceleration of the IMU frame in global coordinates (meter/sec^2)

24-31 Ignored here.

32 Unique key for this row.

The landmarks were kept in separate file where the first 3 columns were forming the $\langle x, y, z \rangle$ tuple and the last one was a key, referencing to a line of `poses.txt`.

In order to make the access to the data entries easier, the iterator pattern was used. Indeed, both `LandmarkParser` and `PoseParser` (see figure 6) expose the data through a `const_iterator` from the STL library. This iterator gives access to `std::pair<unsigned, Poses>` or `std::pair<unsigned, Points>` where the first element is the key for a given *line*, contained in the customs types `Pose3D` and `Point3D` respectively. To make tuple retrieval more direct, a `find()` method was also defined in those parsers, delegating the method call to the corresponding `find()` method on a `std::map` container.

Considering the size of the dataset, this solution based on the standard STL containers seems reasonable. Also, since those containers are backed by Red-Black trees, we can think that after more than 20 years of refinement, the resulting implementation is highly efficient. If the dataset ever grow bigger, a solution based on SQLite might be considered for an easier and more efficient access to the data.

7.6 roscpp

roscpp is a high performance c++ library that provides an API to the ROS system and its *topics*, *services* and *parameters*.

The first step in using roscpp is to execute the `ros::init` function receiving as an argument the identifying name to be used in the ROS system. One must then obtain a `ros::NodeHandle` which is a class that allows for the RAII programming idiom to take place. For example, the RAII technique ease the memory reclamation when an exception is encountered : instead of spreading the memory deallocation logic accross the code, the RAII object will handle this taks. This concept is identitcal to the one behind `std::auto_ptr` or `vtkSmartPointer` for instance. `ros::NodeHandle` also provides namespace resolution so that references to nodes can be implicitely mapped to the declared namespace.

Once the `ros::NodeHandle` is initialized, *publisher* nodes can be created through the `advertise()` method. This one will return a `ros::Publisher` node for topic name and queue size that were provided. In our case, it was decided to reuse two of the buit-in topic names in ROS, namely `geometry_msgs::PoseStamped` and `visualization_msgs::MarkerArray`. This choice was motivated by the idea of maintaining compatibility with the Rviz visualization tool independently developed as part of the ROS project.

Using the newly-created `ros::Publisher` objects, we can finally publish the information through the `publish()` method on each of them, taking as an argument the appropriate topic type.

The `visualization_msgs::MarkerArray` structure contains many members informing in a great detail about each *marker* in a scene. In the context of this project, only a $\langle x, y, z \rangle$ tuple is necessary for each landmarks. Thus, even though only three members of this structure end up being used in our application, we still have to fill all its members in the following way :

```
markers_msg.markers[i].header.frame_id = "/feature";
markers_msg.markers[i].header.stamp = ros::Time::now();
markers_msg.markers[i].ns = "basic_shapes";
markers_msg.markers[i].id = 0;
markers_msg.markers[i].type = visualization_msgs::Marker::POINTS;
markers_msg.markers[i].action = visualization_msgs::Marker::ADD;
markers_msg.markers[i].pose.position.x = it->x;
markers_msg.markers[i].pose.position.y = it->y;
markers_msg.markers[i].pose.position.z = it->z;
markers_msg.markers[i].pose.orientation.x = 0.0;
markers_msg.markers[i].pose.orientation.y = 0.0;
markers_msg.markers[i].pose.orientation.z = 0.0;
markers_msg.markers[i].pose.orientation.w = 1.0;
markers_msg.markers[i].scale.x = 1.0;
markers_msg.markers[i].scale.y = 1.0;
markers_msg.markers[i].scale.z = 1.0;
markers_msg.markers[i].color.r = 0.0f;
markers_msg.markers[i].color.g = 1.0f;
markers_msg.markers[i].color.b = 0.0f;
markers_msg.markers[i].color.a = 1.0;
markers_msg.markers[i].lifetime = ros::Duration();
```

On the other hand, `geometry_msgs::PoseStamped` is less cumbersome to define as all its six members are necessary :

```
msg.header.frame_id = "/attitude_pose";
msg.pose.position.x = pose.position.x;
msg.pose.position.y = pose.position.y;
msg.pose.position.z = pose.position.z;
msg.pose.orientation.x = pose.quaternion.x;
msg.pose.orientation.y = pose.quaternion.y;
msg.pose.orientation.z = pose.quaternion.z;
msg.pose.orientation.w = pose.quaternion.w;
```

where `pose` is an instance of the `Pose3D` type shown in figure 6.

The code for the subscriber node class `TerrainVisualization` (see figure 6) follows the same first steps. However, the `ros::NodeHandle` is used to return an instance of a `ros:Subscriber` node for the given topic names published by the `ImuVisualization` class.

A particularity of this design has to do with the integration of roscpp within a QT application. Indeed, we have that the usual approach for handling callbacks in the subscriber nodes is to use the `ros::spin()` function. However, this function has a blocking behavior, looping indefinitely in order to catch new events on a given topic. Since the QT framework also requires a blocking call on `QApplication::exec()`, the two systems clearly cannot reside in the same thread of execution. Fortunately, roscpp provides a `ros::AsyncSpinner` class that spawn a new thread to execute the event loop. Then a call to `ros::AsyncSpinner::start()` will have the same intended effect but will allow the application to enter the event loop for the QT library.

8 Conclusion and Future Work

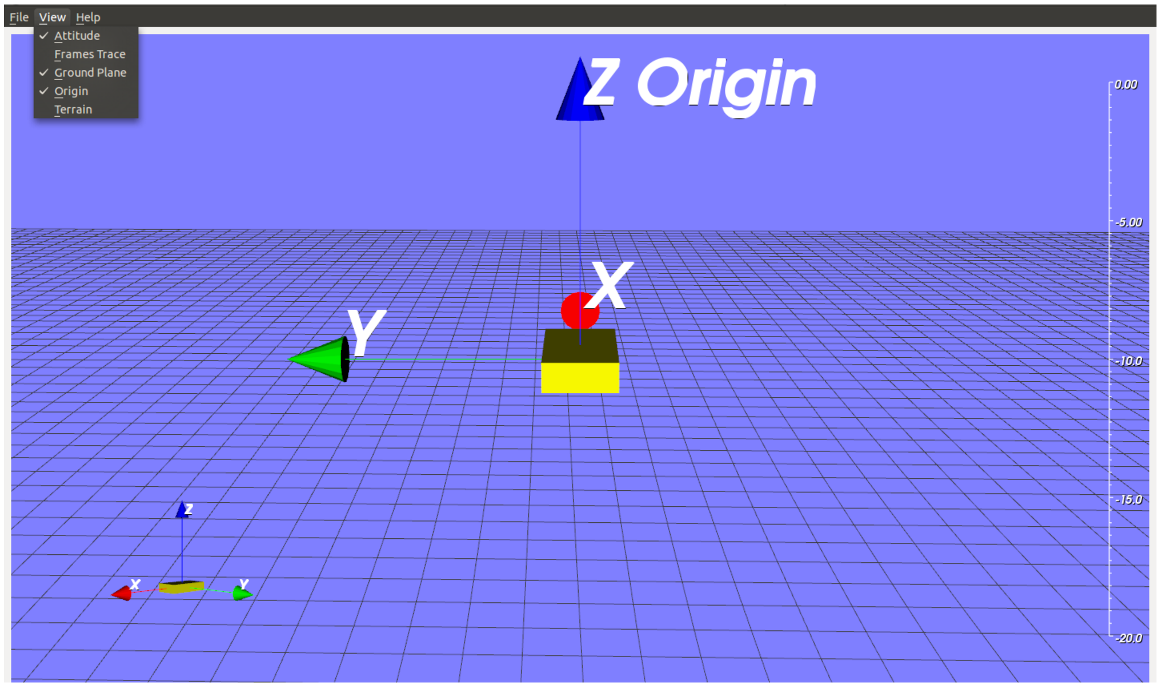


Figure 8: The graphical user interface developed in this project. Multiple visualization options are available from the **View** menu. This picture shows the robot in its initial pose, before Delaunay triangulation, and any quaternion-based rotation.

For this project, an exocentric graphical user interface was developed for representing the robot pose and the estimating Delaunay triangulation for the static landmarks used by the EKF algorithm (see figure 8)

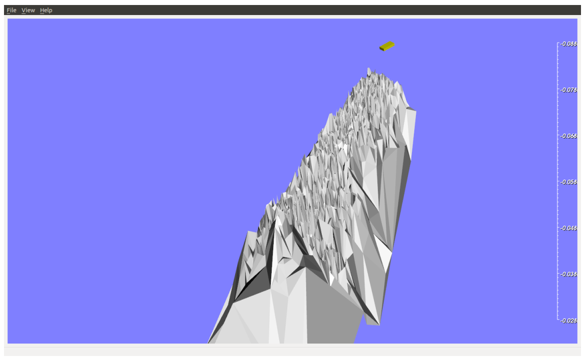
Basics widgets were also developed : a *RobotAttitudeWidget*, *RobotActor*, *TerrainActor* and *ElevationWidget*. The navigational widgets can be enabled from the **View** menu as well as the Delaunay triangulation.

The result of the Delaunay triangulation did provide an appropriate representation of the terrain (see figure 9(a)). However, due the nature of those points that are being estimated from the vision sensors, their variance is high and even after onboard filtering, many outliers remain. Thus, before spending more efforts into better terrain estimation algorithm, a more efficient outliers detection would be needed. A simple approach in order to get a better surface might consist of smoothing the terrain by considering a local neighbourhood. Preliminary experiments with the `vtkSurfaceReconstruction` class based on the PhD work of Hugues Hoppe [6] has however shown that the Delaunay triangulation was still the best approach (see 9(b)).

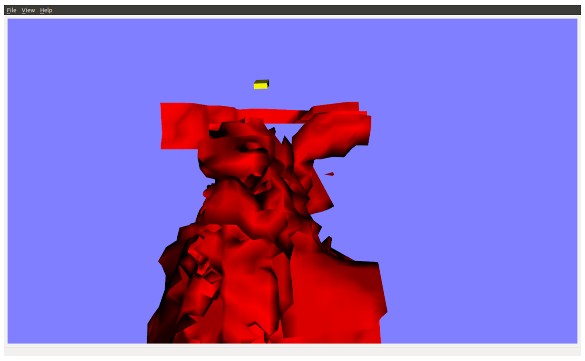
After few minutes of utilization, a performance issue was noticeable because of the Delaunay triangulation on the updated set of points. Note that the implementation provided by VTK does not recompute the whole triangulation after insertion of a new point. However, at the rate at which the simulation was performed (which will most likely be higher than the actual onboard final setup), GUI usability would suffer. Future work on this project might try to look at ways to remove, or simplify the input data set to the triangulation. This might involve carrying a pre-processing phase for outliers detection as reported above, automatic removal of older points, or selective rendering of the part of the scene that is currently being shown.

Development of an egocentric mode might help solving that problem by reducing the amount of points that have to be displayed at any moment in time. With quaternion conversion and geometrical transformations

[6]http://www.research.microsoft.com/-hoppe



(a) Delaunay Triangulation



(b) Surface Reconstruction

developed in this project for moving the robot actor in the scene, the step towards towards that goal is small. Indeed, the same transformation and code will be reusable, this time by applying it to the camera object of VTK.

The choice of using VTK as the underlying graphical toolkit turned out to be beneficial in terms of code quality, but also for development time. Furthermore, contributions from the community are already available for supporting touch and gesture control on interactive surfaces or tablets [7], and support for heads-up display setups is already built-in. In terms of performance, VTK also started to provide GPU acceleration through the CUDA technology.

The learning curve for VTK turned out to be steep. Fortunately, developers benefit from a rather large number of small example programs and a clear, but maybe too vague, class documentation. Investigating the source code directly sometimes appears to be more efficient. During development of the simple widgets presented here, it appeared also that the subclassing mechanism was not being made easy by the framework. Indeed, macros need to be used in order to make the subclasses instantiable from the VTK object factory system. This boilerplate code is cumbersome to use, and makes development more difficult.

However, the architecture developed in this project would still allow efficient development of navitational widgets. An optimal solution to be worked out in the future would consists of the upcoming `QVTKWidget2` class supporting the `QGraphicsView` graphic model from QT. With this combination, it would be possible to design widgets in QT, but overlay them in the VTK viewport. That way, the stability, performance, and rich algorithmic ecosystem provided by VTK could be combined with the qualities of the QT framework. With the new QML declarative language for QT, widgets creation could be simplified to only a couple lines of highly human-readable JavaScript-based code.

References

[1] *Quaternions – Proposed Standard Conventions.* JPL Interoffice Memorandum IOM 343-79-1199, 2 edition, October 1971.

[2] Jennifer Casper, Robin Murphy, and Dr. Robin Murphy. Human-robot interactions during the robot-assisted urban search and rescue response at the world trade center. 33:367–385, 2003.

[3] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, second edition, 2000.

[4] TerrenceW Fong. *Collaborative Control: A Robot-Centric Model for Vehicle Teleoperation.* PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, November 2001.

[7]svn://scm.gforge.inria.fr/svn/parietal-public/Utilities/VTKMultiTouch

[5] Christina Georgiades, Andrew German, Andrew Hogue, Hui Liu, Chris Prahacs, Arlene Ripsman, Robert Sim, Luz-Abril Torres, Pifu Zhang, Martin Buehler, Gregory Dudek, Michael Jenkin, and Evangelos Milios. The aqua aquatic walking robot. In *Proceedings of the IEEEE/RSJ/GI International Conference On Intelligent Robots and Systems (IROS)*, pages 3525–3531. IEEE Press, 2004.

[6] M. Lewis and J. Wang. Gravity-referenced attitude display for mobile robots: Making sense of what we see. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 37(1):94–105, 2007.

[7] Anastasios I. Mourikis and Stergios I. Roumeliotis. A multi-state constraint kalman filter for vision-aided inertial navigation. In *in Proc. IEEE Int. Conf. on Robotics and Automation*, pages 10–14, 2007.

[8] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[9] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization, 1996.

[10] Eric W Weisstein. Quaternion, April 2011. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/Quaternion.html`.