Alexander Meisel

Feature based Sensor Fusion for Victim Detection in the Rescue Robotics Domain

Master's Thesis

Graz University of Technology

Institute for Software Technology

Supervisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa
Co-Supervisor: Ass.Prof. Dipl.-Ing. Dr.techn. Gerald Steinbauer

Graz, May 2014

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ______________________ ______________________

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ______________________ ______________________

Datum Unterschrift

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

In this thesis a victim detection system for mobile robots in search and rescue operations is presented. The system uses feature detectors (e.g. faces and heat) to process the measurements of the robot's sensors and the detected features are then tracked over time to generate feature tracks. The tracking is able to handle false positives as well as false negatives. A rule-based object fusion algorithm combines nearby feature tracks into detected objects. These rules ensure that a detected object contains only reasonable features (e.g. one face per person) and that unspecific features can be shared between objects (e.g. heat signature). An object classifier analyzes each object and its feature tracks and assigns the most likely object type. The objects are then visualized for the human supervisor, who can verify them as true or false victims. The objects are also published so that the other systems of the robot can use them. The proposed system was implemented for a robot using ROS. Finally, the system was evaluated in a RRL-like environment. The results show that the system is able to dedect objects reliable.

Contents

List of Figures

List of Tables

1 Introduction

This thesis presents a framework for victim detection to be used in the RoboCup Rescue Robot League[1]. The Rescue Robot League is an international competition for urban search and rescue robots and part of the RoboCup[2]. Robots of international teams explore a simulated disaster environment and try to find objects of interest like victims, QR codes or hazmat signs. The teams score points for detecting the objects of interest and the accuracy of the generated map of the arena. There are also penalties for harming the victims, colliding with the walls of the arena and detecting false victims. These events could have problematic consequences in a real search and rescue operation. Another important aspect of the competition is that each year the rules get more advanced and more tasks and obstacles are added. The long-term goal of the Rescue Robot League is to develop robots, which can participate in real search and rescue operations without problems.

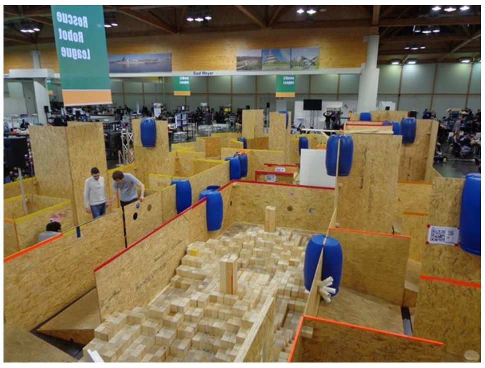


Figure 1.1: The Rescue Arena of the German Open 2014

[1]http://www.robocup.org/robocup-rescue/
[2]http://www.robocup.org

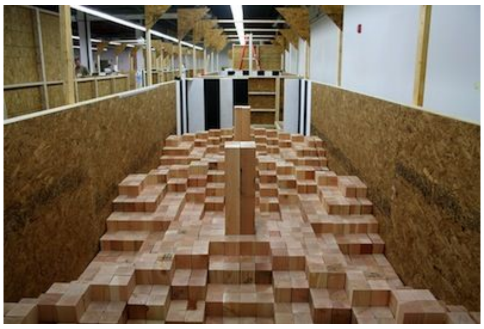


Figure 1.2: Stepfield

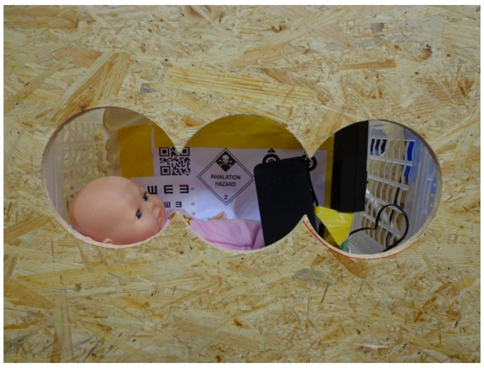


Figure 1.3: Victim Box

The simulated disaster environment (see Figure 1.1) is organized like a maze that contains difficult terrain to make it harder for the robot to move around. These standard test methods from the National Institute of Standards and Technology (NIST) consists of ramps, stairs, stepfields (see Figure 1.2) and closed doors. Ramps, stairs and stepfields were added over the years to foster the development of more complex locomotion system. More difficult obstacles will be added in the future until the robots are good enough to move around in real disaster areas with collapsed buildings and even unstable floors. The closed doors act as shortcuts for robots with manipulators that can open them. The doors also allow the organizers to change the layout of the arena by opening and closing different doors for each team. Some of the obstacles can be avoided by smaller robots. This encourages the teams to build smaller robots.

The Rescue Robot League simulates victims by providing life signs, which are also called features in this thesis. Supported life signs are:

- Form (doll or mannequin parts)
- Visual (eye charts and hazardous materials labels)
- Thermal (heating pad)
- Motion (waving cloth)
- Sound (random numbers)
- CO2 (bicycle tire cartridges)

Each victim has at least two life signs. A heat source and a visual clue are

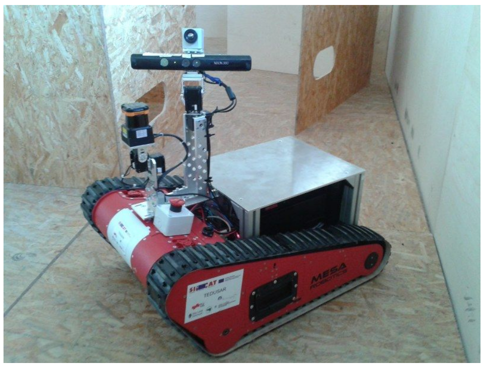


Figure 1.4: The Wowbagger inside a Rescue Arena

always present. The competition uses multiple ways to make the victim detection task harder. It places single life signs around the arena that give a penalty, if wrongly detected as a victim. It uses flashing lights to trick motion detectors, which use visual images for detecting motion. Finally victims are hidden inside boxes (see Figure 1.3) to make the task even harder and to simulate trapped or entombed victims. Those boxes are open on top, open to the side or with a single access hole in any direction. This makes it important to also detect those holes so that the robot can move closer and inspect the inside of the boxes.

The Institute for Software Technology (IST)[3]of TU Graz has developed a robot for search and rescue applications, named Wowbagger (see Figure 1.4), to participate in in the Rescue Robot League. The robot has tracks to move over uneven terrain. It uses a laser scanner to generate a map of the explored environment and localize the robot in it. The robot also has a movable sensor head, which has mounted a RGB-D camera and a thermal camera. The 2 cameras are mainly used for victim detection. Finally the robot has a computer, which runs all the programs needed to operate the robot autonomously. The robot's sensor measurements are transmitted to an operator and can be visualized. The supervisor monitors the autonomous robot, but can also manually control the robot with a joypad.

[3]http://www.ist.tugraz.at/

1.1 Motivation

The short-term motivation of the thesis is to develop a victim detection system that can successfully compete in the Rescue Robot League. The victim detection system must support multiple sensors and feature detectors to handle all required life signs and get as many points as possible. It is also very important that the system works reliable, because finding victims gives more points in the competition than the other tasks. Moreover false positives give a time penalty. This means that the system must be able to handle false measurements.

The long-term motivation is to apply the lessons from the Rescue Robot League in the real world. The final goal is to use robots in real search and rescue operations. This requires improvements all across the board. The terrain in real disaster areas will be even more difficult to navigate. It may even be unstable and dangerous. The environment will be even more cluttered with objects, which will make finding the real victims even harder. Fire, displays and loudspeakers in homes, pets, other searchers and much more may distract and overwhelm the victim detection system. Real operations will also require the cooperation of multiple robots and even the cooperation with human searchers. This can be the simple sharing of detected features and objects or the efficient organization of all the searchers.

1.2 Contribution

The main contribution of this thesis is a framework for victim detection and its evaluation. The framework splits the victim detection task into the following subtasks:

1. feature detection
2. feature tracking
3. object fusion
4. object classification

The framework uses message passing to decouple the feature detection subtask from the rest of the system. This allows users of the framework to add new sensors, feature detectors and even new feature types without modifying the the framework. At runtime the feature detectors can also be started and stopped independent of the system. The decoupling also allows the feature detectors to run on different framerates. Each feature detector can run as fast as the sensor measurements are available.

Tracking features instead of tracking objects also has advantages. One advantage is that the object fusion and object classification subtasks can run on a much lower frequency than the feature detectors and the feature trackers. Another advantage is that the feature trackers discards most false measurements already before they even reach the object fusion subtask.

The object fusion subtask uses a multi-hypotheses clustering algorithm with extra rules that define which feature tracks can or can not be combined into objects. These extra rules prevent an object from having 2 faces, but allow it to have multiple hands or QR codes.

One important advantage of the system is that nearly everything is data-driven. The user can define the feature types, the object types and how they are handled in configuration files.

Finally, the framework supports verification of objects by the user. This allows the operator of the robot to tell the system that a detected object has a wrong type or that the system has missed an object. User verification is very important, because a victim detection system will never work completely perfect. Another use of user verification is the cooperation of robots and humans while searching for victims.

2 Related Work

This chapter describes earlier work of other authors that influenced the development of this victim detection system. The first section describes different feature detection algorithms. Next is an overview of sensor fusion. The final section describes the tracking of objects over time.

2.1 Feature Detection

The paper [VSA03] gives an overview over different color-based skin detectors. It compares different techniques that decide for each pixel if it is skin or not based on the color. It also looks at the effect that the used color space (e.g RGB or HSV) has. Sadly it is very difficult to distinguish between the skin color of the victims and the wooden RoboCup Rescue arena. The influence of sunlight and artificial light sources makes it even harder. Another problem of color-based skin detectors for victim detection is that the skin can be covered in a layer of ash or dust.

Faces are one of the most important features during victim detection. Over the years many different algorithms were invented to detect them. [RBK98] uses neural networks. This thesis uses the detector described in [VJ01] and [LM02] (see 4.2.2). [ASG07] is an improved version of the algorithm.

A new approach for the victim detection is presented in [TPPB08]. It is based on hyperspectral imaging in the near infrared spectral domain. The detector can distinguish between different materials from afar, when the spectra are analyzed. The paper supports the detection of skin, meat, polyester, plastics, wood and carton. It can even recognize them under a thin layer of ash. Another big advantage of this detector is that it can detect undercooled bodies, which is impossible with thermal imaging.

[LLS08] allows to detect more complex object categories like persons, vehicles and animals. Training data is used to find local features (e.g heads, arms and wheels) around interest points and to save their appearance from different directions. The spatial occurrence distribution of each feature is learned from the positions of the feature instances relative to the object center. The algorithm then searches for the features in the input images and each detected feature uses its spatial occurrence distribution to vote for the object's position. An object is detected, if enough features vote for the same location. [JA09] uses the same algorithm to detect persons with a thermal camera.

The paper [DT05] uses Histograms of Oriented Gradient (HOG) to detect persons in color images. HOGs are a good way to describe silhouette contours of objects The algorithm uses a large database of images to learn the HOG of a standing or walking person and compares the new algorithm to existing algorithms trained with the same database. [SA11] uses HOGs to detect persons in the depth image of the Kinect.

The author of [Bur04] gives an overview of the currently available sensors. He compares them based on size, cost, simplicity and robustness. The author also implements feature detectors for a camera, a microphone, a pyroelectric sensor and a thermal camera. Finally he shows that different sensors behave differently under different environmental conditions.

2.2 Sensor Fusion

[KK07] presents a new way to combine the output of simple feature detectors, which produce a large number of false-positives, to detect victims. A genetic algorithm learns relevant neighborhood relations in the 2d image space between features to distinguish between victims and other objects. This knowledge is then used to identify victims by using Markov Random Fields during runtime. The main difference to the victim detection system presented in this thesis is that it only uses the current sensor output. The proposed victim detection system tracks features over time to increase the accuracy of the system.

In [MSK+11] the authors combine the output of different sensors over time to improve the detection rate of victims. The paper uses uses HOGs to detect people and hazmat signs in camera images. It also uses a thermal camera to detect areas within the human body temperature range. An extended Kalman filter is used to update the position and probability of each victim.

The paper [Bur04] uses a fusion method with confidence values. Each feature has a probability and each sensor has a confidence value. The confidence values can change dependent on the environmental condition.

Another approach is presented in [HB03]. The main difference to the above sensor fusion approaches is that it fuses the sensor data before the feature detection step. The proposed method fuses color and infrared videos to detect better contours of humans than one sensor alone is able to.

In the paper [FHL06] multiple techniques to fuse the detected objects from multiple mobile robots are compared. The first technique is a weighted arithmetic mean. The weight is influenced by the distance between the robot and the object, the confidence of the robot's localization and the confidence of the detection. The second technique is a weight grid. Each detected object is rendered as a 2-dimensional Gaussian distribution into the grid. The third technique is a Kalman filter. Finally the paper combines the weight grid technique with the Kalman filter.

2.3 Tracking

The paper [Rei79] presented an algorithm to track multiple targets. It can handle the initiating of new targets, false or missing measurements and even crossing tracks. The algorithm uses multiple competing hypotheses to assign measurements to tracks and create new tracks. For each measurement it creates a hypothesis that it is a false measurement, that it belongs to an existing track and that it creates a new track. The best hypothesis determines the number of tracks, the measurements assigned to them and the current position and velocity of each track. The algorithm also presents techniques to prune hypotheses with a low probability to speed up the tracking. The

Kalman filter [Kal60] is used to predict the movement of tracks, because it helps with missing measurements and even crossing tracks.

[ARS08] combines object detection and tracking to improve victim detection. It uses a part-based object detector, which uses simple detectors to detect parts of a person like hands, heads or feet. It then combines the parts to estimate the position of objects. This makes the detector independent of the pose of the person. The paper improves the detector by tracking the parts with a kinematic limb model over multiple images. The kinematic limb model contains information about the skeleton and the walking cycle of a person.

The authors of [MBM12] present an advanced tracking algorithm for people within groups. The algorithm performs detection-track association as a maximization of a joint likelihood using motion, color appearance and people detection confidence. Each track learns a classifier from the color histogram of the assigned detections to evaluate the color appearance. A Kalman Filter is used to predict the position and velocity on the ground plane. The Mahalanobis distance between a track and a detection is then used to calculate the motion likelihood.

[KUS03] compares the Global Nearest Neighbor and the Suboptimal Nearest Neighbor algorithms for Multiple Target Tracking.

3 Problem Formulation

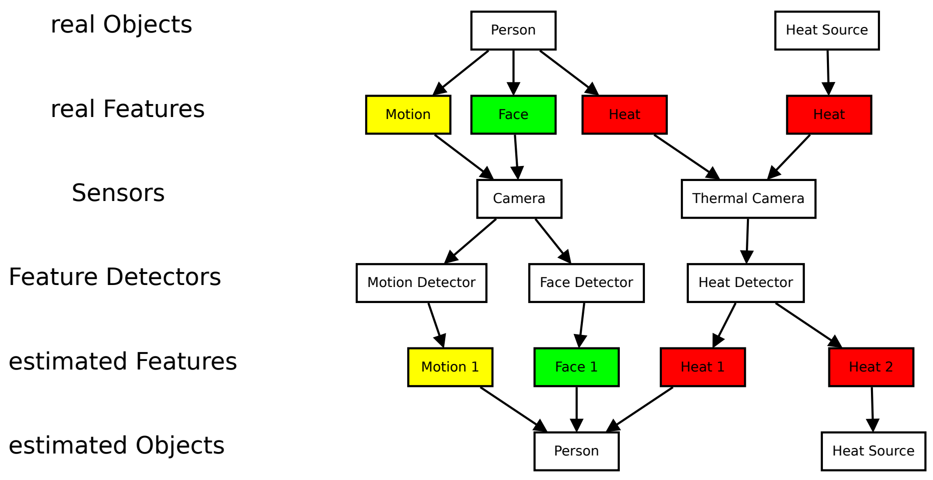


Figure 3.1: An Example of the Problem

In this chapter the victim detection task is formally described. It first describes the objects and their features. Then it describes the robot's sensors and feature detectors. Finally it gives a formal definition of the considered problem. Figure 3.1 shows an example of the with 2 objects and 4 features

There exist n objects $O = \{o_1, o_2, ..., o_n\}$ in the environment. An object o_i has a position $\vec{pos}_i = (x_i, y_i, z_i)^T$ and an object type $obj\_type(o_i) \in OT = \{VICTIM, HEAT\ SOURCE, FACE, HOLE, ...\}$. Each object also has a set of features depended on its object type. A feature f_j has a position $\vec{pos}_j = (x_j, y_j, z_j)^T$ and a feature type $feature\_type(f_j) \in FT = \{MOTION, HEAT, FACE, HOLE, CO2, SOUND, ...\}$.

The robot has a set of l sensors $S = \{s_1, s_2, ..., s_l\}$. The state of the sensor is determined in the 3d space by its 6d pose $(x, y, z, \phi, \theta, \psi)^T$. $(x, y, z)^T$ denotes the position and $(\phi, \theta, \psi)^T$ denotes the orientation. Each sensor provides the data needed to detect a set of feature types. The output of each sensor s_i is the measurement $z_i \in \Re^{M_i}$. For instance $M_i = width \cdot height$ for the cameras used in this thesis. A feature detector $D_i^{type} : \Re^{M_i} \rightarrow \{f_j | feature\_type(f_j) = type\}$ detects a set of features of a feature type $type$ in the measurement z_i of sensor s_i.

The problem is made harder by the inaccuracy of the feature detectors and the sensors they use. The accuracy of a feature detector is defined by the true positive rate and the true negative rate for observing a feature. The true positive rate is the probability that a feature is detected, if there exists an object. The true negative rate is the probability that no feature is detected, if there exists no object.

The first part of the problem is to track the detected features over time. A feature tracking function $T : \{f_1, f_2, ..., f_g\} \rightarrow \{fm_1, fm_2, ..., fm_h\} = fm$ determines the feature tracks from the features detected by all the sensors until now. This can be described as the probability distribution $P(fm | z_{1:l,1:t}, x_{1:t})$. It is the probability of a list of estimated feature tracks fm dependent on the sensor output $z_{1:l,1:t}$ and the robot movement $x_{1:t}$. Each estimated track can be described as $fm_i = (pos_i, feature\_type_i)^T$.

The second part of the problem is to use the feature tracks to estimate the objects located in the environment. An object fusion function $F : \{fm_1, fm_2, ..., fm_h\} \rightarrow \{om_1, om_2, ..., om_p\}$ determines the objects from the feature tracks. This can be described as the probability distribution $P(om|fm)$. It is the probability of a list of estimated objects om dependent on a list of tracked features fm. Each estimated object can be described as $om_i = (pos_i, obj\_type_i)^T$.

The third part of the problem is to assign the correct object type to each object. A classification function $C : \{f_1, f_2, ..., f_k\} \rightarrow OT$ determines the object's type from the features belonging to the object.

4 Solution

This chapter presents a victim detection system that solves the problem described in the previous chapter. An overview of the system can be seen in Figure 4.1. The system consists of multiple processing steps that use the output of the previous step as input.

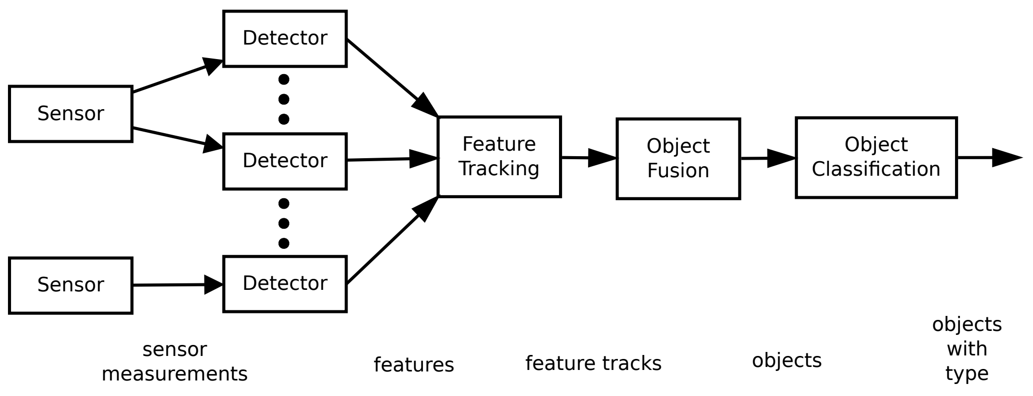


Figure 4.1: System Overview

The first processing step handles the feature detection. It uses a number of feature detectors that process the measurements from multiple sensors to detect features. Currently supported sensors are video cameras, RBG-D cameras (see 4.1.1) and thermal cameras (see 4.1.2). Feature detectors for microphones and CO2 sensors are currently under development. A detector also has to estimate a position in the world coordinate system for each feature. This strongly depends on the sensor type, as can be seen in section 4.1.1 and 4.1.2.

The second processing step handles the tracking of features over time. This step is necessary, because the features received from the last step can be wrong or inaccurate. Detectors can find features, where no features exist, or miss real features. A feature's position can also change from frame to frame drastically. It is affected by the accuracy of the sensor, the feature detector and the transformation to the world coordinate system. Viewing direction is another important factor for cameras. It is impossible to calculate the true center of an object, because cameras can only estimate the width and the height, but not the depth of an object. The feature tracking process combines the features from multiple updates into feature tracks to improve the accuracy. A multi-hypothesis algorithm is used to assign features to existing feature tracks or create new ones. The feature track's probability, position and radius are updated, if it is assigned a feature. The feature track's probability can also decrease, if it is inside the sensor's field of view and is not assigned a feature. Finally feature tracks are deleted, if their probability falls below a threshold. See 4.3 for more details.

The third processing step handles the fusion of feature tracks into objects. A multi-hypothesis clustering algorithm is used to minimize the distance between feature tracks belonging to the same object. Extra rules are needed to handle the special properties of certain feature types. For example an object can only have one face, but multiple QR codes. Certain feature types like heat and motion can be shared between multiple objects. See 4.4 for more details.

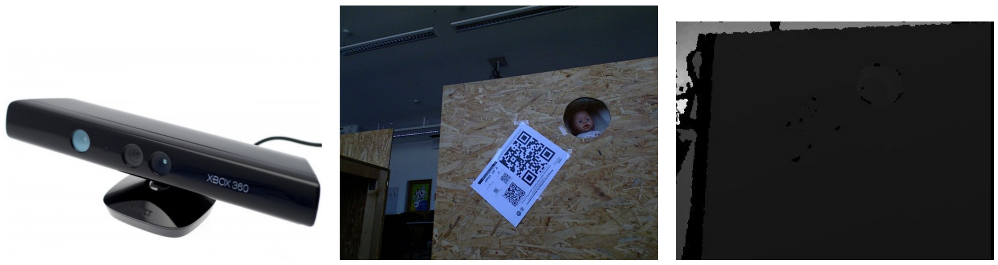
The final processing step handles the classification of the objects. Here each object and its feature tracks are analyzed and assigned the best fitting object type like victim or heat source. See 4.5 for more details.

It is important for the Rescue Robot League and other applications that a human user can confirm a detected victim or mark it as wrongly classified. User verification can also be used to mark objects, after the robot finished inspecting them, to avoid reporting them again. See 4.6 for more details.

4.1 Sensors

This section describes the sensors currently supported and used by the victim detection system.

4.1.1 RGB-D Camera



(a) Device (b) Color Image (c) Depth Image

Figure 4.2: Kinect

The main sensor of the robot is the **Microsoft Kinect**[1](see Figure 4.2(a)) and most features are detected with it. It is an RGB-D camera, which means that it captures RGB color images (see Figure 4.2(b)) and depth images (see Figure 4.2(c)). The Kinect has an RGB camera, a depth sensor and a multi-array microphone. The RBG camera captures color images with a resolution from 640 × 480 pixels up to 1280 × 1024 pixels at a frame rate of 9 Hz to 30 Hz depending on the resolution. The depth sensor consists of an infrared laser projector combined with an infrared camera. The depth sensor captures images with a resolution of 640 × 480 pixels and has a working range of approximately 0.7-6 m.

Transformation using a Depth Image

Converting the feature's 2d position (x,y) in the color or depth image to a 3d world position (X,Y,Z) using the Kinect's depth image consists of multiple

[1] http://en.wikipedia.org/wiki/Kinect

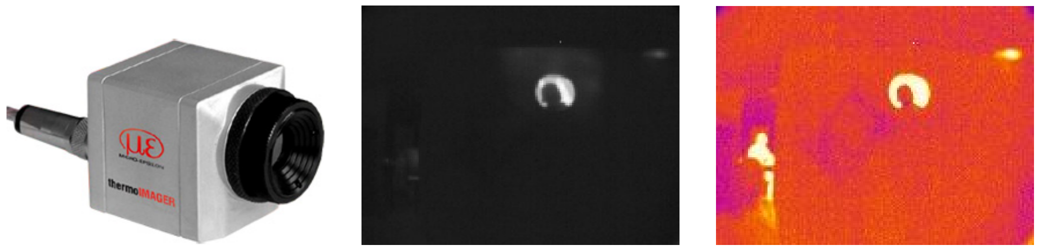
steps. First the value of the depth image at the 2d position is converted to meters. Then the depth and 2d position are used to calculate the 3d position in the camera coordinate system. Finally the local 3d position is transformed into the world coordinate system. See the Equations 4.1, 4.2 and 4.3 for more detail. This method only works correctly inside the working range of the Kinect's depth sensor. A second method for features farther away or for cameras without a depth image can be seen in subsection 4.1.2.

$$depth = \frac{depth\_image(x, y)}{1000.0} \tag{4.1}$$

$$\vec{position}_{camera} = depth \cdot \begin{pmatrix} (x - center_x)/focal\_length \\ (y - center_y)/focal\_length \\ 1.0 \end{pmatrix} \tag{4.2}$$

$$\vec{position}_{world} = transform_{camera\_to\_world} \cdot \vec{position}_{camera} \tag{4.3}$$

4.1.2 Thermal Camera



(a) Device

(b) The Thermal Image is a greyscale image for fast computation.

(c) The Visible Image is a color image for humans.

Figure 4.3: thermoIMAGER TIM 160

Another important sensor of the robot is a **Micro-Epsilon thermoIMAGER TIM 160**[2](see Figure 4.3(a)), which is a compact high speed thermal camera used to detect heat features. The thermal camera captures thermal images (see Figure 4.3(b)) with a resolution of 160 × 120 pixels at a frame rate of 120 Hz. The thermal image is a 16-bit grayscale image, which is easy to process by image processing techniques. The Equations 4.4 and 4.5 allow the simple conversion between temperatures in degree Celsius and the thermal image. The thermal image is less useful for human supervisors, because the useful temperature range of 0 °C to 100 °C looks identically to the human eye. The device driver publishes a second image (see Figure 4.3(c)) to solve this problem. This image concentrates on a much smaller temperature range and maps certain temperatures to certain colors. Blue areas are cold and yellow or white areas are hot.

$$value = temperature \cdot 10.0 + 1000.0 \tag{4.4}$$

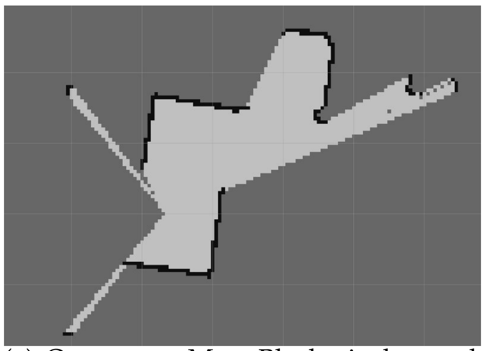
$$temperature = \frac{value - 1000.0}{10.0} \tag{4.5}$$

Transformation using Ray Casting

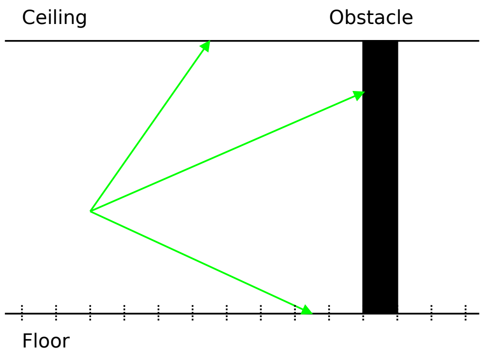
Converting the feature's 2d position (x,y) in an image to a 3d world position (X,Y,Z) without the Kinect's depth image (see 4.1.1) is much more complex and inaccurate. This system uses ray casting in combination with an occupancy map (see Figure 4.4(a)) of the area surrounding the robot. Each cell in the occupancy map represents the probability of occupancy. Low values mean an obstacle-free area, high values mean an obstacle like a wall and -1 is an unexplored area. The occupancy map is used and created by the robot's mapping system.

The first step is to calculate a ray in the camera coordinate system from the 2d position and transform it into the world coordinate system. See Equations 4.6 and 4.7. Then the start and direction of the ray are projected onto the occupancy map. The next step is to perform ray casting. The Bresenham's line algorithm [Bre65] is used to process all cells the ray travels through. The

[2]http://www.micro-epsilon.com/temperature-sensors/thermoIMAGER/thermoIMAGER_160



(a) Occupancy Map: Black pixels are obstacles, light grey pixels are free areas and dark grey pixels are unexplored areas.



(b) Rays can intersect with obstacles, the floor plane or the ceiling plane.

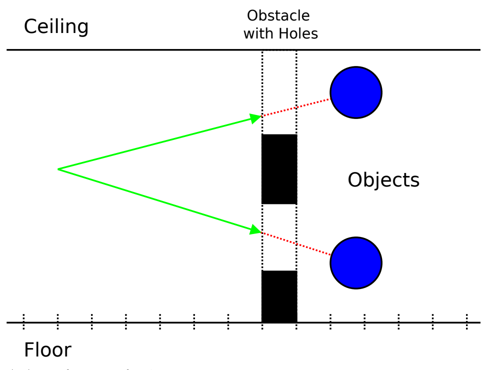
Figure 4.4: Ray Casting Overview

iteration stops, if the ray hits an obstacle. Additional tests are performed to check if the ray hits the user-defined height of the floor or the ceiling. See Figure 4.4(b). These height tests are needed to produce correct results. Without the floor plane objects lying on the floor are projected onto the next obstacle behind them. Without the ceiling plane objects above it and behind an obstacle are projected onto the the obstacles. After a hit the distance between the start point of the ray and the intersection is calculated. The distance is used to calculate the wanted 3d position. See Equation 4.8. The ray casting can fail, if it hits an unexplored cell or the border of the map. The algorithm simply does not know if the ray can travel through that cell or not.

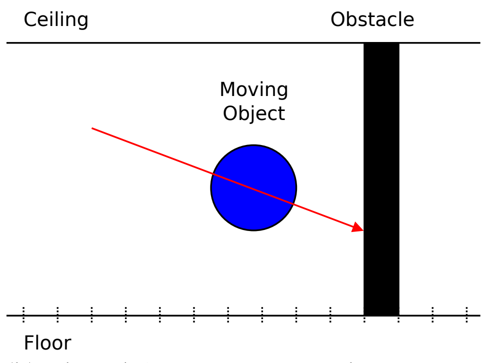
$$\vec{dir}_{camera} = \begin{pmatrix} (x - center_x)/focal\_length \\ (y - center_y)/focal\_length \\ 1.0 \end{pmatrix} \tag{4.6}$$

$$\vec{dir}_{world} = transform_{camera\_to\_world} \cdot \vec{dir}_{camera} \tag{4.7}$$

$$\vec{position}_{world} = \vec{start}_{world} + \vec{dir}_{world} \cdot distance \tag{4.8}$$



(a) The 2d Occupancy Map can not represent obstacles with holes. Rays hit the obstacles instead of the objects visible through holes.



(b) The 2d Occupancy Map does not represent moving objects. Rays hit the obstacles behind moving objects.

Figure 4.5: Ray Casting Problems

This algorithm for calculating the 3d position of a feature has several disadvantages. The biggest disadvantage is that the occupancy map is a 2d approximation of the 3d world. It does not contain information about the height of obstacles. In the real world it is possible to see an object behind an obstacle, if the obstacle is lower than the assumed ceiling or it has one or more holes in it. In contrast the algorithm stops, when the ray hits the first obstacle and returns a false position. See Figure 4.5(a). The second problem is that moving objects are not represented by the occupancy map. This means that the ray can't hit the moving object and continues until it hits the closest obstacle behind the object. See Figure 4.5(b).

4.2 Feature Detection

This section describes the feature detectors currently supported and used by the victim detection system.

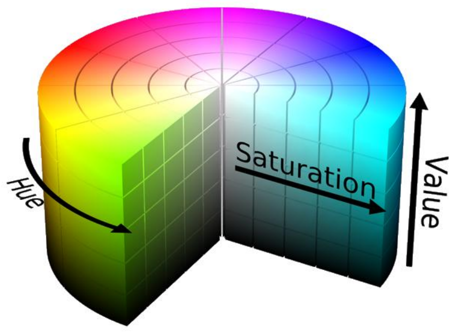


Figure 4.6: HSV Color Space[Sha10]

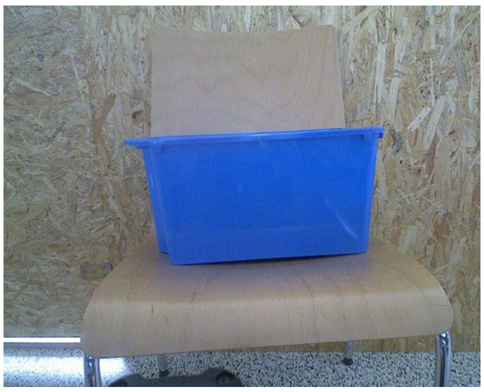
4.2.1 Color Detection

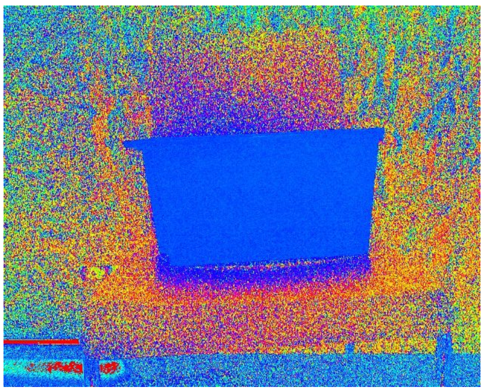
The color detector is the simplest feature detector for the Kinect. It detects areas of a color similar to the target color in the color image. The most difficult part of the algorithm is to choose the correct color space. This detector uses the Hue, Saturation and Value (HSV)[3]color space (see Figure 4.6). The Hue defines the color. Saturation and Value define how the brightness. The target color is mostly defined by a range of hues. The Saturation and Value can vary greatly to allow for the influence of light sources. This is one advantage HSV has over RGB. Another advantage is that it is enough to place a threshold on one axis instead of three to distinguish between two colors.

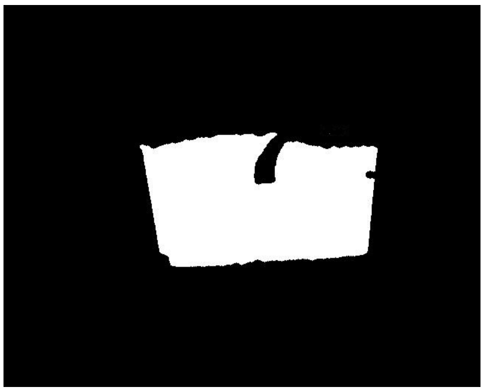
The detector converts the Kinect's color image from the RGB color space (see Figure 4.7(a)) to the HSV color space (see Figure 4.7(b)). It then creates a mask, where only pixel that are close enough to the wanted color are true. Morphological operators [Ser83] are used to improve the result (see Figure 4.7(c)). The opening operator removes areas too small and the closing operator merges nearby areas. Then a segmentation algorithm is used to find all remaining areas in the mask (see Figure 4.7(d)). Finally the 3d positions of the areas are calculated and they are published as features.

The color detector can be used to detect skin, but it has problems while

[3]http://en.wikipedia.org/wiki/HSL_and_HSV


(a) The RGB color image from the Kinect.


(b) The Hue channel of the color image transformed into the HSV color space.


(c) All pixels in the wanted hue range are marked as belonging to a color feature (white). Then the morphological operators are used to remove areas too small.

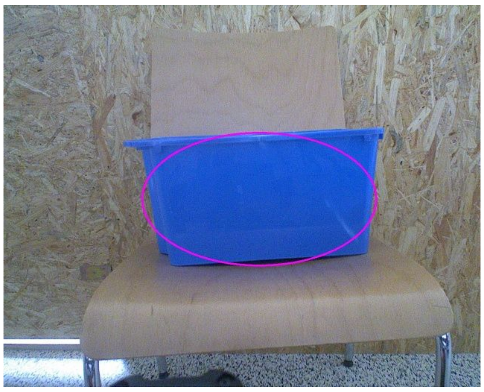

(d) The color feature is marked with a magenta ellipse.

Figure 4.7: Color Detection

used in the Robcup Rescue arena. The color values of the human skin, the baby dolls and the wooden arena are just too similar. See Figure 4.8.

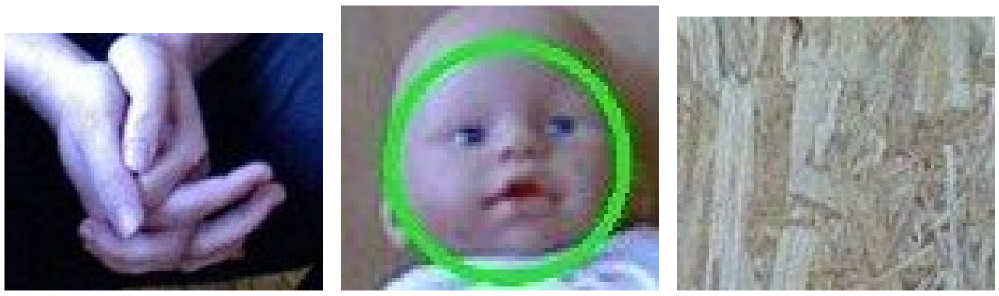

(a) Skin (b) Baby Doll (c) Wood

Figure 4.8: Closeup comparison of skin, a baby doll and the wooden arena shows that the color values are very similar dependent on the lighting.

4.2.2 Face Detection

The face detector uses OpenCV's CascadeClassifier to detect faces in the Kinect's color image. The CascadeClassifier is based on [VJ01] and [LM02]. It is a cascade classifier working with haar-like features for object detection.

Haar-like features are rectangles that are split into multiple rectangles, which can be white or black. The difference between the sums of the pixel values inside the white and dark rectangles is calculated and compared to a threshold. If the difference is above the threshold, the feature exists at the current location in the image. Haar-like features can be used to describe many simple features. Features with two rectangles can detect the border between light and dark areas. Features with three rectangles can detect light or dark lines. An advantage of haar-like features is that summed area tables make them very fast to compute.

The idea behind the cascade classifier is to collect a large number of examples of the wanted object type and find haar-like features that are shared by most examples. For example the eyes are darker than the area above and blow them in most faces. Another example would be that the nose is lighter than the area to the left and right of it. Each haar-like feature is only a weak classifier, but by combining many into a cascade of classifiers it is possible to reliable detect the wanted object. This cascade classifier can be automatically learned form a large collection of positive and negative examples. The OpenCV tutorial[4]contains a learned cascade classifier that is good enough at detecting the faces of persons and baby dolls, but sometimes it wrongly detects faces in the wood grain. See Figure 4.9.

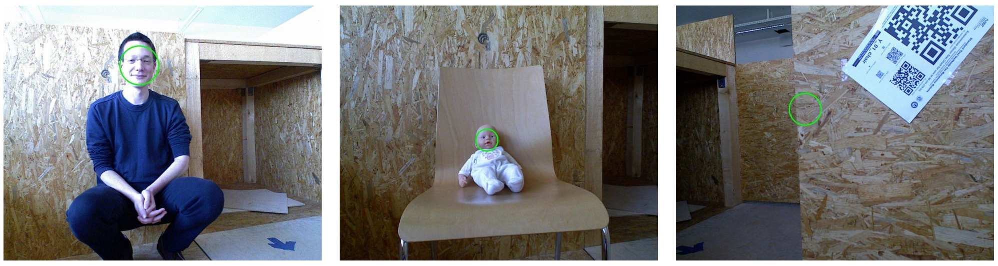

(a) Person (b) Baby Doll (c) A pattern in the wood grain that is repeatedly recognized as face.

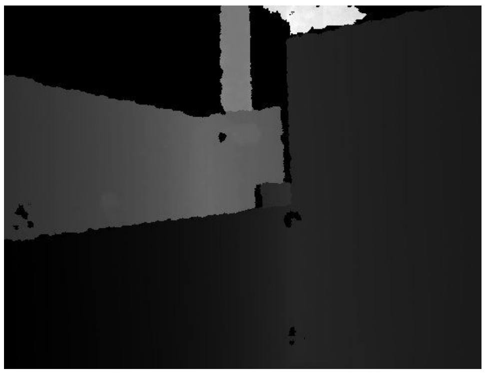
Figure 4.9: Face Detection with detected faces marked with green circles.

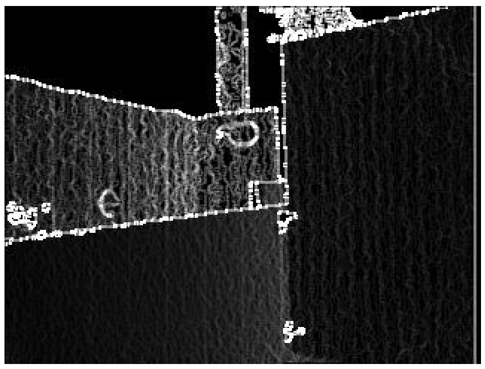
In the detection phase, a search window is moved across the input image and it checks at every location if the classifier detects a face. This is repeated for different sizes of the search window to find faces at different scales. Finally the face's height in meter is calculated with the help of the Kinect's depth sensor. The height makes it possible to discard faces that are too large or too small to belong to a person or a baby doll. It is also an easy way to distinguish between persons and baby dolls.

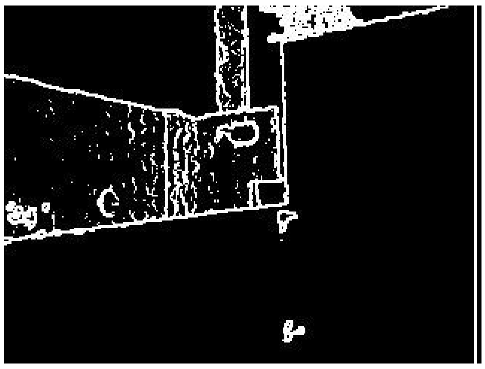
4.2.3 Hole Detection

The hole detection algorithm searches the depth image (see Figure 4.10(a)) for holes. First it calculates the gradient (see Figure 4.10(b)) of the depth image using the Sobel operator. A threshold operator is applied to the gradient image to create a binary image (see Figure 4.10(c)). Pixels with a

[4]http://docs.opencv.org/doc/tutorials/objdetect/cascade_classifier/cascade_classifier.html


(a) Depth


(b) Gradient


(c) Threshold

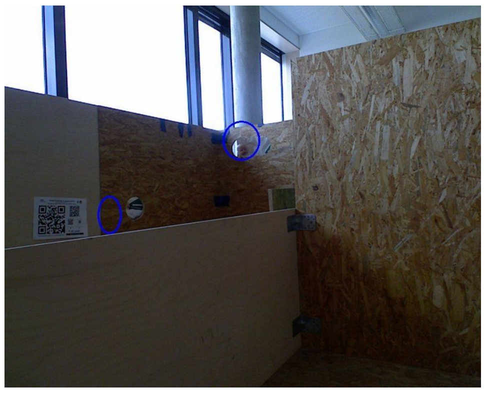
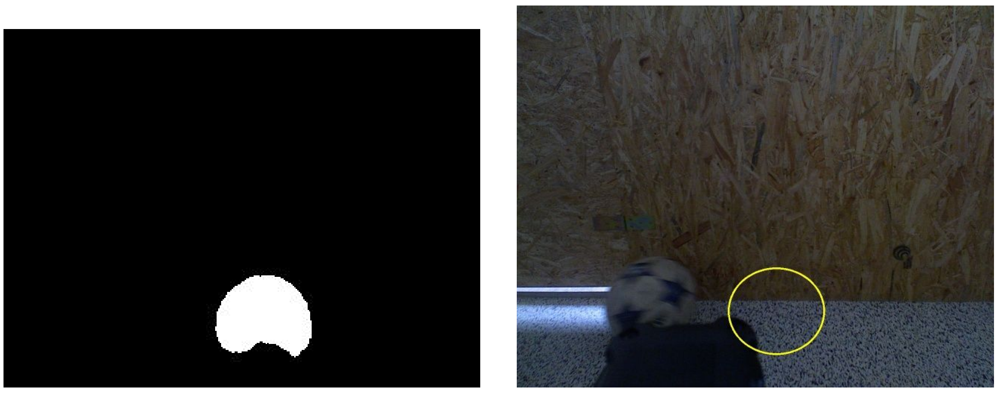

(d) Detected holes marked with blue ellipses.

Figure 4.10: Hole Detection

strong gradient are set to one and other pixels are set to zero. The next step is to search for areas with a low gradient that are completely surrounded by pixels with a high gradient. These areas can be holes or extrusions. Then the depth of an area is compared to the depth of the pixels surrounding it. A real hole has higher depth values, because it is farther away than the area around it. Finally the size of the hole is compared to a user defined minimum and maximum and holes too small or big are discarded. The remaining holes are marked as features in the color image (see Figure 4.10(d)) and published.

4.2.4 Motion Detection


(a) Pixels, where the difference between the old and new depth image is greater than a threshold, are marked as belonging to a motion feature (white).

(b) Motion features are visualized as a yellow ellipse.

Figure 4.11: Motion Detection

The motion detection algorithm compares the current depth image with a previous depth image to detect motion. It calculates the absolute difference between the old and new depth image and applies a threshold operator to create a mask. Pixels that are one belong to a motion feature and the rest of the pixels are zero. The rest of the detector works like the color detector.

The motion detection algorithm uses depth images instead of color images, because they are much less influenced by changing lighting. A motion detector using color images would wrongly detect moving shadows or changing brightness as motion. The Rescue Robot League uses flashing lights to trick color-based motion sensors. Displays showing a video are another error source. The main drawback of using depth images is the small maximum distance of the depth sensor. A solution would be to use the color image for pixels with invalid depth values.

The current implementation has one big drawback. It separates the moving objects from the static background. The problem is that the robot and the Kinect mounted on it move most of the time while searching for victims. This causes large parts of the background to be classified as belonging to a motion feature. The current solution is to only use the motion detection, if the Kinect is not moving or to inspect suspected victims from closeup. Another solution would be to use a more complex and slower motion detection algorithm that works better with a moving camera like [SG99].

4.2.5 QR Code Detection

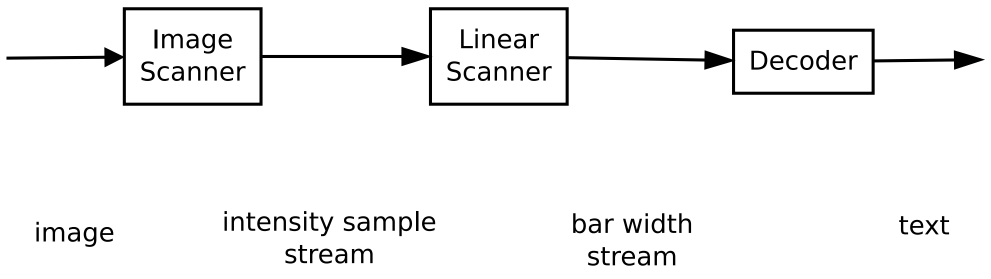


Figure 4.12: Zbar Pipe Overview

The QR code detector uses the Zbar library. [5] It is an open source bar code reader that is inspired by laser scanners. A laser scanner moves a laser

[5] http://zbar.sourceforge.net

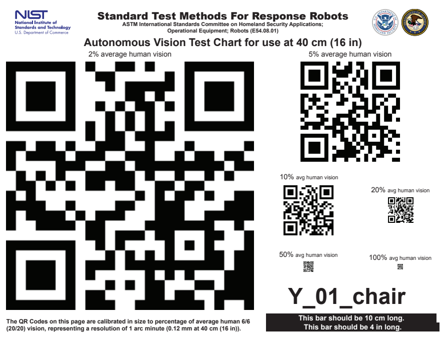

NIST National Institute of Standards and Technology U.S. Department of Commerce
Standard Test Methods For Response Robots
ASTM International Standards Committee on Homeland Security Applications; Operational Equipment; Robots (E54.08.01)
Autonomous Vision Test Chart for use at 40 cm (16 in)
2% average human vision
5% average human vision
10% avg human vision
20% avg human vision
50% avg human vision
100% avg human vision
Y_01_chair
This bar should be 10 cm long.
This bar should be 4 in long.
The QR Codes on this page are calibrated in size to percentage of average human 6/6 (20/20) vision, representing a resolution of 1 arc minute (0.12 mm at 40 cm (16 in)).

Figure 4.13: QR Code Example

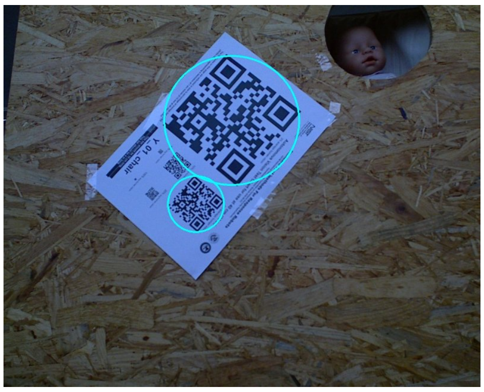


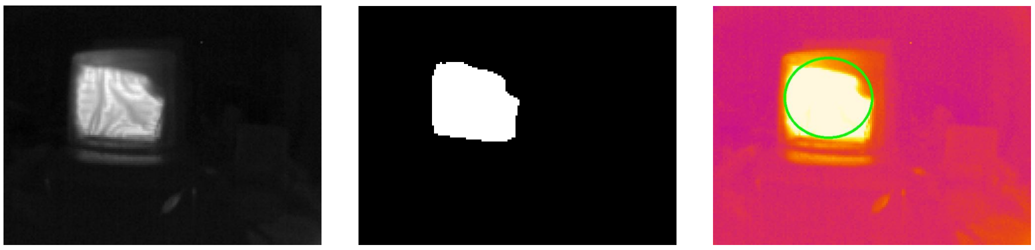
Figure 4.14: Detected QR Codes marked with a cyan circle.

beam back and forth across the bar code and a photodiode measures the intensity of the light reflected back from the bar code. In a similar manner Zbar's image scanner produces a linear stream of intensity samples from a grayscale image. The linear scanner produces a bar width stream from the output of the image scanner by using basic 1D signal processing. Finally the decoder searches the stream of bar widths for recognizable patterns and converts them into text. Figure 4.12 shows the pipeline of Zbar..

Zbar supports different symbologies, which are the mappings between messages and barcodes. The decoded text is stored in the feature and is later processed. Figures 4.13 and 4.14 show a typical example of QR codes inside the Rescue Arena. It is important to note that some of the QR codes are very small and need a very good camera to detect and decode.

4.2.6 Heat Detection

The heat detection algorithm searches the thermal image (see Figure 4.15(a)) for areas with temperatures in a specified temperature range. The heat detector currently works with the temperature range of 25°to 50°Celsius. The values for the minimum and maximum temperature are converted from degree Celsius to the same scale as the thermal image for faster comparison. In the next step a mask is calculated. Pixels inside the temperature range are set to one and the other pixels are set to zero. Morphological operators are used similar to the color detector (see 4.2.1) to remove small areas and merge nearby areas. The result can be seen in Figure 4.15(b). Finally a segmentation algorithm is used to find all remaining areas in the mask. Those areas are marked as heat features in the Visible image (see Figure 4.15(c)) of the thermal camera and published.


(a) Thermal Image

(b) All pixels inside the wanted temperature range are marked as belonging to a heat feature (white).

(c) Heat features are marked with a green ellipse in the colored thermal image.

Figure 4.15: Heat Detection

4.3 Feature Tracking

The next step is to track the features over time. Each feature type has its own feature tracker. See Figure 4.16.

The most important task of the feature tracker is to assign features to feature tracks. A feature track is a collection of features that the tracker believes to be the same feature measured at different times. Each feature can be assigned to an existing feature track or it can create a new one. There are **number of features · (number of feature tracks + 1)** possibilities and the tracker creates a hypothesis for each one. Each hypothesis has a score that allows the tracker to compare them with each other. Equation 4.9 shows how to calculate the score for assigning a feature to a track. In this case a lower score is better, because the goal is to minimize the distance between the features and their assigned feature tracks. The score of a new track has a

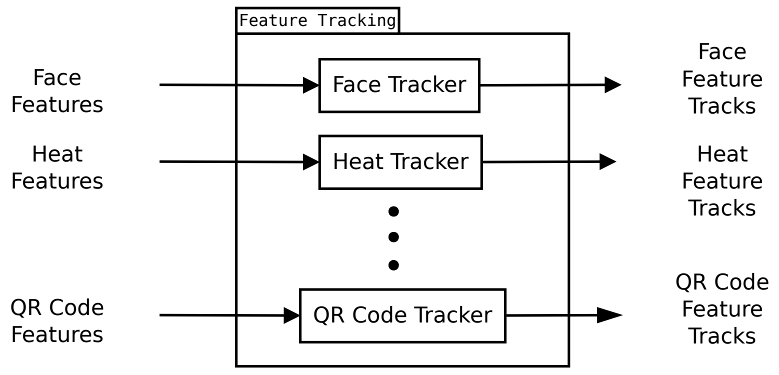


Figure 4.16: Feature Tracking Overview

default value of 1.5, which was determined by experiments with the victim simulator (see Section 5.6). The user can customize the value for each feature type. QR Codes and similar feature types are handled a little different. As mentioned above (see Section 4.2.5) they store additional data as a text. Only features with the same text as a feature track can be assigned to it.

$$score = \frac{\mathrm{d}(\vec{pos}_{track}, \vec{pos}_{feature})}{radius_{track} + radius_{feature}} \tag{4.9}$$

The next step is to find the best hypothesis. Other hypotheses, which share the feature or the feature track with the best hypothesis, are discarded. Depended on the hypothesis the feature is assigned to an existing feature track or initializes a new feature track. See Section 4.3.1 on how to update the position and radius of the track. Finding and processing the best hypothesis is repeated until there are no hypotheses left.

After that the feature tracks, which are visible to the current sensor, update their probability. This requires extra data form the sensor and a visibility test, which depends on the sensor type. The simple visibility test for cameras checks, if the feature tracks are inside the view frustum of the camera. A more complex visibility test uses ray casting similar to 4.1.2. This prevents the deletion of feature tracks hidden behind a wall. Other sensors like microphones use a sphere instead of a view frustum. Visible feature tracks,

Algorithm 1 Feature Tracking

```
Input: set of detected features F
Input: set of existing feature tracks T
Output: set of updated feature tracks T
 1: for all feature f ∈ F do
 2:    for all feature track t ∈ T do
 3:       create a hypothesis $h_{t,f}$ that f is t
 4:    end for
 5:    create a hypothesis $h_{t+1,f}$ that f is a new track
 6: end for
 7: for all feature f ∈ F do
 8:    find best hypothesis h
 9:    mark other hypotheses with the same feature as invalid
10:    mark other hypotheses with the same track as invalid
11:    if h is a new track hypothesis then
12:       create a new feature track from h
13:    else
14:       update an existing feature track with h
15:    end if
16: end for
17: for all feature track t ∈ T do
18:    if t is visible to the sensor's visibility test then
19:       update probability of t
20:    end if
21: end for
```

which were assigned a feature, increase their probability. Visible feature tracks, which were assigned no feature, decrease their probability. See 4.3.2 on how to increase or decrease the probability. Finally feature tracks with a low probability are deleted. See Algorithm 1 for the pseudo code of the feature tracker and Figure 4.17 shows a detailed example.

4.3.1 Position and Radius of a Feature Track

The victim detection system supports multiple algorithms for updating the position and radius of feature tracks. There exists a trade-off between speed, memory and accuracy. Another reason for multiple algorithms is that some specialize in static objects and others in moving objects.

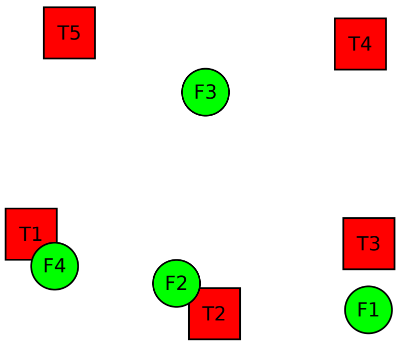
Static Data

The simplest algorithm initializes the position and radius at the creation of the feature track and never changes them. This algorithm is mostly used for user verification (see 4.6).

Arithmetic Mean of the Position and Radius

The standard algorithm sums up the positions and radii of all the assigned features and returns the average of both. See Equations 4.10 and 4.11. The algorithm is very fast, but it needs to check for arithmetic overflows. To save memory each track stores the sum of all past positions in one variable. If the tracker runs long enough the sum can become too large for the variable and cause an arithmetic overflows without the check.

$$\vec{pos}_{average} = \frac{\sum\limits_{i=1}^{N} \vec{pos}_i}{N} \tag{4.10}$$



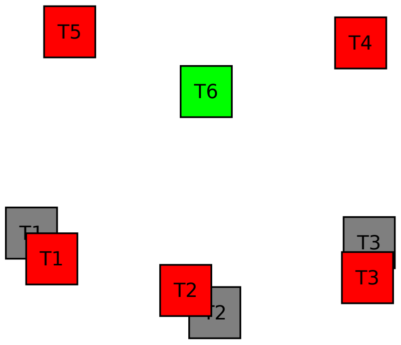
(a) The example contains 5 existing feature tracks (T1, T2, T3, T4 and T5) and 4 features (F1, F2, F3 and F4).

	F1	F2	F3	F4
T1	$h_{1,1}$	$h_{1,2}$	$h_{1,3}$	$h_{1,4}$
T2	$h_{2,1}$	$h_{2,2}$	$h_{2,3}$	$h_{2,4}$
T3	$h_{3,1}$	$h_{3,2}$	$h_{3,3}$	$h_{3,4}$
T4	$h_{4,1}$	$h_{4,2}$	$h_{4,3}$	$h_{4,4}$
T5	$h_{5,1}$	$h_{5,2}$	$h_{5,3}$	$h_{5,4}$
New	$h_{6,1}$	$h_{6,2}$	$h_{6,3}$	$h_{6,4}$

(b) All possible hypotheses ($h_{1,1}$ to $h_{6,4}$) are generated for the tracking. The best hypothesis $h_{1,4}$ (green) is found and the feature F4 is assigned to the track T1. Incompatible hypotheses are marked as invalid (gray).

	F1	F2	F3	F4
T1	$h_{1,1}$	$h_{1,2}$	$h_{1,3}$	$h_{1,4}$
T2	$h_{2,1}$	$h_{2,2}$	$h_{2,3}$	$h_{2,4}$
T3	$h_{3,1}$	$h_{3,2}$	$h_{3,3}$	$h_{3,4}$
T4	$h_{4,1}$	$h_{4,2}$	$h_{4,3}$	$h_{4,4}$
T5	$h_{5,1}$	$h_{5,2}$	$h_{5,3}$	$h_{5,4}$
New	$h_{6,1}$	$h_{6,2}$	$h_{6,3}$	$h_{6,4}$

(c) The process is repeated for the other best hypotheses ($h_{2,2}$, $h_{3,1}$ and $h_{6,3}$). F2 is assigned to T2 and F1 is assigned to T3. F3 creates a new track T6, because it is too far away from T4 and T5.



(d) The positions of the feature tracks T1, T2 and T3 are updated. Finally a new feature track T6 is created.

Figure 4.17: Feature Tracking Example

$$radius_{average} = \frac{\sum\limits_{i=1}^{N} radius_i}{N} \tag{4.11}$$

Kalman Filter for static Objects

The final algorithm uses a Kalman filter [Kal60] to get closer to the real position with each measurement. It is much slower, needs more memory and this implementation assumes that the object is completely static. The matrices P_0, A and H are initialized to the identity matrix and the covariance Q_k of the process noise is a zero matrix. The state vector $\vec{x}_0$ is initialized with the position and radius of the first feature. See Equation 4.12.

$$\vec{x}_0 = \begin{pmatrix} x \\ y \\ z \\ radius \end{pmatrix} \tag{4.12}$$

The Prediction Step predicts the prior state estimate $\vec{x}_k^-$ and the prior estimate covariance P_k^- based on the state transition model A. Here the filter assumes that the objects are static and predicts that nothing changes.

$$\vec{x}_k^- = A\vec{x}_{k-1} = I\vec{x}_{k-1} = \vec{x}_{k-1} \tag{4.13}$$

$$P_k^- = AP_{k-1}A^T + Q = P_{k-1} \tag{4.14}$$

The Correction Step uses the measurement $\vec{z}_k$ to correct the the prior state estimate $\vec{x}_k^-$ from the Prediction Step. The Klaman Gain K_k decreases with each update, which means that the influence of each measurement also decreases with each update.

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1} = P_k^- (P_k^- + R)^{-1} \tag{4.15}$$

$$\vec{x}_k = \vec{x}_k^- + K_k(\vec{z}_k - H\vec{x}_k^-) = \vec{x}_k^- + K_k(\vec{z}_k - \vec{x}_k^-) \tag{4.16}$$

$$P_k = (I - K_k H)P_k^- = (I - K_k)P_k^- \tag{4.17}$$

4.3.2 Probability of a Feature Track

Calculating the probability of the feature tracks is one of the most important tasks of the whole victim detection system. The system supports multiple algorithms and an easy way to switch between them at system start. This makes testing and comparing the different algorithms much easier.

Static Probability

The simplest algorithm initializes the probability at the creation of the feature track and never changes it. It is mostly used for user verification (see 4.6).

Detection Rate

The next algorithm calculates an approximation of the probability that the feature is detected, if the sensor should be able to see it. See Equation 4.18. The term $N_{features}$ is the number of times a feature was assigned to the feature track. The term $N_{visible}$ is the number of times the feature track was visible to the sensor determined by the visibility test. The quality of the approximation depends on the accuracy of the visibility test used by the feature tracker. The downside of this algorithm is that the magnitude of $N_{visible}$ determines how fast the probability changes. If the object has been seen for a long time and $N_{visible}$ is very high, then it may take too long to adapt to changes.

$$P = \frac{N_{features}}{N_{visible}} \tag{4.18}$$

Sensor Probability

The standard algorithm (see Equation 4.19) uses the update formula from [HWB+13]. The term $P(f|z_t)$ (see Equation 4.20) is the probability that a feature track exists given the current measurement. The term P_{Seen} is the probability that a feature track exists, if a feature was detected. The term P_{Unseen} is the probability that a feature track exists, if no feature was detected in the current update. Both values depend on the sensor and feature detector. The term $P(f|z_{1:t-1})$ is the probability that this feature track exists from the previous update.

$$P(f|z_{1:t}) = \left[1 + \frac{1 - P(f|z_t)}{P(f|z_t)} \cdot \frac{1 - P(f|z_{1:t-1})}{P(f|z_{1:t-1})}\right]^{-1} \tag{4.19}$$

$$P(f|z_t) = \begin{cases} P_{Seen} & \text{if feature is detected} \\ P_{Unseen} & \text{if feature is not detected} \end{cases} \tag{4.20}$$

The probability of the feature track starts at 0.5 and increases if a feature is assigned and decreases if no feature is assigned to a visible feature track. The rate of change is fastest around 0.5 and slows down the closer the probability gets to 0 or 1. An advantage over the previous algorithm is that the rate of change does not slow down with the number of times a feature was detected.

4.4 Object Fusion

Now it is time to fuse the feature tracks into objects. The victim detection systems creates hypotheses for all possible combinations of the existing feature tracks. Each hypothesis has a score that is used to compare them. The first feature tracker creates a hypothesis for each of its feature tracks that states that the feature track creates a new object. The next feature tracker creates 1 hypothesis for each of its feature tracks and each hypothesis of the previous feature tracker. These hypotheses state that the previous hypothesis is true and that the new hypothesis's feature track is part of the object. See Equation 4.21 on how to calculate the score. The feature tracker also creates 1 hypothesis for each hypothesis of the previous feature tracker that states that the object contains no feature track of the current feature type. See Equation 4.22. Finally the feature tracker creates a hypothesis for each of its feature tracks that states that the feature track creates a new object. See Equation 4.23. N_t is the number of previous trackers. The term $penalty_{no\,feature} \cdot N_t$ is needed so that the order of feature trackers does not influence the final score. These steps are repeated for the remaining feature trackers.

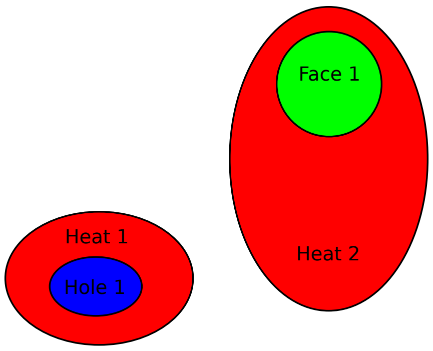
$$score_{feature} = score_{old} + \mathrm{d}(\vec{pos}_{old}, \vec{pos}_{feature}) \tag{4.21}$$

$$score_{no\,feature} = score_{old} + penalty_{no\,feature} \tag{4.22}$$

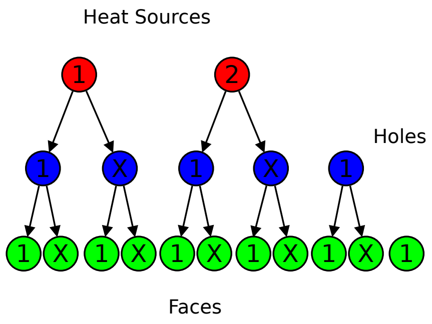
$$score_{new\,object} = penalty_{new\,object} + penalty_{no\,feature} \cdot N_t \tag{4.23}$$

The next task is to find the best hypothesis and create an object from it. The algorithm loops over all the current hypotheses to find the hypothesis with the lowest score. Remember that a lower score means that the feature tracks are closer together. Then an object is created from the feature tracks contained in the hypothesis. See 4.4.1 on how to calculate the position and the radius of the object and see 4.4.2 on how to calculate the probability of the object. Finally all hypotheses, which share one or more feature tracks with the best hypothesis, are marked as invalid. This task is repeated until no valid hypotheses remain.

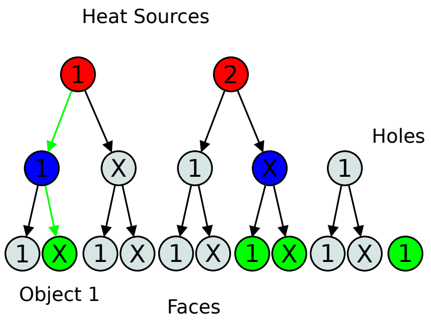
The object fusion algorithm (see 2) above is only the basic version. Some feature or object types need special rules to work properly. For example the heat and motion detectors often detect only one feature, if a group of objects is close enough together. The algorithm must allow multiple objects to share these feature tracks to produce the correct classifications later. The user defines, which feature types can be shared. This makes the validation of the hypotheses more difficult. Sharing a feature track, which can be shared, with the best hypothesis does not make a hypothesis invalid any longer. On the other hand each hypothesis needs at least one feature track, which is



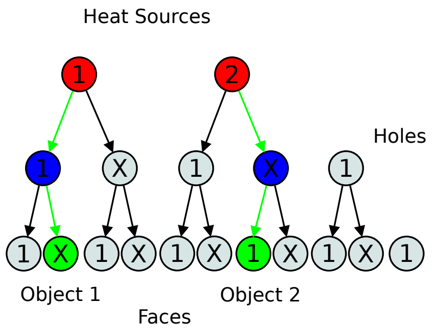
(a) The feature tracks of 2 objects. One of the objects is a victim and the other object is a heated hole, which could contain another victim.



(b) All the created hypotheses, which are all possible combinations of feature tracks. The color shows the feature type. The number shows the id of the feature track. A hypothesis with an X means that the object candidate doesn't have a feature track of this type.



(c) The best hypothesis, which is shown by the green arrows, is selected and turned into an object. Invalid hypotheses, which share one or more feature tracks with the best hypothesis, are marked as invalid (grey).



(d) The process is repeated for the 2.object. Afterwards the algorithm is finished, because all hypotheses are used or invalid.

Figure 4.18: Object Fusion Example

Algorithm 2 Object Fusion

```
Input: set of feature tracks F
Input: set of feature types T
Output: set of objects O
 1: create empty set of hypotheses H
 2: for all feature type $t \in T$ do
 3:     for all hypothesis $h \in H$ do
 4:         create a hypothesis that no track of type t fits h (use Equation 4.23
            for the score)
 5:         for all feature track $f \in F$ of type t do
 6:             create a hypothesis that t fits h (use Equation 4.21 for the score)
 7:         end for
 8:     end for
 9:     for all feature track $f \in F$ of type t do
10:         create a hypothesis that f is a new object (use Equation 4.23 for the
            score)
11:     end for
12: end for
13: repeat
14:     find best hypothesis $h \in H$
15:     remove hypotheses incompatible with h
16:     create an object o from h
17:     add object o to O
18: until no hypothesis is left
```

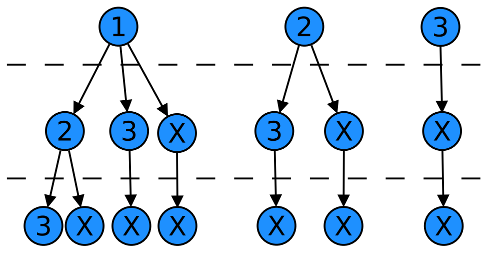


Figure 4.19: Hypotheses of 3 QR Codes per Object

not shared with an existing object, to be valid. This rule is needed to avoid the creation of too many objects.

Another special rule for certain feature types is that more than one feature track of the same type can belong to the same object. The RoboCup Rescue arena contains sheets with multiple QR codes (see Figure 4.13). Each QR code must be detected as a feature track, but the whole sheet must be the object. The algorithm creates hypotheses for all possible combinations of N feature tracks and M feature tracks per objects. Hypotheses that have the same feature tracks in a different sequence are avoided. See Figure 4.19.

The final special rule is that some feature types can not be mixed with other feature types inside an object. Currently QR codes will always have their own objects. This makes the handling of QR codes in the Rescue Robot League much simpler. An object can also only contain one user verification feature.

4.4.1 Position and Radius of an Object

The victim detection system supports multiple algorithms for calculating the position and the radius of an object from its feature tracks.

Circumscribed Sphere

The first algorithm creates a circumscribed sphere around all the feature tracks of the object. The object's position is the mean of all feature tracks. The object's radius is so large that all feature tracks are completely inside the sphere. This algorithm has problems with feature tracks that can be very large like heat or sound. It is even worse with large feature tracks that are shared between multiple objects. The large feature track's position is often far away from the object's true position.

Weighted Arithmetic Mean

The second algorithm calculates a weighted arithmetic mean of the position and the radius of the feature tracks. The weight is user defined for each feature type. See Equations 4.24 and 4.25. The user can customize the weights for each environment or application. One possibility is to set the weights to reflect the accuracy of the feature detectors. Small and accurate feature types like faces and holes would have a higher weight than others. Another possibility is to set each weight to the possibility of the feature track.

$$\vec{pos} = \frac{\sum_{i=1}^{N} weight_i \cdot \vec{pos}_i}{\sum_{i=1}^{N} weight_i} \tag{4.24}$$

$$radius = \frac{\sum_{i=1}^{N} weight_i \cdot radius_i}{\sum_{i=1}^{N} weight_i} \tag{4.25}$$

Inverse Radius

The third algorithm also uses a weighted arithmetic mean to calculate the position and the radius of the object. It uses the inverse radius as the weight. See Equations 4.24, 4.25 and 4.26. Small feature tracks have a larger influence than large ones. This avoids the problem of the first algorithm.

$$weight_i = \frac{1}{radius_i} \tag{4.26}$$

4.4.2 Probability of an Object

Calculating an object's probability from its feature tracks' probabilities is another task that strongly affects the behavior of the whole victim detection system. Once again the system supports multiple algorithms to choose from.

Weighted Arithmetic Mean

The first algorithm calculates the arithmetic mean of the feature tracks' probabilities. This has the disadvantage that adding new feature tracks can lower the object's probability instead of increasing it.

Maximum

The second algorithm loops over all the object's feature tracks and finds the maximum probability. Adding a feature track will never lower the probability, it has the disadvantage that only the feature track with the highest probability is important and all others are ignored. Having more feature tracks should increase the object's probability.

Combination of all Probabilities

The third algorithm combines the probabilities of the feature tracks so that having more of them increases the object's probability. It starts with $P_0 = 0$ and $rest_0 = 1$ and uses the Equations 4.27 and 4.28 to update the probability for each feature track. The algorithm solves the problems of the previous algorithms.

$$P_i = P_{i-1} + rest_{i-1} * P(feature_i) \tag{4.27}$$

$$rest_i = rest_{i-1} * (1 - P(feature_i)) \tag{4.28}$$

4.5 Object Classification

The final task of the victim detection system is to assign each object an object type. For each object the algorithm iterates over the object types and calculates their scores. See Equations 4.29 and 4.30. The term $probability_i$ is the probability (see 4.3.2) of the ith feature track of the object. It only uses the feature track with the highest probability, if there are multiple feature tracks of the same type. The term $weight_i$ is the feature type's weight of the ith feature track. It is defined by the user for each object type. Positive values mean that having a feature track of this type increases the probability of the object being of this object type. Negative values mean that having a feature track of this type decreases the probability of the object being of this object type. The object type with the highest score is assigned to the object. It is important to note that the object types and their weights are user-defined for each application. An example for search and rescue operations can be seen in Table 4.1.

$$score = \frac{\sum_{i=1}^{N} weight_i \cdot probability_i}{weight_{total}} \tag{4.29}$$

	Feature Types						
Object Types	**Face**	**Heat**	**Hole**	**Motion**	**QR**	**Verified V.**	**False V.**
Face	1.0	-2.0	-2.0	0.0	0.0	-5.0	-5.0
Heat Source	-2.0	1.0	-2.0	0.0	0.0	-5.0	-5.0
Heated Hole	-2.0	1.0	1.0	0.0	0.0	-5.0	-5.0
Hole	-2.0	-2.0	1.0	0.0	0.0	-5.0	-5.0
Victim	1.0	1.0	0.0	1.0	0.0	-5.0	-5.0
QR Code	0.0	0.0	0.0	0.0	1.0	-5.0	-5.0
Verified Victim	0.0	0.0	0.0	0.0	0.0	1.0	-5.0
False Victim	0.0	0.0	0.0	0.0	0.0	-5.0	1.0

Table 4.1: Weights of each Object Type

$$weight_{total} = \sum_{i=1}^{N} \begin{cases} weight_i & weight_i > 0 \\ 0 & weight_i \leq 0 \end{cases} \tag{4.30}$$

4.6 User Verification

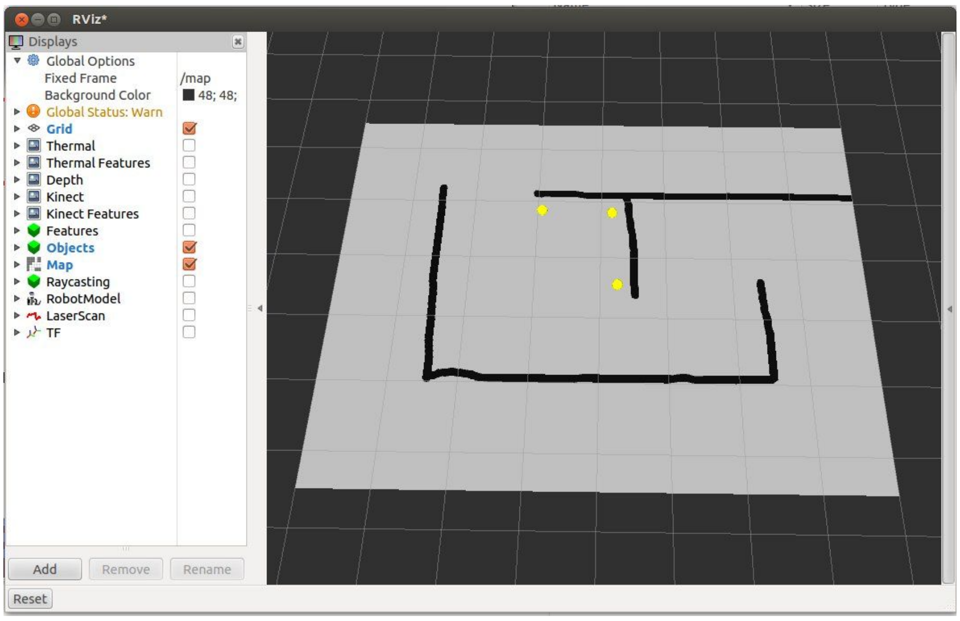
The user verification of victims is an important task of the victim detection system. The user can confirm a detected victim or mark it as wrongly classified by creating verification features of user-defined types. The current system use **Verified Victim** and **False Victim**. They are managed by the feature tracker similar to normal features, but there exist a few differences. The first difference is that each verification feature creates a new feature track and can not be assigned to an existing one. The probability, the position and the radius of a verification feature are set by the user during the initialization and never change. This means that user verification currently only works for nonmoving objects. The probability of a verification feature is 1, because a verification feature should only be created, if the user is absolutely sure about it. During the object fusion the verification features are treated like a normal features, except that each object can only have one of it. Multiple verification features would conflict with each other during object classification. The object classifier treats verification features like normal features and everything depends on the user-defined object types. Currently the **Verified Victim** feature makes sure that the object is assigned the **Verified Victim** object type. The **False Victim** feature makes sure that the object is assigned the **False Victim** object type. The probability of 1 and the positive or negative weights with a large magnitude ensure that the verification features dominate the object classification. The user can delete verification feature, if an error was made.

5 Implementation

This chapter describes the implementation of the framework presented in the last chapter. The Victim Detection System is written in the programming language C++ and uses the Robot Operating System[1](ROS). ROS was first presented in [QCG+09]. It is a set of drivers, software libraries and tools that make developing for robots easier and faster. The idea behind ROS is to split large applications into multiple smaller nodes. Each node solves a specific task and can be reused by other applications. The nodes communicate by message passing. A node can publish a message on a topic or subscribe to a topic and wait for published messages. Each topic only allows messages of a certain message type. The message passing decouples the nodes from each other and allows the programmer to replace a node with another one that solves the same task. ROS even allows the nodes to run on different computers and correctly sends the messages over the network. ROS uses RViz[2]to visualize important 2d and 3d data. It can display maps, robot models, laser scans, point clouds, images and much more. ROS nodes can publish markers[3]to display basic 3d shapes inside RViz.

The Victim Detection System also uses the Open Source Computer Vision Library[4](OpenCV). The library has more than 2500 optimized algorithms for computer vision and machine learning. It can load, convert and save images, detect edges and objects, apply morphological operators and much more.

[1]http://www.ros.org
[2]http://wiki.ros.org/rviz
[3]http://wiki.ros.org/rviz/DisplayTypes/Marker
[4]http://www.opencv.org



Fixed Frame /map
Background Color 48; 48;

Figure 5.1: Example of the GUI of Rviz

5.1 ROS Node raycasting_map

The ROS node **raycasting_map** provides 2 services[5]. Both of those services involve casting rays through the *OccupancyGrid*[6]. The *OccupancyGrid* is created and published by the ROS package *gmapping*[7]. Figure 5.2 shows the occupancy map with drawn rays, start points (blue) and intersection points (red). The node also uses markers to visualize the rays in *RViz*. The markers are bright green arrows that point from the rays' start points to the intersection points. See Figure 5.3.

The first service is called **raycasting** and it returns the 3d position of the intersection between the ray and the occupancy map. The service also returns the distance between the start of the ray and the intersection point. Finally it returns the type of intersection. The algorithm can detect if the ray hits an

[5]`http://wiki.ros.org/roscpp/Overview/Services`
[6]`http://docs.ros.org/api/nav_msgs/html/msg/OccupancyGrid.html`
[7]`http://wiki.ros.org/gmapping`

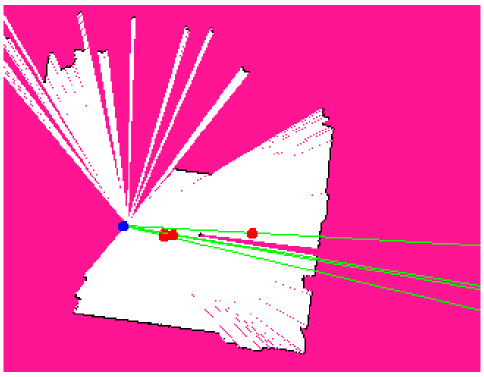


Figure 5.2: ROS node **raycasting_map**: The blue circle is the sensor. The red circles are the positions where the green rays hit an obstacles or the floor.

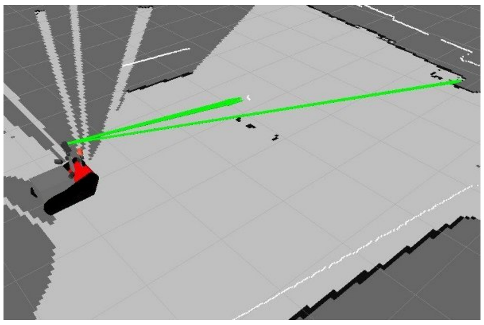


Figure 5.3: Green rays in *RViz*

obstacle, the floor, the ceiling, an unexplored cell or the border of the map. The service is mainly used by the class **TransformationWithRaycasting** of the ROS package **image_to_world_position**. See Section 4.1.2 and 5.2.

The second service is called **is_visible** and it checks if an observer can see a 3d position or if it is occluded by an obstacle. The service tries to cast a ray from the observer to the target and returns true, if the ray reaches the target without intersection. The service is mainly used by the ROS node **object_tracking** to determine the visibility of feature tracks. See Section 4.3 and 5.5.1.

5.2 ROS Package image_to_world_position

The ROS package **image_to_world_position** implements the algorithms described in Section 4.1.1 and 4.1.2 to transform a 2d position (x,y) in an image to a 3d world position (X,Y,Z). The class diagram can be seen in Figure 5.4.

The base class **Transformation** contains the code that is shared between both subclasses. It uses a *tf::TransformListener*[8] to get the transformation from

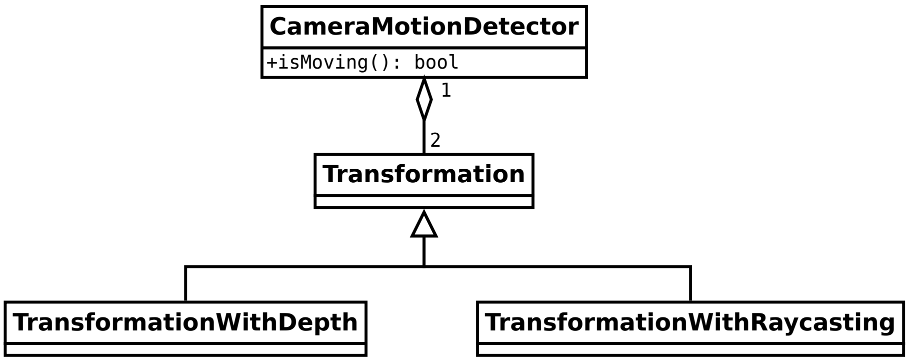


Figure 5.4: Class Diagram of **tedusar_image_to_world_position**

the camera coordinate system to the world coordinate system. The class also contains the functions to calculate a 3d position from a 2d position and a known depth in the camera coordinate system or the world coordinate system.

The subclass **TransformationWithDepth** uses the depth image of the Kinect to calculate the 3d position. See Section 4.1.1. The class uses the ROS package *image_transport*[9]to subscribe to the depth image. It also uses the ROS package *cv_bridge*[10], which converts between ROS Image messages and OpenCV images.

The subclass **TransformationWithRaycasting** uses raycastng to calculate the 3d position. The class calculates the position and direction of the ray from the 2d position and the transformation between the local and global coordinate systems. It then uses the **raycasting** service of the ROS node **raycasting_map**. See Section 4.1.2 and 5.1.

The ROS nodes **feature_detection_kinect** and **feature_detection_thermal** use the class **CameraMotionDetector** to disable the feature detectors, if the camera moves too much. The class transforms the same local point to the world coordinate system each update and calculates how much the

[8]http://wiki.ros.org/tf
[9]http://wiki.ros.org/image_transport
[10]http://wiki.ros.org/cv_bridge

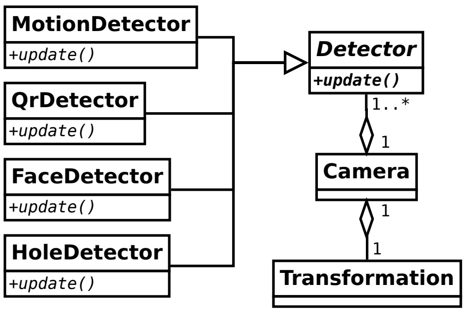


Figure 5.5: Class Diagram of the ROS Node **feature_detection_kinect**

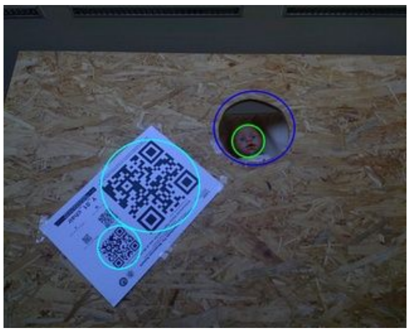


Figure 5.6: Features detected with the Kinect. The color of the ellipse determines the feature type.

point has moved since the last update. It then divides the distance by the elapsed time to get the speed of the point. The function *isMoving()* returns true, if the calculated speed is higher than the allowed maximum. For better results the function continues to return true for a certain duration, even after the camera has stopped moving.

5.3 ROS Node feature_detection_kinect

The ROS Node **feature_detection_kinect** implements the feature detection algorithms for the Kinect from Section 4.1.1. The class diagram can be seen in Figure 5.5. The **Camera** class subscribes to the camera info, color image and depth image topics of the Kinect. It also reads the launch file to determine, which feature detectors are started and what their parameters are. The node has multiple classes for feature detection:

- ColorDetector (see Section 4.2.1)
- FaceDetector (see Section 4.2.2)
- HoleDetector (see Section 4.2.3)
- MotionDetector (see Section 4.2.4)
- QrDetector (see Section 4.2.5)

name	Name of the detected feature. Some feature detectors can run multiple times with different parameters to detect different features.
type	Name of the feature detector.
frame_rate	Number of times the feature detector runs per second.
use_scaling	Use images at a lower resolution to speed up the detection?
probability_seen **probability_unseen**	Sensor probabilities used to calculate the probability of a feature track. See Section 4.3.2.

Table 5.1: Parameters shared by all Feature Detectors

The Table 5.1 shows the parameters that are shared by all feature detectors. The node uses the class **TransformationWithDepth** from the package **image_to_world_position** to calculate the 3d position of each feature.

5.4 ROS Node feature_detection_thermal

The ROS Node **feature_detection_thermal** is very simlar to the previous node. It implements the feature detection algorithms for the thermal camera from Section 4.1.2. The class diagram can be seen in Figure 5.7. The **ThermalCamera** class subscribes to the camera info, color image and thermal image topics of the thermal camera. It also reads the launch file to determine, which feature detectors are started and what their parameters are. The **HeatDetector** class (see Section 4.2.6) is used to detect areas inside a specific temperature range. The node uses the class **TransformationWithRaycasting** from the package **image_to_world_position** to calculate the 3d position of each feature.

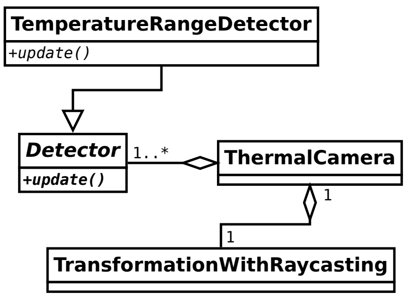


Figure 5.7: Class Diagram of the ROS Node **feature_detection_thermal**

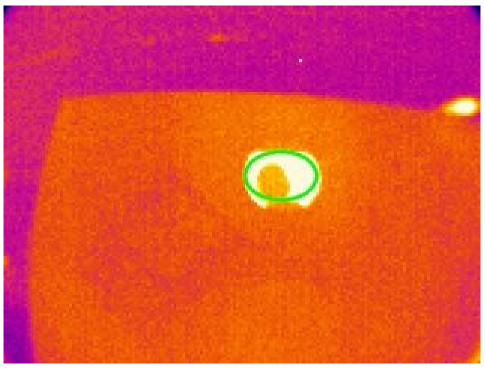


Figure 5.8: Features detected with the Thermal Camera

5.5 ROS Node object_tracking

The ROS Node **object_tracking** implements the feature tracking, object fusion and object classification algorithms from Section 4.3, Section 4.4 and Section 4.5. It subscribes to the features published by the feature detection nodes. Many important parameters to customize the behavior of the algorithms can be set in the launch file. At the end it publishes the created feature tracks and objects.

5.5.1 Feature Tracking

This subsection describes the implementation of the feature tracking algorithm from Section 4.3. See Figure 5.9 for the class diagram. There exists one **FeatureTracker** for each feature type the user wants to track. The number of them and their parameters can be defined in the launch file. Each **FeatureTracker** also contains a list of **FeatureTracks** of its type and is responsible for creating, updating and deleting them. The class **FeatureTrack** contains most of the data needed for tracking. Most of it is stored in its **Score** and **TrackData**. Both of those classes and their subclasses use the Strategy Pattern, which is one of the Design Patterns included in [GHJV94]. It allows the user to select one of the algorithms presented in 4.3.1 and 4.3.2 to update the position, the radius and the probability of the **FeatureTrack**. Different

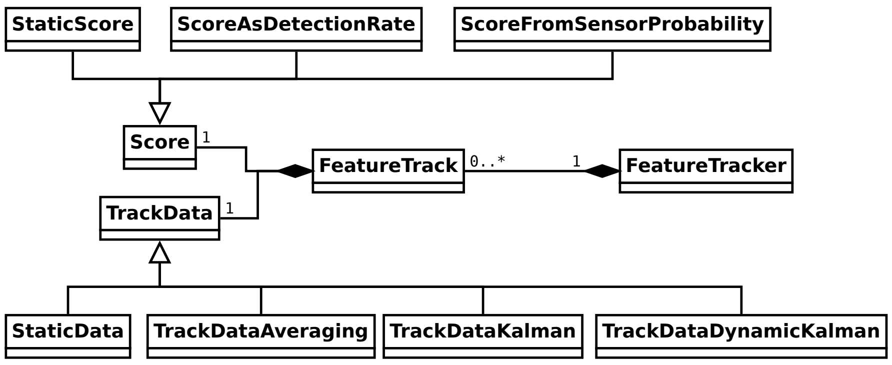


Figure 5.9: Class Diagram of the Feature Tracking Task

classes also allow the different algorithms to store different data. Finally the **FeatureTracks** are published by the **FeatureTrackers**.

5.5.2 Object Fusion

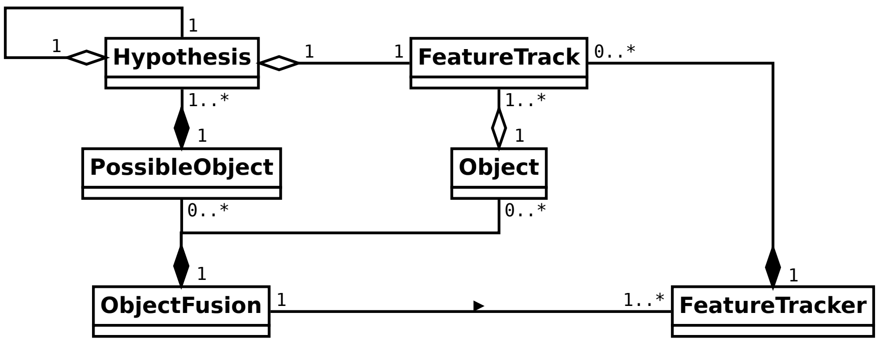


Figure 5.10: Class Diagram of the Object Fusion

This subsection describes the implementation of the object fusion algorithm from Section 4.4. See Figure 5.10 for the class diagram. The class **Object-Fusion** is responsible for creating the **Objects** from the **FeatureTracks**. It uses the **FeatureTrackers** to access all the **FeatureTracks**. The classes **PossibleObject** and **Hypothesis** are used to create all possible combinations of **FeatureTracks** and find the best ones.

The class **Object** is responsible for calculating the position, the radius and the probability of the object from the feature tracks. It implements the algorithms presented in 4.4.1 and 4.4.2. The algorithm to be used can be selected in the launch file. The Strategy Pattern is not used, because the algorithms are simpler and there is no need to store additional data. The **Objects** are published by the class **ObjectFusion** after the object classification.

5.5.3 Object Classification

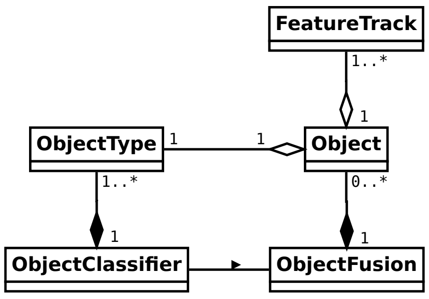


Figure 5.11: Class Diagram of the Object Classification Task

This subsection describes the implementation of the object classification algorithm from Section 4.5. See Figure 5.11 for the class diagram. The class **ObjectClassifier** loads the **ObjectTypes** from the launch file. It also decides, which **ObjectType** is assigned to each **Object**. The class does this by calculating how well each **ObjectType** fits the **Object** and assigning the one with the highest score. The Equations 4.29 and 4.30 show how the score is calculated from the probability of the object's **FeatureTracks**. The weight of each feature type is defined by the **ObjectType**.

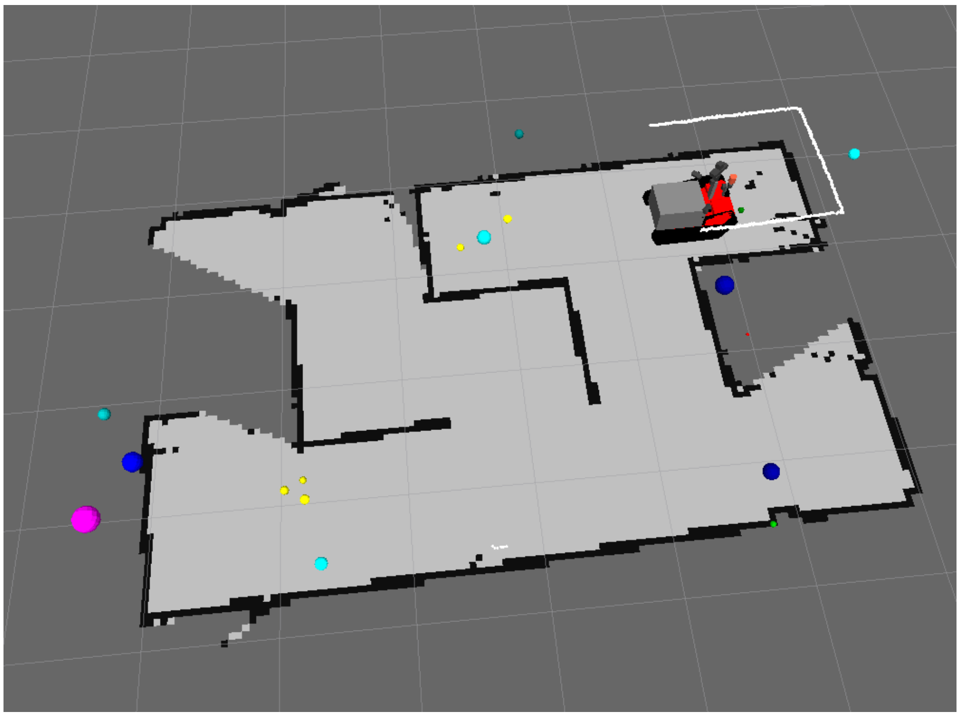
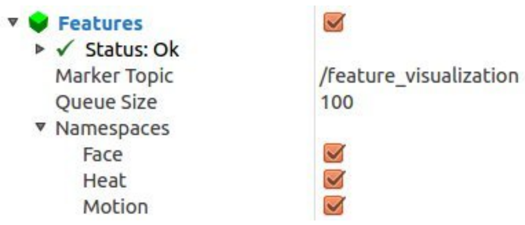


Figure 5.12: The colored spheres are the Markers in RViz. The color shows the feature or object type. The robot and the laser scan can also be seen.



▾ Features	✓
▸ ✓ Status: Ok	
Marker Topic	/feature_visualization
Queue Size	100
▾ Namespaces	
Face	✓
Heat	✓
Motion	✓

Figure 5.13: Namespaces in RViz can be used to toggle the visibility of each feature and object type.

5.5.4 Visualization

The node uses markers to visualize the feature tracks and objects. The markers are published and can be seen in *RViz* (see Figure 5.12). The markers use a sphere as the shape and the position and the radius are set by the feature track or the object. A namespace (see Figure 5.13) groups all the markers of the same feature or object type and it can be used to toggle the visibility of them. The color of the marker is a little more complex. The base color is set by the feature type or object type. It is modified by the probability of the feature track or object. See Equations 5.1 and 5.2. The variable *start* is set to 0.25 to ensure that feature tracks and objects of different types can still be distinguished at low probabilities. All markers with low probabilities would look black without it.

$$factor = start + (1.0 - start) \cdot probability \tag{5.1}$$

$$color_{marker} = color_{base} \cdot factor \tag{5.2}$$

5.6 ROS Node victim_simulation

The ROS node **victim_simulation** (see Figure 5.15) can be used to test the victim detection system without requiring an arena, a robot, victims and feature detectors. It is also ideal to repeatedly test special cases that are hard

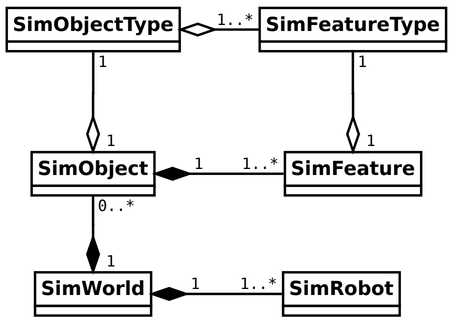


Figure 5.14: Class Diagram of the ROS Node **victim_simulation**

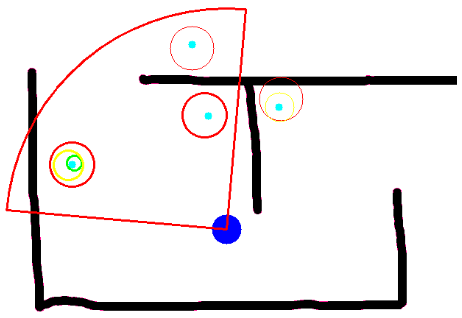


Figure 5.15: The victim simulator allows the user to place objects (cyan dots) with simulated features (red, green or yellow circles) and move the robot (filled blue circle) with its view frustum (red quarter circle).

or slow to reproduce. The node does this by simulating the arena, the robot and the objects. The simulation is run in 2d to keep it fast and simple. More complex tests should be run in the real world.

See Figure 5.14 for the class diagram. The class **SimWorld** subscribes the *OccupancyGrid*, which can be loaded from an image, and uses it as the 2d world of the simulation. The launch file defines one or more instances of the class **SimRobot**. Each robot has a position, an orientation, a radius, a view distance and a view angle. The user can move and rotate the selected robot and switch to other robots. The class **SimFeatureType** has a name, a color and the sensor-dependent data needed for the feature tracking. The class **SimObjectType** consists mainly out of a list of **SimFeatureTypes**, their probabilities of detection and their radii. The **SimTypeMgr** loads the feature types and object types from the launch file and manages them. **SimObjects** are also loaded from the launch file or can be placed anywhere in the world by the user. Each update they randomly place their features around them to simulate noise. They also use the feature's probability to determine if it was detected. Each robot determines if the features are inside its field of view defined by the view distance and the view angle. It also performs a simple 2d raycast to determine if a feature is occluded by the map. Finally all visible **SimFeatures** are published as feature messages.

6 Evaluation

In this chapter multiple tests to evaluate the victim detection system and its feature detectors are described. Each test is divided into the stages setup, testing and evaluation. In the setup stage the scene of the test is configured. The walls, the ramps, other parts of the arena and the objects are placed in the needed configuration. Then the robot is moved to the start location. Finally the positions of all objects relative to the start location are measured and recorded. This data serves as the ground truth in the evaluation.

At the start of the testing stage the heating pads and the robot are switched on and the victim detection system is started. During the test the feature detectors save the input images with marked features for each update. This allows the evaluator the count the number of detected features (true positives) TP, falsely detected features (false positives) FP and missing features (false negatives) FN for each feature detector. The number of true negatives TN is the number of input images without any feature. The number of negatives N is the sum of the true negatives TN and the false positives FP. The number of positives P is the sum of the true positives TP and the false negatives FN. The object tracker saves all the received features into files. The robot can stand still or move around until the end of the test. At the end the object tracker saves all the feature tracks and objects into files. The evaluator can make screen shots of the feature track and object visualization in RViz.

Finally, the recorded data is evaluated with the help of a Matlab[1]script. It uses the previously counted N, TP, FP, TN and FN to calculate the detection rates of the feature detectors. The true positive rate P_{TP} (see Equation 6.1) is the probability that a detected feature is really a feature. The false positive rate P_{FP} (see Equation 6.4) is the probability that a detected

[1] www.mathworks.com/products/matlab/

feature is not a feature. The true negative rate P_{TN} (see Equation 6.3) is the probability that there is no feature, if no feature was detected. Test that have zero negatives are marked with a $*$, because P_{TN} would be infinite. The false negative rate P_{FN} (see Equation 6.2) is the probability that there is a feature, if no feature was detected.

$$P_{TP} = \frac{TP}{P} = \frac{TP}{TP + FN} \tag{6.1}$$

$$P_{FN} = 1 - P_{TP} \tag{6.2}$$

$$P_{TN} = \frac{TN}{N} = \frac{TN}{FP + TN} \tag{6.3}$$

$$P_{FP} = 1 - P_{TN} \tag{6.4}$$

Then the matlab script Automatically assigns the recorded features to the objects recorded in the setup stage. Features, that are too far away from all the objects, are outliers. It calculates the standard derivation of the position and the radius for each cluster of features. It combines the values of each cluster to calculate σ_X, σ_Y, σ_Z and σ_{radius} for each feature detector. The noise of the position is very important, because it determines how reliable features can be assigned to feature tracks. A low noise means that different objects can be closer together without causing problems.

In the next step the evaluator compares the calculated and measured positions of the objects. It is important to note that the measured positions are also inaccurate, because it is difficult to use a measuring tape in the large arena. Another difficulty is that the calculated position of an object depends on the view direction and factors.

Finally, the evaluator compares the detected feature tracks and objects at the end of the test with the real objects. The evaluator counts the number of detected feature tracks and objects $N_{correct}$, the number of false feature tracks and objects N_{false}, the number of feature tracks and objects not detected $N_{missing}$ and the number of objects with a wrong object type $N_{wrong\ type}$. The evaluator also compares the measured and calculated position of the real objects, which were detected, to find out how accurate the positions are.

6.1 Test Under Ideal Conditions

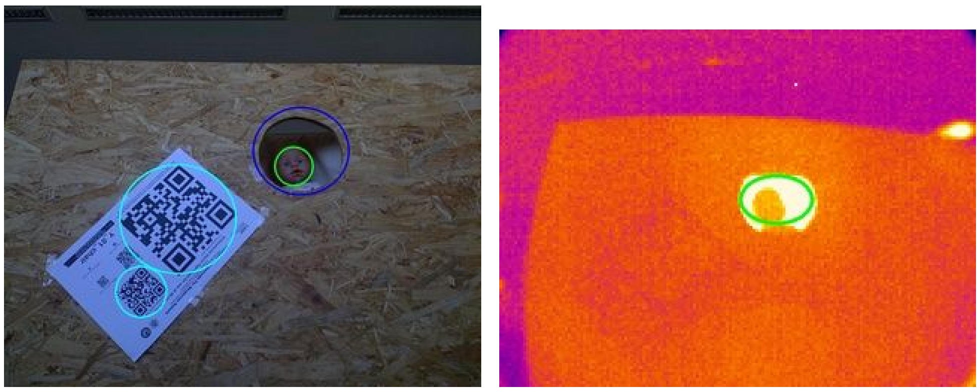

(a) Color Image (b) Thermal Image

Figure 6.1: Scene of the 1.Test

In this test of the evaluation analyzes the performance of the feature detectors and the victim detection system under ideal conditions. The robot and its sensor head are not moving and have a closeup view of 2 objects. The first object is a victim that consists out of a baby doll and a heating pad inside a hole. The second object is a QR code sheet with a big and a small QR code. See Figure 6.1. The Kinect runs for 228 cycles and the thermal camera runs for 505 cycles.

The Table 6.1 shows the detection rates of the feature detectors. The face, heat and hole detectors never missed a feature or produced a false feature. The heat detector detected a person moving outside the arena, which explains why it detected more features than expected at first. The QR code detector did not detect all features, but it also never detected a false feature. An interesting fact is that it detected the smaller QR code more often than the larger one. A possible reason for this is the algorithm of the Zbar library that handles QR code detection at different scales.

Detector	P	N	TP	FP	TN	FN	P_{TP}	P_{TN}
Face	228	0	228	0	0	0	1	$*$
Heat	526	0	526	0	0	0	1	$*$
Hole	228	0	228	0	0	0	1	$*$
QR Code (big)	228	0	140	0	0	88	0.614	$*$
QR Code (small)	228	0	194	0	0	34	0.8509	$*$

Table 6.1: Detection Rates of the 1.Test

Detector	σ_X	σ_Y	σ_Z	σ_{radius}
Face	0.0018686 m	0.0019379 m	0.0013968 m	0.00070899 m
Heat	0.0014698 m	0.0013266 m	0.0049055 m	0.00318 m
Hole	0.0015146 m	0.0015270 m	0.0011861 m	0.00054422 m
QR Code	0.0017155 m	0.001303 m	0.00085959 m	0.00035556 m

Table 6.2: Standard Deviation of Position and Radius of the 1.Test

The Table 6.2 shows the standard deviation of the positions and the radii for each detector. The standard deviation of the position is small compared to the typical distance between features, which makes feature tracking much easier. An interesting fact is that there is no big difference between detectors using the depth image or raycasting for the calculation of the position.

The Tables 6.3 and 6.4 show that the victim detection system correctly detected all feature tracks and objects.

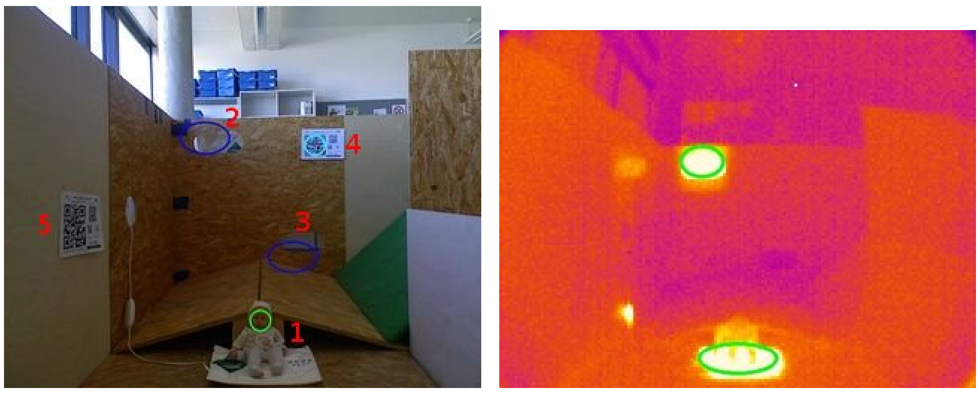
Tracks	$N_{correct}$	N_{false}	$N_{missing}$
Face	1	0	0
Heat	1	0	0
Hole	1	0	0
QR Code (big)	1	0	0
QR Code (small)	1	0	0

Table 6.3: Feature Tracks of the 1.Test

Objects	$N_{correct}$	N_{false}	$N_{missing}$	$N_{wrong\ type}$
Face	0	0	0	0
Heat Source	0	0	0	0
Heated Hole	0	0	0	0
Hole	0	0	0	0
Person	1	0	0	0
QR Code	1	0	0	0

Table 6.4: Objects of the 1.Test

6.2 Long Distance Test



(a) Color Image (b) Thermal Image

Figure 6.2: Scene of the 2.Test

The long distance test of the evaluation analyzes the performance of the feature detectors and the victim detection system under more difficult conditions. It will show that the detection system still works but with lower detection rates. The robot and its sensor head are still not moving. The robot can see 5 objects, which are farther away than in the previous test. The first object is a victim sitting on the floor that consists out of a baby doll and a heating pad. The second object is a hole with a heating pad. The third object is a simple hole. The fourth object is a QR code sheet facing the camera. The fifth object is a QR code sheet perpendicular to the viewing direction of the camera. See Figure 6.2. The Kinect runs for 279 turns and the thermal

Detector	P	N	TP	FP	TN	FN	P_{TP}	P_{TN}
Face	279	0	76	0	0	203	0.2724	$*$
Heat	1160	0	1160	0	0	0	1	$*$
Hole	558	1	557	1	0	1	0.9982	0
QR Code (big)	279	0	29	0	0	250	0.104	$*$
QR Code (side)	279	0	0	0	0	279	0	$*$
QR Code (small)	558	0	0	0	0	558	0	$*$

Table 6.5: Detection Rates of the 2.Test

Detector	σ_X	σ_Y	σ_Z	σ_{radius}
Face	0.0071628 m	0.0025864 m	0.0031929 m	0.00054819 m
Heat	0.0036037 m	0.0037221 m	0.0038086 m	0.0016632 m
Hole	0.0106716 m	0.0070939 m	0.0054066 m	0.0046165 m
QR Code	0.0175356 m	0.0044963 m	0.0055691 m	0.00095211 m

Table 6.6: Standard Deviation of Position and Radius of the 2.Test

camera runs for 580 turns.

The Table 6.5 shows the detection rates of the feature detectors. The face detector didn't produce any false features, but it missed the doll much more often, because of the higher distance. The heat and hole detectors still worked as reliable as before. Finally the detection rates of the QR code detector decreased dramatically. It never detected the smaller QR codes or the big one perpendicular to the viewing direction of the camera. Even the big QR code facing the camera was rarely detected. On the positive side the QR detector never produced a false feature.

The Table 6.6 shows the standard deviation of the positions and the radii for each detector. The values have increased somewhat compared to the previous test.

The Table 6.7 shows the difference between the calculated positions and the manually measured position of the objects. The coordination system is defined so that the x-axis points towards the objects, the y-axis points to the left and the z-axis points upwards. The biggest error is the X-coordinate of the simulated person sitting on the floor. The reason for that is that the

	Position Error			
Objects	**X**	**Y**	**Z**	**Distance**
Heated Hole	-0.00598 m	-0.076099 m	-0.03187	0.083 m
Hole	-0.0546 m	0.024668 m	0.077851	0.098 m
Person	-0.17339 m	0.0867551 m	0.068359	0.206 m
QR Code Sheet	-0.04138 m	0.075086 m	0.067158	0.109 m

Table 6.7: Position Error of the 2.Test

Tracks	$N_{correct}$	N_{false}	$N_{missing}$
Face	1	0	0
Heat	2	0	0
Hole	2	0	0
QR Code	1	0	3

Table 6.8: Feature Tracks of the 2.Test

heat detector uses raycasting to project the position of the heat source on the ground behind the doll. The false position of the heat source influences the position of the object enough to drastically increase the error.

The Tables 6.8 and 6.9 show that the victim detection system nearly detected all feature tracks and objects correctly. It only missed 3 QR code tracks and 1 QR Code object. The 2.Table also shows that all detected objects have the correct type.

Objects	$N_{correct}$	N_{false}	$N_{missing}$	$N_{wrong\ type}$
Face	0	0	0	0
Heat Source	0	0	0	0
Heated Hole	1	0	0	0
Hole	1	0	0	0
Person	1	0	0	0
QR Code Sheet	1	0	1	0

Table 6.9: Objects of the 2.Test

6.3 Moving Robot Test

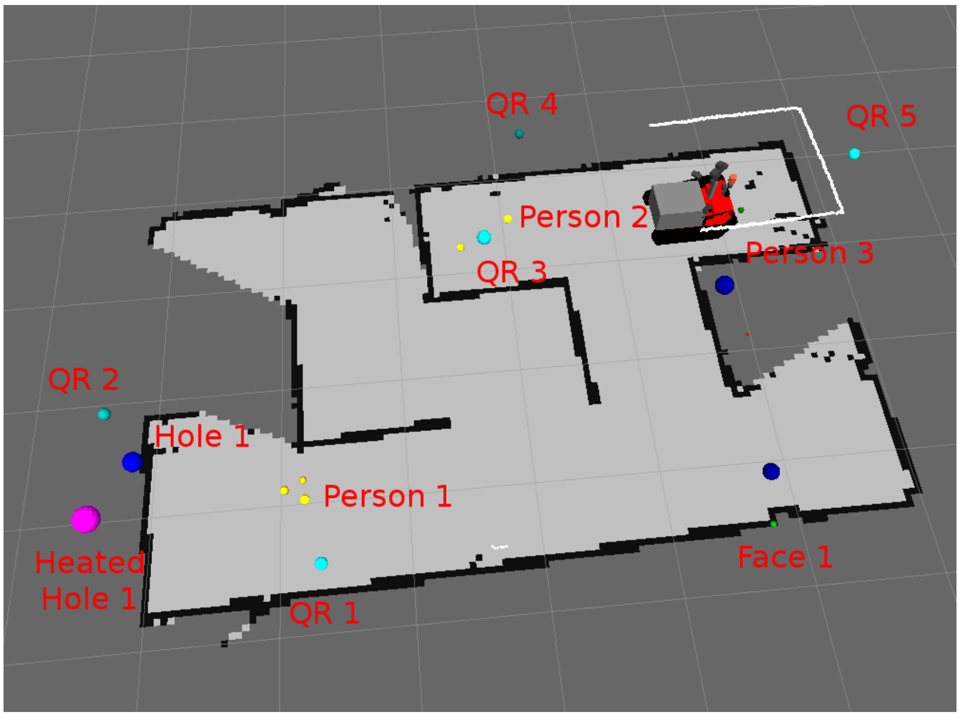


Figure 6.3: Objects of the 3.Test

The moving robot test of the evaluation analyzes the performance of the feature detectors and the victim detection system while the robot explores the whole arena. The robot drives from one location to another and then observes the objects from the new location for a few seconds. The arena contains the same objects as in the previous test and a few more. The extra objects are 2 QR codes, 1 victim, 1 face, which is a baby doll without heating pad and 2 holes. See Figure 6.3. The Kinect runs for 533 turns and the thermal camera runs for 1899 turns.

The Table 6.10 shows the detection rates of the feature detectors. The face detector detected faces more often compared to the previous test, but also produced more false positives. The improved detection rates is most likely caused by seeing the objects from different positions. These false faces are

Detector	P	N	TP	FP	TN	FN	P_{TP}	P_{TN}
Face	292	241	180	59	182	112	0.6164	0.755
Heat	1536	363	978	129	234	558	0.6367	0.6446
Hole	480	53	283	47	6	197	0.5896	0.113
QR Code (big)	321	212	220	0	212	101	0.6854	1

Table 6.10: Detection Rates of the 3.Test

Detector	σ_X	σ_Y	σ_Z	σ_{radius}
Face	0.066356 m	0.044867 m	0.032872 m	0.0029309 m
Heat	0.047901 m	0.021553 m	0.024331 m	0.028772 m
Hole	0.021715 m	0.021705 m	0.028039 m	0.0058677 m
QR Code	0.029344 m	0.019791 m	0.014583 m	0.0023371 m

Table 6.11: Standard Deviation of Position and Radius of the 3.Test

not detected at random positions, but repeatedly at the same ones. The heat detector had worse detection rates than before. It detected a heating pad through a hole and incorrectly projected the position on the wall with the hole. This is a known problem (see Figure 4.5(a)) and a solution is already planned (see Section 7.1). The second problem was that it didn't detect the last heading pad. The most likely reason is that its automatically switched off after 30 min. The QR code detector did never detect the small features, but it also never detected a false feature.

The Table 6.11 shows the standard deviation of the estimated positions and the estimated radii for each detector. The values have increased again somewhat compared to the previous test.

The Table 6.12 shows the difference between the calculated positions and the manually measured position of the objects. The movement of the robot and the different viewing directions only caused an increase of about 30% in the average distance between the calculated positions and the manually measured position.

The Tables 6.13 and 6.14 show that the victim detection system detected most feature tracks and objects correctly. The face detector detected 4 false faces often enough to create feature tracks. See Figure 6.4. The hole detector

	Position Error			
Objects	**X**	**Y**	**Z**	**Distance**
Face 1	0.108 m	-0.012 m	-0.124 m	0.165 m
Heated Hole 1	-0.006 m	-0.076 m	-0.032 m	0.083 m
Hole 1	-0.055 m	0.025 m	0.078 m	0.098 m
Person 1	-0.173 m	0.087 m	0.068 m	0.206 m
Person 2	0.102 m	0.131 m	-0.006 m	0.166 m
QR Code Sheet 1	0.2m	-0.076 m	-0.022 m	0.215 m
QR Code Sheet 2	-0.041 m	0.075 m	0.067 m	0.109 m
QR Code Sheet 3	0.164 m	0.079 m	-0.004 m	0.182 m
QR Code Sheet 4	0.112 m	0.222 m	-0.039 m	0.252 m
QR Code Sheet 5	-0.02 m	0.15 m	0.042 m	0.157 m

Table 6.12: Position Error of the 3.Test

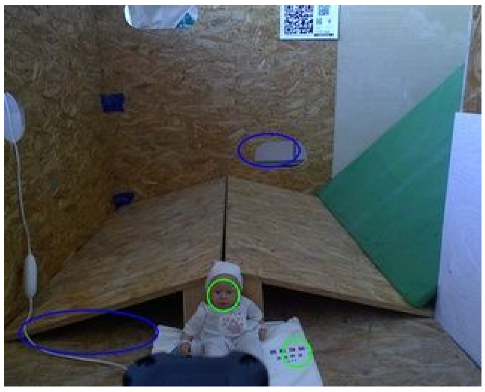
Tracks	$N_{correct}$	N_{false}	$N_{missing}$
Face	3	4	1
Heat	3	1	1
Hole	4	3	3
QR Code (big)	5	0	0
QR Code (small)	0	0	5

Table 6.13: Feature Tracks of the 3.Test

Objects	$N_{correct}$	N_{false}	$N_{missing}$	$N_{wrong\ type}$
Face	1	0	0	0
Heat Source	0	1	0	0
Heated Hole	1	0	0	0
Hole	2	1	2	0
Person	2	3	1	0
QR Code Sheet	5	0	0	0

Table 6.14: Objects of the 3.Test

missed 3 holes and detected 3 wrong holes, because there is always a trade off between detecting too many and too few holes. The heat detector saw the fourth heating pad through a hole in the wall, but the ray casting projected the heat source onto the wall, because the occupancy map can not handle holes in obstacles (see Figure 4.5(a)). This created a heat feature at the wrong location. The system also missed the last victim, because it could not detect the face, the heat source or the hole for unknown reasons. It is possible that the heating pad had turned itself off after 30 minutes and that it had cooled off enough to be no longer recognized as heat source. Another idea is that the shadow across the face cast by the hole prevented the feature detector from recognizing it as a face. On the positive side all detected objects have the correct object type.


(a) Eye Chart as Face

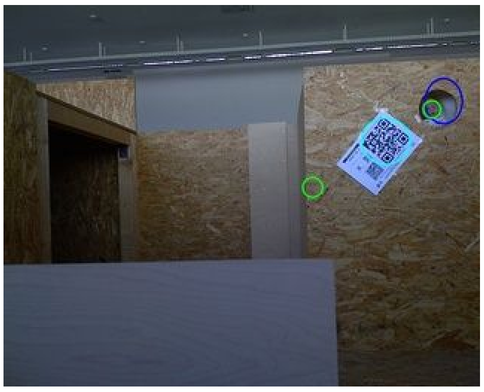

(b) Face in the Wood Grain

Figure 6.4: False Features of the 3.Test

6.4 View Angle Test

In this test the robot watches multiple features on a wall that is 1 meter away. The wall is rotated so that each subtest sees the wall under a different view angle. The view angle is defined for this test as the angle between the wall and the view direction of the robot's sensors. In the first test the wall is perpendicular (90°) to the view direction of the sensors. In the following

	90°	75°	60°	45°
Face	0.74	0.70	0.72	0.66
Heat	1.00	1.00	1.00	1.00
Hole	0.88	0.99	0.63	0.01
QR Code	0.74	0 .70	0.72	0.00

Table 6.15: Detection Rates dependent on the View Angle

tests the view angle is reduced to 75°, 60°and finally 45°. The goal of this test is to determine the influence of the view angle on the feature detection. The true positive detection rate dependent on the view angle is calculated for each sensor and shown in Table 6.15. The detection rate of the face detector is nearly constant around 0.70. The heat detector always found the heat source independent of the view angle. The detection rate of the hole detector drops to zero around 45°. It is interesting that the maximum detection rate is at 75°. The most likely explanation is that the baby doll blocks most of the hole in frontal views. The detection rate of the QR Code detector is nearly constant around 0.70 until it drops to zero at 45°. This test shows that it is not only important that the sensors cover all surfaces, but also that the view angle is important for certain feature detectors. The result must be considered while designing the movement pattern of the robot's sensors.

7 Conclusion

This thesis presented a framework for victim detection that can be used by rescue robots with multiple sensors of different types. The related research chapter has shown existing algorithms for feature detection, tracking, sensor fusion and other topics. Next the thesis presented a problem definition for victim detection. Then it showed an overview of how the framework and its subsystems work. The approach combines feature detection, feature tracking, object fusion, object classification and user verification. The thesis showed how existing and new algorithms can be used to solve those tasks. The different sensors and feature detectors were explained. It showed how the features are assigned to existing feature tracks, which track features over time, or create new ones. Then how feature tracks are combined into objects and how a simple object classifier assigns object types to objects are explained. The last section of the solution chapter showed how user verification affects all the subsystems of the framework. Next implementation details concerning C++, ROS and OpenCV are explained. Finally, it an evaluation of the performance of the framework with multiple tests is presented.

The evaluation showed that the framework is able to provide good results in different scenarios. It can easily handle feature trackers with a low true positive rate, which means that the detector detects a feature only every few updates. Over time those features detected at a low frequency create feature tracks and slowly increase the feature track's probability. It can also handle false features that are randomly detected without creating false victims. Those false positives rarely create feature tracks, because a minimum number of features is needed at the same location to create a feature rack. The framework only has problems with false features that are repeatedly detected at the same location. In this case the framework creates feature tracks and can't distinguish between true and false ones. In this case the responsible feature detector should be improved. The evaluation has also shown that the victim detection framework produces positions that are accurate enough for search and rescue operations.

7.1 Future Work

Now possible ways to improve the victim detection system in the future are discussed.

The simplest way to improve the victim detection system is to add new sensors and feature detectors. New feature types can be added without changing the code of the existing system by adapting the launch file of the ROS node **object_tracking**. CO2 and sound detectors [ILMS14] are currently developed in the lab. The CO2 detector uses a CO2 sensor and reacts to changes in the CO2 level in the environment. It can detect nearby breathing victims and other CO2 sources. The sound detector uses microphones to detect spoken words. Both detectors have the problem that the location of the feature is much harder to determine than with the camera-based detectors presented in this thesis.

The feature tracker is another part of the system that can be improved. Currently it uses a greedy algorithm that makes the locally optimal choice for each received feature, but does not find the optimal solution. This can lead to problems, when multiple objects are close together. A solution would be to use an algorithm that searches all possible combinations and finds the the optimal solution. Another improvement would be to implement the full tracking algorithm from [Rei79]. It is similar to to how the object fusion works in this thesis, but keeps the hypotheses between updates. This allows new features to change the feature assignment of early updates by changing the probabilities of the competing hypotheses. This feature tracking algorithm would be much slower, but better adapted to moving objects.

Another possible addition is a better algorithm for calculating the positions of moving feature tracks. A Kalman filter with a system matrices that supports moving objects would be a possible implementation that uses the past movement to predict the future movement. This makes it possible to correctly track moving objects that are temporary occluded.

The raycaster is another area that can be improved greatly. Currently it uses a 2d approximation of the environment and it has many problems as discussed in 4.1.2. A solution would would be to replace the 2d approximation with a 3d approximation of the environment. On the one hand the raycasting would be slower, because the they rays would have to travel through a 3d map instead of a 2d map. On the other hand the 3d map could handle obstacles of different heights and even holes in obstacles. It even has the advantage that ROS already has a library for 3D occupancy grid mapping. It is called OctoMap[1]and described in [HWB+13]. The victim detection system would need another ROS node called **raycasting_octomap**. The node would provide the same service as the existing ROS node **raycasting_map** and internally use the 3d Octomap instead of the 2d OccupancyGrid. The user could decide which node to start and the rest of the victim detection system would work the same.

[1]`http://wiki.ros.org/octomap`

Bibliography

[ARS08] Mykhaylo Andriluka, Stefan Roth, and Bernt Schiele. People-tracking-by-detection and people-detection-by-tracking. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[ASG07] Yotam Abramson, Bruno Steux, and Hicham Ghorayeb. Yet even faster (yef) real-time object detection. *International journal of intelligent systems technologies and applications*, 2(2):102–112, 2007.

[Bre65] Jack E Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1):25–30, 1965.

[Bur04] Steve Burion. Human detection for robotic urban search and rescue. *Diploma Work*, 2004.

[DT05] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[FHL06] Alexander Ferrein, Lutz Hermanns, and Gerhard Lakemeyer. Comparing sensor fusion techniques for ball position estimation. In *RoboCup 2005: Robot soccer world cup IX*, pages 154–165. Springer, 2006.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[HB03] Ju Han and Bir Bhanu. Detecting moving humans using color and infrared video. In *Multisensor Fusion and Integration for Intelligent Systems, MFI2003. Proceedings of IEEE International Conference on*, pages 228–233. IEEE, 2003.

[HWB$^+$13] Armin Hornung, Kai M Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, 2013.

[ILMS14] Stefan Imlauer, Konstantin Lassnig, Johannes Maurer, and Gerald Steinbauer. Life sign detection based on sound and gas measurements. In *Proceedings of the Austrian Robotics Workshop '14*, 2014.

[JA09] Kai Jungling and Michael Arens. Feature based person detection beyond the visible spectrum. In *Computer Vision and Pattern Recognition Workshops, 2009. CVPR Workshops 2009. IEEE Computer Society Conference on*, pages 30–37. IEEE, 2009.

[Kal60] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

[KK07] Alexander Kleiner and R Kummerle. Genetic mrf model optimization for real-time victim detection in search and rescue. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 3025–3030. IEEE, 2007.

[KUS03] Pavlina Konstantinova, Alexander Udvarev, and Tzvetan Semerdjiev. A study of a target tracking algorithm using global nearest neighbor approach. In *Proceedings of the International Conference on Computer Systems and Technologies (CompSysTech'03)*, 2003.

[LLS08] Bastian Leibe, Aleš Leonardis, and Bernt Schiele. Robust object detection with interleaved categorization and segmentation. *International journal of computer vision*, 77(1-3):259–289, 2008.

[LM02] Rainer Lienhart and Jochen Maydt. An extended set of haar-like features for rapid object detection. In *Image Processing. 2002. Proceedings. 2002 International Conference on*, volume 1, pages I–900. IEEE, 2002.

[MBM12] Matteo Munaro, Filippo Basso, and Emanuele Menegatti. Tracking people within groups with rgb-d data. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 2101–2107. IEEE, 2012.

[MSK$^+$11] Johannes Meyer, Paul Schnitzspan, Stefan Kohlbrecher, Karen Petersen, Mykhaylo Andriluka, Oliver Schwahn, Uwe Klingauf, Stefan Roth, Bernt Schiele, and Oskar von Stryk. A semantic world model for urban search and rescue based on heterogeneous sensors. In *RoboCup 2010: Robot Soccer World Cup XIV*, pages 180–193. Springer, 2011.

[QCG$^+$09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 2009.

[RBK98] Henry A Rowley, Shumeet Baluja, and Takeo Kanade. Neural network-based face detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(1):23–38, 1998.

[Rei79] Donald B Reid. An algorithm for tracking multiple targets. *Automatic Control, IEEE Transactions on*, 24(6):843–854, 1979.

[SA11] Luciano Spinello and Kai Oliver Arras. People detection in rgb-d data. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 3838–3843. IEEE, 2011.

[Ser83] Jean Serra. *Image analysis and mathematical morphology*. Academic Press, Inc., 1983.

[SG99] Chris Stauffer and W Eric L Grimson. Adaptive background mixture models for real-time tracking. In *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, volume 2. IEEE, 1999.

[Sha10] SharkD. The hsv color model mapped to a cylinder. http://en.wikipedia.org/wiki/File:HSV_color_solid_cylinder_alpha_lowgamma.png, 2010.

[TPPB08] Marina Trierscheid, Johannes Pellenz, Dietrich Paulus, and Dirk Balthasar. Hyperspectral imaging or victim detection with rescue robots. In *Safety, Security and Rescue Robotics, 2008. SSRR 2008. IEEE International Workshop on*, pages 7–12. IEEE, 2008.

[VJ01] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–511. IEEE, 2001.

[VSA03] Vladimir Vezhnevets, Vassili Sazonov, and Alla Andreeva. A survey on pixel-based skin color detection techniques. In *Proc. Graphicon*, volume 3, pages 85–92. Moscow, Russia, 2003.