The Stallion

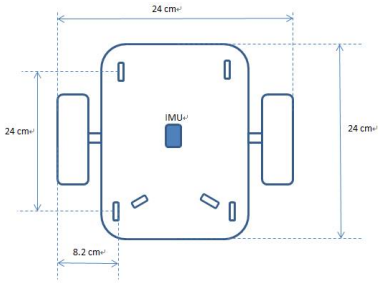
Group 9 Project Report
DD2425

TOBIAS ANDERSSON
MAX LOSCH
DIEGO MARTINEZ MARRODAN
TIAGO SEBASTIAO
LAN WANG
Royal Institute of Technology
December 16, 2014

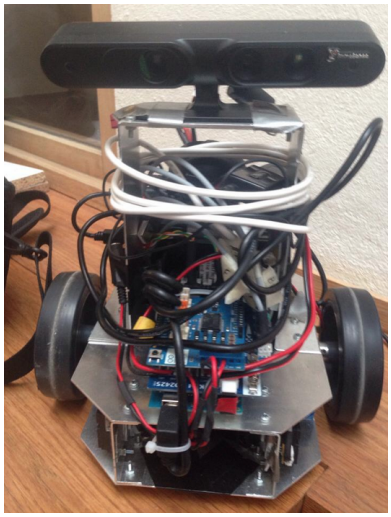
Abstract

This report describes the design decisions, accomplishments and evaluation of the work done by Group 9 during the course DD2425 Robotics and Autonomous Systems. In summary the robot uses an occupancy grid and a topoloical map to navigate in the maze, and the 3D information from the primesense camera together with PCL for object recognition. In the end the system worked well enough to solve the whole task in simple cases.

Contents



(a) Top View and Sensor Placements



(b) Front

Figure 1: Robot

1 About The Robot

1.1 Robot Structure

The construction of our robot took about 1.5 weeks. We managed to make the robot structure as compact as possible, so that it takes minimal space within the maze. The dimensions of the robot can be seen in figure 1a.
Since the robot is constructed with the same length and width of 24cm, when it turns, it will never hit the front wall as long as there was space left in front before it turns. The robot has two layers. On the bottom layer, motors, wheels are mounted in the center and all sensors are fixed on this layer as well. On the top layer, the Arduino board, the camera braket are fixed. Also there are two holes in the top layer, where the NUC and the battery is placed. Since the NUC is standing between two layers with its USB connectors facing sideways, the usb interfaces are hard to reach and are close to the wheels when plugged in. A change in the overall construction was avoided by bending the cables tightly upwards.

1.2 Sensor Placement

There are 6 IR sensors and one IMU sensor integrated at the center of the lower plate. On each side of the robot, two short range IR sensors are mounted with distance in-between as far as possible, in order to get a better estimation of the robot angle. And two long range IR sensors pointed to

the front with an angle adjustment to the robot center. The reason why the front IR sensors are not pointed directly to the front is because if we do that, when there are obstacles come to the area exactly between the two front sensors, nothing will be detected. With crossed sensors, an triangle area will be formed, thus it guarantees front obstacle detection. Actually, we took advantages of such placement when dealing with the 'wall of death '.

There is also an IMU sensor mounted on the bottom layer, we used it for detection of crash. When the robot crashes into an obstacle, the IMU will send the current time to brain to reset. And the robot will try to go backwards and continue its path again.

1.3 Camera Placement

Since the PrimeSense has a minimal working range of 35 cm, the camera is fixed on top of an aluminum tower with height of 28cm above the floor. The system is not sensitive to the camera angle since software calibration of camera angle will be executed when the vision is started. Thus we are able to give position of detected objects precisely without caring a lot about the camera angle.

2 Controlling

The main task of the project requires the robot to move with the highest possible speed. On the other hand the robot should be able to move at a speed that induces as little drift as possible and reduces the risk of errors when hitting an obstacle. And lastly, the vision would have to identify objects from further away, increasing the difficulty of implementation and recognition. Thus, a balance in the velocities have to be found.

The implemented controllers consisted of a forward movement controller, a turn controller and a wall alignment controller. By embedding them in an adapter pattern it was possible to render their usage very versatile (see figure 4). Every controller inherited hereby a controller base which defined an interface that updated the logic and returned a Twist (A Twist is one of many ros pre-baked data structures that can be sent as a message. It stores a position and a rotation). The adapter combines all received Twists to one Twist, entrusting that no contradictions occur. A top logic has to ensure that the right controllers are activated and deactivated. This can be done by activating and deactivating each controller. Controllers like the turn and the forward movement controller respond with a message as soon as they have

stopped or finished their activity.

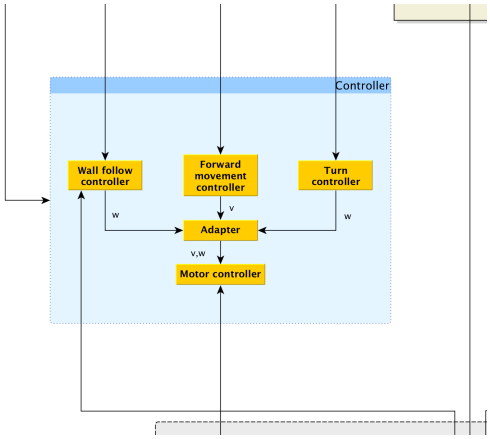


Figure 2: Architecture of the controller package

2.1 Motor controller

Due to internal motor difference, it is not sufficient to control the motors directly. Otherwise the robot would move in an arch, one motor rotating slower than the other. To compensate for that and to also be able to input linear and angular velocities, a PID controller was used. This enabled the robot to drive in a stable straight line.
The motors itself have a static resistance, that have to be overcome whenever the motors should transition from idling to rotating. Overcoming these static resistance could be described by constants (for each motor one) K_{power}, that were added to the result of the PID results. As soon as the static resistance has been overcome and the motor is rotating, the constant K_{power} can be reduced to a value that sustains rotation: $K_{sustain}$. This behavior results in a spiked motor control output as can be seen in figure 3. Although movement without spiking is possible due to the accumulating behavior of the PID controller too, it ensures responsive controlling and avoids the necessecity of increasing the gains which can easily result in overshoots.

2.2 Wall alignment

The wall allignment controller (see figure 4) uses the IR sensors to measure the distances from the robot to the walls, and works as follows: if both walls are close to the robot (distance < 0.4 m), then keep the robot aligned to the

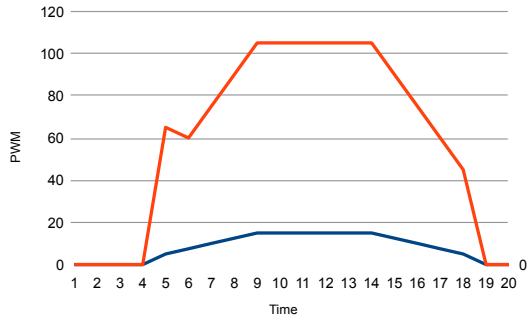


Figure 3: PWM Spiking for initial motor rotation attempt. **Red:** PWM output. **Blue:** Input, e.g. requested linear velocity.

center between both walls. Otherwise control only by the next closest wall, or if no wall is present, do not align.

3 Localization

Observing the created map given the regular odometry, indicated problems with the heading of the robot. A constant drift was observable while turning and because angular errors accumulate during turns, the map got in danger of falling into an unusable state. Completing a complete round trip in the maze, would inevitably lead to overlapping walls and free space, making navigation hard or impossible.

The common and generic approach for localization would be to use some kind of Bayesian Filter, which corrects the odometry by receiving measurements from the environment. Due to very little time until the third milestone, which required to have a basic localization implemented, it was decided to not spend time on any sophisticated solutions like Kalman or Particle Filter. Instead, systematic errors were minimized and the prior knowledge of rectangular walls was exploited to use IR readings to correct the heading of the robot at well defined events.
To minimize systematic errors, two approaches were implemented. The first and easiest approach is to fix the error in the wheel distance (in the following called base distance) by doing manual hill climbing. This can be performed when letting the robot turn for at least 3 complete revolutions and roughly determine if the turning overshooted or undershooted. Based on this information, the base distance can be adjusted accordingly. The process was repeated until there was no significant drift left.
Furthermore, as stated in [1], a systematic error can be non equal wheel

diameters. Due to different weight distributions and that the wheels are made of soft plastic, the wheel diameter can differ in the magnitude of one millimeter. Although this is hardly noticeable in the length of the maze it was decided to give it a try. As described in [1] five clockwise and five counter-clockwise runs were performed in a 4mx4m region. The results were somewhat disappointing though. The calculated difference in the wheel diameters turned out to be too large which made the odometry worse than before. The unexpected result is probably an issue of another systematic error in performing the measurement of the actual position. It would be advised to perform more than 5 runs to reduce these errors.
Having the angular drift minimized was enough to achieve satisfying results. To achieve an overall better localization, it was decided on using the IR sensors and also segmented planes from the point cloud, to estimate the position of the robot more accurately. The IR sensors were used to correct the heading of the robot, by calculating the actual heading from the two side sensors (see equation 1). A precondition that has to be valid to correct Θ this way, is that the wall next to the robot has to be planar. This was approached, by only correcting theta after a turn has been performed and only if the calculated theta did not significantly differ from the current estimated odometry.

$$\Theta = tan^{-1}\left(\frac{d_{ir_{front}} - d_{ir_{back}}}{dist(ir_{front}, ir_{back})}\right) \tag{1}$$

To also correct the position and not only the heading, information about walls were combined with the built map. Splitting the position in a lateral (y-axis) and a longitudinal (x-axis) part, enables to simplify the correction into two steps. The longitudinal correction can be solved by casting a ray from robot to the forward facing wall and comparing it with casting a ray in the constructed grid map. As the point cloud gives quite accurate results when close and while standing, the x position of the robot can be easily corrected. A very similar approach can be used to correct the lateral direction. By comparing the result of a physical IR raycast of the side sensors with a raycast in the grid map, one can derive the correct y position in the map. Both lateral and longitudinal correction only works, when a map is given and not updated. For exploration, only the heading correction can be used.

The final solution did not use the lateral and the longitudinal correction. Although implemented it could not be tested properly due to changed priorities.

4 Mapping and navigation

In order to navigate in the maze three map layers are used. The first layer is an occupancy grid, which is based on data from the IR sensors. The second layer is something we call the "seen map" which is an estimation of which areas of the map that the primesense has seen. The third layer is a topological map.

4.1 Occupancy grid

An occupancy grid is essentially a matrix where each cell is either occupied, free or unknown. The internal representation contains the probability that a cell is occupied in log odds notation [2]. An unknown cell contains our prior belief that a cell is occupied, in our model that was $l(0.5)$, i.e. it is as likely to be occupied as free. Any cell with a value higher than the prior is considered to be occupied, and any cell with a smaller value is considered to be free.
When a cell is within the field of view of an IR sensor the probability of that cell being occupied is updated according to the following fomula:

$$p_{x,y} = p_{x,y} + p_t - p_{prior} \tag{2}$$

where p_t is the tentative belief that the cell is occupied. The cell where the sensor hit an obstacle and the surrounding cells will have a high tentative belief, and the cells between the hit cell and the robot will have a low tentative belief. By taking previous observations in to account, the model is less sensetive to sensor noise. [3]
The grid has a resolution of one cell per square centimater, and a fixed size of 1000 by 1000 cells. The robot always starts in the middle of the grid, which means that it can go five meters in any direction without going out of bounds, which was enough.
The main useage of the occupancy grid is when a new node is placed in the topological map. In order to decide which directions of the new node contain unexplored space the occupancy grid is used in combination with the seen map. A direction is considered unexplored if the ratio of occupied cells and seen cells is below the respective thresholds.

4.2 Seen map

The "seen map" is contains an estimation of which areas of the map that the primesense has seen. It is implemented by marking a trapezoid as seen in front of the robot in each time step. As previously stated, it is used in

conjunction with grid map to decide which areas of the map are unexplored. This saves us a lot of unnecessary exploration.

4.3 Topological map

The topological is our primary tool for navigation. Nodes are placed when the robot encounters an obstacle, when an intersection is detected, when an object is recognized, and at the starting position. When a node is placed the four compass directions (north, east, south and west) are marked as either unexplored or blocked, according to the previously stated condition. This does not apply to the direction from which we came, which is connected to the previous node instead. When we return to a node a connection is also made to the previous node.
To navigate in the maze the topological map offers a service that let's the master node navigate to the next point of interest, which can be either the closest unexplored area, or an object, depending on which mode is activated. Dijkstra's algorithm is used to find the shortest path.

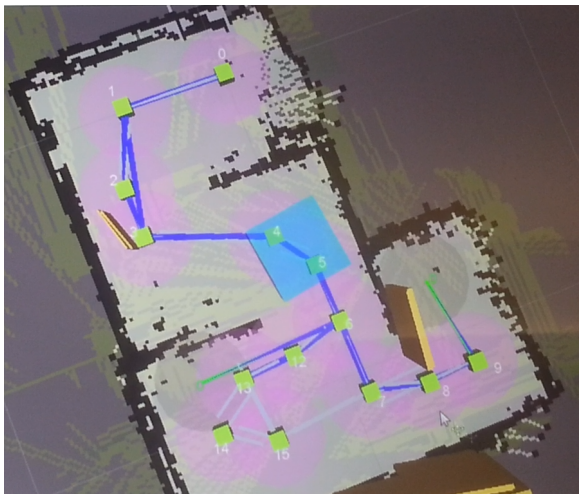


Figure 4: A screenshot of all map layers in rviz. The robot is the large cube. Nodes are represented by the small cubes (except objects, which are the points with an edge to them without a cube). The have seen map is represented by the yellowish lines. In this image the robot is running phase two. It has recognized two objects, returned to the start, and is now on its way to fetch the objects. The edges that make up the TSP tour are blue.

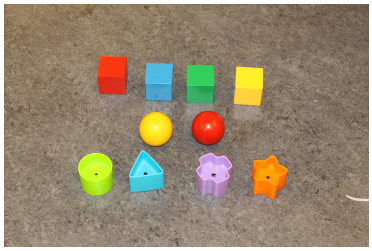


Figure 5: Object data set

4.4 Path following

Although an actual controller, the path follower (in figure 8 the top most instance called *Goto*), takes a path of points in the map, optimizes and follows it. Because the path follower is going node by node, resulting in stops at every node, the optimization is necessary to reduce the travel time. It is done by doing a greedy approach of removing nodes that can be skipped because the path between them is obstacle free. For that a raycast service was implemented in the mapping, that traces a ray for a given distance and checks for obstacles. Due to some noise in the grid map, it did not perform as well as expected.

5 Vision

Although the main intent of the vision was to detect the given set of objects (see figure 5, it was later on also used to exploit wall and ground information for mapping and localization functionality.

5.1 Object detection

It was decided upon using the 3D information from the primesense to reduce susceptibility to illumination changes and to gain more information about the structure of objects. Processing of the point cloud is mostly done with using the library *Point Cloud Library*. The detection and recognition can be divided in 4 different steps (see figure 7a).

- **Calibration:** Takes the ground plane from the segmented planes and calculates the camera pitch and height.
- **Transformation and subsampling:** Receives a calibration matrix from the calibration and transforms the point cloud accordingly. Simultaneously it downsamples the point cloud to reduce the number of points and so increase the processing speed in later steps.

- **Plane segmentation:** Takes in the subsampled point cloud and downsamples it even further so that only 1500 points are left. Instead of using the regular approach of using voxel grid downsampling, offered by the PCL library, the downsampling is performed by randomly picking 1500 points. This is sufficient to have enough features on the planes. RANSAC is applied to find all planes, until only a fraction of points is left unclassified (see result in figure 6a). To make use of the planes in mapping this step also calculates the oriented bounding box that the points on the plane encompasses. Because two walls can lie on the same plane, a cluster segmentation is necessary to make the distinction. The embedded euclidean cluster extraction turned out to be very expensive. It was therefore decided to implement a 1D cluster segmentation. This can be achieved by projecting all points on the orthogonal vector of the planes normal vector, sorting them and finding gaps.
- **ROI extraction:** Takes in the segmented planes and removes all points in the point cloud that lie on that plane with regard to some distance threshold. Euclidean cluster extraction, as implemented in the PCL library is applied to find clusters. Clusters that contain points lieing outside a given distance to the ground plane are immidiately dismissed. The remaining clusters define the regions of interests (ROIs) and can be used for further processing.

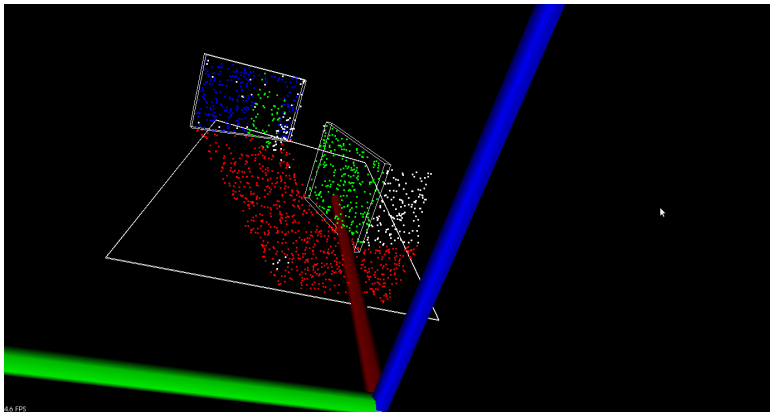
5.2 Object recognition

The object recognition consists of three phases itself (see figure 7b).

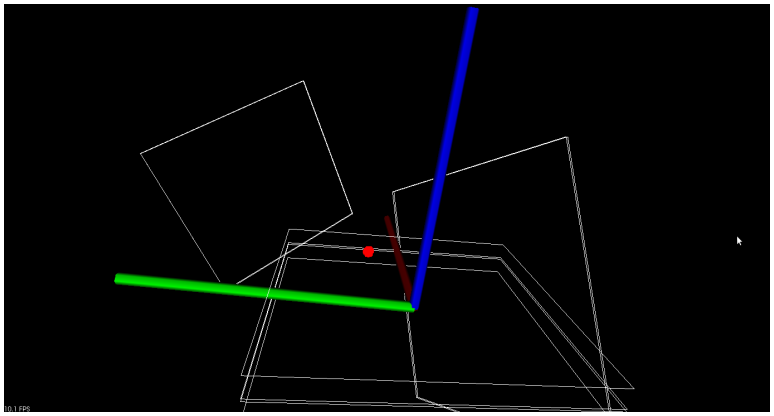
5.2.1 Shape classification

Regarding shape classification, it relies mainly on exploiting the topology of the points lieing in the ROIs. Three different shapes can be classified by applying different settings of RANSAC:

- **Spheres:** Normals are estimated and RANSAC with model Sphere is applied.
- **Cylinders:** Normals are estimated and RANSAC with model Cylinder is applied.
- **Cubes:** RANSAC with model Plane is applied to exhaust all planes. Multiple conditions are checked to verify that the planes describe a cube. They mostly involve checks for perpendicularity regarding the ground and other planes.

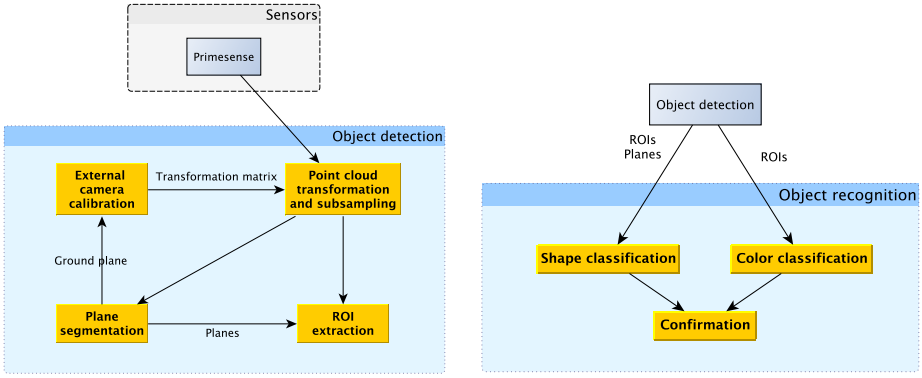


(a) Segmented planes. **Green:** Case when two unconnected walls are interpreted as one plane. Only the largest part is taken. **White:** Non segmented points.



(b) Segmented regions of interest. **Red:** ROI of a sphere. **Skewed rectangles:** Planes from (a). **Bottom sandwich:** Ground plane and maximum distance for accepting a cluster as a region of interest.

Figure 6: Results of the object detection



(a) Object detection (b) Object recognition

Figure 7: Vision - Process flow

All the other shapes can not be reasonably detected with RANSAC. The major problem with those shapes though, is the generated point cloud by the Primesense. Due to the whole in the top and bottom, the logic of the Primesense can not generate points especially around the region of the whole, rendering the extracted clusters in the ROI extractor very small. Therefore they are often filtered out for being classified as noise.

5.2.2 Color classification

The color information is extracted from the point cloud which carries RGB information for each point. Because color is highly susceptible to illumination changes, the perceived color can vary dramatically. This makes it hard to obtain a robust but consistent color classification algorithm.
The analysis is made per color possible and not all at once, even though the algorithm is the same whatever the color. We started by analising the HSV (hue, saturation, value) values we got from the regions of interest, mainly the hue, which is what defines the color. We then tried to find meaning in those values, mainly their average. We concluded that for the most part, the overall average of hues of all the points in the region of interest would be enough to obtain a good estimate of the color. So we defined an area of each mean color value around which the color will be. The bigger the margin, the more robust the algorithm would be. However, because of the specific colors, the margin could not be too large, or the added robustness would not be reliable, as colors would start to mix. Because different conditions may lead to very different results, we weighted two times the previous result based on specific situations, mainly, the percentage of points that have the color being tested relative to the total number of points, and the probability

of being of the color being analysed, i.e., considering the point clouds don't usually have that many points, we give a higher weight to probabilities above a certain value. Because the blob of interest may have points not only belonging to the object we want, but also from the ground or wall (if the object is very close to one), the number of points that have a color similar to orange, color of the ground, is also relevant in deciding the true color of the object, and hence helps in determining the weight of the probability of the color being analysed. The end result is a somewhat robust algorithm and consistent as far as our testing went.

5.2.3 Confirmation

To filter out false positives, a confirmation instance was added, that accumulates shape and color classifications over a specific time frame. If a combination of shape and color occurred more often than others and it's ratio is greater than a threshold, it is returned as a confirmed classification. Otherwise, if there is no classification for multiple frames whatsoever, the accumulation is reset.

6 Exploration

The robot is controlled by a master node called the "brain". The brain is a state machine that is implemented with the ROS package SMACH. In order to use SMACH the brain had to be implemented in python and it is the only node in the project which does not use C++. The basic strategy of the brain is to go forward until it detects something interesting. When an obstacle is encountered, it turns left, right, or back in order of preference. When an object is detected the brain will go closer or further away from the object to reach the optimal recognition distance, then turn towards the object, and then activate the recognition, to attempt to recognize the object. When a previously visited node is detected the brain will enter follow graph mode. Follow graph mode will take the robot to the closest unexplored area in phase one, and will follow the whole TSP tour and visit all objects in phase two. The brain also has a state to recover from a crash reported by the IMU, which is sadly a little too unsophisticated, but still helps quite a lot. When a crash is detected the state of the brain will be reset, i.e. stop, and reset all flags. Then the robot will back away from the crash site, turn towards the direction at the current node that is unexplored (if any), and then resume operation.

7 Performance

Our system did perform reasonably well in the end, and managed to get us third place in the competition. We have all the parts required to perform the main task of the project, i.e. navigate in the maze and recognize objects, return to the start, and then fetch the objects using the map constructed in the first phase. Sadly we did not have time to test the phase two logic in time for the competition, and it was therefore not used. The whole system together only works well in simple mazes though. Since some parts of the system are not robust enough, the chance of everything working fully at the same time is not very high in the hard part of the maze.
Control works alright, but could be better. We can't move as fast some other groups, because of the high risk of crashing.
Mapping and navigation works very well, drift in the map is never a problem unless we crash. Sometimes we place too many nodes in open areas, which results in time being wasted on pointless exploration.
The vision worked well for the simple objects, but we can't recognize the hollow objects when standing still. Experiments were made with a "shake controller" that should shake the robot back and forth when detecting an object, but that did not work well enough.
Integration (i.e. the master "brain" node) worked fine until the end when we added a lot of features that needed to be integrated really fast. Then it got messy very quickly and not robust enough.

8 Conclusions

The project was a real time killer and showed impressively how hard it is to work with real data and ever changing conditions. Although the maze environment can be considered quite statically and simple, it took some real effort to integrate everything properly.

9 Lessons learned

Regarding group work everybody has learned that it is important to maintain a persistent contribution. The first weeks were scattered with periods of time were few people invested time and so all the major changes happened in the last weeks. This obviously results in hassle and cluttered code, leading to bugs.
Regarding vision, there were problems with the shape classification since RANSAC didn't work well. Instead of only relying on the point cloud, an image based approach for object recognition with OpenCV might have been

worth a try. Contours of the object can be approximated or detected with line or corner detector, and different shapes should have different characters regarding contours. A Bayes Classifier could have been used to improve the color recognition.

A Topological map has been implemented, but it took a significant amount of time to adjust it properly for the given problems. A more generic approach by using navigation on the grid map could have led to better results, regarding simplicity and versatility.

More time should have been spent on making the whole system more robust to quirks of the enviroment. The IMU turned out to be a good choice to extract information about touches with obstacles. The actual crash handling however, should have been more sophisticated. It would have been advantageous to implement some kind of local obstacle avoidance.

References

1. J. Borenstein and Liqiang Feng. Measurement and correction of systematic odometry errors in mobile robots. *Robotics and Automation, IEEE Transactions on*, 12(6):869–880, Dec 1996.

2. Wikipedia. Logit — Wikipedia, the free encyclopedia, 2014. [Online; accessed 16-Dec-2014].

3. Cyrill Stachniss. Slam course - 10 - grid maps, 2013. [Online; accessed 16-Dec-2014].

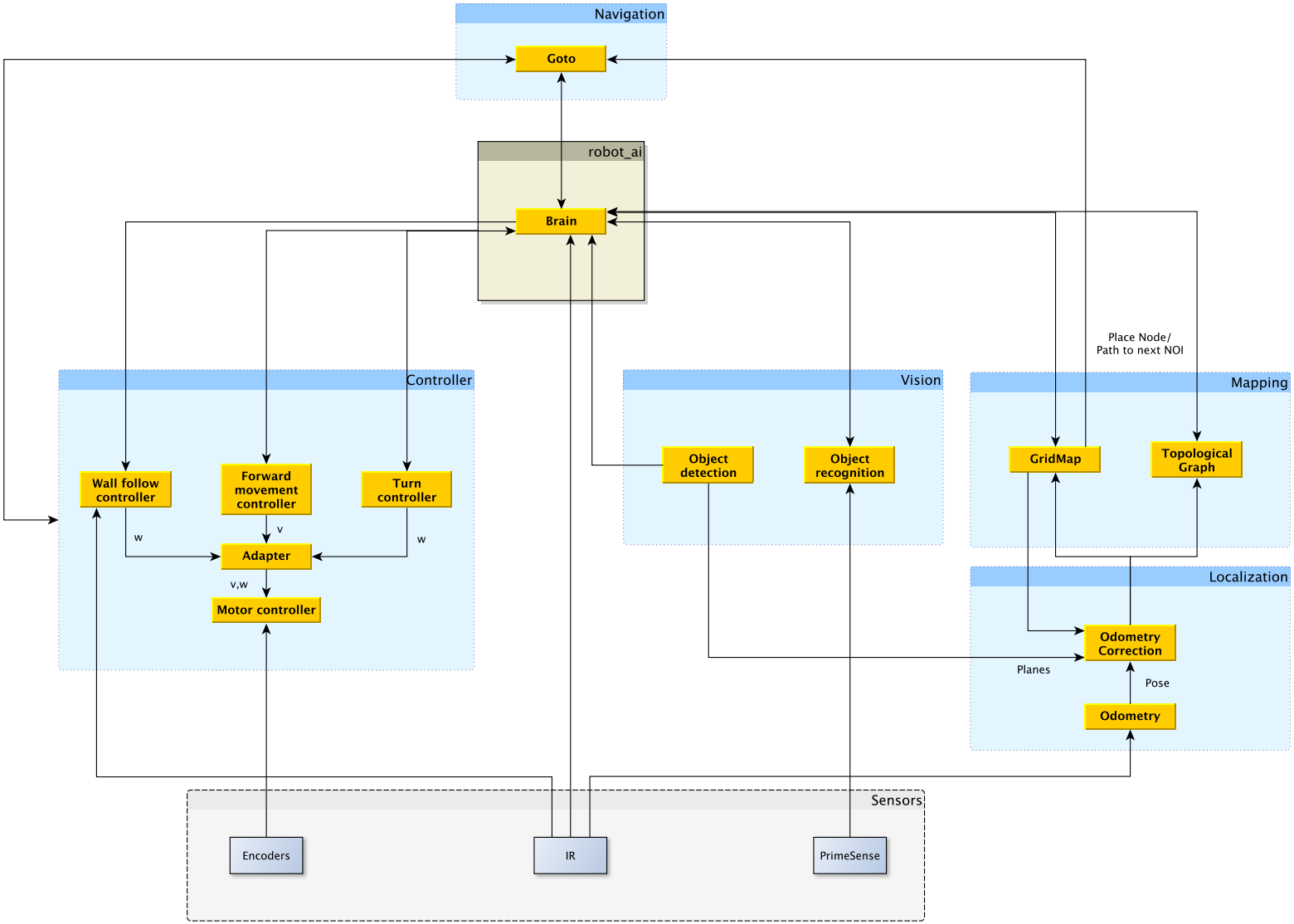


Figure 8: Main architecture of the system