Optimize picture scanning

Masterthesis Applied Mathematics

December 2014

Student: H.W.P. de Wilde

First supervisor: A.J. van der Schaft

Second supervisor: N. Brouwer, BMW Forschung und Technik

Abstract

Self driving cars need to be able to check whether there is danger on the road or not. Cars are equipped with cameras and the pictures are scanned for objects. Correctly detecting all the pedestrians in the image takes a lot of time. Research is done to make the detector better, but the detector often still searches through the whole picture. A lot of places in the picture are scanned that, with the use of only global information about the scene, can be discarded. Cars are equipped with a lot of other sensor systems, which can provide the detector with information about the scene. This information should be processed in such a way, that the detector only scans the image in the right places. This will help to reduce the search time and to improve the quality of the detections. This thesis describes a way to model the search space of the detector, which will lead to a framework to optimize picture scanning.

Contents

1 Introduction

In the field of advanced driver assistance systems and in particular in the field of autonomous driving, a lot of research is done to improve the intelligence of the car. It is really important for the car to know what is happening around him. The car is going to be equipped with accurate sensors. For a human inside a car the most important source to gain information is vision. Therefore some of the sensors are cameras. The information that a camera generates comes most basically in the form of pictures. Pictures form an important basis to make the car intelligent and to predict danger. This thesis focuses on optimizing the way pictures are scanned.

Searching for objects in pictures is a time consuming and difficult task. The computer has to be able to correctly detect objects and therefore needs to know what a particular object looks like. The computer needs clear defining features of the objects. In the field of driver assistance systems one of the interesting objects to detect is the pedestrian. Especially for pedestrians, general features are difficult to define. Pedestrians can have many different shapes and colors. In the development of detectors the so called Histogram of Oriented Gradients (HOG) is a widely used method to define the features of shape. Features like warmth and the motion of the body are normally not considered, but could help to decrease false positives of the detector. A false positive is an object that is detected by the detector to be a pedestrian, but what is actually something else. When only features about the shape of the pedestrian are considered, it is hard to avoid a lot of false positives. A tree could for example be seen as a pedestrian because of its strong vertical appearance.

A lot of false positives occur in places in the image where a human brain immediately knows that it cannot be a pedestrian. This occurs for example when the detector detects pedestrians in the air or as a smaller part of another bigger object. Features about location or surroundings could help to improve the detection of the right pedestrians. Another issue of the current detectors, and the first reason to start with this thesis, is the time it takes to scan a complete image. The target is to use the detectors in a real-time setting. A lot of research is done to improve the detector it self, but in this thesis the goal is to reduce the number of places to be searched. The approach taken hopefully has the extra benefit that it avoids a lot of false positives and therefore it actually makes the results of the detector better too.

The idea is to scan the picture only in the interesting areas and to use the information about the environment to do this. It is important to think about the way the information about the environment can be used to improve the quality of the detections. Detectors should only search in the regions of the picture where a pedestrian could possibly appear. This means it doesn't have to look in trees, buildings or air to find a pedestrian. Imagine the detector is given a general picture and it needs to find pedestrians. In theory a pedestrian could appear everywhere in the picture. Only with the use of information about the scene in the picture, it is possible to define places that do not (necessarily) need to be searched. Then optimization can be applied to find the list of patches that the detector should scan. The expected benefits are twofold: first the quality of the detections is expected to improve and secondly only parts of the image need to be searched, which means the detector needs less time to find all the pedestrians.

2 Detector and search space model

This chapter describes the steps that are taken to define a good model for the detector and for the search space. It also explains why certain steps are taken. The diamond and cube model are both models of the detector. The diamond model will not be used in later chapters, while it is replaced by the cube model. It is added to describe aspects of the detector model and to give more insight.

2.1 Diamond model

Normally a detector uses the so called sliding window to search objects in a picture. A detector takes a patch in the picture (a smaller part of the whole picture) and returns whether it believes there is a pedestrian or not. In the sliding window method, every patch from left to right and top to bottom is chosen and checked by the detector. The detector literally slides his window over the whole image. It is interesting to know how big the step size in the sliding window method can be taken, while still all pedestrians are correctly detected.

Therefore, the quality of the detector needs to be tested. As mentioned before, the detector works on patches. The pedestrian does not need to be exactly in the center of the patch and also doesn't need to have exactly the same size as the patch. In this way every patch will lead to an area around its center where pedestrians will be detected. This is the origin why in the rest of the paper the center of the patch and the center of the pedestrians are taken as the reference points of the object.

A model is needed that holds assumptions about the detection quality of the detector. In the literature [Dollar et al., 2012], the performance of a detector is often evaluated using the overlap between the bounding box of the ground truth (the labeled pedestrian) and the bounding box where the detector sees the pedestrian. The bigger the overlap, the better the detector performs. This idea was the inspiration to use overlap between a possible target and the patch as a measure to define the area of the picture that is covered by one patch. In other words: The overlap between the bounding box of a pedestrian and the patch will determine whether we assume the pedestrian will be detected or not. This idea forms the basis of the diamond model.

2.1.1 Two dimensions

The overlap of two rectangles, $s1$ and $s2$, is given by the following equation. Both $s1$ and $s2$ have their own height and width in pixels. The values x and y are the relative distances in pixels between the reference points (the centers) of the rectangles. The formula only hold for $|x| < \max(s1.w, s2.w)$ and $|y| <$

$\max(s1.h, s2.h)$.

Example pedestrian	Example patch
$s1.w = 300$	$s2.w = 350$
$s1.h = 300$	$s2.h = 350$

$$\text{overlap}(s1, s2, x, y) = \min(s1.w, s2.w, \frac{s1.w + s2.w}{2} - x) \times \min(s2.h, s2.h, \frac{s1.h + s2.h}{2} - y)$$

The overlap needs to be normalized to make it usable for comparison. Normalization can be done in many different ways and in this thesis the following normalization is used. The result is the function q, used to measure quality of the detector.

$$q(s1, s2, x, y) = \frac{2 \cdot \text{overlap}(s1, s2, x, y)}{s1.w \cdot s1.h + s2.w \cdot s2.h}$$

Note that q will always be between 0 and 1. The normalization could have been chosen in another way too, but for the current purpose this normalization suffices. The function defines the region in the image that is covered by one single patch. Assume as an example that the detector will correctly find pedestrians if $q > 0.8$ (the value 0.8 is called the threshold). To see what area is covered by a particular patch, fix the size of the patch and the size of a pedestrian with reasonable size and consider the relative distance between both. The value of q is plotted in figure 1 for the relative x- and relative y-position between patch and pedestrian. Furthermore, the plane $q = 0.8$ is plotted.

The area that is covered by one single patch is particularly interesting. This is the area where pedestrians will be found by that patch and it is given by the area where $q > 0.8$. The function q is clearly symmetric in relative positions x and y and therefore the area where $q > 0.8$ will also be symmetric. The exact size and shape of the area depends on the size of the patch and the pedestrian. However, the shape is important: an octagon. While detectors are designed for particular objects, the proportion of the target object (ratio between height and width) is fixed. In this thesis the focus is put on detecting heads of pedestrians and therefore the proportion is normally around one, which means the bounding box of the target is normally a square.

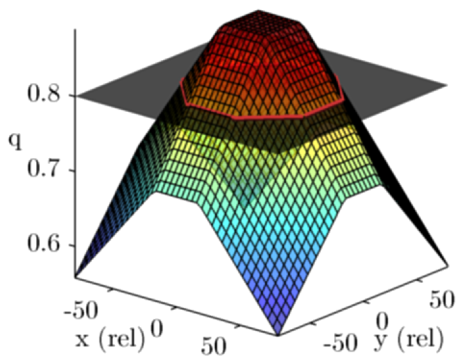

Figure 1: Function q plotted

Now assume all the pedestrians in the image of a certain size (say 300×300 pixels) have to be found. This means a tiling of the octagons has to be defined such that the whole picture is covered. Covering by octagons is not really straightforward, while there exists no regular tiling and a complete cover will always give overlap between different octagons. Therefore the tiling is chosen relatively simple and patches are placed in a grid. So there is only a distance in x-direction or a distance in y-direction defined for subsequent patches.

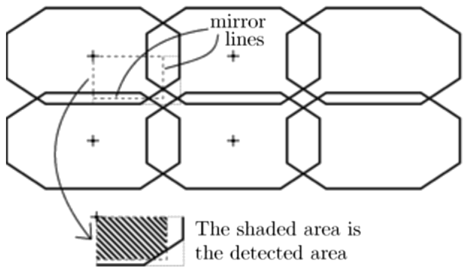

Figure 2: Tiling of octagons

Pedestrians are detected in the area that is covered by the tiling of octagons. A part of the tiling is given in figure 2 and it shows that there is a small area in between the octagons that is not covered. It is interesting to know how much of the total area is covered by the octagons. The mirror lines form a rectangle that is partly covered by a piece of one octagon. It is easy to see in figure 2 that the percentage of the total covered area is equal to the percentage of covered area of the rectangle that is formed by the mirror lines. The corner points of the octagon can be calculated analytically and the percentages of the area that is covered can then be calculated at once for many different step sizes. The size of the target is fixed at 300×300 pixels. For different step sizes and several different patch sizes, the resulting hit percentage against the number of patches needed to cover a 2592×1920 pixel image, are shown in figure 3.
With use of the above picture, the preferred optimal choice of detector size for a particular target can be found. The step sizes are not shown in figure 3, but every point belongs to a specific combination of step size in x- and y-direction.

2.1.2 Three dimensions

Covering the whole image with patches is the first step, but what about different sizes of patches? A detector cannot detect a very small target in a very large patch. Therefore the image needs to be covered by patches of different sizes. The detector typically has a smallest and a biggest target that it can detect in a patch. Now fix again the size of the patch and additionally change the size of the target. Remember here once more, that the proportion of the target is always fixed. Different sizes of targets lead to different sizes and shapes of

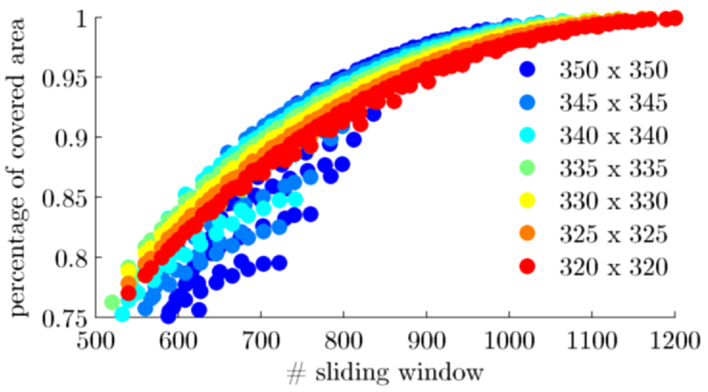

Figure 3: Covering percentages for different step sizes

octagons. This is shown in figure 4.

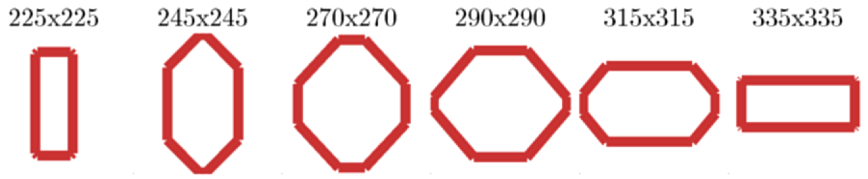

Figure 4: Octagons for several pedestrian sizes for the same patch (250×300)

When the shapes are plotted on top of each other and more steps in between are added, a 3d-shape arises, shown in figure 5.

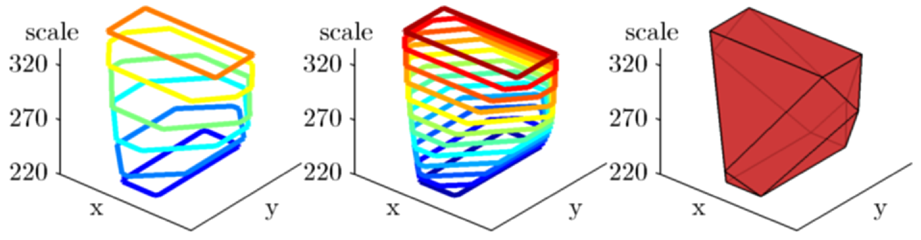

Figure 5: Formation of a diamond

If all pedestrians in the whole picture need to be found, the whole picture should be searched. When besides also different sizes of pedestrians need to be found, the whole picture should be searched on different scales too. But how big is the step between subsequent patch sizes? The model as described above can give an answer to this. Every patch leads to a diamond-like-shape. A point inside the diamond corresponds to properties of a pedestrian: size and position. Note that proportion was initially already fixed. When the properties of a pedestrian lay inside the diamond it is assumed to be detected by the corresponding patch. That means that the pedestrian will also be detected if it is a little bit smaller or a little bit bigger than the patch. The size of the diamond helps to find the next patch size that needs to be selected.

2.2 Search space model

The end of the previous sections leads to a description of the initial search space. Pedestrians in the picture can appear on different locations and on different scales; the detector has three dimensions to search: x- and y-position and scale. Therefore the search space of a picture can be seen as a subset of the three dimensional space defined by a range in x- and y-position and a range in scale. The whole picture (this gives the range in x- and y-position) could for example be searched for pedestrians with sizes ranging from 200 to 800 pixels. As explained before, under the chosen assumption every patch gives rise to a diamond. This diamond is a small subset of the search space. The problem of finding the right patches to scan the picture is now transformed into a problem of finding a way to place the diamonds such that the diamonds together completely fill the search space.

A regular tiling of diamonds that fill the search space doesn't exist. In particular since the diamonds in the problem have the strange property to have a fixed size depending on the size of the patch and the diamonds obviously cannot be rotated. Like in the case of the octagon tiling, a simple tiling can be chosen where the diamonds partly overlap eachother. Why are we actually working with a diamond shape? The diamond-shape is a consequence of the chosen overlap and quality function and the assumption that this function defines the pedestrians that a patch can detect. Any other assumption about which pedestrians the detector detects in a single patch could also be good. Deciding whether the model based on overlap matches the real detector, is not part of this thesis because it strongly depends on the used detector and it is important to keep it general. What at least stands out is that the model that has been used until now is not flexible; there is only one parameter to cover the properties of the detector: the threshold (see figure 1). Therefore another model is described in the next section.

2.3 Cube model

The need for a more flexible model for the detector arises. Central question: Within what range of a patch it can be assumed that the pedestrians will be detected? In this section a new and more flexible model for the detector is introduced.

As described above, the image needs to be searched in three directions: x-position, y-position and scale. Every patch leads to a covered range in x-direction, a covered range in y-direction and a range in scale. The diamond was an example of such a covered combination of ranges. The new model will not be an octagon that changes on every combination of sizes, but will be a rectangle that is equal for every combination of sizes. The rectangle and the different possible sizes are given by ranges. Every range can be separately chosen, which makes the model more flexible. The range in x-direction is defined as a percentage of the size of the patch itself. So, the smaller the patch, the smaller the region it covers. This is a natural assumption and the same holds of course for the y-direction. The range in scale direction is defined by a lower and upper bound. That could for example mean that a detector is able to correctly detect pedestrians of sizes between 300×300 and 380×380 in a patch of size 400×400. These lower and upper bounds are again given as a percentage of the size of the patch. In figure 6 the definition of the parameters is illustrated with example values.

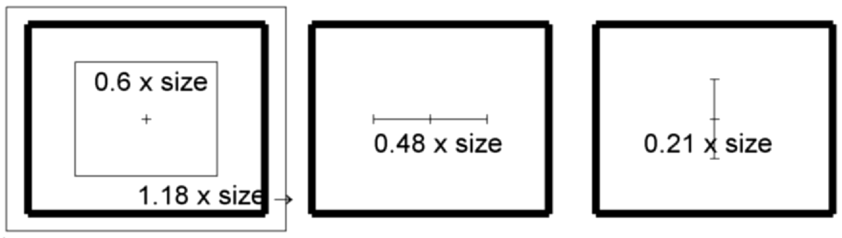

Figure 6: Parameters of the second model

All the four percentages above can be chosen independently. This makes the model more flexible and it is likely to capture the performance of a real detector better. For every particular detector the right parameters have to be chosen. This can be done by testing within which range the detector correctly determines the occurrence of a pedestrian.

The big difference between the diamond-shape, is that the range in x and y direction where pedestrians are assumed to be detected, do not depend on the scale of the target. The three dimensional shape can again be visualized in the same way it was done with the diamond in figure 5, by just drawing the shapes for different sizes on top of each other, see figure 7.
The interpretation of the model of the cube as shown in figure 7 is the same as the model of the diamond in figure 5. The important explanation can be reviewed at the end of section 2.1.2.

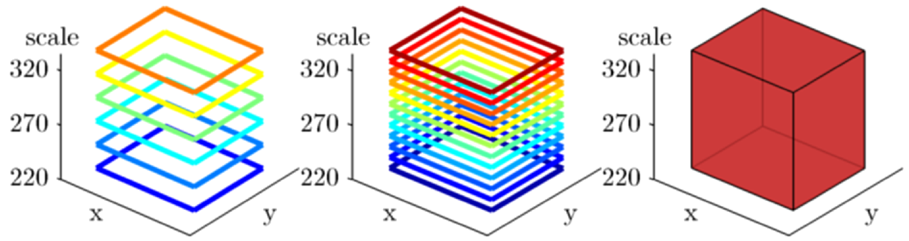

Figure 7: Formation of a cube in the same way as in figure 5

3 Fill the initial search space

The rest of the thesis uses the cube model and the search space model as described in the previous chapter. This chapter explains how to create an initial list of patches that according to the cube model completely fills the initial search space in an efficient manner.

The initial search space is given as a range in x-, y- and scale-direction and therefore looks like a big cube. The model of the cube gives a straightforward way to fill this initial search space: simply side by side. It works perfectly in the x- and y-direction. But, since the size of the cubes get smaller when the size of the patches get smaller, the straightforward way to make a filling for he complete search space, is to place them layer by layer. Because the cubes get smaller, this means that you need more patches to find the smallest pedestrians in the picture. The first and last step of the process of filling the initial search space layer by layer is shown in figure 8.

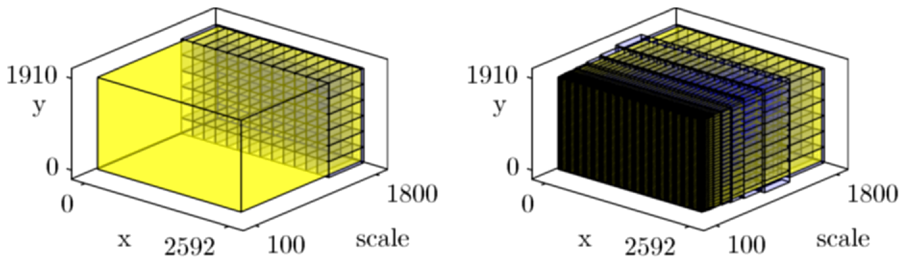

Figure 8: An example of filling the search space starting with the biggest cubes

With the settings used in this example, 46691 cubes (eq. patches) are needed to fill the initial search space. Something appears to go wrong at the smallest size. The smallest cubes are needed to fill the last part of the search space. These cubes are chosen smaller than necessary, since they are chosen to connect to the cubes that are one size bigger. It would be better to choose them to have the size that covers until exactly the smallest possible pedestrian (the lower bound of scale of the search space). Choosing its size like that, makes the smallest cubes larger (larger is not possible). The result is that there are less cubes needed to cover the search space on the smallest target size and while the cubes are really small, the total number of cubes in this layer is really reduced by this change.

As a consequence, the second smallest size of cubes in turn are also chosen too small. It can be chosen bigger, while the range of the smallest patches is now a little bit increased. Choosing the second last size a little bit bigger has the same advantage that we just saw by increasing the smallest size: there are fewer cubes needed to cover the whole image. Applying this principle to all the other layers too, it turns out that the same search space, with exactly the same settings, needs only 27048 patches to be completely covered (see figure 9). There is only one conclusion possible: the search space should be filled up starting with the smallest cubes first. It's not a formal proof, but coming up with a more efficient way to fill the search space is likely to be impossible.

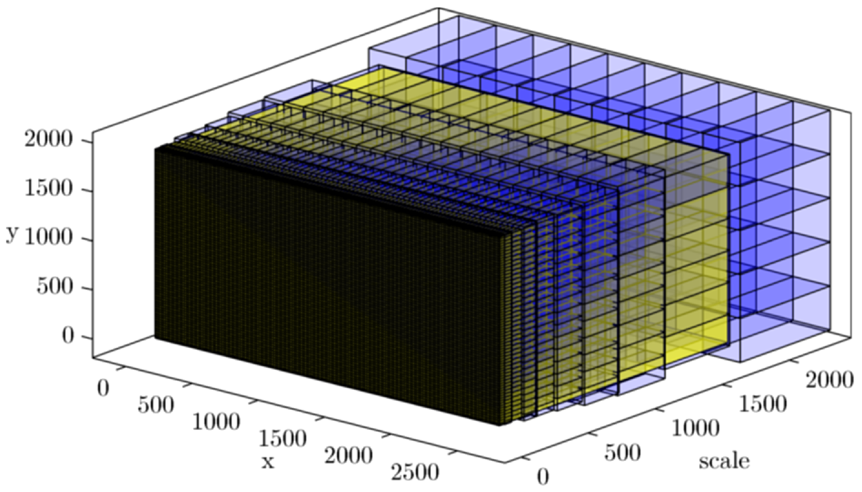

Figure 9: An example of the most efficient initial filling; starting with the smallest cubes

The way of filling the search space as just described, results in a distribution of the small cubes. This in turn corresponds to a list of positions and sizes of patches. This list can be used as input for the detector to search the whole picture. But, as mentioned in the introduction, there is a desire to search only parts of the image and that will be the topic of the rest of the thesis.

4 Environment model

A good starting point for improving results on the search strategy of a picture are the just created model for the detector and the initial search space. Without any information about the scene in the picture, the target (head of a pedestrian) can be anywhere. In this case, scanning the whole image would be the only solution to find all pedestrians and a way to find the corresponding list of patches is given in the previous chapter. To be able to take the next step, we definitely need information about the scene in the picture. This information could provide areas where a pedestrian could be expected or not. In the configuration of a camera in front of a car you could for example expect that pedestrians will not appear in the air. But where is the air? Normally it is in the upper area of the picture, but be aware: a pedestrian in front can be big, so that his head will be visible in the upper area of the picture. Therefore it is important to make a difference between the area that should be searched for small pedestrians and for big pedestrians. Something similar holds for the ground plane. In the image the ground plane normally goes up in the picture. The whole perspective in the image is really important for drawing the right conclusions about where to search. The camera is mounted into the car and the car is standing on the ground, this already means that certain assumptions about the scene can be drawn.

Other really important information to reduce the search space is the existence of buildings, other cars, sidewalks etc. Buildings and cars near to the car could reduce the search space. Sidewalks can give a lot of information about the possibility of pedestrians to appear there or not. All this information is really important for drawing conclusions about where to search and where not. Notice that a lot of this information strongly depends on the particular scene. This means there is a strong need to have live information about the scene.

Information about the scene can also be static. Information about for example ground plane and air does not differ too much from scene to scene. Besides, statistics of labeled data can be used to draw reasonable conclusions about the regions where pedestrians appear.

But how can this information be used to reduce the search space? The first step is to take care of the areas where you don't have to look at all. This is for example given by the existence of buildings, cars, the ground plane or air. Looking 'behind' buildings or other big objects is useless. The word 'behind' is marked, while the picture you look at is a two dimensional projection of the world. An object that would normally be behind a building is seen in the picture at the same position as the building, but with a smaller size. Therefore having information about buildings and big objects in the picture is only useful when you know how far the object is away. Only then the size of the object can be estimated and you can assume that you don't need to search for any smaller objects in this region of the picture. Exactly this issue makes the distance information about the objects in the picture really important.

The cars that will have a camera in front will therefore (and because of a lot of other purposes) be equipped with laser scanners. These laser scanners are the sensors that can provide distance information about objects in front of the car. Transformed back to the two dimensional image plane, this will help to estimate the distance to objects in the picture.

The data from the laser scanners can also help to reduce the search space in front of objects. If the laser scanner detects an object, it means that there is no object between the laser scanner and the detected object and therefore these sizes can probably also be removed from the search space. Note that the whole beam of the laser scanner needs to be transformed to the image plane, so that the right sizes on the right locations are removed form the search space.

Creating this environment model is a big and interesting task, where a lot of developers are working on. A big part of the environment model is already available or will at least be available soon. This thesis therefore assumes that an environment model is available. The more important question is how this information can be used to find the right patches to search.

What the next chapters assume to be available, is an environment representation in terms of regions in the picture where a certain range of pedestrian sizes is not expected to appear. In other words: a subset of the search space that does not need to be searched. Subsets of the search space are three dimensional shapes. It is common to describe these by meshes. With use of the environment model it is possible to describe meshes where the detector doesn't need to search. What these meshes exactly look like, is a matter of environment interpretation. The meshes form the basis to find the patches that efficiently cover the remaining parts of the search space. These remaining parts are together called the reduced search space. Globally, there are two different ways to cover the reduced search space: The first method is to slice away the patches from the previous initial list of patches that cover the initial search space and the second method is to find the area where we want to search and to cover this area from scratch. In the next chapter we will focus on this step.

5 Find the patches

Following the discussion in the previous chapter, this chapter will discuss the two main ways to reduce the search space and more important to find the patches that fill this reduced search space. The first method is based on the list of patches that cover the whole picture and slices away the positions that do not need to be checked. This method is therefore called the slicing method. The other method turns the procedure around and starts with the meshes defining the environment and finds a distribution of patches that cover these meshes. This method is therefore called the covering method.

Both methods will be compared afterwards. Two short conclusions can already be revealed: The slicing method is computationally much faster, while the covering method can be used to find a much better (perfect) solution when computation time is not important.

5.1 Slicing method

The slicing method starts with an initial list of patches that should intentionally cover the initial search space. A procedure to construct such a list is described in chapter 3. Besides the starting list, the method uses the environment model. The environment model is given as a list of meshes that together define the search space as explained in chapter 4. The slicing method essentially throws away the patches that cover areas that do not need to be searched. This is mainly a geometrical task, that can be computationally very expensive. Matlab is chosen to write the framework and functions to do these tasks. With the use of built-in Matlab functions and the packages geom2d [Legland, 2005] and geom3d [Legland, 2009] a lot of tasks can efficiently be computed. A function from the package geom3d is able to decide very fast whether points lay inside or outside a given mesh. This function is used to check if the corner points of each small cube from the initial list lay in or outside the reduced search space. Note that this procedure contains a small simplification: instead of checking whether the whole small cube is in or outside the reduced search space, only the corner points are checked. The result is a list of the cubes that (almost) completely cover the reduced search space.

The performance and results of performed tests of the code [de Wilde, 2014] show that the generated list covers the reduced search space completely. Some small cubes are only contributing with a very small part of its volume, for example only the very corner touches the search space. This small cube is therefore not really important. A way to delete them afterwards is to calculate how much it contributes and find a threshold or rule to delete it when the contribution is too less. This idea has been implemented, but it appeared to be computationally a very expensive task while the number of extra deleted patches is in comparison too low. Because of this low gain for long computation time, this idea is removed from the code and is not used for further elaboration.

5.2 Covering method

The reason to start with the development of the covering method is the wish to have a better distribution of patches. This means, covering the to-be-searched areas in the picture such that fewer patches are needed. However, the whole geometry of the search space needs to be considered when choosing the very best distribution. This can be a very computationally expensive task. This section explains several different strategies to cover the search space. Those strategies all have a certain own balance between computation time and precision. In general more precision means that more computation time is needed.

The approach chosen to find a cover for the reduced search space is to create and solve a binary integer program (BIP) [Bertsimas and Tsitsiklis, 1997]. This is a natural approach, while a function has to be minimized under certain constraints: Minimize the number of patches, while the whole search space remains covered. A big difficulty is to find a way to transform the geometry into constraints. A description of the boundaries in terms of inequalities is not what is needed; a description of the areas that are covered and uncovered by a certain choice of patches is needed and needs to satisfy the constraint that everything needs to be covered. Recall the general mathematical form of a BIP; it is given by the following equations:

$$\begin{array}{llll} \text{Minimize} & z = f^\top x & f \in R^n & x \in R^n \\ \text{such that:} & Ax \leq b & A \in R^{m \times n} & b \in R^m \\ & x_i \in \{0,1\} & \forall i \in \{1,..,n\} & \end{array} \tag{1}$$

The first approach is to slice the geometry into several parallel layers. Those layers correspond to the layers that were used to fill the initial search space as in chapter 3. The layers therefore look like slices from the reduced search space and every layer has a very specific own two dimensional geometry. Slicing one of the cubes that correspond to a patch, results in a small rectangle. The problem of covering the search space around the sliced layer, now corresponds to the problem of covering a two dimensional geometry by rectangles. Note that the two dimensional geometry is a result of a sliced combination of meshes, which means it has the form of a two dimensional polygon.

The procedure of slicing the reduced search space into layers is a simplification. It reduces the three dimensional problem into a series of two dimensional problems. It would have been possible to create a similar three dimensional problem, but the computation time for the two dimensional problem will turn out to be long enough, which implies that a three dimensional extension would not be interesting to consider any way.

Therefore this subsection focuses on the two dimensional simplified problem. In two dimensions, the problem is known as the polygon covering problem. While this problem is a basic problem, it appears in different research areas.

Already around 1980 - 1990 researchers worked on it and showed that it is an NP-hard problem [Franzblau and Kleitman, 1984]. In the last years again some papers appeared about the subject of polygon covering [Stoyan et al., 2011]. It is for example used to create an optimal atlas [Hoffmann, 2001]. The papers all propose methods to approach the problem. Often they impose special assumption about the geometry that do not apply to our problem and therefore no good methods have been found in literature. The author of this thesis wanted to exploit the possibilities and difficulties himself and this is described in the upcoming sections.

5.2.1 Complete covering

How to transform the problem of covering a two dimensional polygon into a BIP? The idea is to transform the polygon into a pixelated image. That means, creating a raster of cells over the polygon and fill every cell that is part of the polygon. This pixelated polygon defines the area that needs to be covered and can be used to set constraints on every pixel. Every single pixel needs to be covered by at least one rectangle. But every pixel will also serve as the position vector of a potential patch. The polygon will be covered by identical patches, while the slices of the reduced search space are exactly chosen in the layers of the original filling.

Every pixel in the pixelated image serves as the position vector of a potential patch and every pixel also defines a constraint, saying by how many rectangles it needs to be covered (zero or one). Therefore every pixel gives rise to a decision variable. More precise: for every pixel there is an associated decision variable that has value 0 if it will not serve as the position vector of a patch and it will have value 1 if it does serve as a position vector of a patch. In other words: whether there will be a patch placed on that location or not. Secondly, every pixel gives rise to a constraint. Therefore we need to count for every pixel, which potential patches would cover it. These patches are located just above and just left to the pixel that needs to be covered. That means that whenever a patch is located just above and just left to the pixel, the pixel is covered and the constraint is satisfied. If more than one patch is placed there, the pixel is covered twice or more often and the constraint is satisfied as well. The constraint has therefore the form of a summation over unweighted decision variables that should be bigger than or equal to one. The result of the complete problem, when written in BIP form is given below.

The pixelated image has m rows and n columns.

$$a, x \in \{1, .., n\}$$
$$b, y \in \{1, .., m\}$$
$$p_{ab} = \begin{cases} 1 & \text{if pixel in column a and row b needs to be covered} \\ 0 & \text{otherwise} \end{cases}$$
$$s_{xy} = \begin{cases} 1 & \text{if a patch is placed in column x and row y} \\ 0 & \text{otherwise} \end{cases}$$
$$w : \text{fixed width of the patch in pixels}$$
$$h : \text{fixed height of the patch in pixels}$$
$$\text{minimize } z = \sum_{x=1}^{n} \sum_{y=1}^{m} s_{xy}$$
$$\text{s.t.} \sum_{x=\max(0,a-w+1)}^{a} \sum_{y=\max(0,b-h+1)}^{b} s_{xy} \geq p_{ab}$$

The equations have to be written in standard matrix form as given by equation (1). The constraints in the above formulation have to be multiplied by -1 to match the standard form. Matlab uses linear indexing for matrices and this is also applied to the decision variables s_{xy} and the constraints with RHS p_{ab}. This can be seen as listing the decision variables and constraints in the order from top to bottom, from left to right (column by column). The entries in A are 0 or 1 and an example of the matrix is given in the figure 10. The matrix shown here belongs to the example on the left in figure 12. The entries of the given vector f are all 1 and the entries of the given vector p_{ab} are the linear indexing of the to be covered pixelated image.

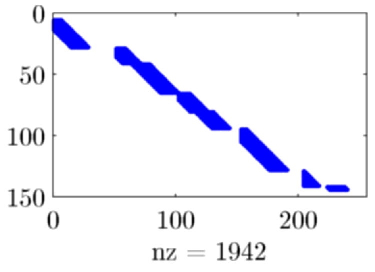

Figure 10: The entries of A

Note that all the entries in the matrix A and the vectors f and b are 0 or 1 and that all the decision variables can only adopt binary values. Actually this problem can be seen as a set covering problem, known from literature [Caprara et al., 1999]. Set covering problems are NP-hard and therefore solving it is a very time consuming task. Especially when you notice that the number of decision variable and constraints is really big; every pixel gives rise to a variable and to a constraint. Without down scaling the image of 4096×3072 pixels (the image size of the camera from the car), this would result in 12582912 decision variables and a similar amount of constraints. Downscaling the image and the corresponding patches would really help to create the solution faster. The steps downscaling, solving, and then upscaling again generate little round-off errors. Those round-off errors can be negotiated when the computation time is really important.

A way to solve the created BIP is to use the function *bintprog* from the Optimization Package in Matlab, designed to solve a BIP. The latest version of the Optimization Package that could be used for calculations during the writing of this thesis was version 6.4 (R2013b) 08-Aug-2013. Whenever a BIP is solved with a BIP solver, this solver *bintprog* is used.

The method and solutions of the covering method are demonstrated with the use of a simple binary image: the shape of a star that needs to be covered.

The pixels will be numbered from top to bottom and from left to right. That means for example that if the second decision variable turns out to have value one, there will be a patch placed in the second row in the first column. Note that the position vector is given at the upper left corner of the patch. Furthermore this means that the constraint matrix A also respects this way of numbering the pixels. This way is chosen while the index numbering in Matlab is defined in the same way. The constraint matrix A has a simple structure, see figure 10. Note that in this figure, some unnecessary constraints are removed. Only the constraints where the right hand side is 1 are needed, while the others, where the right hand side is 0, are always satisfied.

Besides unnecessary constraints, there are also a lot of unnecessary decision variables. When the picture contains a lot of zeros (corresponding to the pixels that do not need to be covered), there will be a lot of positions that when a patch is placed there, it doesn't contribute to the area that should be searched. Only the positions in the image where a patch causes overlap with the to-be-covered polygon, are interesting to use as decision variable. A method similar to the Minkowski sum [Lee et al., 1998] can be applied to the rectangle and the polygon to get the interesting decision variables. A function to create this Minkowski sum is included in the implementation.

The number of decision variables can further be reduced. Every single rectangle that would be placed at a point that results from the Minkowski sum will overlap the polygon with at least one pixel. However, this overlap could easily be covered by another rectangle too. The latter maybe covers other pixels of the polygon too and can therefore be favoured. The former rectangle is therefore not interesting to consider and the decision variable corresponding to its position can be removed from the optimization problem.

After selecting the right decision variables and constraints, a solution can be found by the BIP-solver *bintprog*. As mentioned before, this can take really long when a big image is selected. The solution that is returned should consist only of zeros and ones. This solution needs to be transformed back in to coordinates for the patches. The definition of the decision variables is very important there. After the right transformation, the solution can be shown and in the case of the star, the cover looks like figure 11.

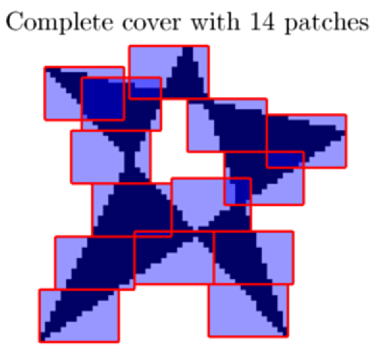

Figure 11: An example of a complete cover

The steps are worked in out a Matlab function, called *createcover*. The code can be requested [de Wilde, 2014].

5.2.2 Horizontal and vertical covering

As seen before, the computation time for the complete covering method is really long. Although the complete covering method gives the real optimal covering, it can also be really interesting to consider a rougher covering method, where solutions can be computed in less time. Rougher here means that less locations are considered to use for patches. Two ways are chosen: only cover the polygon in a vertical or horizontal manner. The idea to solve the problem is exactly the same, but some simplification can be made. Solutions of both the vertical and horizontal covering according to the BIP-solver are given below. These methods are also included in the function *createcover*, which can be requested [de Wilde, 2014].

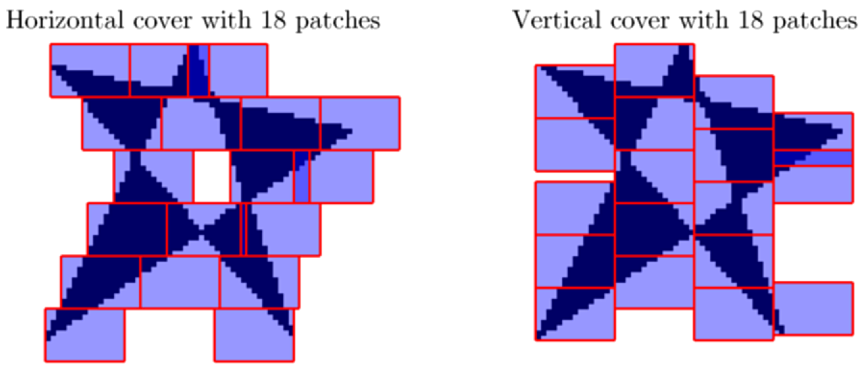
Horizontal cover with 18 patches
Vertical cover with 18 patches

Figure 12: Two simplified ways to cover a polygon

The structure of the constraint matrix A for the horizontal and the vertical cover look quite simple. Investigating the structure and purpose of these matrices could maybe lead to faster solution methods that could approximate the optimal solution really quick. As a test, the solutions of the simple matrix equation Ax = b were examined and apparently it gives very good results. Of course the solutions of these matrix equations, as computed by Matlab ($x = A\backslash b$) do not necessarily have to be integer. It unfortunately doesn't work for the complete covering, since the solution then contains a lot of non-integer values. The structure of the matrix A in case of the horizontal or vertical covering method however, seems to be a motivation for Matlab to return an (almost) complete binary solution. The horizontal and vertical covering problems of figure 12 are as an example solved by the equation $Ax = b$ and this results in a cover of respectively 18 and 19 patches. Compared to the results found by using the BIP as in figure 12, this is really not a bad result. This example shows that the results are very good, although not necessarily optimal. The computation time however is shorter and that shows that this is an interesting field to do more research.

6 Calculations

The developed methods and corresponding code is now capable to be applied to a picture. A picture from the camera of a test car is taken. This picture gives the information about the environment and for this example the meshes of the environment model are defined by hand. The first step is to outline the big objects that are located within the range of the environment where pedestrians should be detected. The second step is to estimate how far the corner points of these outlines are away from the camera. Then this distance has to be transformed into the expected size of the target pedestrians. In the example a detector for detecting heads of pedestrians is applied and therefore the target size in centimeters is approximately 25 cm. For the transformation, the pinhole model for the camera can be used, where features about the camera needs to be applied. In the example a simple estimation is applied by hand, while at this moment it is just used to demonstrate the principles discussed in this thesis.

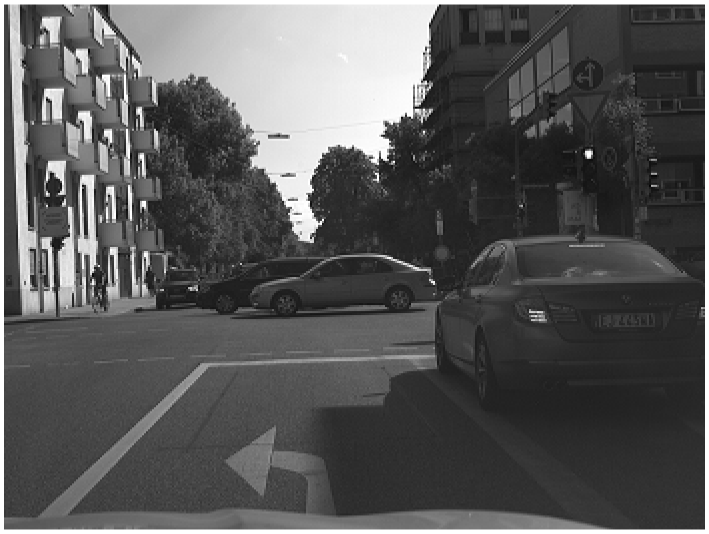

Figure 13: The picture from a test car used as an example

Three cars are visible in the picture within the range of the environment where pedestrians should be detected. Also an estimation of the ground plane is made. This ground plane can also be used to find a plane for the air. Pedestrians that are standing on the ground plane are particularly interesting and standing on the ground plane means that they will not appear really high in the picture. The further away from the camera, the smaller the pedestrians appear in the picture and therefore the air and ground plane approach each other in the direction of smaller scales. Figure 14 shows the result of the environment model.

The yellow mesh is the result of the complete search space where the ground plane and air plane are sliced away. The inside of this mesh should be searched,

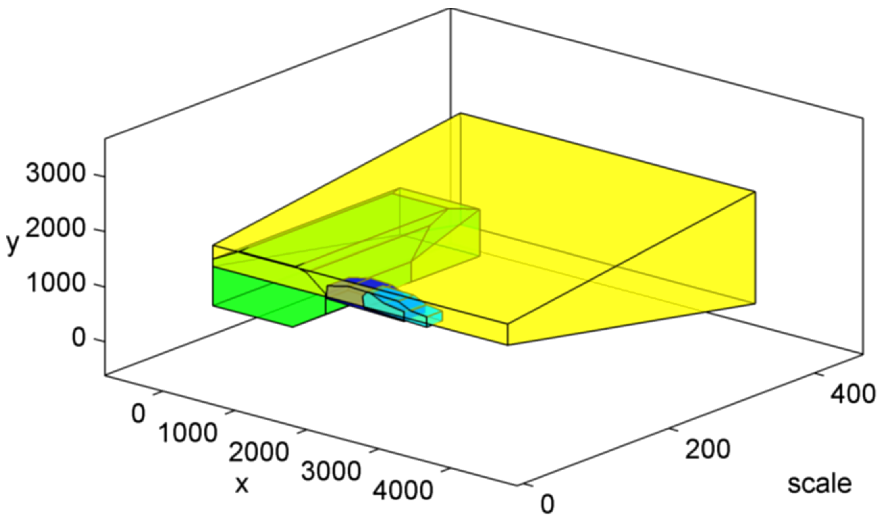

Figure 14: The environment model of the example

except the parts that are occupied by the three cars; the green and the light and dark blue meshes. The two blue cars are just close enough to be within the interesting distance to the camera. The green car is closer and therefore the green car removes a bigger part of the search space.

The settings for the initial search space are chosen to match the picture. That means that the x-range is $[0, 4096]$ and the y-range is $[0, 3078]$. This is equal to the width and height of the image in pixels. The range of scale, that means the range of sizes of the heads of pedestrians that should be detected, is chosen to be $[60, 400]$. This range is approximated by hand. It means that the smallest pedestrians to be detected are as far away as the cars that are crossing the street and the biggest pedestrians to be detected are standing just in front of the camera.

The settings for the detector are an approximation of the quality that a real detector could have. The proportion is chosen to be 1. The patches will be squares, matching approximately the proportion of a head. The maximum detection range in X and Y direction are chosen equal, namely $0, 5$. In short this means that the distance to the next patch will be $0, 5$ times its size (so patches overlap). The scale lower and upper bound and chosen respectively $0, 65$ and $1, 00$. This means that a pedestrian with exactly the same size as the patch is the biggest pedestrian that will be detected in the patch and a pedestrian with $0, 65$ times the size of the patch is the smallest pedestrian that will be detected in the patch.

The above mentioned settings always need to be available and should therefore be set in the beginning of the code.

6.1 Apply slicing method

The first step of the slicing method is the creation of the initial list of patches. This list is created in the way that is explained in chapter 3. The list for the example here consists of 10.284 patches.
Then this list of patches is going to be sliced by the meshes. A list of the corner points of all the small cubes is created and with the use of the function *inpolyhedron* [Sven, 2012] it is checked whether one of the corner points lays inside the reduced search space defined by the meshes. As long as one corner point lays in the reduced search space, the cube will be kept and otherwise it will be thrown away. The result is a list of only 2.074 patches.

Note that the biggest part of the patches is sliced away by the ground and air plane. Without slicing away the cars and therefore only slicing away the air and ground, the initial amount of 10.284 patches is already reduced to 2.658 patches. Additionally the big car in the front slices away another 584 patches, resulting in the 2.074 patches that was mentioned before. The two cars in the distance do not slice away anything. That's because the cars are to far away and all the patches around the cars are still filling a part of the reduced search space. The result of the slicing method therefore is 2.074 patches.

6.2 Apply covering method

The covering method starts with the meshes and fills it up from scratch. The method itself has different parameters to choose. There are three different covering methods: the complete cover, the horizontal cover and the vertical cover. These methods are tested separately. Besides this choice, there are two solvers to choose. For the complete covering method only the BIP solver works (Binary Integer Programming). For the horizontal and vertical covering method both the BIP solver and the EQ solver (Equation) work. Furthermore the image needs to be downscaled in order to keep the number of decision variables low enough. Downscaling the image appeared to be tricky. A too small scale results in too small patches (which ofcourse also need to be scaled) and a too large scale results in a too big problem. The calculations are split into the different covering methods.

6.2.1 Complete covering method

For the complete covering method there is only one solver available, the BIP solver. This is the more time consuming solver. Unfortunately the complete covering method appeared to be too complicated to give results. Even when choosing the smallest scaling, it appeared to be impossible to retrieve a solution for the example. The problem is the complexity of the problem. Every pixel of the downscaled image is added as a decision variable and for the size of the example this simply results in too many decision variables.

6.2.2 Horizontal covering method

The horizontal covering method can be solved with either the BIP solver or the EQ solver. Both methods work fine for this covering method. First the BIP solver is applied. A scaling of $0,10$ is used. The result is a list with 1900 patches. Other scaling factors could also be examined, but here we stick to $0,10$. Using a bigger scaling can cause memory overload and using a smaller scaling makes the results unusable.

The same scaling is applied for the EQ solver, which calculates the patches by solving the equation $x = A\backslash b$. The calculation is faster and it results in a covering of 1912 patches. This is slightly more than the solution of the same problem with the BIP solver. This is because the BIP solver really finds the best solution to the problem and the EQ solver just approximates it.

6.2.3 Vertical covering method

The same methods as the horizontal covering method can be applied to the vertical covering method. It results in 1889 patches when using the BIP solver and it results in 1905 patches when using the EQ solver. The number of patches are similar to the number of patches of the horizontal covering method.

7 Discussion

The previous chapter shows the results of the introduced models. This chapter will give an overview of the structure of the code used to calculate the results. This structure will show the different steps and the calculation time of different steps of the previous example can then be compared. This will show which steps take more time then others (all tests are performed on the same example as in the previous chapter). This chapter is added to make a good comparison of the different methods possible and also to show which steps can be improved as an extension of this thesis.

In general the slicing method is more practical. It doesn't have problems with running out of memory and the complexity grows linear with the size of the problem. The slicing method basically consists of two parts: first the creation of the initial patch list for the whole search space and then the selection of the patches that are part of the reduced search space. It takes only 0.08 seconds to create the initial patch list with 10.284 patches. The most expensive step is the second, the selection of the patches belonging to the reduced search space, performed by the function *slicePatchListByMultipleMeshes*. For the example this takes around 30 seconds. The problem is that the eight corner points of all the 10.284 patches are checked to lay in or outside each mesh separately. The code could further be optimized to reduce this number of checks by selecting other corner points or to combine corner points. However, it would be much more efficient to define the initial search space more precise. The air and ground plane already remove 80% of the patches. The air and the ground plane are part of the more or less static objects in the picture, since the air and ground plane are always approximately in the same position in the picture. Slicing away static objects from the patch list does not have to be executed in real time. This can be done before. The benefit is that the function *slicthePatchListByMultipleMeshes* only needs to be applied to 20% of the patches. To show this benefit in terms of calculation time observe the following. In the example shown, the dynamic objects only slice away a small part of initial search space. A shorter initial patch list can be created by first processing the algorithm with only the air and the ground plane. The resulting patch list can be used as input for the algorithm and now only dynamic objects need to be sliced away. Remember that the whole algorithm took approximately 30 seconds for the previous example. When using the shorter initial patch list, only 3.5 seconds are needed to slice away the cars. The result is a final list with 2093 patches. That is slightly more than the result of slicing away the cars and ground and air plane simultaneously.

The covering method can easily become too big to be solvable. Downscaling the problem can help to reduce complexity, but this causes loss of precision. However, whenever a solution of the covering method is given, the result is guaranteed to be optimal. Any solution of the slicing method will always be worse or equally good as the solution of the covering method. The covering method can therefore be used to find a (theoretic) lower bounds for the amount of patches to compare other methods. Even if the geometry is really irregular. The covering method will never be suitable to give real time solutions.

The sequence of pictures are taken from a moving car and therefore the objects in a picture are often also visible in the next picture. The strategy of the current algorithm is to find the list of patches based one a single picture. Extending the current algorithms to create patch lists based on multiple pictures can maybe speed up the calculation time, since information about the past can be used for the new picture.

The slicing method works faster and more practical then the covering method. It is more suitable to be extended for using the solutions of the past. When using the slicing method, it is really beneficial to make the initial search space as small and precise as possible. That means that static information, like the air and ground plane is used to create an initial search space. Additional to the static information that defines the initial search space, there is the desire to use information about dynamic objects. This information is delivered in real time and therefore needs to be calculated in real time. The algorithm is not working in real time at this moment. Program and code optimization has not yet been applied.

The approach taken in this thesis, where we started with nothing and came up with different optimization problems is probably the core content of the thesis. The problem has gone through its first transformation into an equivalent geometric problem. For this new equivalent problem, the thesis also describes different solution methods. These methods have just been compared, but unfortunately they do not yet meet the requirement of performing in real time. Different improvements can still be made there.

8 Future work, improvements and possible extensions

Searching only parts of the search space and furthermore to do this really effectively is a straightforward way to make make results of detectors faster and better. Research has to be done to define what regions in the image are interesting and why. This seems trivial, but it isn't. A great part of research can be done in the dynamic definition of regions of interest. What makes a region interesting? What technical tools and measurement data is available to robustly define those regions? Those definitions will most likely result in percentages defining the importance of regions. How to handle these percentages? Can these percentages be added to the idea and approach that was taken in this thesis?

The dynamic aspect of the problem has to be exploited. Using the list of patches from the past to find the new patch list gives a complete new dimension to the problem. The slicing method seems to be more suitable to be extended in this direction.

When using the current approach and techniques, interesting mathematical research can be done by developing a very specific solution method for the vertical and/or horizontal covering problems as mentioned in the end of section 5.2.2. The sparsity of the resulting matrices in the inequality $Ax \leq b$ could lead to solution methods similar to solving $Ax = b$. This solution method should guarantee a binary (optimal) solution. The vertical and horizontal covering problems can be seen as a series of one dimensional covering problems and can therefore be solved as such. This could lead to faster solving algorithms.

References

Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific Belmont, MA, 1997.

Alberto Caprara, Matteo Fischetti, and Paolo Toth. A heuristic method for the set covering problem. *Operations research*, 47(5):730–743, 1999.

H.W.P. de Wilde. Matlab files belonging to master thesis, 2014. URL `jelmerdewilde at gmail dot com`. Code requests can be send to the mail adresss.

Piotr Dollar, Christian Wojek, Bernt Schiele, and Pietro Perona. Pedestrian detection: An evaluation of the state of the art. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(4):743–761, 2012.

Deborah S Franzblau and Daniel J Kleitman. An algorithm for covering polygons with rectangles. *Information and Control*, 63(3):164–189, 1984.

Michael Hoffmann. Covering polygons with few rectangles. In *Abstracts 17th European Workshop Comput. Geom*, pages 39–42, 2001.

In-Kwon Lee, Myung-Soo Kim, and Gershon Elber. Polynomial/rational approximation of minkowski sum boundary curves. *Graphical Models and Image Processing*, 60(2):136–165, 1998.

David Legland. geom2d - two dimensional geometry package, 2005. URL `www.mathworks.com/matlabcentral/fileexchange/7844-geom2d`. [Published 13 Jun 2005 (Updated 27 Oct 2014); accessed 18 December 2014].

David Legland. geom3d - three dimensional geometry package, 2009. URL `www.mathworks.com/matlabcentral/fileexchange/24484-geom3d`. [Published 19 Jun 2009 (Updated 13 Oct 2014); accessed 18 December 2014].

Yu G Stoyan, Tatiana Romanova, Guntram Scheithauer, and A Krivulya. Covering a polygonal region by rectangles. *Computational Optimization and Applications*, 48(3):675–695, 2011.

Sven. inpolyhedron - are points inside a triangulated volume?, 2012. URL `www.mathworks.com/matlabcentral/fileexchange/37856-inpolyhedron`. [Published 20 Aug 2012 (Updated 27 Feb 2014); accessed 18 December 2014].