Politecnico di Milano

Scuola di Ingegneria Industriale e dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

Using Simultaneous Localization And Mapping techniques in Robogames

---

Autore:
Andrea Salvi
Matricola 721247

Relatore:
Prof. Andrea Bonarini
Correlatore:
Prof. Matteo Matteucci

Anno Accademico 2012-2013

Contents

List of Figures

List of Algorithms

List of Tables

Computer vision techniques are becoming more and more used in modern technological objects that we use daily; one of the most promising applications of it is robotics, which thanks to the continuous hardware improvements in the embedded and mobile fields managed to use more and more complex algorithms in order to make automata more intelligent and reactive to the world that surrounds them.

This thesis project wants to show the applications of SLAM algorithms (Simultaneous Localization and Mapping) and in particular Visual SLAM for robots controlled by ARM system-on-chips and microcontrollers, taking as a sample use case a competitive game between two robots for the control of a playing field filled with obstacles.

1. Introduction

1.1. General description of the work

This project is aimed at investigating the potentiality and possible implementations of *Simultaneous Mapping And Localization* (SLAM) algorithms for autonomous robots using low-end embedded hardware. SLAM is a very active field of research that aims to let a robot be aware of its surroundings and navigate in an environment without previous knowledge of it, by detecting and then tracking interesting features using the sensors at the robot's disposal; in particular, *visual SLAM* is based only on the input coming from one or more cameras, devices that have become common and ubiquitous thanks to the rise and the advancements of mobile computing technologies.

1.1.1. What is a Robogame

The most suitable application testbed for SLAM is a Robogame; Robogames are interactive games with autonomous robots, which should involve human players both physically and behaviorally. The main objective of this field of research is the experimentation of new techniques and algorithms concerning computer vision, artificial intelligence and robotics in order to deliver engaging games, which should be implemented on reasonably cheap devices in order to make the games themselves affordable.

1.2. Scientific and technological issues

The biggest challenge that this work tries to tackle is *to make the robot aware of its own position* within its environment: this is exactly what SLAM algorithms attempt to do. This is very relevant for the robotics research community because it is a fundamental step to build robots that can be truly autonomous from direct human control.

Given the complexity of these algorithms, the *limited computational capabilities* that are inherent to embedded devices have been certainly an obstacle during the development of the project.

Another challenging aspect is, of course, *the design of an artificial intelligence routine* that is adequate for the purposes of the game.

1.3. Structure of the thesis

The thesis is organized as follows. Firstly, an analysis of the implemented game is presented, including its rules, purpose, preparation and evolution. Then, the requirements and the logical project of the implementation are presented, highlighting the scientific aspects where I have mostly focused. Next, the existing hardware devices and software packages used to produce this project are introduced, and then details about the software architecture and implementation will be given. Finally, the experimental results obtained with the implemented prototype are presented, as well as possible future improvements.

2. Problem analysis

In this chapter, the analysis of the designed game is presented: first the rules of the game are reported, then their representation as a finite state machine is introduced.

2.1. Game description

2.1.1. Rules, participants and setting

2.1.1.1. Players and playing field

The game is focused on the capture and retention of a flag placed within the playing area, dubbed *Control Point*, which is contested between two robotic players, *Red* and *Blue*, that are identical with the exception that the former is autonomously controlled by an AI system and the latter is controlled by a human player through a standard gamepad for videogames.

The playing field, which is setup at every match by the human player, is divided into four parts:

- the Red and Blue *Homefields*, which act as a shelter for their owners, repairing any damage they might have sustained in combat;
- the *Midfield*, where the core of the game takes place; it is filled with visual obstacles which can provide cover for guerrilla actions such as ambushes;
- the aforementioned Control Point, situated in an arbitrary position within the Midfield. To make things interesting, more than one Control Point areas could be layed out by the human player, but *only one* will be activated by the game controller application.

Both players at the beginning of the match start in their Homefields, and from there they begin to explore the game arena. Obviously, the autonomous robot is put at a disadvantage, as it does not know where the Control Point could exactly be.

2.1.1.2. Capturing, neutralizing and contending the Control Point

The Control Point starts out as being neutral; whenever one of the robots is in its immediate vicinities, a 5-seconds timer starts; if the robot manages to stay close to the flag for all that time, it *captures* the Control Point. From this moment for every second that the flag is in possession of the player, it will gain a point.

When the current owner of the Control Point is not defending and its foe comes close to the control zone, the flag starts to be *neutralized*: if the robot is able to make its stand for at least 5 seconds without getting critically damaged – see section 2.1.1.3 on the following page – and without letting the other robot coming close, the neutralization process gets completed: from this instant, the original holder of the flag stops gaining points and the flag goes back to a neutral state, ready to be recaptured again.

If both players are close to the flag, it is said to be *contested* – in this case, neutralization or capture operations are halted and reset, although the current player owning the flag will keep on earning points toward the final victory.

2.1.1.3. Combat and retreat

Both robots are equipped with a "rifle", and thus are able to shoot each other, granted that they have a visual on their target and *both* robots are outside of their Homefields.[1]

To hit its opponent a robot has to move in a position so that the enemy lies within a reticule in the center of its camera's image and then issuing a *shoot* command. If this is satisfied the opposing robot is said to be hit, and enters into *retreating mode*: it cannot be hit anymore, but it is forced to move back to its Homefield without being able to do anything else until this is accomplished.

A robot can shoot only when the opponent is in its viewing range, and only every 2 seconds to allow the weapon to reload.

2.1.1.4. Victory conditions

To win a match, one of the robots should reach the threshold of 30 score points before the other robot does, within the game time limit of 5 minutes. If after such a time no robot has managed to reach the threshold, then the robot that scored most points wins. If both robots have the same score, then the game is declared as *draw*.

2.1.2. Evolution of the game

The evolution of the game has been designed through a finite state machine, with respect to the position of the robots within the playing field and to the current state of the Control Point; this is represented in Figure 2.1 on page 21, where states in which no one controls the Control Point are filled in white; states in which either Red or Blue control the CP are filled with the corresponding color; states in which either Red or Blue are close to the CP have a border colored in the same way; finally, if the CP is contested, the border is colored in green. Moreover, for additional clarity, the final state is not represented.

For the sake of keeping the game description simple enough and not to explode the state machine into hundreds of nodes, there are actions that do not trigger an actual state transition, as their event wouldn't bring a critical change to the gameplay. An example of this could be a robot shooting its opponent: if the shot hits the target then its health is decreased by one point, but all rules given for the state would still be valid.

To keep track of the exact current situation of the game, some variables can be defined, both for each player and for the game as a whole.

For a more in-depth description of the state machine, including its states and transitions, please refer to Appendix A on page 67.

2.2. Possible and desirable behaviors for the robots

Given the structure and the game mechanics of Robotic Battlefield Control, there are many possible strategies that can be implemented for the autonomous robot, either more defensive or more aggressive: in the first case the artificial intelligence will try to avoid a frontal confrontation with the human player as much as it can, trying instead to find good places from where it can setup ambushes or from where it can have a good visual of the surroundings; in the second case, the AI will decide to attempt a frontal assault to the Control Point, uncaring of possible

[1] This limitation has a very simple reason: as each Homefield acts as a repair area, they could act also as a perfect camping ground from where to snipe the opponent without fear of retaliations and for the sake of the game such a situation is not desirable.

traps that could be orchestrated by the opponent. In this section, the various aspects that should be taken into consideration when planning a strategy for the game are explained.

2.2.1. Exploring and moving into the the game world

Exploration can be done in different ways, depending also on the conformation of the playing field.

In places where there are many obstacles, a player should take advantage of those to conceal itself from the sight of its opponent while moving, but at the same time limiting its own awareness of the surroundings. Moreover, extra caution has to be taken when entering very narrow paths that could finish with a dead end: these are very complicated to handle for autonomous robots, especially if it wishes to reach a point just beyond the obstacle; to see an example of this, refer to section 3.3 on page 32.

In zones that don't offer much cover, it is always advisable to drive along the shortest route from the starting point to the goal, since from a tactical point of view, the player is going to be exposed visually to the opponent, which could then fire upon it.

2.2.2. Capturing, defending and neutralizing the Control Point

The Control Point is obviously the cornerstone of the game, and around it revolves much of the tactical reasoning behind the implementation of a strategy. First of all, no matter what kind of policy is chosen to explore and traverse the playing field, the primary task for any player is to find out where the flag is, since it is vital to capture it in order to win the game.

Once the strategic location is found and captured, the robot has to decide how to defend it.

- the first possible solution is to stay in the immediate vicinities of the Control Point so that the player can intervene as soon as the opponent gets *too close* to it; this approach has the advantage that the presence itself of the player is deterring the possibility that the CP falls into the enemy's hands – since if it enters the flag area, the flag itself would become *contested* – but it has the disadvantage that the defending robot will be vulnerable to *sneak attacks* from angles where it is not looking, so the player will have to pay attention to have its sides and/or its back covered, or at least do not stay still in the same position for too much time;

- another solution is to leave the Control Point area and look for higher grounds, or for a position in the playing field that allows the robot to keep track visually of what is happening in the surroundings of the flag. This approach has the advantage that the position of the defending player is less obvious for the opponent, but at the same time it leaves the Control Point unguarded and, if something happens, the defender's reaction time will be much greater than what it could be if it was closer to the flag[2].

On the other hand, as an attacker the player will have to consider whether it is best to go straight to the flag and try to *neutralize or capture it as soon as possible*, or try first to *locate where the enemy could be*, "destroy" it and then proceed to the neutralization or capture. The former method is quicker, avoiding to give away too many points to the opponent, but it is riskier as there are more chances for the defender to sneak undetected and kill the attacker averting the assault completely; the latter strategy is more defensive and probably has more chances of success, but at the cost of giving more of an edge to the defender in terms of points toward victory.

[2]Note that even if the defender robot is not present at the CP, *it still gains points toward victory*. This stops only after the opposing robot successfully neutralizes the Control Point.

2.2.3. Shooting

Shooting tactics depend on two parameters: the shooter's *movement* and its *distance* from the target – to see how the game simulates this, refer to section 3.2 on page 31.

Therefore the shooting player has to take into consideration whether it wants to get more close up to have better chances to hit the other robot – risking to be spotted and shot down in the meantime – or whether it should keep moving or stay still while pulling the trigger – risking to miss the target *and* to be detected without gaining any advantage over the enemy.

2.2.4. Retreating

Retreating doesn't involve any particular tactic, besides that a retreating robot should follow the shortest route from where it is to its homefield, in order to resume playing as soon as possible.

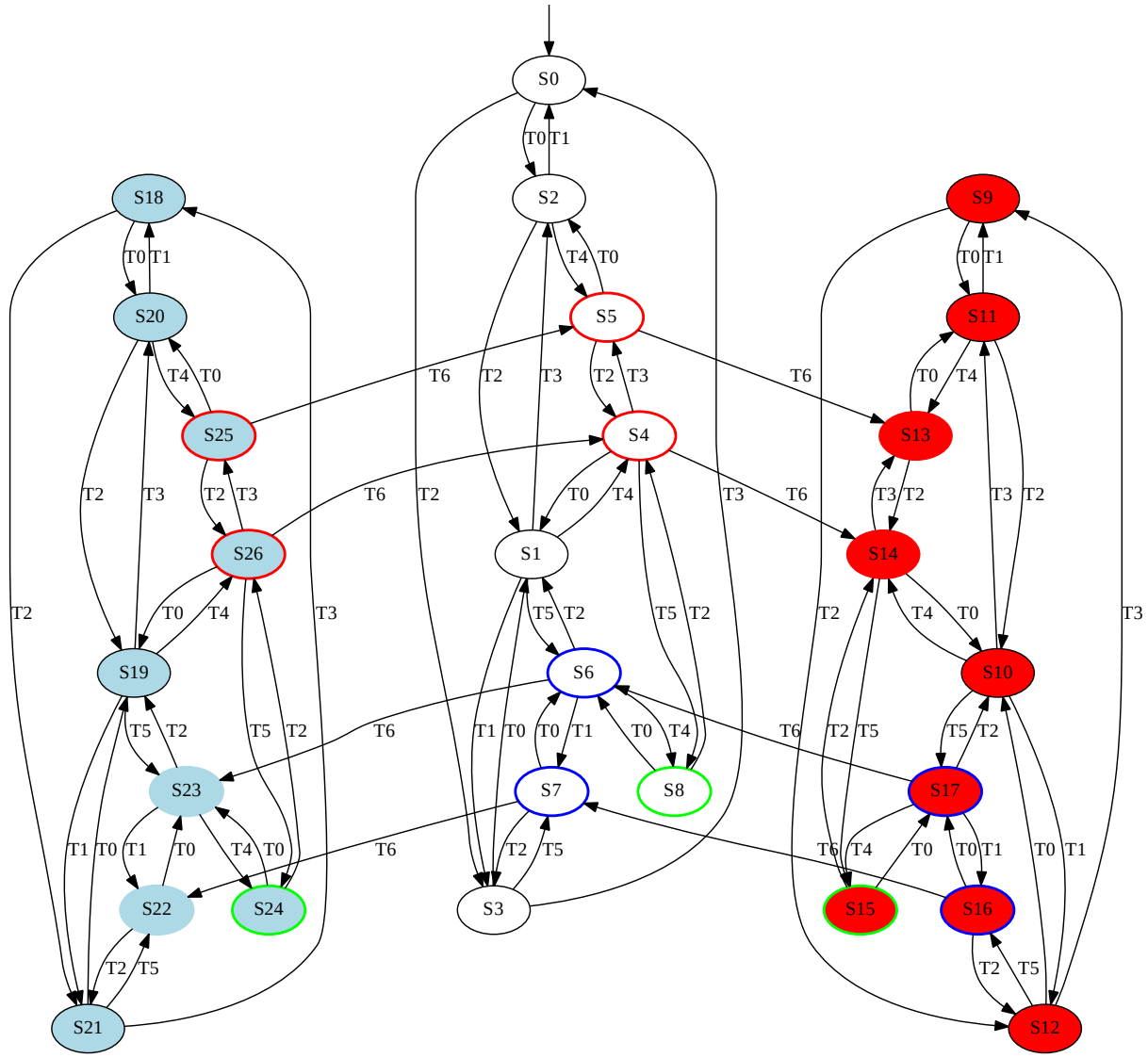


Figure 2.1.: The state machine graph for Robotic Battlefield Control.

3. Requirements and logical project of the implementation

In this chapter, I will explain the project requirements that have been highlighted by the analysis of the game that I have implemented, together with the purposes of the Robogames effort as a whole; please refer to section 1.1.1 on page 15 to know more about it.

3.1. The perception of the robot's own position

One of the major challenges was to make the robot *aware of its own position in the playing field*, since it can change at every game and therefore it is highly de-structured and no strong assumption can be made about it: this means that at the beginning of the match, the autonomous robot knows nothing about the environment surrounding it, besides what it knows thanks to the rules of the game (see section 2.1.1 on page 17):

1. it is standing in its own Homefield;
2. somewhere there is an opponent robot and another Homefield that only it can access;
3. somewhere there is a Control Point that needs to be captured and defended.

Given that the game is intended to be played indoor or in restricted outdoor environments, it was clear that a technique that is able to cover large-scale territories wasn't needed nor usable: a GPS (*Global Positioning System*) uplink requires the robot to remain outdoor so that it could "see" and connect to the network of GPS satellites that are placed all over Earth; moreover the accuracy that is provided by the system is too coarse for it to be adequate to the purposes of the game. Another possibility was to use beacons, but I didn't choose to use it in order not to structure the environment too much and not to require an initial calibration for the robot.

The next idea was to use IR (*Infra-Red*) sensors make a scan of the robot's immediate surroundings, eventually assembling them on an array that could continuously cycle back and forth in order to get readings of many points in space at different angles with respect to the robot's current bearing. This approach however still has severe shortcomings, because infrared sensors cannot discern between two different obstacles; the only thing they can perceive is the distance to any object that is directly in front of them. This is an information that could be used to estimate the robot's *odometry* – that is, the distance travelled between two points in time – in case the obstacle density of the environment is fairly high, but it is not enough to build a map of the surroundings sufficiently detailed for the robot to plan routes within the environment that has been already explored. Despite this, I have kept IR sensors to help the robot navigating its immediate surroundings – read section 3.3.2 on page 33 for more details.

3.1.1. Simultaneous Localization And Mapping

The problem of maintaining a representation of the working environment of a robot as it explores it is called *Simultaneous Localization And Mapping* (SLAM) [30]. The basic idea behind it is to track *a sparse* stochastic *map of high quality and reliable features* [31] using the information

obtained from its sensors that can be used to describe the environment that surrounds the robot and to work out the position of the robot itself within this environment at the same time; in this way, the robot is able to navigate its surroundings and move among the obstacles between itself and its goal.

For those who are not familiar with the topic, it is strongly suggested to read Davison's original articles [30, 31] as well as Ceriani's PhD thesis [24]; in the remainder of this section I will only talk about the specific algorithms and techniques that have been adopted for the implementation of SLAM in this system.

3.1.1.1. Visual SLAM and MonoSLAM

One of the kind of sensors that can be used in a SLAM algorithm is a camera; this particular instance of the problem is called *Visual SLAM* and has been one of the hot topics of robotics and computer vision researches since the last decade. There are two main reasons to use cameras as sensors for SLAM:

- first of all, they have a great intuitive appeal since sight is one of the primary senses animals and humans alike use to orient themselves;
- cameras are compact, non-invasive, and well-understood thanks to extensive scientific research [31] – and today also cheap and ubiquitous, in contrast with equipment such as laser sensors [33] that are more expensive, bulkier and require more power to work.

However, one of the biggest issues of using cameras is the inherent impossibility to recover both the range and the bearing of a scene feature when using a single image. The case of using one single camera is known as the *Monocular SLAM* problem [23, 31], which is opposed to the stereo- and multi-camera approaches, that can take advantage of epipolar geometry [38] to reconstruct the three-dimensionality of the observed scene with a certain degree of confidence. A single camera actually allows quite accurate measurement of bearing, but it only allows a uniform uncertainty on the depth of the feature, being its position equally likely along the entire viewing ray. For this reason, in Monocular SLAM the observer needs to gather data from different points of view in order to develop a stronger confidence about the feature position [23]. This is the main reason why, in Visual SLAM, the map of landmarks[1] is inherently stochastic and not deterministic.

It should be pointed out that there is a marked difference between *Visual Odometry* and Visual SLAM: the former – also called *Structure from Motion* – is about working out the trajectory of the camera and to reconstruct the three dimensional structure of the scene [66], and it is something that traditionally has been done *offline* through *batch processes*, while (Visual) SLAM aims to be an *online* algorithm [30], working in *real-time* and with the goal of reaching an efficiency such that it is possible to process the video stream at a sufficiently high rate (e.g., 30 frames per second) [31].

It is also important to notice that the uncertainty about the estimates of any two scene features, observed from the same pose, is not uncorrelated as one might expect, because the pose where the observations have been gathered is uncertain as well. Moving from a pose to another, the new point of view of the scene might let the robot examine only a subset of the scene features that were previously observed; the estimates of both the newer features and the robot pose itself can be updated by integrating the new measures; this in turn updates also the scene features that were seen previously. Moreover, the correlation between the estimates of the features grows with the subsequent observations. After some time the features and the robot pose will be correlated to each other [32, 23].

[1] From here onwards, *landmark* indicates a registered feature within the SLAM system.

Robotic Battlefield Control is bound to be set in room-sized environments which makes the repeated localization of the same features within the scene relatively easy and, therefore, helps the system to reduce the inevitable drift of the stochastic map of landmarks from the *ground truth* [31]. This is called *loop closure* and the approach I have followed to tackle it is discussed in details in section 3.1.1.7 on page 31.

3.1.1.2. Filtering: Extended Kalman Filter

As the *state* of the system is not completely known, new information has to be incorporated iteratively and this data has to be assumed as incomplete and prone to *errors*. This procedure is called *filtering* or *estimation* and throughout the years many techniques have been formulated to address this problem. When modelling the error within the currently available information with Gaussian noises, the *Kalman Filter* [27, 58] – part of the recursive Bayesian estimators family – in its extended variant (EKF) can be used to solve the SLAM problem; the extended variant of the filter is necessary due to the non-linear nature of the dynamic part of the state transition relationship with respect to the state variables and from the non-linearity of the measurement equation [23]. Other Bayesian estimators are the *Unscented Kalman Filter* [17, 26, 40] and *Particle Filters* [19, 56].

Current Visual SLAM systems based on the EKF have to deal with several shortcomings, such as:

- a limitation on the number of landmarks that can be treated at the same time, due to the computational complexity given by the upkeep of the coupled pose and scene covariance; these landmarks could be just a few, maybe even limited to those that can be seen in the current view. In some unfortunate cases these are too few to fully determine the pose with an acceptable confidence [66], increasing the overall uncertainty;
- for the same reason the map of the scene cannot be too big [66]; however this particular issue has been solved thanks to *sub-mapping* – that is, dividing the overall scene into multiple local maps [48];
- it has been shown that the Extended Kalman Filter suffers from inconsistency due to linearization errors [18]. In fact, when the angular uncertainty grows beyond just a few degrees the filter becomes overconfident and underestimates the uncertainty of the estimates it produces [66].

Alternative methods to filtering are represented by non-linear optimization procedures [61] such as what has been presented in [42, 62] and more pertinently to SLAM in [43, 65], which has been demonstrated to obtain good results; however, I have decided to use the EKF approach because:

- it is a more established and simpler technique with a lot of research already been done on it;
- as stated before the game doesn't need to map a large area, and therefore the number of landmarks that have to be treated should not be too large;
- the parametrization I have chosen, which I am going to present in the next section, mitigates linearization errors in a way similar to what non-linear optimization techniques do.

3.1.1.3. Parametrization: World-centric

A first big distinction in SLAM parametrization is the coordinate system that is going to be kept as reference. In the *world-centric* approach the initial pose of the robot becomes the origin of the

global reference system and every actor within the map – the robot's pose and all the landmarks – are related to this reference. This is the simplest approach computationally-wise, but also has a tendency to increase pose uncertainties constantly, until one of the previously-seen landmarks is detected again, triggering a loop closure [39]. On the other hand, a *robot-centric* approach takes the current pose of the robot as reference; this implies that all landmarks' poses are recalculated at every loop in light of the new pose of the robot. This technique has demonstrated to mitigate the uncertainty that is typical of the world-centric approach, but with a high increase in the computational cost of the algorithm [24].

Given the limited hardware that the robot uses, I have decided to adopt a world-centric reference for the SLAM implementation.

3.1.1.4. Parametrization: Framed Inverse-Depth

The second choice in SLAM parametrization is how we encode the information about the robot's and the landmarks' poses.

Using Euclidean coordinates would be the most intuitive idea, however, as Davison has shown in his original work [30], this is not possible without *delaying* the initialization of the landmarks as 3D features until some parallax has been acquired, meaning that the camera has to move enough so that the uncertainty about the *depth* of the 3D point is below a reasonable threshold. Unfortunately delayed initialization hampers *data association* when the feature has been observed very few times, exactly when it is most important to get it right [23].

Undelayed initialization has five solutions, which differ in how 3D points are represented in the filter state. Here, I will show only the *Framed Inverse-Depth* (FID) technique, presented by Ceriani and others in [23].

The FID parametrization is based on the previous *Framed Homogeneous Point* (FHP) representation proposed by Ceriani in [25], which in turn extends the concept of *anchor point* introduced by Solà in the *Anchored Homogeneous Point* (AHP) parametrization [59]. In FHP the anchor consists, for every registered feature $\mathbf{y}$, of the full camera pose $\Gamma = \left[\mathbf{t}^T \mathbf{q}^T\right]^T$ – $\mathbf{t}$ being the translation and $\mathbf{q}$ the rotation of the camera with respect to the world reference – in which the landmark has been added into the filter; the feature is then fully identified by its *viewing ray* $\mathbf{r}^C$ in the camera frame and by the inverse distance ω along the ray itself. Only the first two components $\mathbf{p}_\pi = [u, v]^T$ of the viewing ray are saved, as the third is always equal to 1 in the camera frame, and they are equal to the point coordinates on the *normalized* and *undistorted* image plane.

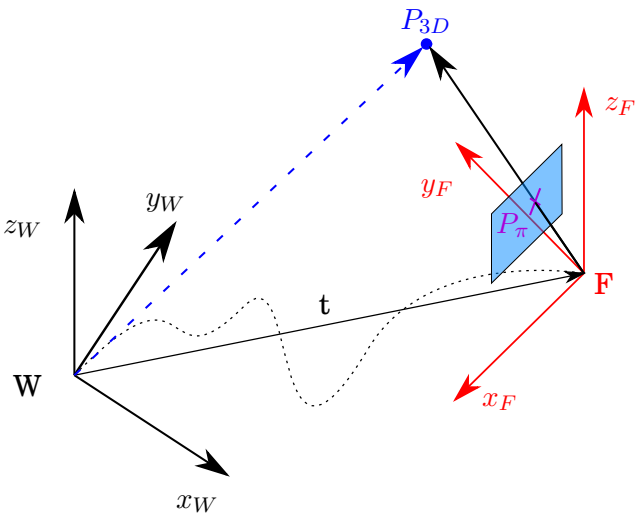


Figure 3.1.: Visual representation of the FHP and FID parametrizations.

$$\mathbf{y}_{FHP} = \left[\mathbf{t}^T \mathbf{q}^T \mathbf{p}_\pi^T \omega\right]^T \tag{3.1}$$

$$\mathbf{y}_{FID} = \left[\mathbf{t}^T \mathbf{q}^T \omega\right]^T \tag{3.2}$$

The real difference between FHP (Eq. 3.1) and FID (Eq. 3.2) is that the latter saves the viewing ray out of the filter; this is feasible because, if the camera is calibrated correctly, the

camera image has a sufficiently high resolution and is not much noisy, then $\mathbf{p}_\pi$ can be assumed as accurate enough even at landmark initialization time. This has three main benefits:

1. The representation of a single landmark is more compact;
2. The filter's consistency is not potentially harmed by a wrong estimation of the viewing ray due to errors of the camera pose being erroneously compensated by changes in the viewing ray itself [23];
3. When more than one feature is initialized with the same camera frame, they all have the same values in the covariance matrix of the filter, thus making it rank deficient. Furthermore this introduces singularities in the covariance matrix, making it not invertible. To solve this issue, when adding multiple features from the same position they should share the same common frame – making the overall representation even more compact [23].

As in FHP, the *feature initialization* (Eq. 3.3) is a linear operation, meaning that no information about uncertainty is dropped because of linearization.

$$\mathbf{y}_{FID}^{new} = \left[\mathbf{t}^T, \mathbf{q}^T, \omega_0 \centerdot \|\mathbf{r}^c\|\right]^T \tag{3.3}$$

To conclude, the *feature estimate* of landmark $\mathbf{y}_i$ with camera pose Γ at time t is given by Eq. 3.4:

$$feat_{FID}^{3D}(\Gamma, y_i) = \mathbf{R}(\mathbf{q}_t)^T \left(\omega(\mathbf{t} - \mathbf{t}_t) + \mathbf{R}(\mathbf{q}) \begin{bmatrix} \mathbf{p}_\pi \\ 1 \end{bmatrix}\right). \tag{3.4}$$

Finally it is possible to see that FID – and also FHP – refine past camera poses every time a registered feature gets updated by the Kalman filter. This looks like a violation of the EKF assumption that the current pose is a summary of the whole camera trajectory, but what really happens is a recursive filtering and smoothing of selected past camera poses and 3D features, giving a result that is comparable to non-EKF, optimization-based approaches [23].

For all the reasons I have mentioned, I have decided to adopt the Framed Inverse-Depth parametrization for my project.

3.1.1.5. Feature detection: FAST

Real-time feature detection and tracking [55] is a very important part of the visual SLAM problem, as we need to achieve good and robust results using the least possible resources, as SLAM is an extremely intensive application per se. *Robustness* in feature detection means to be able to deal with unmodelled clutter in the scene environment and with motions such as rapid translations, rotations and accelerations that can hamper the system by inducing large prediction errors. To tackle these issues Rosten and Drummond [49, 50] proposed to combine two popular approaches to tracking, *point-based* and *line-based*, both of which have their own advantages and disadvantages.

Point trackers:

1. *are robust to large, unpredictable inter-frame movements* since point features have strong characteristics that make them easily recognizable; however, these characteristics can vary greatly due to the sudden change in the scene structure due to motions and therefore obtaining a static *point cloud* for large scenes is considered as not feasible [49];
2. *produce measurement errors that approach a Gaussian distribution*, as they are largely due to pixel quantization and therefore they are independent from each other; nonetheless *the*

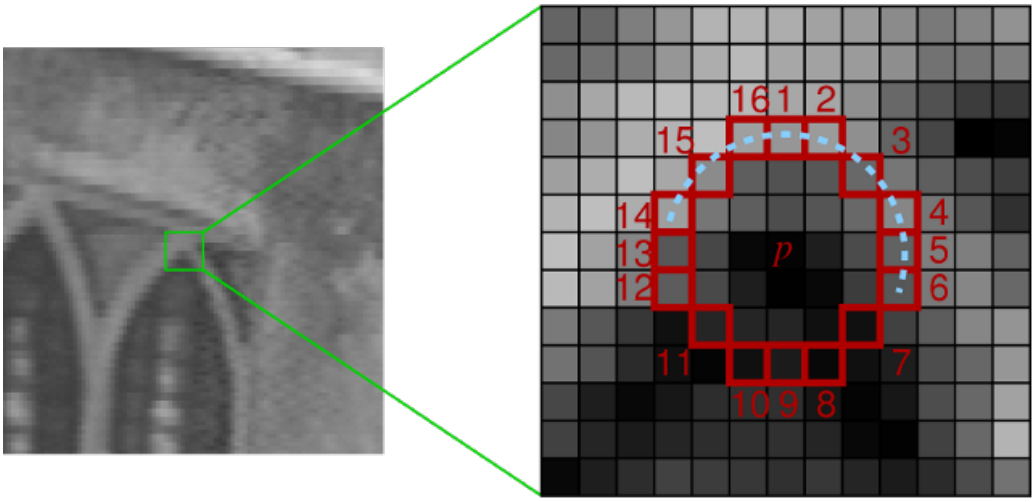


Figure 3.2.: FAST feature detection.

posterior measurement covariance is inaccurate, because the errors in the 3D point cloud are not independent [49].

On the other hand, edge trackers:

1. *are very stable under a wide range of lightning conditions and perspective changes* [49];
2. *need a 3D edge model of the object to be tracked*, which however can remain static because of the above point [49];
3. highly invariant features are not discriminative and therefore *we need strong prior information about the model's pose* to avoid incorrect edge correspondences – and even given this, edges can still be detected incorrectly, leading to pose estimates with large errors [49].

The result of the two researchers' work culminated into the proposal of the *Features from Accelerated Segment Test* (FAST) feature detector. To assess whether pixel p represents a feature or not, FAST examines a circle of 16 pixels – i.e. a *Bresenham circle* of radius 3 – surrounding it: if the intensities of at least 12 contiguous pixels are all above or all below the intensity of p by some threshold t, then p is declared a feature. For each pixel $x \in \{1, \dots, 16\}$ on the circumference having intensity I_x, its state S_x with respect to intensity I_p can be either

$$S_x = \begin{cases} darker: & I_x \leq I_p - t \\ similar: & I_p - t < I_x \leq I_p + t \\ brighter: & I_p + t \leq I_x \end{cases} . \tag{3.5}$$

In the example of Figure 3.2, p is *not* considered a feature because only 9 pixels on the circumference are lighter than p itself.

To run the test in a shorter time, at first only the pixels at the cardinal points are examined, since the aforementioned condition can be true only if at least three of these points are either all brighter or darker than p.

The last issue to tackle is the *non-maximal points suppression*, that is the pruning of those features that are adjacent to another feature with a stronger *corner response function* V, which in FAST's case is the sum of the absolute difference between the pixels in the contiguous arc and the center pixel p, that is

$$V = \max \left(\sum_{x \in S_{bright}} |I_x - I_p| - t, \sum_{x \in S_{dark}} |I_x - I_p| - t \right) \tag{3.6}$$

where S_{bright} and S_{dark} are respectively the set of brighter and darker pixels on the circumference of candidate feature p.

In the tests run by Rosten and Drummond by using a PAL video source [50], FAST managed to obtain impressive results on newer and older hardware alike – up to 80 times faster than SIFT [45], 38 times faster than the Harris edge detector [37] and 6 times faster than SUSAN [57] – and it was the only algorithm able to achieve real-time computation capability on both the tested systems.

3.1.1.6. Data Association: One-point RANSAC

As the new camera pose and landmark prediction is prone to error, a method for *associating* these estimates to actual image features is needed. A first quick data association technique is to search for a feature that has a high *cross-correlation* value – i.e., it is similar to – the registered one within the *bounding box* defined by the uncertainty of the landmark pose itself and the *jacobian* of the prediction; however this procedure can easily lead to wrong associations that will harm the data consistency and accuracy of the whole filter. This problem highlights the necessity of an algorithm that retains only the features that concur with a certain movement hypothesis, the so-called *inliers*, while rejecting all the others, the *outliers*.

For this purpose, a technique like the *Random Sample Consensus* (RANSAC) [34] can be utilized: the core idea behind it is to compute model hypotheses from randomly-sampled minimal sets of registered features and then verify these on the other landmarks; the hypothesis that shows the highest consensus is then selected as the solution. The number of hypotheses n_{hyp} necessary to ensure that at least one spurious-free hypothesis has been tested with probability p can be calculated with

$$n_{hyp} = \frac{\log(1-p)}{\log(1-\epsilon^m)} \tag{3.7}$$

where ϵ is the *inlier ratio* and m is the minimum number of features necessary to instantiate the model [29]; The usual approach is to adaptively compute this number of hypotheses at each iteration, assuming the inlier ratio is the support set by the total number of selected features in this iteration [38].

For the unconstrained motion with 6 degrees of freedom of a calibrated camera, it has been shown [46, 60] that $m = 5$ points are needed to solve the relative pose problem; this can be intuitively understood by noticing that since the parameters we need to determine are six – three rotations and three translations – the sixth can be found out once the other five are given. However less data can be used in several circumstances:

1. if *movement is constrained* like on a car [53] proceeding on a planar surface, where the degrees of freedom of motion are reduced to two – that is, the rotation angle and the radius of curvature – therefore reducing the required number of points to $m = 2$ for planar motion [47] and $m = 1$ for planar and non-holonomic motion [52].

2. if *extra information* for the camera motion is introduced, specifically the probability distribution function that the EKF propagates over time, without adding any kind of constraint over the camera movement itself [28].

Algorithm 3.1 One-point RANSAC

1: **function** $onePointRansac(\mathbf{x}_{k|k-1}, \mathbf{P}_{k|k-1}, th, \mathbf{h}_{k|k-1}, \mathbf{S}_{k|k-1}, \mathbf{z}_{ic})$
2: **begin**
3: $n_{hyp} = 1000$
{Phase one: pick low-innovation inliers by running random hypotheses}
4: $\mathbf{z}_{li\_inliers} = [\,]$
5: **for** $i = 0$ to n_{hyp} **do**
6: $\mathbf{z}_i = select\_random\_match(\mathbf{z}_{ic})$
7: $\mathbf{x}_i = EKF\_state\_update(\mathbf{z}_i, \mathbf{x}_{k|k-1})$
8: $\mathbf{h}_i = predict\_all\_measurements(\mathbf{x}_i)$
9: $\mathbf{z}_i^{th} = find\_matches\_below\_a\_threshold(\mathbf{z}_{ic}, \mathbf{h}_i, th)$
10: **if** $sizeof(\mathbf{z}_i^{th}) > sizeof(\mathbf{z}_{li\_inliers})$ **then**
11: $\mathbf{z}_{li\_inliers} = \mathbf{z}_i^{th}$ {$\mathbf{z}_i^{th}$ is now the current best hypothesis}
12: $\epsilon = \frac{sizeof(\mathbf{z}_{li\_inliers})}{sizeof(\mathbf{z}_{ic})}$
13: $n_{hyp} = \frac{\log(1-p)}{\log(1-\epsilon)}$
14: **end if**
15: **end for**
{EKF update of the filter using low-innovation inliers}
16: $\left[\mathbf{x}_{k|k-1}, \mathbf{P}_{k|k-1}\right] = EKF\_update(z_{li\_inliers}, \mathbf{x}_{k|k-1}, \mathbf{P}_{k|k-1})$
{Phase two: rescue high-innovation inliers}
17: $\mathbf{z}_{hi\_inliers} = [\,]$
18: **for** every match $\mathbf{z}_j$ above threshold th **do**
19: $[\mathbf{h}_j, \mathbf{S}_j] = measurement\_prediction(\mathbf{x}_{k|k}, \mathbf{P}_{k|k})$
20: $\upsilon_j = \mathbf{z}_j - \mathbf{h}_j$
21: **if** $\upsilon_j^T \mathbf{S}_j^{-1} \upsilon_j < \chi^2_{2,0.01}$ **then**
22: $\mathbf{z}_{hi\_inliers} = [\mathbf{z}_{hi\_inliers}; \mathbf{z}_j]$
23: **end if**
24: **end for**
{EKF update of the filter using high-innovation inliers}
25: **if** $sizeof(\mathbf{z}_{hi\_inliers}) > 0$ **then**
26: $\left[\mathbf{x}_{k|k}, \mathbf{P}_{k|k}\right] = EKF\_update(z_{li\_inliers}, \mathbf{x}_{k|k}, \mathbf{P}_{k|k})$
27: **end if**
28: **end**

Both conditions are satisfied by my project's requirements, but of course the second one bears the most significance as it allows the exploitation of the EKF characteristics, without forcibly restraining movement to be planar.

The algorithm integrates itself into the EKF loop after the *prediction* phase and a first, coarse *data association* between the landmarks registered within the filter and the features that can be seen on the current image. The procedure is divided into two main phases:

1. Initially (Algorithm 3.1, lines 3-14), several *random movement hypotheses* are run; this is accomplished by picking $\mathbf{z}_i$ from the set of *individually compatible* matches $\mathbf{z}_{ic}$ that have been found previously, and then run an EKF update of the sole state vector[2] in a temporary variable $\mathbf{x}_i$, then predicting the pose of registered features $\mathbf{h}_i$, and then finding all the matches $\mathbf{z}_i^{th}$ that fall below threshold *th*, for example 1 pixel. These are called *low-*

[2]This is where the algorithm differs more other RANSAC-based techniques: not only the assumed matches $\mathbf{z}_{ic}$, but also the entire prior knowledge given by the EKF state vector is involved.

innovation inliers. If $\mathbf{z}_i^{th}$ is bigger than the last best hypothesis' consensus $\mathbf{z}_{li\_inliers}$, then it is taken as the new best hypothesis and therefore n_{hyp} gets recalculated according to the new ratio between the current number of inliers and the total number of assumed matches. The hypotheses loop terminates once at least n_{hyp} iterations have been done. The (real) state vector and covariance matrix of the filter are then updated (line 16).

2. After this (lines 17-24) the algorithm tries to rescue *high innovation inliers* by cycling through every match $\mathbf{z}_j$ above threshold *th*, predicting where it should be in the image according to the updated vector state and covariance matrix, and then executing a *chi-square test* between the feature position and the prediction to check for individual compatibility. All the matches that pass the test are included in $\mathbf{z}_{hi\_inliers}$ and then are used to update the filter once again (lines 25-27).

3.1.1.7. Algorithm complexity, map reliability and loop closure

The complexity of EKF-based SLAM techniques grows quadratically with the number of features stored within the filter; therefore keeping anything beyond a hundred landmarks within it can be taxing for a laptop computer – let alone for an embedded device – to the point that the computation speed decreases so much that 30 frame per second can't be processed anymore.

Framed Inverse-Depth already tries to minimize this by sharing the camera frame pose for all the features that have been acquired in that frame, but of course features that haven't been seen in a long time will be eventually removed by a filter *clean-up policy*. This has two reasons to exist:

1. sometimes the filter adds features that are not easily re-detectable because of different and changing light conditions or because of noise in the camera's output images. If these landmarks are kept within the filter, their uncertainty grows for every iteration in which they stay as unobserved, therefore littering the filter with useless data;

2. features within the filter that haven't been recognized since many cycles will increase their uncertainty over time.

Both these issues can potentially hamper the reliability of the map itself, so there is the need to keep the landmarks within the EKF filter current, while retaining information that can be considered reliable – even if kept outside of the filter as fixed data, therefore losing information about the correlation with all the other features currently processed in the filter itself.

A solution of this problem can be to separate the *mapping* and the *pose tracking* tasks, in a similar way to what Klein and Murray proposed in their *PTAM* (*Parallel Tracking And Mapping*) [41] algorithm: the map would be sparse and made of the sum of the landmarks that are either currently in or out of the filter; when an out-of-filter feature is seen again, the information regarding its pose and its relative uncertainty can be used to update the robot's current pose.

3.2. The autonomous robot's decisions

The autonomous robot needs to take a lot of decisions and plan its actions accordingly:

- To what the other player is doing;
- To the current state of the game, which indicates what is possible to do and what is not;
- To the position of the robot itself within the playing field.

Some of these actions are *forbidden*: for example, a robot should not collide with an obstacle, and it cannot shoot an opponent while either player is standing in its own homefield (see section 2.1.1.3 on page 18); therefore, the artificial intelligence routine, given the constraints defined by the gameplay mechanics and physical space, has to plan the actions that it can *possibly* accomplish. The idea is then to be able to *react* to what happens in the game world, but also to be able to come up with a strategy to win the game.

3.3. The autonomous robot's movements

The autonomous robot needs to have two kinds of movement planning procedures: the first one is localized in a shorter time-frame, in order to avoid colliding into *obstacles*, both static and dynamic ones, not just to preserve the integrity of the robot itself but also for the safety of the users and the environment.

The other path planning procedure should aim to reach a long-term goal – such as reaching the Control Point – but it has to take into consideration the obstacles and *dead-end paths* that are present between the robot's current position and the goal itself. The space of all the possible states or configurations of the robot is called *configuration space*, and it generally has more dimensions than the cartesian physical space in which the robot moves: as an example, 3D movement has a 6-dimensional C-space in which we have to consider the three carthesian coordinates x, y and z and three bearings, one for each axis: φ, ψ and θ.

3.3.1. Path planning: obstacles and dead-end paths

Path planning is interested in finding a path between a starting configuration S and a goal configuration G, while remaining in the C-space subset C_{free} which is known to be free of obstacles. The fact that the autonomous robot moves in a completely unknown environment makes impossible for it to use a predetermined map, and since the sensors on the robot do not permit to create a detailed obstacles map, the only feasible thing to do would be to use the *potential fields* method [36]. The pivotal part of the algorithm is the definition of a heuristic *potential function* U that estimates the distance from any configuration to the goal; this function is composed of an *attractive term* U_{attr}, which is a C-space metric that becomes bigger as the configuration gets closer to G, and a *repulsive term* U_{rep} that penalizes configurations that come too close to obstacles; these potentials define a force F obtained from their negated gradients. A possible formulation then could be

$$U_{attr}(\mathbf{x}) = \frac{1}{2}\xi P_{goal}^2(\mathbf{x}) \tag{3.8}$$

$$U_{rep}(\mathbf{x}) = \begin{cases} \frac{1}{2}\eta\left(\frac{1}{P_{obstacle}(\mathbf{x})} - \frac{1}{P_{threshold}}\right)^2 & if\ P_{obstacle}(\mathbf{x}) \leq P_{threshold} \\ 0 & otherwise \end{cases} \tag{3.9}$$

$$U(\mathbf{x}) = U_{attr}(\mathbf{x}) + U_{rep}(\mathbf{x}) \tag{3.10}$$

$$F(\mathbf{x}) = F_{attr}(\mathbf{x}) + F_{rep}(\mathbf{x}) = -\nabla U(\mathbf{x}) = \left|\frac{\partial U}{\partial \mathbf{x}}\right| \tag{3.11}$$

where $\xi, \eta > 0$ and $P_y(\mathbf{x})$ is the euclidean distance between the current robot configuration x and object y. Equation 3.9 considers a $P_{threshold}$ so that the algorithm won't take into consideration obstacles that are too far from the robot.

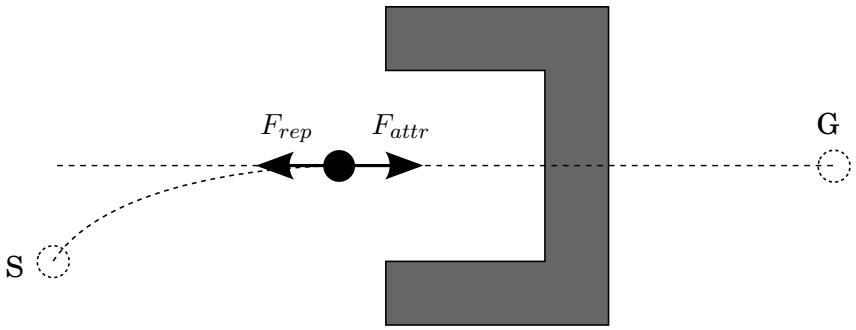


Figure 3.3.: The local minima problem in potential field path planning.

However, using potential fields has the shortcoming that robots can become trapped in *local minima* generated by dead-ends and particularly wide walls that reside on the shortest path to the goal (Figure 3.3) that is normally selected by a *best-first search* algorithm.

To alleviate this, Barraquand and Latombe [20, 21, 44] proposed the *randomized potential field* approach, which uses *random walks* to attempt to escape local minima when best-first search becomes stuck: essentially the robot performs a random series of moves until U has been lowered by a certain threshold or a certain number of moves have been attempted; if after a certain number of attempts the robot is still stuck, it backtracks to a previous position and then restores the best-first policy for choosing the next move.

A modification of this algorithm could be the following (Algorithm 3.2 on the following page): if the goal location is unknown, follow a certain exploration policy, or if it is known follow a best-first search; in either case, if there is an obstacle ahead – i.e. `get_next_waypoints()` returns an empty set – follow the obstacle on the right until a `timeout`, and then go back to use the potential fields planning again.

3.3.2. Measuring distance: IR range finder sensors

Infra-red range finders (Figure 3.4) are simple and economic sensors that allow the detection of the distance to an object in their view (*analog range finder*) or a high or low signal indicating that said object is respectively closer or farther than a predefined distance (*digital range finder*).

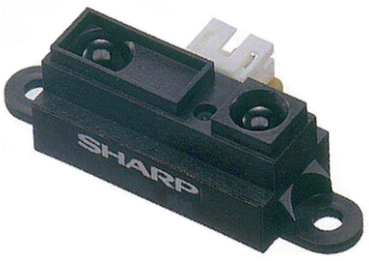


Figure 3.4.: A Sharp IR range finder.

All IR range finders use triangulation to compute the distance or the presence of objects in the field of view (Figure 3.5 on page 35): an IR light emitter releases an impulse, which travels along the field of view until it hits an object or indefinitely; in the former case, the light is then reflected and received by a small CCD array on the sensor itself. The incident angle radius of the reflected light varies with the distance d to the object, and since the distance between the emitter and the receiver is known, it is possible to easily compute d. This method of ranging is relatively immune to interference from ambient light (being only affected by strong IR emissions like direct sun) it is also indifferent to the color of the object being detected.

Every sensor is designed to react within a certain distance range, for example from 10 to 80 cm. Since trigonometry is involved, the function that maps distance to the sensor's returned

Algorithm 3.2 Potential Field path planning

```
1: function potentialFieldPlanning(x, goal, ξ, η, startTime, timeout)
2: begin
3: waypoints = get_next_waypoints()
4: if sizeof(waypoints) > 0 and goal is known then
5:    U_best = get_potential(x, goal, ξ, η)
6:    wp_best = [ ]
7:    for wp in waypoints do
8:       U_wp = get_potential(wp, goal, ξ, η)
9:       if U_wp > U_best then
10:         U_best = U_wp
11:         wp_best = wp
12:      end if
13:   end for
14:   return wp_best
15: else if sizeof(waypoints) > 0 and goal is not known then
16:   return exploration_policy(waypoints)
17: else
18:   if startTime ≡ 0 then
19:      startTime = get_current_time()
20:   end if
21:   if get_current_time() ≥ startTime + timeout then
22:      startTime = 0
23:      return potentialFieldPlanning(x, goal, ξ, η, startTime, timeout)
24:   else
25:      wp = follow_obstacle_on_right(waypoints)
26:      return wp
27:   end if
28: end if
29: end
```

signal is *non-linear*, and presents two critical zones: the first one is when the object is very close to the sensor, since the signal drops sharply and it can be confused with a longer range reading; the second one is when the object is very far, as minimal variations of signal intensity can mean a non-insignificant distance variation. To address these issues it is advisable to pass-through the signal either by hardware or software means, in order to reject any reading that is either too close to the minimum range or too far to have a reliable reading.

Another issue of these sensors is that the IR beam width is pretty narrow, and therefore they need to be used in arrays – and possibly even on a swivelling platform – if the user desires to sense distances from a multitude of directions. This issue is particularly problematic if the obstacles are relatively thin, as the sensor needs to be exactly on the obstacle's path to see it.

3.4. Recognizing areas and actors

There is the need to recognize *interactive components* and *actors* – i.e. robot players – of the game, differentiating them from the rest of the environment and possibly without using any additional hardware component than what it is used by the robot already. Coloring them with a specific hue, it is then possible to filter every other color out. In this way, it is then feasible to

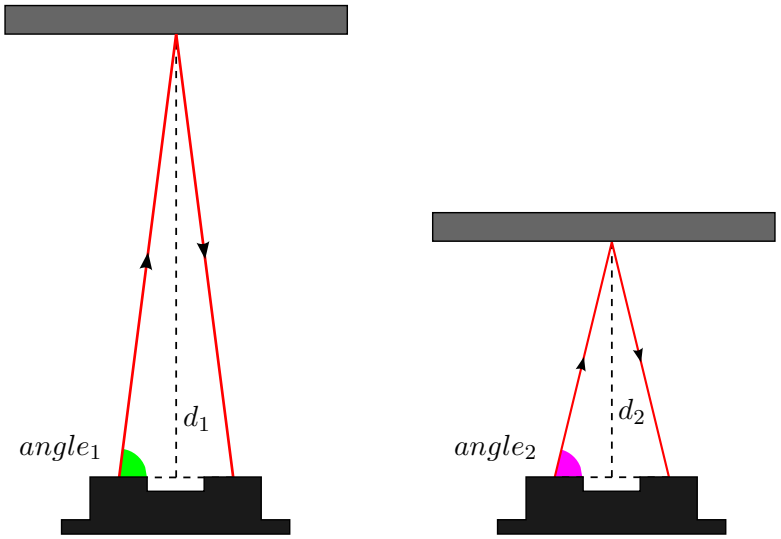


Figure 3.5.: How IR range finders work.

recognize items and actors just by visual means.

4. Preexisting components

In this chapter, I present the existing hardware and software components that have been used to create Robotic Battlefield Control along with the guidelines and motivations that led me to those decisions. Their application will be explained throughout chapter 5.

4.1. Design and development philosophy

One of the core ideas behind Robogames is to use cheap, off-the-shelf technology wherever possible in order to decrease the overall cost of the game itself – so that it could be affordable for most of the potential consumers – but of course must be balanced with the features and characteristics that these technologies deliver to the developer, so that he or she is not too constrained in what can be possibly done. This has influenced many of the design choices that have been made to make Robotic Battlefield Control a reality:

- *Small, embedded computing platforms* have been favored over more complex solutions like mini-PCs; thanks to the mobile computing revolution brought by smartphones and tablets, most modern *System-on-Chips* have comparable computing power to what small notebook computers can provide, yet consuming only a fraction of the power that is necessary to operate them;
- *Computing power shouldn't be laid to waste*; this means that software has to be as lean and clean as possible, and existing solutions that are too bulky or too expensive on the requirements costs had to be discarded;
- *Ad-hoc solutions* should be designed and implemented if no existing product can satisfy the needs of the project.

4.2. Hardware components

4.2.1. Arduino

Arduino [2] is a single-board microcontroller, designed and manufactured in Italy, with both the purpose of teaching microelectronics and the creation of interactive objects and environments. There are many board variants tailored to different needs and kind of applications, but they share the same base concept: an AVR *microcontroller* or an ARM system-on-chip that can be programmed through an USB connection and the RS-232 protocol on a standard personal computer, without the need of a chip programmer. Every board presents to the user a host of input and output pins and ports, both analog and digital, to connect sensors, actuators and any other electronic equipment or device.

Moreover, Arduino boards can be complemented with *shields*, printed circuit expansion boards that integrate perfectly with the original PCB and offer additional features such as GPS antennas, LCD displays, network adapters and more.

The most popular Arduino model is the *Arduino Uno* (Figure 4.1), and it is the one I have chosen for the project: it is based on the ATmega328 microcontroller, which can be powered

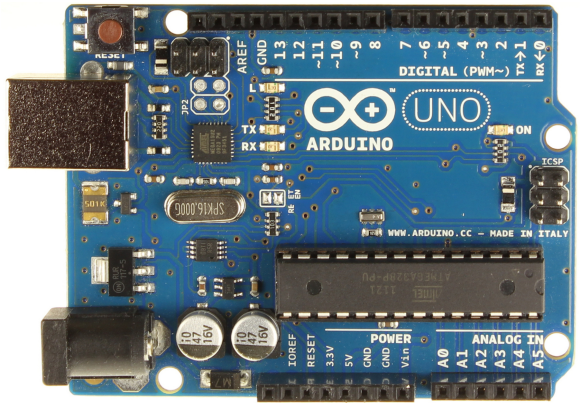


Figure 4.1.: Arduino Uno, revision R3.

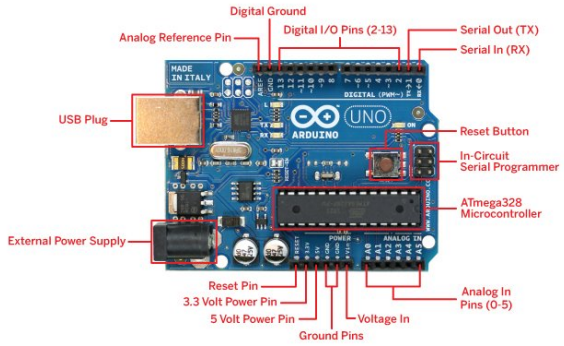


Figure 4.2.: Pin-out diagram of an Arduino Uno.

either directly via the USB communication cable or a jack to a 7-12V external power source; it sports 14 digital *General Purpose Input/Output* (GPIO) pins and 6 analog input pins.

It has to be noted (Figure 4.2) that 6 digital pins are able to send *Pulse-Width Modulated* (PWM) signals that can be used to control servos and wheel engines. However, since the wheel motors require more current than the maximum 40 mA that a single Arduino pin can provide, the PWM signals provided by the board have been input to two external PWM drivers (Toshiba TB6612FNG, Figure 4.3), one for the right wheels and one for the left ones, that could be connected directly to the battery pack that powers the whole robot.

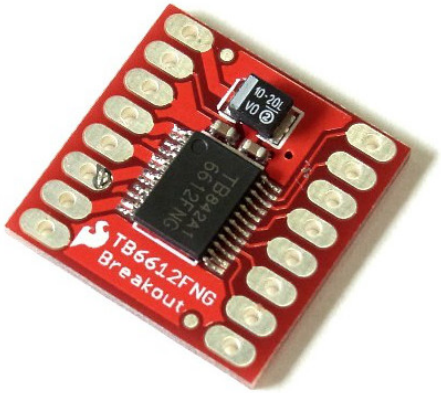


Figure 4.3.: The PWM driver used for the robot.

4.2.1.1. Programming for an Arduino: sketches

A piece of software written for Arduino is called a *sketch*, and it is written in a programming language based on *Processing*, which is very C-like – and indeed, the user's code is automatically converted to standard C++ before compilation by the development toolchain. In a nutshell, a sketch is composed of two main functions: `setup()` and `loop()`. The former is run only once when the board is powered up, for tasks such as pin initializations; then the latter function is run in an infinite loop until the board gets powered down. Sketches can also link to external libraries and headers, as long as they are known to the *Integrated Development Environment* (IDE).

4.2.1.2. Communications

Arduino can communicate with other hardware in a variety of ways:

- through an RS-232 compatible serial connection via USB or via the first two digital GPIO pins;
- through *Inter-Integrated Circuit* (I2C) connections, also via GPIO digital pins;

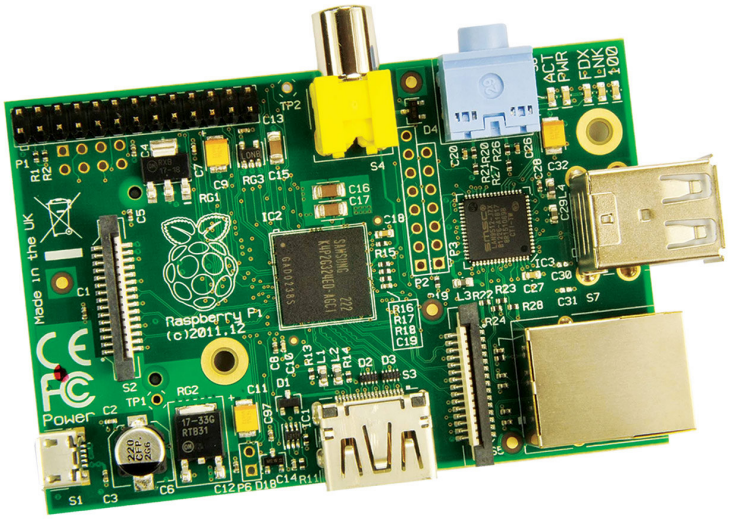


Figure 4.4.: A RaspberryPi, in its B variant.

- through any device available to the microcontroller via connected shields, such as wired and wireless *Ethernet* adapters.

4.2.2. RaspberryPi

The *Raspberry Pi* [9] is a credit-card sized, single-board computer developed by the *Raspberry Pi Foundation* in the United Kingdom. It has been designed initially to be used for teaching computer science and programming in schools, but it has been rapidly shown also to be a good platform for embedded electronics projects.

The original goal of the board influenced much the design of the board, which had to be as cheap as possible; the RasPi is manufactured in two configurations, *Model A* and *Model B*, which are largely the same but differ in a few key areas. The core of the system is the Broadcom BCM2835 System-on-Chip, which includes an ARMv6 processor clocked at 700 MHz, a VideoCore IV *Graphics Processing Unit* and 512 MB of RAM. Other features include a USB 2.0 port (two for Model B boards), a 3.5 mm jack audio out port a Composite and an HDMI video ports. It uses an SD card for booting and persistent data storage, and it can be powered either via a standard microUSB phone charger or a standard USB cable connected to a self-powered hub.

The RaspberryPi runs many flavors of Linux distributions, including the Debian-based *Raspbian* [11], the Fedora-based *Pidora* [8], *ArchLinux* [1] and others. For this project I have chosen to use Raspbian as it is the most mature and stable available environment.

4.2.2.1. Programming for a Raspberry

Programming can be done in any language which has a compiler or cross-compiler toolchain [10] for the Linux on ARMv6 architecture; therefore, development can be done either directly on the machine or on an external computer with any IDE.

4.2.2.2. Communications

Every RaspberryPi has a number of integrated input/output ports:

- a Fast Ethernet port (only for Model B boards);

Figure 4.5.: An ODROID-U2 board without (on the left) and with (on the right) its heatsink case.

- 17 GPIO pins, with support for I2C, *Serial Peripheral Interface* (SPI) and *Universal Asynchronous Receiver/Transmitter* (UART) protocols.

It has to be noted that these capabilities can be expanded through the use of USB-based network adapters, such as Wi-Fi or Bluetooth ones.

4.2.3. ODROID

ODROID [4] boards are a series of compact single-board computers developed by the South Korean company Hardkernel. They are more expensive than the RaspberryPi but also more powerful, integrating multi-core ARM processors and more RAM, and are therefore able to achieve notebook-grade computing performances.

The model of the board I have used is the *ODROID-U2* (Figure 4.5), which features a Samsung Exynos 4412 Prime Quad-core CPU clocked at 1.7 GHz, a Mali-400 GPU and 2 GB of DDR2 RAM. Similarly to the Raspberry, it uses a microSD card for booting and persistent data storage. Other features include two USB 2.0 ports, a 3.5 mm jack audio out port and a Micro HDMI video port. The device needs to be powered by a 5V, 2A power source.

All ODROID boards can run the ARMv7 versions of the Android or Ubuntu Linux operating systems.

4.2.3.1. Programming for an ODROID

As for the RaspberryPi, software for ODROID boards can be developed in any programming language supported by the running OS.

4.2.3.2. Communications

To communicate with other devices, the ODROID-U2 board integrates only a Fast Ethernet port and an UART socket. However, as with Raspberries USB network adapters can be use to extend the networking capabilities of the device.

4.2.4. The robot chassis: ArduQuad

The chosen robot chassis is the model *Pirate* from DFRobot, which sports a 4WD, differential drive motor system which is also Arduino-ready. It has lots of space where to put additional

Figure 4.6.: The *Pirate* robot chassis, without any component mounted with the exception of motors and wheels.

hardware, and – last but not least – it is very cost effective. For this and future Robogames projects, I have called this platform *ArduQuad*.

4.3. Software components

4.3.1. OpenCV

OpenCV (Open Source Computer Vision Library) [7] is a free software project originally led by Intel – and now by the Willow Garage robotics research lab and the Itseez company – to create a computer vision and machine learning library to foster machine perception techniques and their applications in scientific research and commercial products.

The library is highly modular, and offers a wide range of highly optimized algorithms for algebra, image processing and analysis, cameras calibration, object detection, and computational photography; it has been used extensively by both well-established companies like Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, Toyota and by the robotics scientific community, which keeps on expanding the library itself with newer algorithms. Most of these procedures can take advantage of *CUDA* [6] or *OpenCL* [63] *General Purpose GPU* (GPGPU) acceleration routines when present and supported by the hardware.

OpenCV is compatible with most current desktop and mobile operating systems such as Windows, Mac, Linux, Android and iOS; it is written in C and C++, but has a lot of bindings for a

host of other languages such as Python, Java, MATLAB, C# and others.

4.3.2. Libav / FFmpeg

Libav [5] is a collection of free software programs and libraries to process and play multimedia streams, including those from web cameras. It originated as a fork of the *FFmpeg* project, whose API is very similar and mostly compatible with the original one; however, Libav developers don't want to retain this compatibility with FFmpeg, and the two libraries might irreparably part ways in the future.

I have chosen Libav over FFmpeg as it is readily available for Debian and Debian-based Linux distributions, while the adoption of FFmpeg would have required a recompilation of several key system components – since both frameworks share the same names for their libraries and cannot therefore coexist on the same system – that I assessed as unnecessary, given that the two libraries for now give the same kind of performances.

4.3.3. TinyXML2

TinyXML2 [64] is an open-source XML parser library written with the precise purpose of being small and fast, and therefore adequate for use on embedded systems. It is based on the *Document Object Model* (DOM) concept, and it offers full UTF-8 text support – although it lacks the capability of parsing either *DTD* or *XML schema* definitions. It is written in C++ and easily embeddable in any project as it is composed of only two files, a header and the matching implementation file.

As the limitations of the library are not relevant for my project, I have selected this library over more complete – but also more cumbersome – libraries like *Apache Xerces* or *libxml2*.

4.3.4. YARP

YARP (Yet Another Robot Platform) [16] is a set of libraries, protocols, and tools to keep software modules and devices used on a robot cleanly decoupled, by giving to the user a way to communicate in a way that abstracts from the underlying network, hardware, operating system and the rest of the software architecture used by the robot. Software modules then can be also shared among projects and switched with other ones as new requirements arise and new hardware components are introduced: all these parts – whether physical or logical – can change rapidly, but the interfaces that keep them together are far more stable. This is even more true for robot software, as it is typically hardware-specific and task-specific.

Therefore, there is a need to:

1. factor out the *details of the data flow* between programs from their source code, so that the algorithms used for specific tasks can be changed quickly as long as the communication interface is known;

2. factor out the *details of devices* used by the program from their source code, so that new components can be introduced without much effort.

YARP is then an attempt to make robot software that is more stable and long-lasting.

The components of YARP are divided into three separated libraries:

- *libYARP_OS*, which is the core of the middleware and interfaces with the underlying operating system to implement OS-neutral data streaming across different programs and machines on the same network.

- *libYARP_sig*, that performs audio-visual signal processing tasks;
- *libYARP_dev*, which provides drivers for common robot devices such as framegrabbers, digital cameras, motor control boards, etc.

For my project I have used only the first two libraries, as all the hardware devices, except for the gamepad and the camera, are handled at a lower level by the Arduino board. I have decided to use YARP instead of Robot Operating System middleware [12] because the latter, even if much more powerful, is also much more complex and requires a lot of computational power, that neither the RaspberryPi or ODROID can fully provide.

4.3.5. Mr. BRIAN

Mr. BRIAN (Multilevel Ruling Brian Reacts by Inferential ActioNs) is a decisional engine for behavior-based autonomous agents, developed within the developed within the *AIRLab* of Politecnico di Milano. The engine was designed to be as general as possible, abstracting over application environments and structure of the agents in order to be usable in many different situations, with no assumptions over the tools and libraries used to build the application logic.

4.3.5.1. Behaviors

A *behavior* is a simple functional unit that cares about the completion of an elementary goal, basing its decisions on inputs coming from sensors or the robot's *knowledge base*. In complex systems, autonomous agents have to deal with more than one behavior in order to achieve a particular set of tasks.

Behaviors are *modular*: each of them is unaware what the other behaviors are designed for, or which behaviors are proposing a certain action. They are organized into a *hierarchical structure* according to their priority, so that each behavior reacts not only to contextual information but also to actions proposed by lower-level – i.e., less critical – behaviors that could potentially interfere with its own goal. In this way, a behavior knows what the other lower-level behaviors would like to do, and it can try to achieve its goal while trying to preserve the actions proposed by others as much as possible.

Therefore, behaviors work cooperatively to generate the global behavior of the robot, which is composed of high level commands that have a correspondence to specific set-points for the actuators of the robot.

4.3.5.2. Fuzzy Predicates

Mr. BRIAN uses *fuzzy logic predicates* [67] to represent the activation and motivation conditions of behaviors, as well as its internal knowledge; each predicate is a triple composed of a *label*, a *truth value* that is computed by a fuzzy evaluation of a crisp – i.e., non-fuzzy – value or a composition of other fuzzy predicates, and a *reliability value* that takes into account the quality of the data source (which could be noisy, for example as in sensor data).

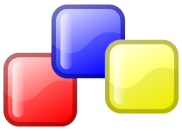
There are two kinds of fuzzy predicates in Mr. BRIAN:

1. *ground fuzzy predicates* are related to data directly available to the agent through its input interface, and have a truth value corresponding to the data's degree of membership with respect to a labelled fuzzy set;
2. *complex fuzzy predicates* are compositions of fuzzy predicates obtained by fuzzy logic operators.

Figure 4.7.: The VTK visualization pipeline.

4.3.6. wxWidgets

wxWidgets [15] is a cross-platform widget toolkit and tools library for creating *Graphical User Interfaces* (GUIs) abstracting from the underlying operating system and GUI *Application Programming Interface*. It supports Windows, Mac, Linux/Unix and more exotic operating systems such as OS/2 and AmigaOS, and renders the user interface with the native toolkit that is available on each of these platforms.



The library is written in C++ but several wrappers for Python, Perl and C# are also available. GUIs can also be designed through XML-based files that can be manipulated either from a plain text editor or from specific GUI designer applications such as *wxGlade* [14]; by using externally defined interfaces, programs do not have to be recompiled and linked every single time the interface itself is changed.

Beyond pure and simple GUI functionality, the library gives the user other cross-platform features such as network programming, multi-threading, image loading and saving in many popular formats and database support, but thanks to wxWidgets' modular approach, only the parts that are of interest for the developer can be used.

Initially developed at the University of Edinburgh for scientific research purposes, wxWidgets is now widely used by enterprises, other education centers and users worldwide.

4.3.7. VTK

The Visualization ToolKit, or VTK for short [13] is an open-source, freely available library and toolset for 3D computer graphics, image processing and visualization, supporting a wide variety of visualization algorithms including scalar, vector, tensor, texture, and volumetric methods. It is cross-platform and runs on Windows, Mac, Linux and other Unix-like operating systems; it is written in C++ but also offers several wrappers for interpreted languages including Tcl/Tk, Java, and Python. It has a suite of 3D interaction widgets, but it can also integrate with other widget toolkits such as GTK, QT and wxWidgets.

The library was designed around a few core goals [54]:

- to conceive the library as many *simple and well defined modules*, to ensure maximum *flexibility* and ease of integration into larger systems;
- to base it on *standard components and languages* to encourage the software's adoption;
- to try to *abstract from the underlying 3D rendering library* – currently OpenGL – as much as possible, as they are in constant evolution and might be superseded by other libraries in the future.

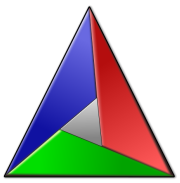
The core concept in VTK is the *visualization pipeline* (Figure 4.7), which consists of five main stages, stacking on each other:

1. *sources* which provide initial data input – a geometry and its associated topology – either from files or generated programmatically;

2. *filters* that can modify source data in some way, like resize, interpolation and merge operations;
3. *mappers* that basically convert geometry and topology data into renderable 3D objects;
4. *actors* which are instances of an object defined by a mapper;
5. *renderers* and *renderer windows* that constitute the user's actual viewport of the 3D scene.

4.3.8. CMake

CMake [3] is a cross-platform, open-source program to manage the build process of software using a compiler- and OS-independent method. The main idea behind CMake is to create platform-specific *makefiles* depending on software structure and requirements specifications written in CMake-specific configuration files; this approach makes software portability and also *cross-compilation* a much easier task.



Other CMake features include:

- the handling of both *in-place* and *out-of-place builds*; the ability to build a software project outside the source tree is a key feature, ensuring that if a build directory is removed the source files remain unaffected;
- *system introspection*, i.e. it is able to determine automatically what the target system could and could not do –similarly to what the *autotools* suite and the *pkg-config* system utility do on Linux;
- the ability to write configuration files for IDEs, such as *Visual Studio* on Windows and *Eclipse* on Linux;
- a less convoluted and more powerful language to define software projects than what most platform-specific makefile specifications can offer.

Since almost all the tools and libraries I have used offer native support to CMake, it was sensible for me to use it as well for my software project.

5. The project architecture

In this chapter I introduce the overall software-hardware architecture that has been created for the project, firstly with a general overview and then presenting each component separately.

5.1. Overview

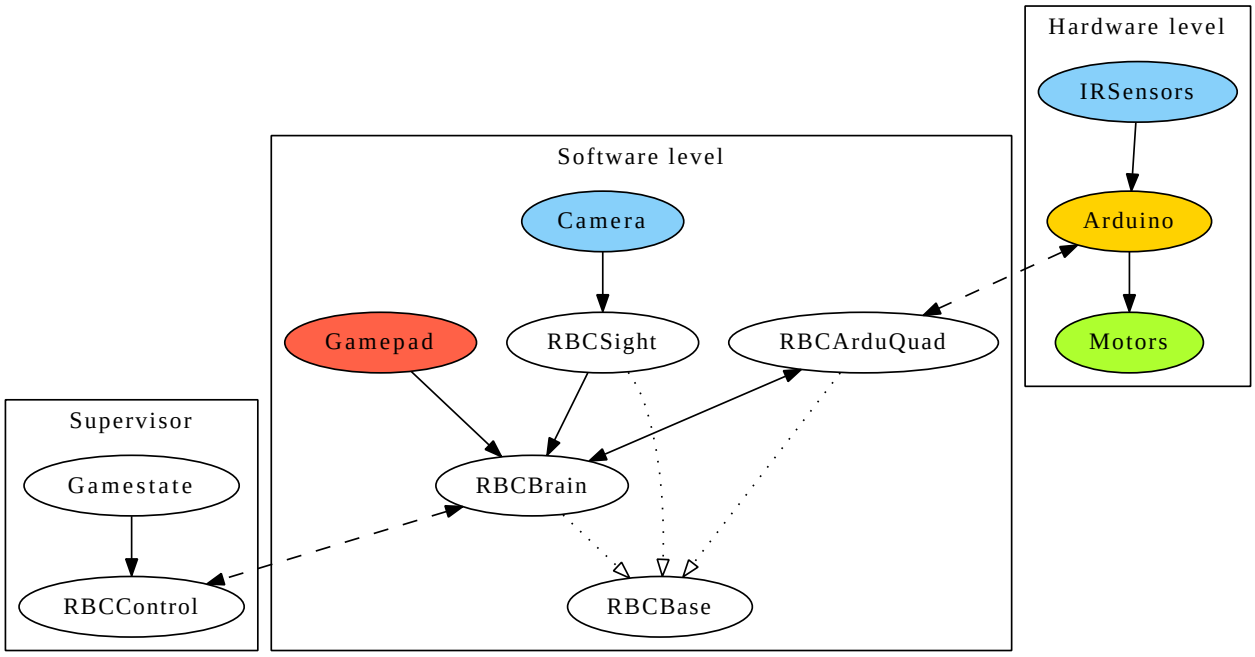


Figure 5.1.: The project architecture. Dashed lines represent remote connections, solid ones are local connections and dotted ones denote class hierarchy.

The project's architecture structure is divided into three main blocks:

1. the *hardware level* of the robot, focused on actuating motors and reading sensors;
2. the *software level* of the robot, whose purpose is to act as a decision-making center, analyzing visual perceptions and the overall game situation. This of course is not true for the human-controlled robot, as it is controlled from a gamepad;
3. the *game supervisor*, which is implemented on an external computer, housing the game state machine and controlling all robots involved in the game. The game state machine is not replicated on each robot and it is instead centralized in the controller so that there cannot be any confusion about the current state of the game that could be introduced in a completely distributed system by message passing latencies between the playing robots.

This subdivision wasn't tailored only to the specific game I wanted to implement, but thought to be as general as possible in order to maximize flexibility and reuse of the architecture.

5.2. The robot hardware level

The robot hardware level is centered around an Arduino Uno board (see section 4.2.1 on page 37) that receives distance readings from the IR sensors mounted on the robot chassis (see section 4.2.4 on page 40), and commands the PWM drivers that in turn actuate the wheel motors.

The control loop is based on a *message-passing paradigm* between the Arduino firmware and the RBCArduQuad library – which is explained later in section 5.3.3 on page 50. The loop (Figure 5.2) has two operation modes, *stand-by* and *active*: when on standby, the microcontroller stays still and it awaits for a `START` signal coming from the serial connection to the higher logic routines; as soon as this message arrives the robot enters in active mode, in which it receives sensors readings, sends a move request, and then waits until a timeout to receive a command back. If this command is a `HALT`, then the Arduino falls back into standby mode, otherwise if it is a `MOVE` command, it actuates motors. Then, no matter if a command has been received or not, the microcontroller sends the sensors feedback to the higher logic, and repeats the loop.

The command synchronization is necessary because otherwise the higher level logic might send move commands at a far too high rate for Arduino to process, effectively overflowing the serial communication buffers and therefore crashing the connection.

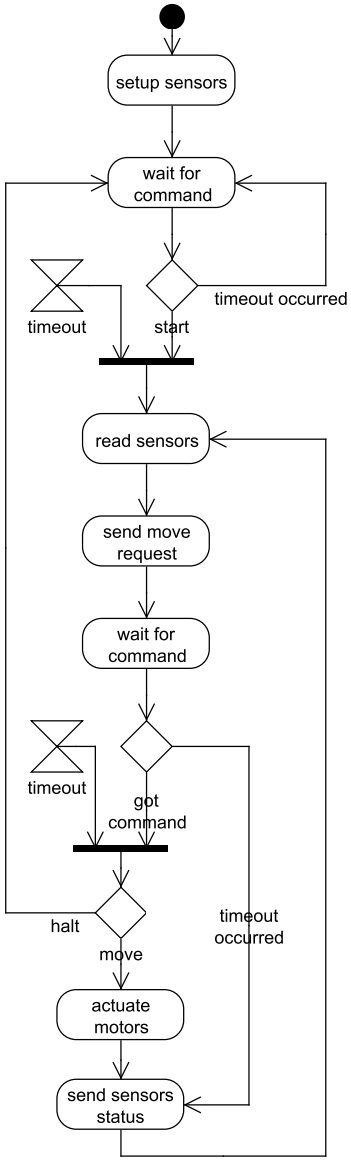


Figure 5.2.: The UML activity diagram for the Arduino firmware code.

5.3. The RBC Framework

The robots are equipped either with a RaspberryPi (see section 4.2.2 on page 39) or an ODROID (see section 4.2.3 on page 40) embedded computer which run the software level of the architecture – what I call the *RBC Framework* – that is constituted by a set of *loosely coupled Units* – i.e. modules – that work together to form the application logic of the robot.

Communication (Figure 5.1 on the preceding page) between these modules is achieved through passing of XML-based text messages via the YARP framework (see section 4.3.4 on page 42), which is able to abstract over the actual medium used to deliver the messages: data passed between units on the same machine are passed via shared memory buffers, while if it is sent to remote machines are transferred via the TCP network protocol. The presence of a *name-server*[1] enables the communication system to function similarly to the *publish-subscribe paradigm*, as in without the modules even being aware of how each other are made or composed of, besides the exchanged data format.

In this way, Units can be freely distributed on a number of different machines if it is ever needed to do so by specific hardware or application constraints.

[1]The name-server is launched by the RBCControl application, which is explained in section 5.3.5 on page 52.

5.3.1. RBCBase

RBCBase – as the name suggests – is the foundation of all the other modules, as it defines the basic structure and features of a `Unit` class and the messages serialization and marshalling mechanism.

A Unit is in practice a named *thread* that refers to a certain robot, and it can be in three possible execution modes: *stopped*, *running* and *terminating*; the third state exists to allow a *delayed de-initialization* of the thread itself, due to the different scheduling policies the underlying operating system might have. Once started with the `start()` method, the Unit will periodically call a `loop()` method until either the `stop()` method is invoked or the Unit itself gets deleted.

Each Unit has also a bidirectional *port*, named as `/robotName/unitName` : this is where all outgoing data will be written by the Unit itself, and the place where other Units will send data to it. To check for new messages, Units have to call the `receiveMessages(bool block)` method, which can be called either in *blocking* or *non-blocking* mode according to the passed argument; in the former case, the Unit will wait indefinitely for at least *one* message to arrive, while in the latter one the method will return immediately if no messages are present in the buffer; the handling or rejection of each message is done through the `parseMessage(const char* xml)` method, which must be implemented by every subclass of Unit.

5.3.1.1. Anatomy of a message

Messages are encapsulated in YARP `yarp::os::Bottle` objects, and each instance can contain a number of XML-based messages. The simplest form of a message is:

```
<message robot="MyRobot" sender="MyUnit" time="1394552610" id="1"/>
```

where Unit *MyUnit* of robot *MyRobot* has sent a message with an ID value of 1. The ID value is crucial as a message structure can be reused to carry the same kind of data but with different meanings, for example the position of another robot or the position of certain landmarks within the physical environment.

A message can also have a body containing arbitrary data, for example:

```
<message robot="Player" sender="ArduQuad" time="1394552610" id="100">
  <arduquadData>
    <irReads irFront="32.41" irRear="25.54"
      irFrontRight="0.0" irFrontLeft="10.12"/>
    <movement thrust="0" dir="S"/>
  </arduquadData>
</message>
```

which carries the distances read by the IR sensors on the ArduQuad chassis, together with the thrust and direction values of the motors.

The serialization and marshalling of these messages is done through the `XMLMessage` class and its derivatives, which use TinyXML2's facilities (see section 4.3.3 on page 42) to achieve this; the library implements various subclasses already, but the user can define new ones.

5.3.2. RBCBrain

RBCBrain is the module that acts as the robot's strategy planner and executor. Its pivot is Mr. BRIAN (see section 4.3.5 on page 43) which acts as decisional engine, taking information from

ArduQuad's sensors, the environment map (explained in section 5.3.4.4 on page 52) and the game supervisor (see section 5.3.5 on page 52); its tasks include:

- *game strategies*, i.e. deciding what to do in order to win the game: in the case of Robotic Battlefield Control, find the Control Point, capture it and defend it;
- *react to events* that happen in the environment: for example, the visual contact of an opposing robot;
- *decide where to go next* and *avoid obstacles*.

To do this, the hierarchical structure of Mr. BRIAN's behaviors is exploited: game strategies have the lowest priority, as they are broad, long-term goals that could be overrided by more impelling needs and opportunities given by events triggered by external actors and restrictions imposed by the robot knowledge or by the conformation of the playing field; finally, the obstacle avoidance routines have the outmost importance, as the robot needs to do it regardless of any other policy.

5.3.3. RBCArduQuad

RBCArduQuad is the module that acts as a bridge between the higher-level logic and the robot's hardware level.

The `Serial` class provides a serial RS-232 communication link with the Arduino board and methods for reading messags and writing commands on the link. Messages are in the form

```
COMMAND [; ARGUMENT]* <newline>
```

Where `COMMAND` is a character signifying a command (for example '`a`' to acknowledge a previously received message, '`m`' to issue a motor actuation order) followed by a list of optional arguments separated by semicolons, and then terminated by a newline character. There is also a helper class called `RobotStats` that parse status messages coming from the Arduino, containing information such as the IR sensor readings, the current direction and power applied to the motors on a scale from 0 (still) to 255 (full throttle).

An instance of each of these classes is used by the `ArduQuadUnit` class to communicate with the Arduino board, transmitting to and receiving messages from the robot's decisional center (explained in section 5.3.2 on the previous page).

5.3.4. RBCSight

RBCSight is the module that is concerned about everything related to vision: video stream processing, SLAM (see section 3.1.1 on page 23), Visual Odometry and the environmental map. The pivotal center of the module is the `CameraUnit` class, which initializes all other classes that are relevant to these purposes and provides visual information to *RBCBrain*.

5.3.4.1. Video stream processing

Video streams, whether coming from a real camera or a recorded video file, are managed by implementations of `VideoInput` (Figure 5.3), namely `Camera` and `FileReader`; these classes can then use either OpenCV (see section 4.3.1 on page 41) or libav (go to section 4.3.2 on page 42) as a backend to provide actual video stream reading capabilities. Frames can be either returned directly in the form of a OpenCV `cv::Mat` object or as an RGB image that can be streamed through a YARP network ports. The latter approach is exactly what it is used in the framework: in this way multiple programs across the network – such as RBCControl, explained in section 5.3.5 on page 52 – can visualize the video stream with ease if they ever need to.

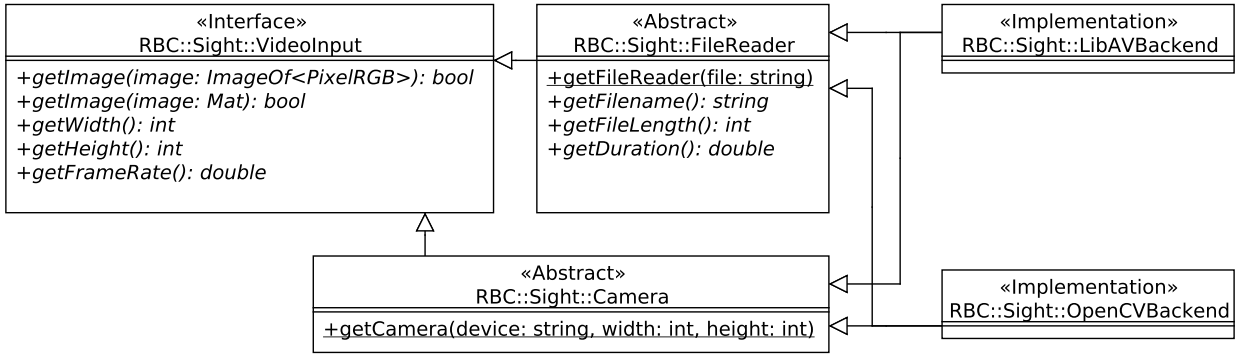


Figure 5.3.: UML Class diagram for the video stream handling classes.

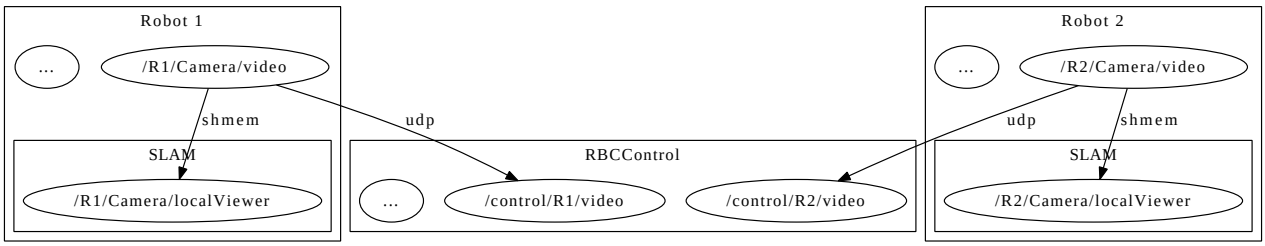


Figure 5.4.: A robot video stream is transmitted over the network so that it can be used by multiple programs and modules.

5.3.4.2. SLAM and Visual Odometry

SLAM is implemented using the world-centric, Framed Inverse-Depth algorithm presented in section 3.1.1.4 on page 26. It massively uses OpenCV's functions and classes to execute all algrbraic calculations. The implementation is divided in three main classes: `Slam`, `FID2DSlam` and `LandmarkData`. `Slam` provides a template for world-centric SLAM implementations and methods that are common to every possible implementation such as feature recognition, image patch matching and bounding box calculations; `FID2DSlam` is the actual implementation of the FID algorithm presented earlier, and finally `LandmarkData` is a helper class that stores information about landmarks such as their anchor's position within the EKF filter.

5.3.4.3. Robots and items recognition

To make a robot easily recognizable by other robots, it is dressed with a specific color hue so that it can be detected through blob recognition techniques. RBCSight accomplishes this by using OpenCV's functions to threshold only certain ranges of HSV color-space values, which are given to the robot by the game supervisor at the beginning of the match. Once the three components – Hue, Saturation (colorfulness) and Value (brigthness) – are filtered in three separated binary images, they are combined through a bitwise AND operator to obtain a final binary picture that identifies the blob.

This technique is less accurate but faster than Connected Component Labelling [51], as labelling requires further steps such as extracting edges with detectors such as Canny [22], connectivity checks on every pixel of the image: the robot is not interested in finding *all* blobs in the image, but only those that it is interested in.

5.3.4.4. Environment map

The environment map is tightly tied to the SLAM classes, as its landmarks are composed of both visual features that are still within SLAM's EKF filter, and features that have been removed from the filter itself. This information is then enriched with datacoming from the blob recognition classes, for example the position of the Control Point.

5.3.5. RBCControl

RBCControl is a desktop application to create and supervise a robotic game developed with the RBC Framework. It allows the user to define connections to robots and edit a game definition through the Gamestate library explained in section 5.4, and it is written using the wxWidgets (see section 4.3.6 on page 44) and VTK (see section 4.3.7 on page 44) libraries for visualization tasks, as well as YARP to handle network connections.

5.3.5.1. Robot initialization and handling

RBCControl relies on Gamestate's actor definitions (see section 5.4.1) to store information about robot initialization; this allows the user to edit most aspects of the game through a single interface.

The application identifies each robot through a unique *name*, and remotely executes robot applications through SSH connections; once this connection is established and the robot program is running, RBCControl connects to two robot-provided YARP ports, `/<RobotName>/log` and `/<RobotName>/Brain`, respectively in order to receive the program's textual log produced by FFLog (read section 5.5 on page 56) and to send commands and data to the application itself.

5.4. Gamestate

Gamestate is a library to define and execute gameplay logic for robot games, in the form of a *finite state machine* (FSM). Although as it was initially thought to be used by RBCControl, it can be easily used stand-alone in other projects. A state machine can be either defined programmatically or through an XML textual description; the resulting FSM can then be saved to an XML file for later use.

The central class of Gamestate (Figure 5.5 on the facing page) is `StateMachine`, which allows the user to fire `Events` triggered by players, *start*, *stop* and *reset* the FSM. A peculiar feature of Gamestate is that a state machine can be *multi-layered*: an action in the current state can make a call to another graph, allowing therefore the game to be divided in different sub-sections while keeping the overall structure of the single graphs relatively simple. To track where the game is at, the class uses an *execution stack* with pointers to the current `Graph` and `State`. Finally, by exploiting the *Observer* design pattern [35], `StateMachine` allows subclasses of `Observer` to subscribe and be notified when important things happen, such as the notification of an event or the value change of a *variable*.

5.4.1. Actors

To define players, interactable objects and zones, Gamestate uses instances of the `Actor` class. Every object has a unique *name*, a type and a series of *attributes* stored as string key-value pairs, to allow maximum flexibility in their use.

Each `Actor` can then be bundled into a `Group`; every instance of this class is homogeneous – a group can contain either only players, items, etc. – so players can then form up into teams and

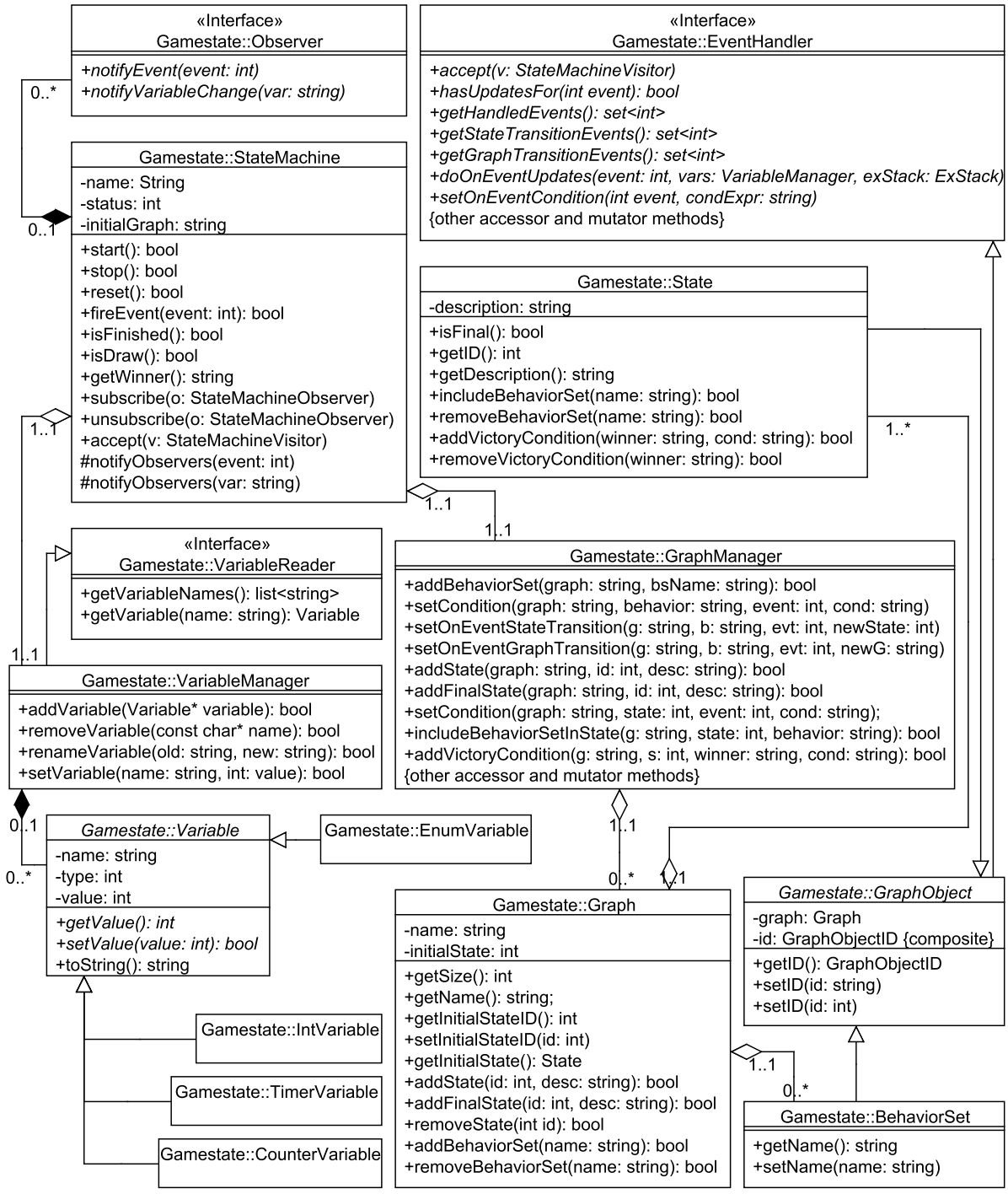


Figure 5.5.: The UML class diagram for the main classes of the Gamestate library.

other objects can be clustered into sets. Finally, if needed a `Group` can decide to activate only a part of its members, to introduce a degree of variability ad uncertainty in games: as an example, in Robotic Battlefield Control the player can lay out multiple Control Points, but only one will be activated by the game.

5.4.2. Variables

Variables in Gamestate are defined by the abstract class `Variable`, which at the core treats all variables as integer numbers, but their behaviors can be very different. For example, an `EnumVariable` acts like an enumeration in programming languages – in which only a few select values are valid, while subclasses of `ThreadVariable` such as `CounterVariable` and `TimerVariable` increase or decrease their value independently of what happens during the game, and can trigger `Events` when their maximum or minimum value is reached. This behavior can also be achieved by `IntVariable` instances.

Variables are handled by `VariableManager` helper class; to permit read-only access to `Variable` instances from external software components, the class implements the `VariableReader` interface.

5.4.3. Graphs

Every `Graph` object is identified with a unique *name* and has an *initial state* which acts as the *entry point* of the graph itself, as with standard finite state machines. Graphs are managed through the `GraphManager` helper class, and can hold two kinds of objects: *states* and *behavior sets*, both of which contain a number of *state updates*.

5.4.3.1. States

A `State` is one of the possible situations or stages of the game. It is represented by an ID number which is unique within the `Graph`, and can be set as either *final* or *non-final*.

Non-final states represent normal situations that can happen as the game unfolds, while final states are the *exit points* of a graph: when the game evolution reaches a final state of the main game graph, the game is considered over and a *winner* – or a *draw* – can be declared; if the final state belongs to any other graph, then the execution of the current graph is terminated and the previous graph is resumed (more on this in section 5.4.4 on page 56).

5.4.3.2. OnEnter Updates, OnEvent Updates and Behavior Sets

State updates come in two flavors: *OnEnter* and *OnEvent*.

`OnEnterUpdates` are triggered as soon as the game enters in a new state. This kind of update can act only on `Variable` values or behaviors, such as setting a new current value or stopping or restarting a timer.

On the other hand, `OnEventUpdates` respond to a specific `Event` being triggered, and in addition to updating variables' values they allow the game to move *either* to another `State` or to another `Graph`. Since updates of this kind can be often repeated in distinct groups within the same graph, they can be defined in *behavior sets*.

Each `BehaviorSet` instance is named and can be referenced in any `State` that has been defined within the `Graph` where the set is declared. As an example, Robotic Battlefield Control implements a behavior set named "combat", in which all Events related to fighting actions are enclosed: in this way the definition of states in which combat between robots is allowed is much simpler and without repetitions that could also make the definition itself more prone to error.

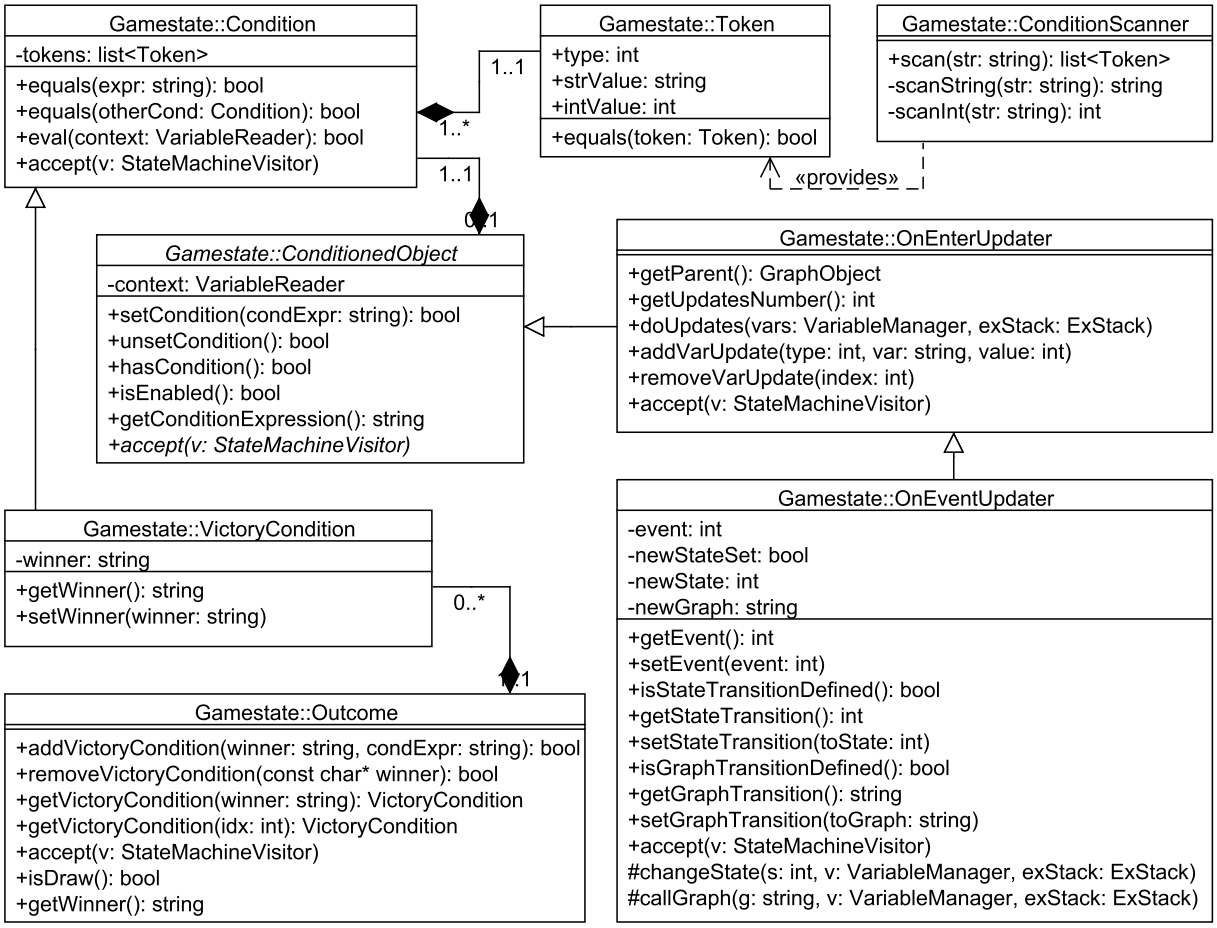


Figure 5.6.: The UML class diagram for Conditions and Conditioned Objects within Gamestate.

5.4.3.3. Conditioned Objects

Some classes within Gamestate are *conditioned* (Figure 5.6), meaning that they are logic-wise in a *true* state only when a certain *boolean expression* is satisfied. Expressions are written in a very C-like syntax: an example taken from Robotic Battlefield Control could be

```
red.visual_enemy_contact == 1 and blue.activity != 6
```

meaning “Blue is in Red’s sight and Blue is not fleeing”.

This kind of condition is particularly useful for state updates, since the user can then define subtle behavior nuances depending on the variables’ actual value, without having to define many states or behavior sets with only slight changes. When a `ConditionedObject` has no expression attached to it, it is always considered as true/enabled.

5.4.3.4. Outcome and Victory Conditions

All final `State` objects[2] can define an `Outcome` of the game; outcomes are essentially a list of `VictoryCondition` instances – one for every participating player or team – that are checked

[2]Of course, outcomes make sense only for final states that reside in the main graph of the game.

once the final state itself is reached. Obviously, there can be only a single winner party; if any of these conditions is not met, then the game is declared a draw.

5.4.4. Events and Event Handling

Events in Gamestate are represented by numeric IDs, and they can either be fired by a value variation of a Variable, or by the user directly.

Once `StateMachine` receives an event notification (Figure 5.7 on the next page) through the method `fireEvent()`, it checks whether the event is valid in the current State; if it is, the state machine's `Variable` objects are modified according to the OnEvent updates declared for that event. If no transition is defined in the update, then the event handling procedure terminates, otherwise the ExStack object is updated according to the kind of transition – either to a new `State` or `Graph`.

If there has been a transition, what happens next depends on the kind of `State` that has been reached:

- if the new `State` is marked as non-final, then its OnEnter updates are executed and the handling procedure ends;
- if the new `State` is final and it is located in a subgraph – that is, a graph called by another graph – then the `Graph` execution is considered over, its pointer removed from the `ExStack` and the previous `Graph` is resumed, executing the OnEnter updates of its last visited `State`;
- if the new `State` is final and it is located in the main `Graph`, then the game is declared to be over.

5.5. FFLog

The *Fast and Flexible Log library*, or *FFLog*, is a very small and expandable logging utility that allows a program to redirect its textual messages to the console, a file, a network socket and any other I/O software facility.

The main class of the library is `Log`, and offers a series of methods to write messages with a different level of criticality – debug, information, warning and error messages – which allows for C-like string formatting; only the messages that are at least as critical as the current log level will be posted. The `LogPolicy` interface is inherited by classes that want to supply a way for FFLog to write messages: examples of this are `ConsoleLogPolicy`, `FileLogPolicy` and `YarpLogPolicy`, and users can easily define more policies in their own programs.

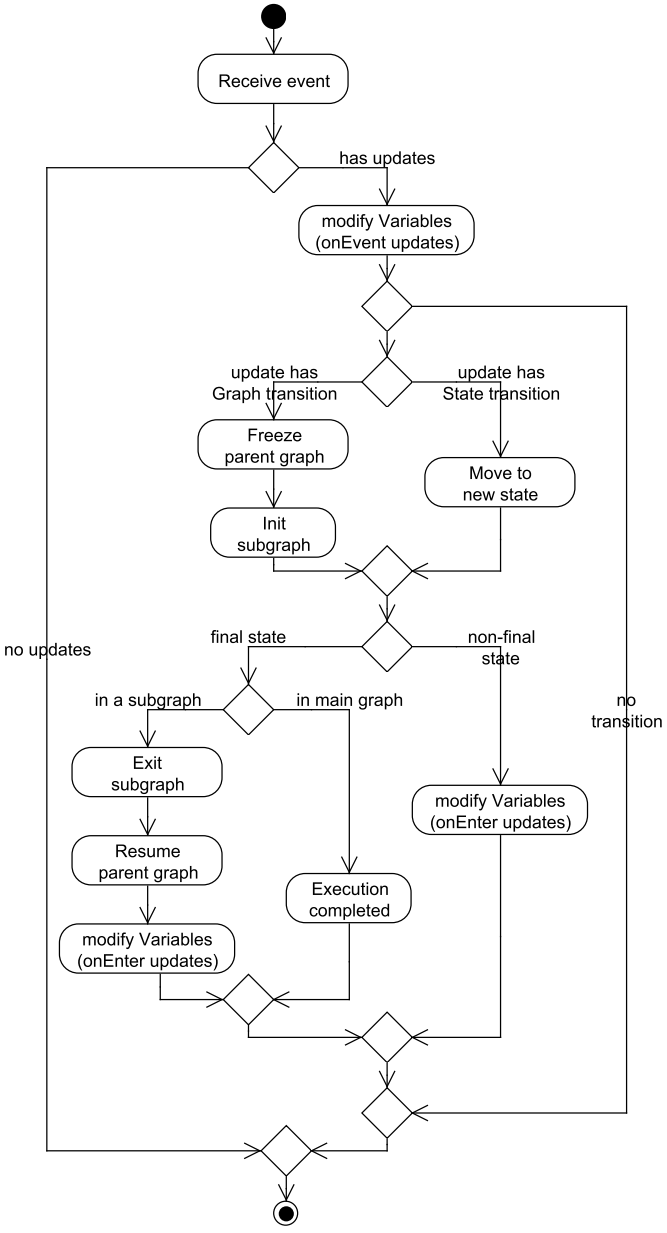


Figure 5.7.: The UML activity diagram for event handling.

6. The implementation

In this chapter I show the results of what I managed to achieve during the development of the project.

Most of the time was spent on developing the SLAM algorithm, as there are no concrete C/C++ implementations of it which does not involve the use of bulky frameworks such as the Robot Operating System, and the core goal of the project was to make everything as light as possible in order to make computations for the autonomous robot feasible directly on its computer board, without the intervention of an additional remote computer – as for navigational purposes, the autonomous robot should be in theory independent from everything else.

6.1. The game logic

6.1.1. Gameplay

The gameplay mechanics are implemented through a Gamestate state machine, as explained in appendix A on page 67. It has to work on an external computer, as having it running on robots would bring synchronization issues between the two participants.

6.1.2. The autonomous robot AI

The autonomous robot's artificial intelligence is divided into two main blocks: the Mr. BRIAN decisional engine and the *path planner*. The former executes all the behaviors that are required for the robot to win the game while abiding by its rules, while the latter implements the potential fields algorithm explained in section 3.3.1 on page 32 and bases its decisions on data provided by RBCSight's environmental map (section 5.3.4.4 on page 52), and it is launched whenever Mr. BRIAN requests a new waypoint where to move.

6.1.2.1. Mr. BRIAN behaviors

Mr. BRIAN's *behaviors* are divided into four priority levels; in this section I will list all of these behaviors, together with their activation conditions (*cando* and *want*, which respectively denote the *possibility* and the *actual wish* of fulfilling a certain action) and the variables they set when they are activated. For a full list of variables and variable types, please refer respectively to table 6.1 on the next page and table 6.2 on page 61.

- **Level 1**: this is the least important level, and it is used for short-term navigation purposes.
  - **GoToWaypoint**: the robot moves towards the current waypoint. If the robot is not aligned with the waypoint's bearing, it carries out an on-place rotation until its heading is aligned with it; when it moves forward, its speed decreases proportionally with the proximity of an obstacle.

    cando: whenever a waypoint is set (`WaypointDistance` is different from *Unset*);

    want: always;

    variables: `LeftEngines`, `RightEngines`.

	Variable	Type	Description
Input	`ObstacleFront`	IRLongRange	The distance, in meters, from an IR sensor to an obstacle. The range depends on the variable type.
	`ObstacleBack`	IRLongRange	
	`ObstacleFrontLeft`	IRShortRange	
	`ObstacleFrontLeft`	IRShortRange	
	`WaypointDistance`	Distance	The beeline distance, in meters, from the robot's current position to the current waypoint.
	`WaypointBearing`	Heading	The bearing, in degrees, of the current waypoint with respect to the robot's current heading. In other words, objects that are directly in front of the robot have a bearing of 0°, objects that are to the right have a bearing of 90°, and so on.
	`CPStatus`	Ownership	The current ownership of the Control Point.
	`SelfActivity`	Activity	The current activity of the robot.
	`SelfOutcome`	Outcome	The current game outcome for the robot.
	`SelfSeesEnemy`	Boolean	Set to true whenever the robot sees its opponent, false otherwise.
	`SelfPosition`	Position	The playing field zone where the robot currently is.
	`SelfActivity`	Activity	The current activity of the robot.
	`SelfKnowsCP`	Boolean	Set to true if the robot knows the location of the Control Point, false otherwise.
Output	`Destination`	Position	Tells where the robot wants to go. Not that this doesn't indicate a precise waypoint, just a region of the playing field.
	`RequestWaypoint`	Boolean	Set to true whenever the decisional engine requests a new waypoint to the path planner, false otherwise.
	`Shoot`	Boolean	Set to true when the robot wants to shoot its weapon.
	`LeftEngines`	Speed	The speed at which the left wheel motors should rotate.
	`RightEngines`	Speed	The speed at which the right wheel motors should rotate.

Table 6.1.: The variables used in the definition of Mr. BRIAN's behaviors for the game.

VARIABLE TYPE	UNIT	MF NAME	MF TYPE	VALUE
IRLongRange	m	Unknown	Singleton	0
		VeryClose	Trapetium	[0.10, 0.15, 0.20, 0.25]
		Close	Trapetium	[0.20, 0.25, 0.40, 0.45]
		Far	Trapetium	[0.40, 0.45, 0.50, 0.50]
IRShortRange	m	Unknown	Singleton	0
		Close	Trapetium	[0.05, 0.05, 0.10, 0.15]
		Far	Trapetium	[0.10, 0.15, 0.15, 0.15]
Distance	m	VeryClose	Triangle_OL	[0, 0.05, 0.20]
		Close	Trapetium	[0.05, 0.20, 0.45, 0.50]
		Far	Trapetium	[0.45, 0.50, 0.80, 0.90]
		VeryFar	Triangle_OR	[0.80, 0.90, 1]
Heading	deg	North1	Trapetium	[0, 0, 30, 60]
		East	Trapetium	[30, 60, 120, 150]
		South	Trapetium	[120, 150, 210, 240]
		West	Trapetium	[210, 240, 300, 330]
		North2	Trapetium	[300, 330, 360, 360]
Ownership	enum	Neutral	Singleton	0
		Blue	Singleton	1
		Red	Singleton	2
Outcome	score	Losing	Triangle_OL	[-30, -5, 0]
		Draw	Triangle	[-10, 0, 10]
		Far	Triangle_OR	[0, 5, 30]
Activity	enum	Move	Singleton	1
		Capture	Singleton	2
		Neutralize	Singleton	3
		Defend	Singleton	4
		Retreat	Singleton	5
		Contest	Singleton	6
Position	enum	RedHomefield	Singleton	0
		BlueHomefield	Singleton	1
		Midfield	Singleton	2
		CP	Singleton	3
Boolean	enum	False	Singleton	0
		True	Singleton	1
Speed	enum	VeryHighBck	Singleton	-255
		HighBck	Singleton	-191
		MedBck	Singleton	-127
		LowBck	Singleton	-63
		Halt	Singleton	0
		LowFwd	Singleton	63
		MedFwd	Singleton	127
		HighFwd	Singleton	191
		VeryHighFwd	Singleton	255

Table 6.2.: The variable types used in the definition of the autonomous robot's behaviors.

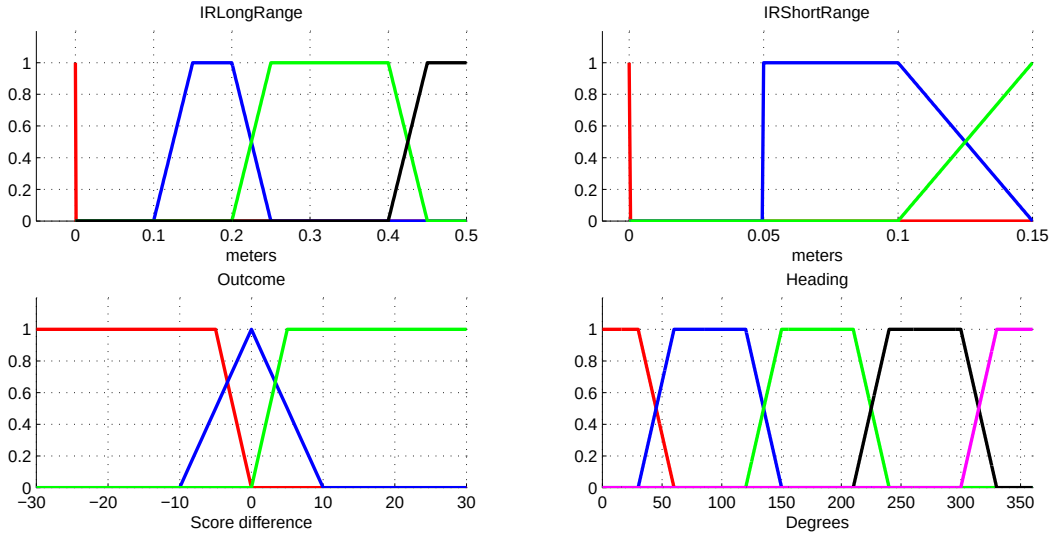


Figure 6.1.: Fuzzy membership functions of several fuzzy variables.

- **RequestWaypoint**: the robot asks for a new waypoint.

  cando: always;

  want: when the current waypoint has been reached (`WaypointDistance` is *VeryClose*);

  variables: `RequestWaypoint`.

- **Level 2**: this level of behaviors is used for strategic reasoning: depending on the current game situation, the robot will activate the most appropriate behavior.
  - **SearchForCP**: the robot searches for the position of the Control Point.

    cando: always;

    want: only when the Control Point location is still unknown (`SelfKnowsCP` is set to false);

    variables: `Destination`.
  - **SearchForOpponent**: the robot searches for the opposing robot.

    cando: always;

    want: the Control Point must be under the robot's control (`CPStatus` is set to *Red*) and the robot is winning the game (`SelfOutcome` is clearly *Winning*);

    variables: `Destination`.
  - **CaptureCP**: the robot captures or neutralizes the Control Point.

    cando: when the robot knows the Control Point location (`SelfKnowsCP` is set to true) and the CP does not belong to *Red*;

    want: the robot is not at the CP position already;

    variables: `Destination`.
  - **DefendCP**: the robot defends a currently owned Control Point.

    cando: when the robot knows the Control Point location (`SelfKnowsCP` is set to true) and the CP belongs to *Red*;

want: the robot is not winning the game (`SelfOutcome` is either *Losing* or *Draw*) and the robot is not already at the CP position;

variables: `Destination`.

– **ShootOpponent**: the robot shoots at its opponent; the robot might want to rotate in order to aim to its enemy.

cando: whenever the robot sees its opponent (`SelfSeesEnemy` is set to *true*);

want: always;

variables: `Shoot`, `LeftEngines`, `RightEngines`.

- **Level 3**: this level is used to override any strategic reasoning.
  – **Retreat**: the robot returns back to its Homefield.

  cando: always;

  want: when `Activity` is set to *Retreat* and the robot is not at its Homefield already.

  variables: `Destination`.
- **Level 4**: this is the most important level, as it is used to avoid collisions with objects.
  – **DoNotCollide**: the robot halts in order to prevent a collision with an obstacle.

  cando: always;

  want: whenever either `ObstacleFront` or `ObstacleRear` are *VeryClose*.

  variables: `LeftEngines`, `RightEngines`.

6.2. SLAM on the development computer

When run on my development computer (Figure 6.2), the SLAM algorithm managed to achieve a decent performance by finding and tracking 20 to 30 features with a frequency of 15 frames per second. In the figure, all features recognized in the frame by FAST are marked in blue, while features that are currently in the filter and have been successfully detected by the algorithm are marked in green. Red squares indicate where the algorithm presumed to find the landmarks, while the magenta boxes are their respective bounding boxes.

Odometry is quite accurate, as it can predict with a 2-3 centimeters resolution the fact that the camera moves back and forth between two points of the plane.

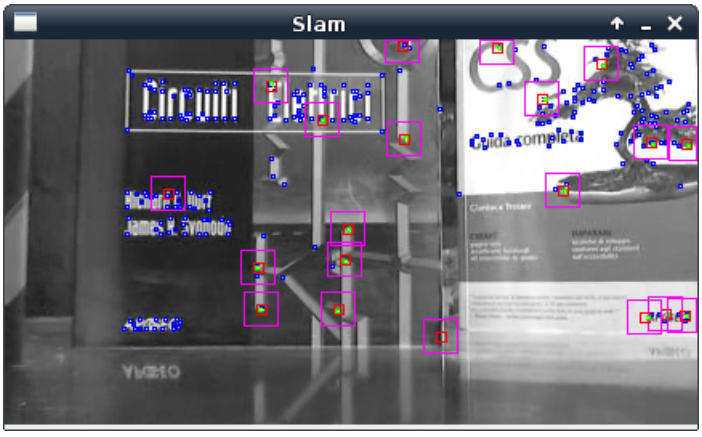


Figure 6.2.: A frame captured by the camera while running my SLAM implementation.

7. Conclusions and future developments

Both the ODROID and the RaspberryPi platforms proved to be not good candidates for the implementation of a visual SLAM algorithm. The causes are mainly two:

1. *The overhead introduced by an off-the-shelf USB web camera is high* relatively to the computational power of the ARM chips present on both computer boards: on the Raspberry, the CPU topped 60% usage just with the *RBCSight* module running without SLAM (see section 5.3.4 on page 50). This is probably due to the current implementation state of *USB Video Class* (UVC) webcam drivers in the Linux kernel – version 3.10.25 for the Raspberry, 3.8.13 for the ODROID – and most importantly the *overhead generated by the USB port* drivers, even when using a modest resolution such as 432x240 pixels; this is also verifiable when examining the performances of the USB wi-fi network adapters, which often drop packets and are able to produce a throughput that is a fraction of what the network itself can support; this happens also when using drivers provided directly by the network interface producers.

2. *The core of the EKF-SLAM algorithm isn't really prone to parallelization*, as the sequence of operations that has to be carried out is strictly sequential in nature. The only operation that can be carried out in parallel is the map construction (see section 3.1.1.7 on page 31), as pose tracking requires read/write access to the whole EKF filter data. This of course taxes the processor, and is an hindrance for the exploitation of multi-core architectures such as the Samsung Exynos found on the ODROID.

Despite my efforts to mitigate these two issues, I have not managed to find a solution that could bring a reliable visual SLAM implementation on embedded devices.

7.1. Future developments

Of course, the SLAM algorithm has lots of room for improvement: for example it should exploit the sparseness of certain matrices used by the SLAM algorithm [23], but in my opinion the fundamental problem of the algorithm is its lack of parallelizable parts. Therefore, my conclusion is that EKF-SLAM is not adequate either for ODROID, nor for RaspberryPi boards and that there is a need for new proposals that can exploit multi-core architectures if we want to see this kind of technology running on embedded architectures, which are clearly trending towards clusters of many-core processors, together with the rise of technologies such as OpenCL which is enabling general-purpose computation capabilities (GPGPU, General Purpose Graphics Processing Unit) on programmable video chipsets.

A. The Robotic Battlefield Control state machine

In this appendix I explain how the state machine of the game is implemented through the Gamestate state machine library (see section 5.4 on page 52).

A.1. Variables

Variables in the game are of two types, *player* or *global*. The former are related to a player exclusively (and therefore are named in a dot-notation fashion, like `red.position`), while the latter are variables that refer to the game as a whole. Table A.1 on the next page reports all variables that are used in the Robotic Battlefield Control game.

A.2. States

The states of the game are roughly in three subgraphs, depending on the current status of the Control Point, which starts out as being neutral – i.e. no one is controlling it, and therefore no one is gaining points. In the following sections I will list all of the states, expliciting the value of variables when the state forces them to have a certain value and omitting them when this is not the case. The overall graph of the state machine can be seen in Figure 2.1 on page 21.

A.2.1. Neutral Control Point

Ss: Beginning of the match (not shown)

Red: `position` = Homefield, `score` = 0, `activity` = Explore, `visual_enemy_contact` = false

Blue: `position` = Homefield, `score` = 0, `activity` = Explore, `visual_enemy_contact` = false

Game: `game_time` = 0, `cp_timer` = 0, `cp` = Neutral

S0: Both players are in home field

Red: `position` = Homefield, `activity` = Explore, `score` < max

Blue: `position` = Homefield, `activity` = Explore, `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Neutral

NAME	TYPE	VALUES	DESCRIPTION
`position`	enum	"Red Homefield", "Blue Homefield", "Midfield", "Control Point"	**Player variable**. The position of the player within the playing field.
`activity`	enum	"Move", "Capture", "Neutralize", "Defend", "Retreat", "Contest"	**Player variable**.What the player is doing at the moment. Notice that *Move* does not necessarily mean that the robot is moving: it can be doing anything that is not mentioned in the other possible activities. Shooting is not mentioned as a possible activity as it is an instantaneous action.
`visual_enemy_contact`	bool	true or false	**Player variable**. Indicates whether the robot is seeing its opponent.
`score`	int	[0, max_score = 80]	**Player variable**. Indicates the advancement toward the game's goal for a player.
`cp_timer`	timer	[0, max_time = 5]	**Global variable**. Indicates the amount of time passed since a CP capture or neutralization attempt by a player.
`cp`	enum	"Neutral", "Red", "Blue"	**Global variable**. Indicates which player owns the Control Point.
`game_time`	timer	[0, game_timeout = 600]	**Global variable**. Indicates how many seconds have passed since the start of the game; when it reaches its threshold, the match is declared over.

Table A.1.: The variables that define the state of the game in Robotic Battlefield Control.

S1: Both players are in midfield

Red: `position` = Midfield, `activity` = Explore | Retreat, `score` < max

Blue: `position` = Midfield, `activity` = Explore | Retreat, `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Neutral

S2: Red is in midfield, Blue is in home field

Red: `position` = Midfield, `activity` = Explore | Retreat, `score` < max

Blue: `position` = Homefield, `activity` = Explore, `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Neutral

S3: Blue is in midfield, Red is in home field

Red: `position` = Homefield, `activity` = Explore, `score` < max

Blue: `position` = Midfield, `activity` = Explore | Retreat, `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Neutral

S4: Red is conquering CP, Blue is in midfield

Red: `position` = CP, `activity` = Capture, `score` < max

Blue: `position` = Midfield, `activity` = Explore | Retreat, `score` < max

Game: `game_time` < game_timeout, `cp_timer` > 0 && `cp_timer` < max_time, `cp` = Neutral

S5: Red is conquering CP, Blue is in home field

Red: `position` = CP, `activity` = Capture, `score` < max

Blue: `position` = Homefield, `activity` = Explore, Health = max, `score` < max

Game: `game_time` < game_timeout, `cp_timer` > 0 && `cp_timer` < max_time, `cp` = Neutral

S6: Blue is conquering CP, Red is in midfield

Red: `position` = Midfield, `activity` = Explore | Retreat, `score` < max

Blue: `position` = CP, `activity` = Capture, `score` < max

Game: `game_time` < game_timeout, `cp_timer` > 0 && `cp_timer` < max_time, `cp` = Neutral

S7: Blue is conquering CP, Red is in home field

Red: `position` = Homefield, `activity` = Explore, `score` < max

Blue: `position` = CP, `activity` = Capture, `score` < max

Game: `game_time` < game_timeout, `cp_timer` > 0 && `cp_timer` < max_time, `cp` = Neutral

S8: Neutral CP is disputed

Red: `position` = CP, `activity` = Explore, `score` < max

Blue: `position` = CP, `activity` = Explore, `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Neutral

A.2.2. Red Control Point

S9: Both players are in home field

Red: `position` = Homefield, `activity` = Explore, `score` > 0 && `score` < max

Blue: `position` = Homefield, `activity` = Explore, `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Red

S10: Both players are in midfield

Red: `position` = Midfield, `activity` = Explore | Retreat, `score` > 0 && `score` < max

Blue: `position` = Midfield, `activity` = Explore | Retreat, `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Red

S11: Red is in midfield, Blue is in home field

Red: `position` = Midfield, `activity` = Explore | Retreat, `score` > 0 && `score` < max

Blue: `position` = Homefield, `activity` = Explore, `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Red

S12: Blue is in midfield, Red is in home field

Red: `position` = Homefield, `activity` = Explore, `score` > 0 && `score` < max

Blue: `position` = Midfield, `activity` = Explore | Retreat, `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Red

S13: Red is defending CP, Blue is in homefield

Red: `position` = CP, `activity` = Defend, `score` > 0 && `score` < max

Blue: `position` = Homefield, `activity` = Explore, `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Red

S14: Red is defending CP, Blue is in midfield

Red: `position` = CP, `activity` = Defend, `score` > 0 && `score` < max

Blue: `position` = Midfield, `activity` = Explore | Retreat, `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Red

S15: CP is disputed (Red prevents Blue from neutralizing)

Red: `position` = CP, `activity` = Explore, `score` > 0 && `score` < max

Blue: `position` = CP, `activity` = Explore, `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Red

S16: Blue is neutralizing CP, Red is in homefield

Red: `position` = Homefield, `activity` = Explore, `score` > 0 && `score` < max

Blue: `position` = CP, `activity` = Neutralize, `score` < max

Game: `game_time` < game_timeout, `cp_timer` > 0 && `cp_timer` < max_time, `cp` = Red

S17: Blue is neutralizing CP, Red is in midfield

Red: `position` = Midfield, `activity` = Explore | Retreat, `score` > 0 && `score` < max

Blue: `position` = CP, `activity` = Neutralize, `score` < max

Game: `game_time` < game_timeout, `cp_timer` > 0 && `cp_timer` < max_time, `cp` = Red

A.2.3. Blue Control Point

S18: Both players are in home field

Red: `position` = Homefield, `activity` = Explore, `score` < max

Blue: `position` = Homefield, `activity` = Explore, `score` > 0 && `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Blue

S19: Both players are in midfield

Red: `position` = Midfield, `activity` = Explore | Retreat, `score` > 0

Blue: `position` = Midfield, `activity` = Explore | Retreat, `score` > 0 && `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Blue

S20: Red is in midfield, Blue is in home field

Red: `position` = Midfield, `activity` = Explore | Retreat, `score` > 0

Blue: `position` = Homefield, `activity` = Explore, `score` < max && `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Blue

S21: Blue is in midfield, Red is in home field

Red: `position` = Homefield, `activity` = Explore, `score` > 0

Blue: `position` = Midfield, `activity` = Explore | Retreat, `score` < max && `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Blue

S22: Blue is defending CP, Red is in homefield

Red: `position` = Homefield, `activity` = Explore, `score` < max

Blue: `position` = CP, `activity` = Defend, `score` > 0 && `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Blue

S23: Red is defending CP, Blue is in midfield

Red: `position` = Midfield, `activity` = Explore | Retreat, `score` < max

Blue: `position` = CP, `activity` = Defend, `score` > 0 && `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Blue

S24: CP is disputed (Blue prevents Red from neutralizing)

Red: `position` = CP, `activity` = Explore, `score` < max

Blue: `position` = CP, `activity` = Explore, `score` > 0 && `score` < max

Game: `game_time` < game_timeout, `cp_timer` = 0, `cp` = Blue

S25: Red is neutralizing CP, Blue is in homefield

Red: `position` = CP, `activity` = Neutralize, `score` < max

Blue: `position` = Homefield, `activity` = Explore, `score` > 0 && `score` < max

Game: `game_time` < game_timeout, `cp_timer` > 0 && `cp_timer` < max_time, `cp` = Blue

S26: Red is neutralizing CP, Blue is in midfield

Red: `position` = CP, `activity` = Neutralize, `score` < max

Blue: `position` = Midfield, `activity` = Explore | Retreat, `score` > 0 && `score` < max

Game: `game_time` < game_timeout, `cp_timer` > 0 && `cp_timer` < max_time, `cp` = Red

A.2.4. Game over

S27: Game over (not shown)

`red.score` = max | `blue.score` = max | `game_time` = game_timeout

A.3. Transitions

A.3.1. Movement

Movement transitions determine a state change as one of the robots moves in a new zone of the playing field.

T0: Red enters in the midfield.

T1: Red enters its home field.

T2: Blue enters in the midfield.

T3: Blue enters its home field.

T4: Red arrives to the Control Point.

T5: Blue arrives to the Control Point.

A.3.2. Capturing and neutralizing

Any time a capture or a neutralization attempt is successful, `cp_timer` signals a **T6** event to announce it.

A.3.3. End-game

All end-game transitions shift the game to enter the final state (**S27**). **T7** triggers when the game timer expires, while **T8** and **T9** are triggered when respectively the Red and Blue robot reach their score threshold.

A.3.4. Combat

Transitions that are related to combat are all self-loops – they don't move the game execution to a new state. To be able to fire, a robot needs to see the opponent, and this opponent has to be healthy, i.e. its activity variable must not be set to "Retreat".

T10: Red gets visual contact of Blue: `red.visual_enemy_contact` is set to true.

T11: Red loses visual contact of Blue: `red.visual_enemy_contact` is set to false.

T12: Blue gets visual contact of Red: `blue.visual_enemy_contact` is set to true.

T13: Blue loses visual contact of Red: `blue.visual_enemy_contact` is set to false.

T14: Red shoots and hits Blue: `blue.activity` is set to "Retreat".

T15: Blue shoots and hits Red: `red.activity` is set to "Retreat".

B. The Gamestate XML file syntax

In this appendix I am going to illustrate the XML syntax to define a Gamestate state machine by providing excerpts from the Robotic Battlefield Control state machine definition file.

B.1. The overall structure

The root element for a state machine is `<stateMachine>`, which takes a `name` and a reference to the `main graph` as attributes. This element then contains lists of `<actors>`, `<variables>` and `<events>`, followed by declarations of graphs.

```
<stateMachine name="Robotic Battlefield Control"
  mainGraph="main">
  <actors>
    <!-- ...actor declarations... -->
  </actors>
  <variables>
    <!-- ...variable declarations... -->
  </variables>
  <events>
    <!-- ...event declarations... -->
  </events>
    <!-- ...graph declarations... -->
</stateMachine>
```

B.2. Actors

An actor is an item or player that has a role in the game. Each `<actor>` element takes a `name` and a `type` as attributes, and can contain one or more `<attribute>` elements which are key/value textual pairs. Attributes are just a mean for the user to store information about the items and players, they are not directly used within Gamestate.

```
<actor name="red" type="player">
  <attribute key="autonomous">true</attribute>
  <attribute key="remoteCommand">
    ssh $USER@$IP_ADDRESS $ROBOT_NAME</attribute>
</actor>
```

Actors can be put into groups; if the `random` attribute is set to n, then only n of all the listed items will be activated throughout a match.

```
<group type="item" random="1">
  <actor name="control_point_1" type="item">
    <attribute key="color">#FF0000</attribute>
  </actor>
  <actor name="control_point_2" type="item">
    <attribute key="color">#00FF00</attribute>
  </actor>
  <actor name="control_point_3" type="item">
    <attribute key="color">#0000FF</attribute>
  </actor>
</group>
```

B.3. Events

Events are indicated by a simple numeric ID, with an optional textual description.

```
<event id="0" desc="Red enters in the midfield" />
```

B.4. Variables

B.4.1. Integer (int)

Integer variables are defined with `<int>` elements, which can be used in two ways:

1. The default variable's behavior is to act as a boolean, accepting either 0 or 1 as possible values. In the following example, the variable `red.visual_enemy_contact` is declared like this, and is initialized with the value *false*.

   ```
   <int name="red.visual_enemy_contact" value="0">
   ```

2. Alternatively, the user can define different minimum and maximum values for the variable; moreover, the user can let the variable trigger an event as soon as one of the limit values are reached, through the `<onMinValueReached>` and `<onMaxValueReached>` declarations.

   ```
   <int name="red.health" min="0" max="4" value="4">
     <onMinValueReached event="30" />
   </int>
   ```

B.4.2. Enumeration (enum)

Enumerations variables are denoted by the `<enum>` element and are similar to integers, but unlike those, the user can define explicitly all the valid values, giving them also an optional string as `description`.

```
<enum name="red.position" value="0">
  <value desc="Red Homefield">0</value>
  <value desc="Blue Homefield">1</value>
  <value desc="Midfield">2</value>
  <value desc="CP">3</value>
</enum>
```

B.4.3. Timer

Timers are defined by `<timer>` elements, which lets the user specify a duration for the `timer` itself and the `event` ID that should be triggered once the timer reaches the defined threshold.

```
<timer name="game_time" time="600" event="10" />
```

B.4.4. Counter

Counters, indicated by the `<counter>` element, are integer variables that are automatically updated every `every` seconds of an amount specified by the `incr` parameter; when they reach their `threshold`, they trigger an event specified by the argument of `event`.

```
<counter name="red.score" initial="0" threshold="30"
  event="11" every="1" incr="1" />
```

B.5. Graphs

Each definition of a graph in a state machine is enclosed in a `<graph>` element, which has a `name` and an `initial state` ID as attributes; the element then incorporates a series of state declarations within a `<states>` element, followed by a list of zero or more behavior set declarations.

```
<graph name="main" initialState="0">
  <states>
    <!-- ...state declarations... -->
  </states>
  <!-- ...behavior set declarations... -->
</graph>
```

B.5.1. States

States are defined by `<state>` elements, and each of them is characterized by a unique `id` code and an optional textual `description`.

A state can have an enclosed `<onEnter>` element in which one or more `<update>` actions can be listed; each update is defined by an `action` – either *add*, *set*, *reset*, *stop*, *start* – which result is dependent on the kind of variable it acts upon: for instance, both "starting an integer variable" and "adding 2 to a timer" have no sense; finally, an action can have a value that could influence the result of the action: for example adding -2 to an integer variable will actually *decrease* that variable by 2. `<onEvent>` updates are similar, but are triggered by an event, and in addition they can contain either a `<moveToState>` or a `<moveToGraph>` statement.

Lastly, states can include a behavior set through the `<include>` elements.

```
<state id="0" desc="Neutral CP, both players are in home field">
  <onEnter>
    <update action="set" var="red.position" value="0" />
    <update action="set" var="blue.position" value="1" />
    <update action="set" var="red.activity" value="1" />
    <update action="set" var="blue.activity" value="1" />
    <!-- no action if timer already started -->
    <update action="start" var="game_time" />
  </onEnter> <!-- common events -->
  <!-- include a behavior set -->
  <include set="common" />
  <!-- onEvent updaters -->
  <onEvent ref="0"> <!-- Red enters midfield -->
    <moveToState ref="2" />
  </onEvent>
  <onEvent ref="2"> <!-- Blue enters midfield -->
    <moveToState ref="3" />
  </onEvent>
</state>
```

B.5.1.1. Final States

States can be set as final; in this case they contain an `<outcome>` element, which is a list of conditioned `<winner>` statements that refer to a player `<actor>`.

```
<state id="27" desc="Game over" final="true">
  <outcome>
    <if cond="red.score == 30">
      <winner ref="red"/>
    </if>
    <if cond="blue.score == 30">
      <winner ref="blue"/>
    </if>
  </outcome>
</state>
```

B.5.2. Conditions

`<onEvent>` updates can be conditioned by a certain boolean expression.

```
<if cond="blue.activity != 5">
  <onEvent ref="13">
    <update action="set" var="red.visual_enemy_contact"
      value="1" />
  </onEvent>
</if>
```

B.5.3. Behavior Sets

Behavior sets are named lists of `<onEvent>` updates, which can be conditioned or not conditioned.

```
    <behaviorSet name="common">
      <if cond="blue.activity != 5">
        <onEvent ref="13">
          <update action="set" var="red.visual_enemy_contact"
            value="1" />
        </onEvent>
        <onEvent ref="14">
          <update action="set" var="red.visual_enemy_contact"
            value="0" />
        </onEvent>
      </if>
      <if cond="red.activity != 5">
        <onEvent ref="15">
          <update action="set" var="blue.visual_enemy_contact"
            value="1" />
        </onEvent>
        <onEvent ref="16">
          <update action="set" var="blue.visual_enemy_contact"
            value="0" />
        </onEvent>
      </if>
      <!-- Game time expired -->
      <onEvent ref="10">
        <moveToState ref="1000" />
      </onEvent>
    </behaviorSet>
```

Bibliography

[1] The ArchLinux ARM page for the RaspberryPi. http://archlinuxarm.org/platforms/armv6/raspberry-pi.

[2] The Arduino official website. http://arduino.cc.

[3] The CMake project website. http://www.cmake.org.

[4] Hardkernel's website. http://www.hardkernel.com.

[5] The Libav project website. http://libav.org.

[6] NVIDIA's website. http://www.nvidia.com.

[7] The OpenCV project website. http://opencv.org.

[8] The Pidora project website. http://pidora.ca.

[9] The RaspberryPi and RaspberryPi Foundation official website. http://www.raspberrypi.org.

[10] RaspberryPi official software resources. https://github.com/raspberrypi.

[11] The Raspbian project website. http://www.raspbian.org.

[12] The Robot Operating System project website. http://www.ros.org.

[13] The Visualization ToolKit (VTK) project website. http://www.vtk.org.

[14] The wxGlade project website. http://wxglade.sourceforge.net.

[15] The wxWidgets project website. http://wxwidgets.org.

[16] The YARP project website. http://wiki.icub.org/yarpdoc.

[17] J. Andrade-Cetto, T. Vidal-Calleja, and A. Sanfeliu. Unscented transformation of vehicle states in SLAM. *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 323–328, 2005.

[18] Tim Bailey, J. Nieto, J. Guivant, M. Stevens, and E. Nebot. Consistency of the ekf-slam algorithm. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 3562–3568, Oct 2006.

[19] T.D. Barfoot. Online visual motion estimation using fastslam with sift features. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 579–585, 2005.

[20] J. Barraquand and J.-C. Latombe. A Monte-Carlo algorithm for path planning with many degrees of freedom. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 1712–1717, 1990.

[21] J. Barraquand and J.-C. Latombe. Robot motion planning: A distributed representation approach. *International Journal of Robotics Research*, 10(6):628–649, December 1991.

[22] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679–698, Nov 1986.

[23] S Ceriani, D Marzorati, M Matteucci, and DG Sorrenti. Single and multi camera simultaneous localization and mapping using the extended kalman filter on the different parameterizations for 3d point features. *Journal of Mathematical Modelling and Algorithms in Operations Research*, 2013.

[24] Simone Ceriani. *Conditional Independent Visual SLAM with Integrated Bundle Adjustments*. PhD thesis, Politecnico di Milano, 2013.

[25] Simone Ceriani, Daniele Marzorati, Matteo Matteucci, Davide Migliore, and Domenico G Sorrenti. On feature parameterization for ekf-based monocular slam. *proceedings of 18th World Congress of the International Federation of Automatic Control (IFAC)*, pages 6829–6834, 2011.

[26] Denis Chekhlov, Mark Pupilli, Walterio Mayol-Cuevas, and Andrew Calway. Real-time and robust monocular slam using predictive multi-resolution descriptors. In George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Paolo Remagnino, Ara Nefian, Gopi Meenakshisundaram, Valerio Pascucci, Jiri Zara, Jose Molineros, Holger Theisel, and Tom Malzbender, editors, *Advances in Visual Computing*, volume 4292 of *Lecture Notes in Computer Science*, pages 276–285. Springer Berlin Heidelberg, 2006.

[27] E. Ward Cheney and David R. Kincaid. *Numerical Mathematics and Computing*. Cengage Learning, 2012.

[28] Javier Civera, Oscar G. Grasa, Andrew J. Davison, and J. M. M. Montiel. 1-point ransac for ekf-based structure from motion. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3498–3504, 2009.

[29] Javier Civera, Oscar G. Grasa, Andrew J. Davison, and J. M. M. Montiel. 1-point ransac for extended kalman filtering: Application to real-time structure from motion and visual odometry. *Journal of Field Robotics*, 27(5):609–631, 2010.

[30] Andrew J. Davison. Real-time simultaneous localisation and mapping with a single camera. 2003.

[31] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. 2007.

[32] H. Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *Robotics Automation Magazine, IEEE*, 13(2):99–110, June 2006.

[33] E. Milios F. Lu. Globally consistent range scan alignment for environment mapping. 1997.

[34] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

[35] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[36] Giuseppina Gini and Vincenzo Caglioti. *Robotica*. Zanichelli, 2003.

[37] Chris Harris and Mike Stephens. A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference*, pages 147–151, 1988.

[38] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2004.

[39] Kin Leong Ho and Paul Newman. Loop closure detection in slam by combining visual and spatial appearance. *Robotics and Autonomous Systems*, 54:740–749, 2006.

[40] S.A. Holmes, G. Klein, and D.W. Murray. An o(n2) square root unscented Kalman filter for visual simultaneous localization and mapping. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31(7):1251–1263, 2009.

[41] Georg Klein and David Murray. Parallel tracking and mapping for small AR workspaces. In *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*, Nara, Japan, November 2007.

[42] Georg Klein and David Murray. Parallel tracking and mapping on a camera phone. In *Proc. Eigth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'09)*, Orlando, October 2009.

[43] K. Konolige and M. Agrawal. Frameslam: From bundle adjustment to real-time visual mapping. *Robotics, IEEE Transactions on*, 24(5):1066–1077, Oct 2008.

[44] J.-C. Latombe. *Robot Motion Planning*. Kluwer, Boston, MA, 1991.

[45] D.G. Lowe. Object recognition from local scale-invariant features. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157, 1999.

[46] D. Nister. An efficient solution to the five-point relative pose problem. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(6):756–770, June 2004.

[47] D. Ortín and J. M. M. Montiel. Indoor robot motion based on monocular images. *Robotica*, 19(3):331–342, May 2001.

[48] P. Pinies and J.D. Tardos. Large-scale SLAM building conditionally independent local maps: Application to monocular vision. *IEEE Trans. Robot.*, 24(5):1094–1106, 2008.

[49] Edward Rosten and Tom Drummond. Fusing points and lines for high performance tracking. In *IEEE International Conference on Computer Vision*, volume 2, pages 1508–1511, October 2005.

[50] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *European Conference on Computer Vision*, volume 1, pages 430–443, May 2006.

[51] H. Samet and M. Tamminen. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 10(4):579–586, Jul 1988.

[52] D. Scaramuzza, F. Fraundorfer, and R. Siegwart. Real-time monocular visual odometry for on-road vehicles with 1-point ransac. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 4293–4299, May 2009.

[53] Davide Scaramuzza, Andrea Censi, and Kostas Daniilidis. Exploiting motion priors in visual odometry for vehicle-mounted cameras with non-holonomic constraints. 2011.

[54] William J. Schroeder, Ken Martin, and William E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *IEEE Visualization*, pages 93–100, 1996.

[55] J. Shi and C. Tomasi. Good features to track. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR '94., 1994 IEEE Computer Society Conference on*, pages 593–600, Jun 1994.

[56] Robert Sim, Pantelis Elinas, and James J. Little. A study of the Rao-Blackwellised particle filter for efficient and accurate vision-based SLAM. *International Journal of Computer Vision*, 74(3):303–318, Jul 2007.

[57] Stephen M. Smith and J. Michael Brady. SUSAN – a new approach to low level image processing. *International Journal of Computer Vision*, 23(1):45–78, 1997.

[58] J. Solà, A. Monin, M. Devy, and T. Lemaire. Undelayed initialization in bearing only slam. In *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 2499–2504, Aug 2005.

[59] Joan Solà. Consistency of the monocular ekf-slam algorithm for three different landmark parametrizations. In *ICRA*, pages 3513–3518. IEEE, 2010.

[60] Henrik Stewenius, Christopher Engels, and David Nister. Recent developments on direct relative orientation. *{ISPRS} Journal of Photogrammetry and Remote Sensing*, 60(4):284 – 294, 2006.

[61] Hauke Strasdat, J M M Montiel, and Andrew J Davison. Real-time monocular SLAM: Why filter? *IEEE International Conference on Robotics and Automation*, 2010.

[62] Hauke Strasdat, J. M. M. Montiel, and Andrew J. Davison. Scale drift-aware large scale monocular slam. In Yoky Matsuoka, Hugh F. Durrant-Whyte, and José Neira, editors, *Robotics: Science and Systems*. The MIT Press, 2010.

[63] The Khronos Group. OpenCL homepage.

[64] Lee Thomason. The TinyXML2 project website. http://www.grinninglizard.com/tinyxml2/index.html.

[65] Sebastian Thrun and Michael Montemerlo. The graphslam algorithm with applications to large-scale mapping of urban structures. *International Journal on Robotics Research*, 25(5):403–430, 2006.

[66] B. Williams and I. Reid. On combining visual slam and visual odometry. In *Proc. International Conference on Robotics and Automation*, 2010.

[67] Lotfi Asker Zadeh. *Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers by Lotfi A. Zadeh (Advances in Fuzzy Systems: Application and Theory)*. World Scientific Pub Co Inc, 1996.