Semantic annotation of urban scenes: Skyline and window detection

MSc Thesis

written by

Tjerk Kostelijk
mailtjerk@gmail.com
(born June 14th, 1983 in Alkmaar, the Netherlands)

under supervision of **Isaac Esteban** and **Prof. dr ir Frans C. A. Groen**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

Master of Science in Artificial Intelligence

at the *Universiteit van Amsterdam*.

Date of the public defense:	**Members of the Thesis Committee:**
Juli 6th, 2012	Prof. dr ir Frans C. A. Groen
	dr. P.H. Rodenburg
	dr. Arnoud Visser

Universiteit van Amsterdam

Contents

Abstract

We build an automatic system that adds semantics to a urban scene . First we use images taken from different angles to extract the skyline. Then we apply a Hough transform to extract the contour of a building. Next we use projective geometry to extract a 3D model of the building. Finally two methods of window detection are presented, the first is based on cornerfeatures and the second is based on Histograms of Houghlines. The product is a 3D annotated scene where the building and windows are labeled.

1 Introduction

When we humans look at an urban scene we immediately can tell which part represents a building, a tree, a door, a window or a parked car. Even if the scene suffers from high occlusion (a tree occluding the largest part of a building) or extreme perspective distortion (a building seen from the corners of your eye) we perform this task with a very high accuracy. For a computer system however, this task is far from trivial.

Let us address the question that arises many times in Artificial Intelligence: Why are we humans so good in this task? What can we learn from ourselves and how can we apply this on a computer system?

The most important reason of our excellent visual perception is that we combine a series of depth cues (which enables us to experience depth) with top down processing (which enables us to classify objects).

One of the most important depth cue is stereopsis. We use two eyes and look at the same scene from slightly different angles. This makes it possible to triangulate the distance to an object with a high degree of accuracy.

Besides the depth cue we use top down processing to perceive objects. If we want to perceive a window, we tap into the neurons that are activated according to our (generalized) description of a window. I.e. because windows are often rectangular shaped and the color stands out we tap into the neurons that perceive straight parallel lines, orthogonal corners and intense color changes. These visual processes are extremely informative if we want to build a computer system that acts accordingly.

This thesis is about our work of an automatic system that adds semantics to a urban scene inspired on the human brain. We focus on the detection of the building contour, the 3D information of the building and the detection of windows. The product is a 3D annotated scene where the building and windows are labeled.

Before we explain our methods let us first share the variety of applications that use semantical interpretation of urban scenes.

1.1 Application examples

3d City models

Manual creation of 3d models is a time consuming and expensive procedure. Therefore semantic models are used for semi automatic 3d reconstruction/modelling. The semantic understanding is also used in 3d city models which are generated

from aerial or satellite imagery. The (doors and) windows are mapped to the detected 3D model to increase the level of detail. Some other applications can automatically extract a CAD-like model of the building surface.

Historical buildings documentation and deformation analysis

In some fields of research, historical buildings are documented. The complex structures that are contained in the facades are recorded and reconstructed. Window detection plays a key role in this. Another field of research is the analysis of building deformation in areas containing old buildings. Window detection provides information about the region of interest that could be tracked over time for an accurate deformation analysis.

Interactive 3D models

There are some virtual training applications that are designed for emergency response who require interaction with a 3d model. For the simulation to be realistic it is important to have a model that is of high visual quality and has sufficient semantic detail (i.e. contains windows). This is also the case for a fly-through visualization of a street with buildings. Other applications that require semantic 3d models are virtual tourism, visual impact analysis, driving simulation and military simulation systems.

Figure 1: Simulation environment

Augmented reality

Some mobile platforms apply augmented reality using facade and window detection to make an accurate overlay of the building. An example overlay is the same building but 200 years earlier. Semantical information is used to not only identify a respective building, but also find his exact location in the image. The accuracy and realistic level of the 3D model are vital for a successful simulation. And because the applications are mobile, very fast building understanding algorithms are required. Window detection plays an important role in these processes as the size and location of the windows supply an effective descriptor that can be used for robust and fast building identification. Furthermore it provides an accurate alignment of the overlay.

Building recognition and urban planning

Building recognition is used in the field of urban planning where the semantic 3d models are used to provide important references to the city scenes from the street level. Building recognition is done by using large image datasets where the buildings are mostly described by local information descriptors. Some approaches try to describe the 3D building with laser range data. Some methods fuse the laser data with ground images. However, those generated 3D models are a mesh structure which do not make the facade structure explicit. For a more accurate disambiguation, other types of contextual information are desired. The semantical interpretation of the facade can provide this need. In this context, window detection can be used as a strong discriminator.
We can conclude that semantic interpretation plays an important role in the interpretation of urban scenes and is applied in a wide range of domains.

1.2 Thesis outline

The outline of this thesis is as follows:
We start with explaining some basic computer vision techniques in Chapter 2. These techniques are the driving force behind the algorithms used in both skyline detection and window detection. In Chapter 3 we explain our first method of scene interpretation. We simulate the eyes by using images taken from different angles to extract the skyline in a images. In Chapter 4 this result is used to extract the contour of a building. Next we combine this result with projective geometry to extract a 3D model of the building. In Chapter 5 we present two window detection methods that are very similar to the way humans detect windows. We respectively detect the windows of an unrectified and a rectified scene. The thesis finishes with an overall conclusion.

Many methods used in this thesis are distinct, therefore we choose to present the discussion, results and future research for each method separate. Also the chapters are independent from each other. Therefore, a reader only interested in one particular topic may read only the associated chapter describing it while skipping the other chapters.

2 Preliminaries on Computer Vision

In this chapter we discuss the basic computer vision techniques that are used for the skyline detection, and window detection. Furthermore we discuss 3rd party software, the *FIT3D toolbox* [**?**] which is used in 3D building extraction and facade rectification.

2.1 Hough transform

2.1.1 Theory

A widely used method for extracting line segments is the Hough transform [**?**]. In the Hough transform, the main idea is to consider the characteristics of a straight line not as its image points $(x1, y1)$, $(x2, y2)$, but in terms of the parameters of the straight line formula $y = mx + b$. i.e., the slope parameter m and the intercept parameter b.
The Hough transform transforms the line $y = mx + b$ to a point (b, m) in parameter space. With this representation it is impossible to describe a vertical line as the slope m is infinite. Therefore it is better to use a different pair of parameters, denoted r and θ . These are the Polar Coordinates.
The parameter r represents the distance between the origin and the line and θ is the angle of the vector orthogonal to the line. Using this parameterization, the equation of the line can be written as

$$y = \left(-\frac{\cos\theta}{\sin\theta}\right)x + \left(\frac{r}{\sin\theta}\right)$$

$$r = x\cos\theta + y\sin\theta$$

This means that a straight line in (x, y) space appears as a sinusoidal curve in the Hough parameter (r, θ) space. Let's see an example, the following image is transformed into the space (r, θ).

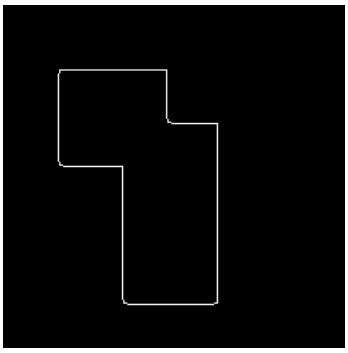
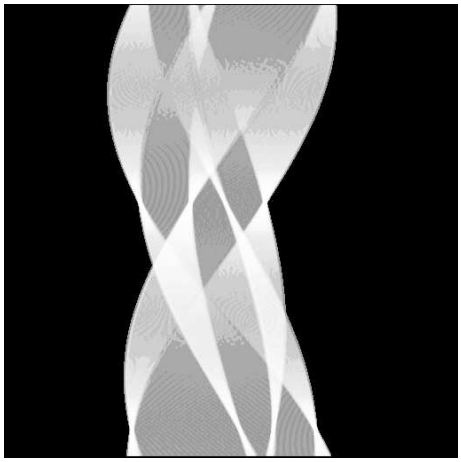


Figure 2: An input image, consisting of eight straight lines, for the Hough transform

(a) (r, θ) values



(b) (r, θ) accumulator array (quantized)

Figure 3: Hough transform

As you can see for every edge point in Figure 2 a curve is generated in (r, θ) space in Figure 3(a). On eight positions (white dots) the number of intersecting sinusoidal curves is high, these position correspond to the eight separate straight line segments in Figure 2 .
This makes the problem of detecting straight lines in finding peaks in the Hough parameter (r, θ) space.

2.1.2 Implementation

The input of a Hough transform is a binary image. In our research it is the output of the skyline detector (3). In the case of window detection (5) it is the output of an edge image.

The Hough transform develops an accumulator array of a quantized parameter space (r, θ).
It loops through the binary image and for each positive value it generates all possible lines, quantized (r, θ) pairs, that intersect with this point. For each candidate it increases a vote in the accumulator array. Lines (r, θ) that receive a large amount of votes i.e. the dots in Figure 3(a) are the found straight lines in the (x, y) space.
These positions are found by looking for local maxima in the accumulator array.

2.1.3 θ constrained Hough transform

The accumulator array consist of two dimensions r and θ. θ typically ranges from $[-90..90)$ resulting in 180 unique bins. Note that a line with $(r, \theta) = (t, j)$ can also be represent by the identical $(-t, j - 90)$, e.g. $(4, 135) == (-4, 45)$. This makes it possible to represent every line with the interval $[-90..90)$.

Sometimes we want to find lines that have a certain angle. For example the skyline of a building will appear about horizontal. If we want to detect windows we would like to detect edges in the horizontal and vertical directions. This can easily achieved by adjusting the θ range. For example if one would detect lines in the horizontal direction of a photograph of a building taken by a user, $\theta = [-10..0..10]$.Although only $\theta = 0$ presents an exact horizontal line we broaden the interval because the user hardly ever holds the camera exactly orthogonal.

2.1.4 Matlab parameters

We used a standard Matlab implementation of the Hough transform for straight lines. This implementation comes with some interesting parameters:

The *minimum length* parameter specifies the minimum length that a line must have to be valid. This is especially interesting if we want to detect a large straight skyline or if we want to discard lines that are to small to form for example a window.

Furthermore it contains the parameter *FillGap* that specifies the distance between two line segments associated with the same (r, θ) pair. When this inter line segment distance is less then the *FillGap* parameter, it merges the line segments into a single line segment. In our application this parameter is of particular interest when we want to merge lines that are interrupted by for example an occluding tree or street lamp.

2.2 Coordination systems

In this thesis three different coordinate reference systems are used. The first one is located at the image level and describes the location of the pixel in 2D. It is called the Image Coordinate Frame (ICF), in Figure 4 (x_0, y_0) span this frame. The second is the Camera Coordinate Frame (CCF) and it involves the projection of the image point (x,y) to the retina of the camera in 3D. The origin of the CCF is equal to the center of the camera. If the camera rotates, it rotates around this point. The coordinates are expressed in (XYZ).
The last system is the World Coordinate Frame (WCF) and it is used to describe the positions of the 3D model of the scene and the position of the cameras in the world in (ijk). An overview of the systems can be found in Table 1.

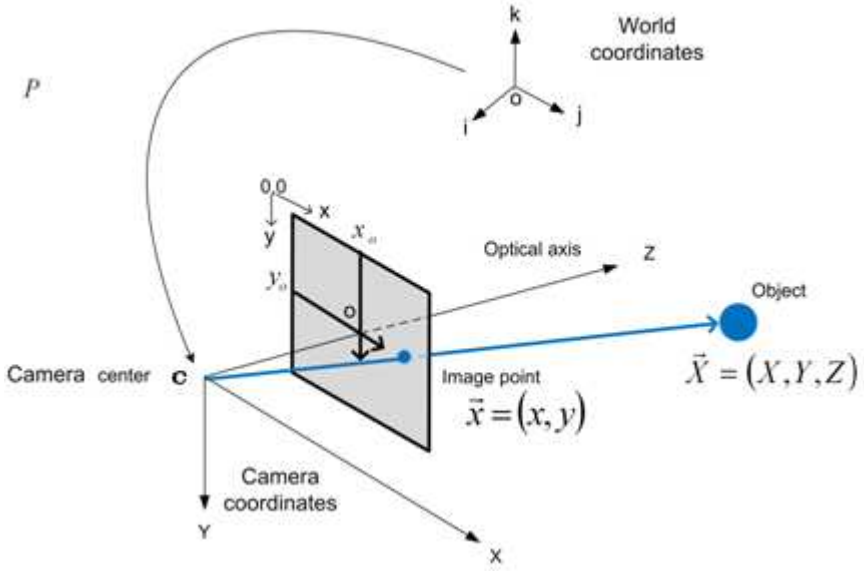


Figure 4: Coordinate systems

Table 1: Coordinate reference systems with their properties

Reference system name	abreviation	dimensions	axis
Image Coordinate Frame	ICF	2D	x,y
Camera Coordinate Frame	CCF	3D	X,Y,Z
World Coordinate Frame	WCF	3D	i,j,k

2.3 Camera calibration

Camera calibration is done to determine the relationship between what appears on the image (or camera retina) and where it is located in the 3D world. This involves calculating the camera's intrinsic and extrinsic parameters.

Intrinsic parameters

The intrinsic parameters contain information about the internal parameters of the camera. These are focal length, pixel aspect ratio, and principal point.
These parameters come together in a calibration matrix K. The Floriande dataset (originated from the Fit3D toolbox [**?**]) comes with a calibration matrix K. For the other datasets we calculated K with the Bouguet toolbox [**?**]. This method involves taking images of a chessboard in different positions and orientations. An algorithm detects the grid on the chessboard and monitors its transformations under the different images. If enough images are given (at least 10) and the images contain enough variety in chessboard pose, the Bouguet toolbox can calculate the intrinsic parameters quitte accurate. More details about this method can be found in [**?**].

Extrinsic parameters

The extrinsic parameters present the center of the locations of the camera and the camera's headings. These are unique for every view. In some systems these are recorded by measuring the cameras position and headings at the scene. In other systems this is computed afterwards (from the images). We calculated the values also afterwards and used existing software for this: *FIT3D toolbox* [**?**], details of this process is explained next.

2.4 FIT3D toolbox ??

The *FIT3D toolbox* **??** is used for several aims in this thesis. It is used in the window detection module to rectify the facades. and the skyline detection used *FIT3D* to extract a 3D model.
In order to extract this 3D model a series of frames (originating from different views) is used to estimate the relation of the camera coordinates to the world coordinates, Next, the result is used to extract a point cloud of matching features. Finally this point cloud is converted to planes which correspond with the walls of the building.
Because the toolbox plays an assisting role we explain the steps briefly. Detailed knowledge about the methods can be found in **??**.

2.4.1 Multiple views

FIT3D uses multiple views to gather information about the 3D structure of the building. The toolbox comes with a prepared dataset of 7 consecutive images (steady (zoom, lightning, etc.) parameters) of a scene. *FIT3D* doesn't

require the input images to be chronological however they need to have sufficient overlap.

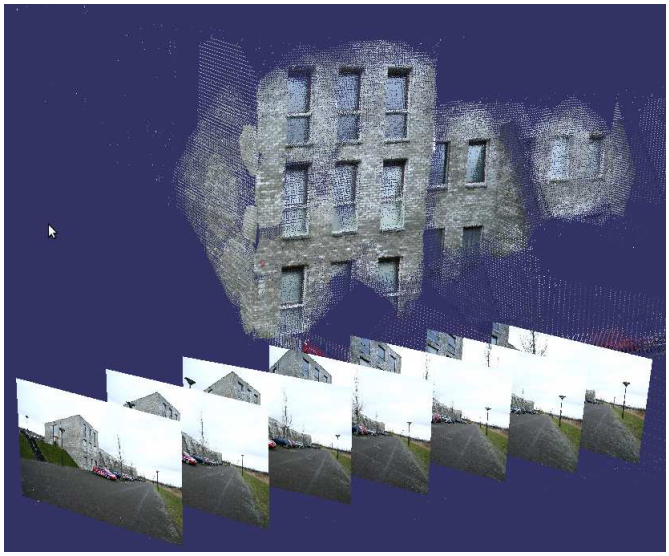


Figure 5: Example series of 7 consecutive frames, (dataset: *FIT3D toolbox*[**?**])

2.4.2 Relating the camera coordinates to the world coordinates

The different views are used to estimate the relation of the camera frame coordinates to the world coordinates, (the extrinsic parameters). In other words the different positions of the camera centers and camera's heading are estimated. This is done by calculating the relative motion between the different views.

The frame to frame motion is calculated by extracting about 25k SIFT features of each frame. Next, SIFT descriptors are used to describe and match the features within the consecutive frames. Not all features will overlap or match in the frames therefor RANSAC is used to robustly remove the outliers. After this an *8-point algorithm* together with a voting mechanism is used to extract the relative camera motion.

The frames are matched one by one which returns an estimation of the camera motion. Because this estimation is not accurate enough, a 3-frame match is done This result is more accurate but comes with a certain amount of reprojection error which is minimized using a numerical iterative method called *bundle adjustment*.

From every frame the camera motion is stored relative to the first frame. The motion is stored as a rotation and translation in homogeneous form (3x4) $[R|t]$.

This is gathered for every frame in a (7x3x4) projection matrix P. P can be used to translate camera coordinates of a specific view to 3D world coordinates and vice versa.

2.4.3 3D point cloud extraction

The next step is to use this projection matrix P to obtain a set of 3D points. These correspond to the matching SIFT features of the different views. This is achieved using a *linear triangulation* method.
The result is an accurate 3D point cloud of the building, see Figure 5. Next a RANSAC based plane fitter is used to accurately fit planes through the 3D points, see Figure 6.

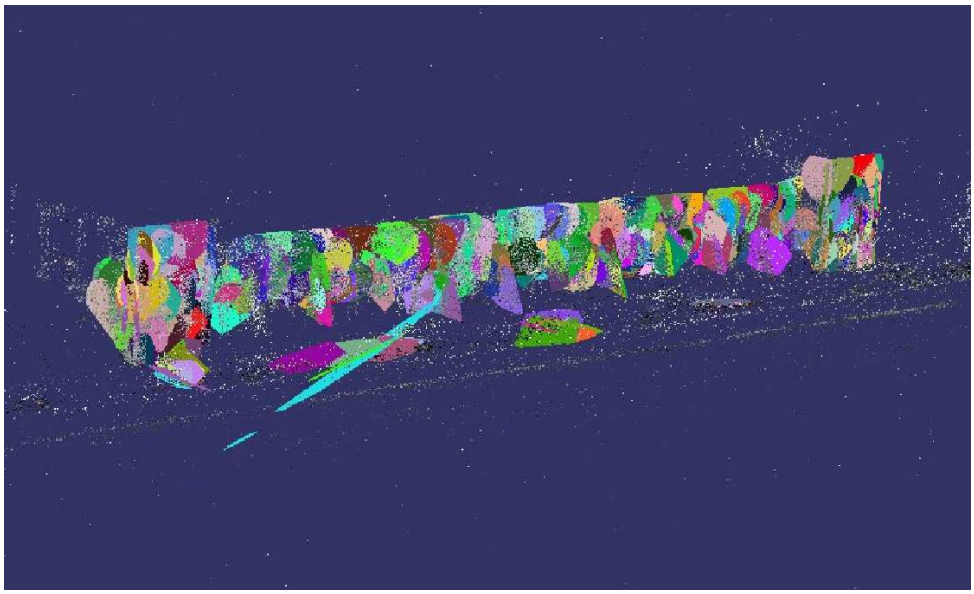


Figure 6: Fitted planes

In this thesis the plane fitting step is skipped and a new method is used to obtain the 3D model. This is explained in 4.

3 Skyline detection

3.1 Introduction

If we take a regular image on which both sky and earth are present, there is often a clear separation between them. This separation is called the skyline. The detection of this skyline has proven to be a very successful computer vision application in a wide range of domains ranging from object detection, guiding micro air vehicles, car localization, etc.
In this research skyline detection is applied on different views of a scene to estimate heights of facades of a 3D model. This is a novel application for skyline detection.

The organization of this chapter is as follows: first we give a summary of related work on skyline detection. Next we explain how we developed a new robust skyline detection algorithm. Then we present and discuss some results and, finally, conclusions are given.

3.2 Related work

Castano et al. [**?**] present a clear introduction of different skyline detection techniques.

Detection of dust devils and clouds on Mars

In [**?**], mars Exploration Rovers are used to detect clouds and dust devils on Mars. Their approach is to first identify the sky and then determine if there are clouds in the region segmented as sky. The sky is detected by an innovative algorithm that consists of three steps. First they place seeds in a sliding window whenever the homogeneity of the window is high. Then they grow this seeds in the direction of edges which are estimated using a Sobel edge detector. Finally each pixel located above the grew seeds is classified as sky.

This seed growing method looks like the a very sophisticated one, as it is accurate and autonomous. However, in our research we have a stable scene with sharp edges at the building contour so this method would be an implementation overkill.

Horizon detection for Unmanned Air Vehicles

In this domain [**?**], scientists detect the horizon to stabilize and control the flight of Unmanned Air Vehicles.
S.M. Ettinger et all [**?**] use a horizon detector that takes advantage of the high altitude of the vehicle, in that way the horizon is approximated to be a straight line. This straight line separates the image into sky and ground. They use color as a measure of appearance and generate two color distributions: one for the sky and one for the ground. They use the covariance and the eigen values of the

distributions to guide a bisection search for the best separation. The line that best separates the two distributions is determined to be the skyline.

This work is not applicable for detecting a building contour as the straight line assumption doesn't work. But it needs to be mentioned that some ideas for section 4.2.4 are inspired by this method.

Planetary Rover localization

Cozman et al. [**?**] use skyline detection in planetary rovers to estimate their location. To recover the rover's position they match image structures with a given map of the landscape (hills, roads, etc) and align both images. The matching process was first based on feature matching. In an improved version the matching process was done by searching for correspondences among dense structures in the image and on the given map, so called signal based matching. The advantage of their algorithm is the simplicity and effectiveness, this could make their algorithm suitable for this project. A big drawback is that they prefer speed over accuracy. To increase accuracy, the detector is part of an interactive system where an operator refines the skyline. For our application the skyline detector must operate without any user interaction. This brings us to our research question.

3.3 Method

Research question

Can we build a skyline algorithm that operates without any user interaction, is simple, is fast, and yet accurate enough to provide a solid skyline that can be used to estimate wall heights?

3.3.1 Situation and assumptions

As the Rover method [**?**] is simple and effective we used it as a basis and build a custom algorithm with higher accuracy on top of that.

Situation

Before we present the method, we define the situation and make some assumptions.

Definition: skyline in urban scene

A skyline in an urban scene is a set of points of the size w (where w is the width of the image) where each point describes the location of the transition from the sky to an object (e.g. a building) which is connected to the earth.

How are we going to detect this sky-building transition point?
In general, the color of the sky is very different than the color of the building. A color-based edge detector would be an intuitive decision as this produces edges on regions where intense color transitions appear. However, the sky and the building itself also contains color transitions (caused by for example clouds and windows). So how do we determine the right transition (edge)?

One of the solutions is to increase the threshold of the edge detector. In this way the detector will only return intense color transitions. Note that this will only pay off if the building-sky transition is the biggest transition in the image. Its easy to see that this is a tricky assumption as other objects may contain sharper color transitions. Furthermore it would not be robust to a change in the lightning conditions, influenced heavily by the weather.

To solve this problem we draw an assumption that is based on the idea of [**?**]. Instead of using the sharpest edge we take the most upper sharp edge and classify this edge as the skyline.

Top sharp edge assumption

The first sharp edge (seen from top to bottom) in the image represents the skyline.
Having defined the situation and assumptions we now explain our algorithm.

3.3.2 Related algorithm

As our algorithm is based on a related algorithm presented in [**?**], this is described first.
The algorithm uses three main steps first it applies a smoothing preprocessing step then it calculates the intensity gradient to find a big color transformation and finally it searches for the highest transformation.
The preprocessing step is used to increase the difference between sharp and vague edges, and to let sharp edges stand out more and vague edges disappear. The preprocessing consists of a Gaussian smoothing operator on the input image. Next the smoothed image is sliced in $\#w$ pixel columns. Each column represents the values of a discretized function. These values are transformed to their derivative, called the smoothed intensity gradient. The values of this column are high when a big change in color happens (e.g. an edge is detected) at that location on the image. Next the system walks through the values of a column, starting from the top. When it detects a value with a gradient higher then a certain threshold it stores its y-position (the height) and continues to the next column. After the position in each column of the highest sharp edge is determined the algorithm is done. The result is a set of y coordinates of length w, that represent the skyline.

3.3.3 Improved algorithm

Taking the smoothed intensity gradient is a computational cheap way to detect edges. It also has a big disadvantage because it is not robust to vague edges (they don't survive the threshold). It is not surprising that the algorithm in [**?**] was used in an interactive system where the user has to refine the result.

Our aim is to develop an autonomous skyline detector, the only user interaction that we allow is to provide the system some parameters. Furthermore the vague edges need to be detected if they are part of the skyline. We will now discuss the adaptations that we developed with respect to the related algorithm.

The column based approach of the related algorithm seems to be very useful and is therefore unchanged. To be robust to vague edges we explored and tested edge detecting types that are different then the smoothed intensity gradient based method.

The output of the different edge detection techniques was studied on an empirical basis and the Canny edge detector [**?**] was a clear winner. This is probably because Canny is a more advanced edge detector. It uses two thresholds, one to detect strong and one to detect weak edges. It includes the weak edges in the output, but only if they are connected to strong edges. In Table 2 we list Matlabs built-in edge detectors together with the method explanation.
In the section 3.4.1 one can find the results of the different edge detection methods.
We classify Canny as the most robust edge detector, and plug it into the skyline detection algorithm: the Canny edge detector outputs a binary image, therefore the column inlier threshold is set to 1, which means that it finds the first pixel that is white. This is, as in the related algorithm, done from top to bottom for every column in the image.
Because we know we are looking for sharp edges, we improved the algorithm by introducing two preprocessing steps. First the contrast of the image is increased, this makes sharp edges stand out more. Secondly the image undertakes an extra Gaussian blur, this removes a large part of the noise.
The system now has several parameters which have to be set manually by the user:

- Contrast
- Intensity (window size) of Gaussian blur
- Edge detector threshold

If the user introduces a new dataset these parameters need to be configured as the image quality and lightning condition are scene depended.

Table 2: Different edge detectors explained, Source: Matlab Documentation

Name	method
Sobel	The Sobel method finds edges using the Sobel approximation to the derivative. It returns edges at those points where the gradient of the image is maximum.
Prewitt	The Prewitt method finds edges using the Prewitt approximation to the derivative. It returns edges at those points where the gradient of the image is maximum.
Roberts	The Roberts method finds edges using the Roberts approximation to the derivative. It returns edges at those points where the gradient of the image is maximum.
zero-cross	The zero-cross method finds edges by looking for zero crossings after filtering the image with a filter that has to be specified.
Laplacian	The Laplacian of Gaussian method finds edges by looking for zero crossings after filtering the image with a Laplacian of Gaussian filter.
Canny	The Canny method finds edges by looking for local maxima of the gradient of the image. The gradient is calculated using the derivative of a Gaussian filter. The method uses two thresholds, to detect strong and weak edges, and includes the weak edges in the output only if they are connected to strong edges. This method is therefore less likely than the others to be fooled by noise, and more likely to detect true weak edges.

3.4 Results

3.4.1 Edge detection

The edge detection results of the other methods can be found in the appendix (A.1).

Figure 7: Edge detection results. Method: Canny

3.4.2 Skyline detection

Datasets

The skyline detection algorithm was applied on three datasets.

Table 3: Dataset properties

Name	Resolution	Source	Location
Floriande	1728x1152px	Fit3d [**?**]	Unknown
Bram	3072x2304px	Author	Amsterdam, 'Postjesweg'
Anne	3072x2304px	Author	Amsterdam, 'Van Spilbergenstraat'

The output of the skyline detector on the *Floriande* dataset [**?**] can be seen in Figure 8. We show the effect of different thresholds of the edge detector on the *Anne* dataset (Figure 9) and the *Bram* dataset (Figure 10).

Figure 8: The output of the skyline detector. The skyline elements are marked red.

Figure 9: The output of the skyline detector with a too high threshold (0.70)

(a) Because of the hanging streetlight, a large part of the building is not detected. Threshold=0.3

(b) The desired part of the building is detected. Threshold=0.7

Figure 10: The skyline detector on two different thresholds

3.5 Discussion

Consider Figure 8, the largest part of the building edge is detected. This is a good result, given the algorithm operates without any user interaction.

Some parts of the skyline are located on a streetlight or a tree. This is because the system assumes that the first sharp edge (seen from top to bottom) is always the skyline. Other sharp edged objects that appear above the building, for example an aircraft, will also produce outliers. This is a disadvantage of the column based approach.

As every object above the building can produce an outlier, a solution is desirable. One solution of this problem is to increase the threshold, this is shown in Figure 10. In this way even sharp edges above the building contour won't be part of the skyline. However, this comes with a risk: a too large threshold can discard the skyline itself. To show this we used the increased threshold of the *Bram* dataset and applied it to the *Anne* dataset, Figure 9.

We decided to keep the threshold low and, as this outlier removal process is the part where we can add some artificial intelligence, we developed a separate module for the outlier detection.

3.6 Conclusion

Let's answer our research question.

Research question

Can we build a skyline algorithm that operates without any user interaction, is simple, is fast, and yet accurate enough to provide a solid skyline that can be used to estimate wall heights?

Beside some scene dependent parameters like the threshold of the skyline the system works without any user interaction. No manual refinement step is needed because the algorithm is robust and accurate enough to provide a base for the next module in the system. Furthermore the algorithm is simple and has a low complexity.
It is interesting to point out that the skyline detector is a stand alone method and it can be optimized individually without any knowledge of the other modules of the project. Some ideas for future research are shared next.

3.7 Future research

3.7.1 Automatic thresholding

As the threshold is manual and scene dependent, a method for automatic thresholding is desired. There are many studies on this topic, most of the methods

are based on the statistical analysis of the image. Detailed literature research needs to be done and an implementation must be made to provides a value for the threshold. If this is done the system would be 100% automatic.

3.7.2 Advanced skyline detection

Although the outlier removal procedure is done in a separate module, it would be interesting to develop a skyline detector which is more robust to outliers. If we take a look at the related work we observe that most work is based on detecting parts that are classified as sky and parts that are classified as ground. The idea of detecting the sky and ground could be replaced by detecting the sky and the building. We could use color to discriminate the building from the sky. Different color models could be used (HSV, normalized RGB, etc.). Because the buildings exists of repeating bricks, texture would also be a good discriminator. We could apply the existing column based approach by using the highest pixel (for every column) that is classified as a building pixel.

The results could be refined by applying detailed edge detection in the region of the estimated skyline. In this way the outliers (e.g. the streetlight and the tree) could be detected, no secondary outlier removal procedure is needed and the skyline detector would be very robust.

4 Extracting the 3D building

4.1 Introduction

In the previous chapter we explained our skyline detection algorithm which extracted the skyline of a scene. The output is a set of 2D points which was collected for a sequence of images. The aim of this chapter is to use this set of points together with an aerial 2D model of the building and the 3D pointcloud obtained by the FIT3D toolbox[**?**] (2.4) to generate a 3D model of the building.

Research question

Is it possible to use a set of (noisy) skyline points together with an aerial 2D model and a 3D pointcloud obtained by *FIT3D toolbox[**?**]* to generate a 3D model of the building?

We present a stepwise solution.
First *Openstreetmap* is used to obtain a 2D topview of the building (the different parts of the 2D model represent the walls of the building). Next the 3D pointcloud obtained by the *FIT3D toolbox[**?**]* is used to align this 2D model in the scene. After this the set of points returned by the skyline detector is transfered to a set of lines. Then each line segment is assigned to a part (wall) of the aligned 2D model. After this the line segments are projected to vertical planes spaned by the 2D model. The result is used to estimate the height values of the walls of the 2D model. The 2D model is transformed according this height values to a 3D model. We will now elaborate on each step.

4.2 Method

4.2.1 Extracting the 2D model

The basis of the generated 3D model is its ground-plane: a 2D model originated from *Openstreetmap*. *Openstreetmap* (Figure 11) is a freely accessible 2D map generated by users all over the world. It contains information about streets, building contours, building functions, museums, etc. We are interested in the building contours therefor we take a snapshot of a particular area and extract this building contour. This is a set of ordered points where each point corresponds to a corner of the building. Next we link these points to line segments which represent (the topview of) the walls of the building.

4.2.2 Aligning the 2D model

We want to align the 2D model extracted from *Openstreetmap* in the scene at the right location with the right aspect ratios. From *FIT3D[**?**]* we have the 3D point cloud of the building in world coordinates (Figure 12). The first challenge is to obtain a topview of the 3D point cloud. How can we determine the direction

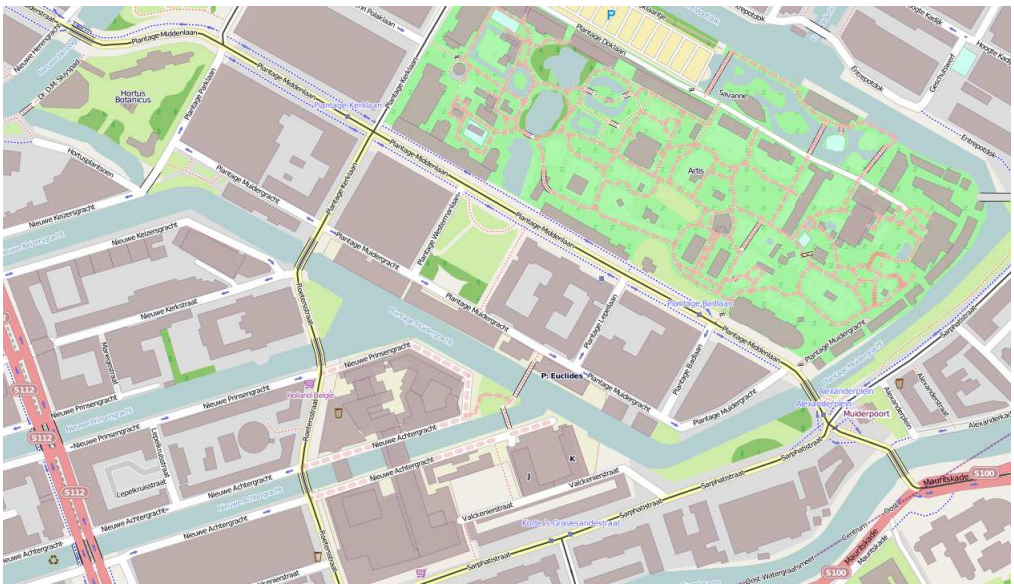


Figure 11: Openstreetmap with annotated buildings

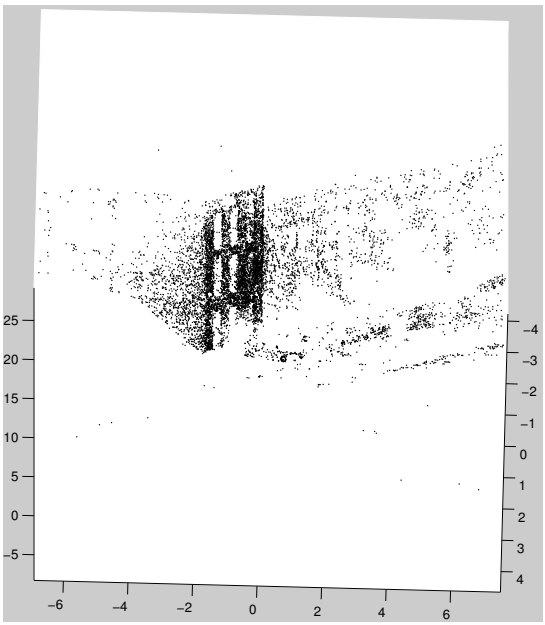


Figure 12: 3D point cloud of the walls of the building

of the top view? We want to project in the direction that is most parallel to

the walls and orthogonal to the ground-plane of the 2D model.

Gravity aligned walls assumption

The walls of the building are aligned in the opposite direction of the gravity which is orthogonal to the 2D basis from ***Openstreetmap***. This means we assume the images taken upright: the camera's $roll = 0$ (Figure 24).

Now we have defined the wall direction, we obtain the topview of the 3D point cloud by discarding the y-dimension of the points. Note that this is equivalent to a projection to the x,z plane. The result is a set of 2D points that represent the topview of the 3D point cloud (Figure 14).
We determine the walls by fitting line segments in the point cloud. We annotated these line segments manually. (If the system needs to operate automatically, RANSAC can be used to fit lines in this point cloud.)

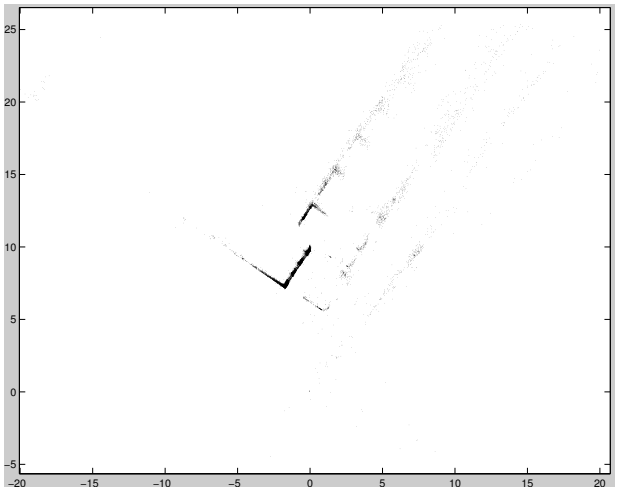


Figure 13: The projected 3D point cloud of the walls of the building

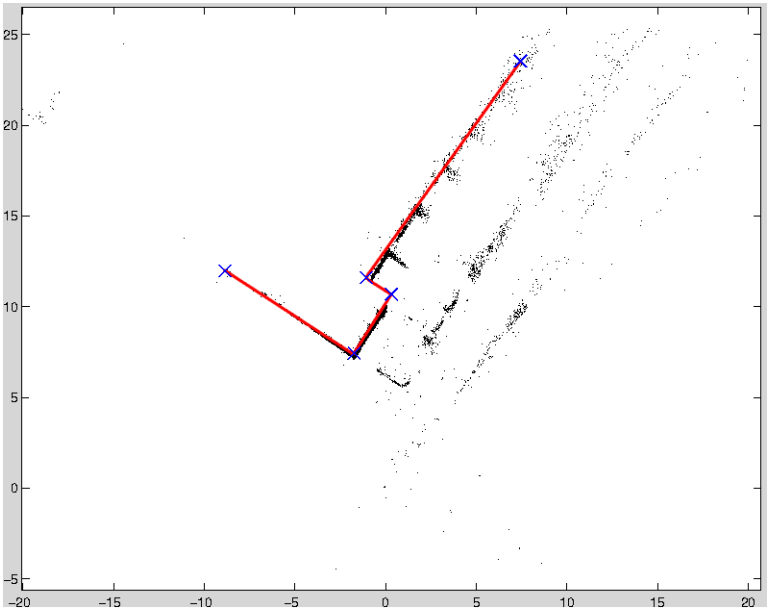


Figure 14: The top view building walls represented the fitted line segments, M1

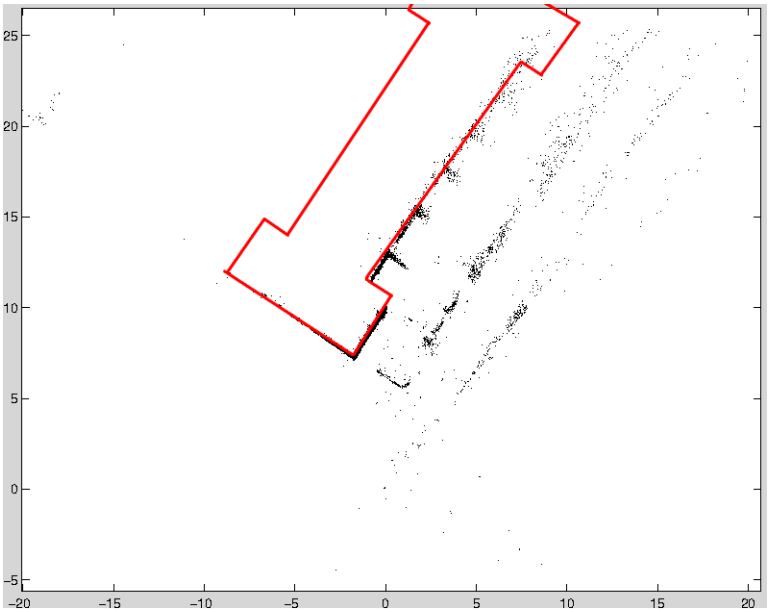


Figure 15: M2, the 2D model extracted from *Openstreetmap* aligned width the point cloud

The next step is to align the 2D *Openstreetmap* model $M1$ with these line segments of the projected pointcloud $M2$. First the endpoints of the line segments, which correspond to the wall corners, are extracted in both models. Not that the walls of $M2$ that are located at the back of the building are often missing because they are occluded by the front walls. We only consider the corners that are present in both models. Next the correspondences between these corners is annotated manually. This is used to generate a set of linear equations which are solved in closed form [**?**]. The result is a matrix A which represents the rotation, translation and scaling that is needed for a coordinate of $M1$ to be transformed to $M2$. Finally A is applied on all cornerpoints of $M1$ which results in an aligned 2D model (Figure 15).

4.2.3 Transfering the aligned 2D model to 3D

Because the 2D model is based on aerial images it contains no information regarding the height of each wall. In the next section we explain how we obtain the precise height values.

For the sake of presentation we use an average building height to generate a rough estimate of the 3D model. The 3D model is generated by taking 2D model and extend it in the opposite gravity direction. An example of the 3D model can be seen in Figure 16.

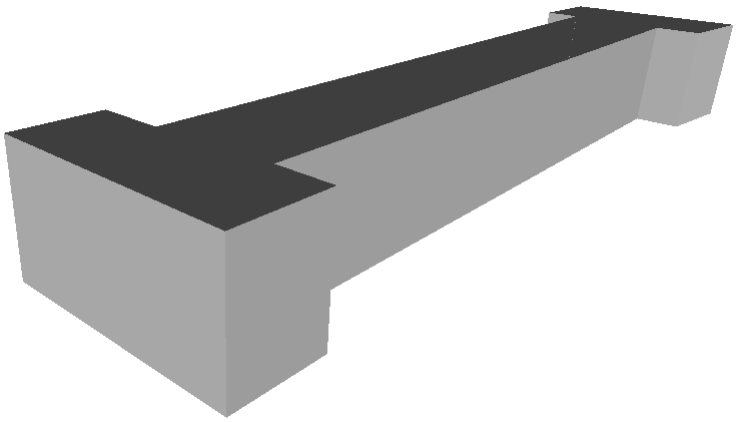


Figure 16: An example of a basic 3D model, generated by extending a basis (2D) model from *Openstreetmap* to an average building height

4.2.4 Extracting line segments

Because we want to estimate the height of the building walls of the 3D model we need to know how high the walls in the 2D images are. To estimate this we first

determine which part of the skyline is part of the building contour. Straight lines in the skyline area are likely to come from the building contour. If we have a method that extracts straight line segments from the skyline, we can use these lines segments to estimate the height of the walls of the 3D model. In this section we explain how we extracting this straight line segments.

Assumptions

Many urban areas contain buildings with a flat roof. Therefore the contour of the building is always formed by a set of straight line segments. Furthermore the contour of the building is always aligned with the upper side of a building wall. If we assume a flat roof we can find the height of the building walls without having to concern for (complex) roof types.

Flat roof assumption

We assume each building has a flat roof, implicating that the building contour is aligned with the topside of a building wall. The building walls may have different heights but the roof should be flat.

Hough transform

As was discussed in chapter 2, a widely used method for extracting line segments is the Hough transform [**?**]. We regard this as a suitable method because it is used a lot for this kind of problems. This is probably because it is unique in its low complexity (compared to other methods like *RANSAC*, who often use an iterative approach). For a detailed explanation of the Hough transform can be found in section 2.1.

The input of the Hough transform that is build in in *MATLAB* is a binary image. This is in our case the output of the skyline detector (chapter 3).
If a pixel is classified as a skyline pixel (a pixel that lies on the skyline according the skyline detector), the Hough transform increases a vote value for every valid line (r, θ) pair that crosses this particular pixel. Lines (r, θ) pairs that receive a large amount of votes contain a large amount of skyline pixels.
Because the algorithm detects straight lines containing only skyline pixels it returns only the straight parts of the skyline. As these straight skyline parts resemble the building contour we found exactly what we where looking for.

Results of the Hough transform on the output of the skyline detector are displayed and evaluated in the Result section (4.3).

4.2.5 Project the skyline to the 3D model

The Hough transform of the previous section returned a set of 2D line segments which likely present parts of the building contour. As we want to estimate the building wall heights we need to determine the wall of the building that is most likely responsible for that line segment. We can solve this problem by projecting the line segments to a specific part of the 3D model. To This is done in three steps, first the camera is calibrated, next we project the line segments to the the building and finally we determine the specific building part that is associate with a line segment.
To project the line segments to the 3D model we need to know where the camera was positioned and heading when it took the photo (extrinsic parameters). Furthermore we need to know in what way this camera transformed the image (zoom factors, lens distortion etc.) (intrinsic parameters). This process is referred to as Camera Calibration and is explained in (2.3)

From image point (2D) to possible points in scene (3D)

What can we do if we computed the camera calibration parameters? The line segment that was returned by the Hough transform consists of two endpoints v and w. These endpoints are in 2D but are recorded in a 3D scene and therefore present a 3D point in space. We don't know which point this is as for example we don't know the distance from the 3D point to the camera that took the picture. However, because we calibrated the camera (2.3) we can reduce these possible points in 3D space to a line. Next we explain how we calculate this for one 2D image point (for example a line segment endpoint).

We know:

- $\vec{x} = (x, y)$, the image point (in the camera coordinates (XYZ))
- $\vec{x}_h = (x, y, 1)'$, the homogeneous coordinate of the image point
- $\vec{c}$ and $\vec{h}$, the camera's extrinsic parameters (the camera's center and heading in world coordinates (ijk))
- K, the camera's intrinsic parameters
- P, the projection from camera coordinates (XYZ) to world coordinates (ijk). It contains implicitly the intrinsic and extrinsic parameters (the camera's center $\vec{c}$ and heading $\vec{h}$) and is applied to transfer the camera image points $\vec{x}$ (XYZ) to world coordinates (ijk)

The two coordinates that span the line of possible points in 3D space are calculated as follows:

- $\vec{c}$, the location of the center of the camera in world coordinates (ijk)

- $\vec{x}_{ijk} = PK'\vec{x}_h$, the image point that lies on the retina of the camera expressed in world coordinates (ijk)

This is illustrated in Figure 17.

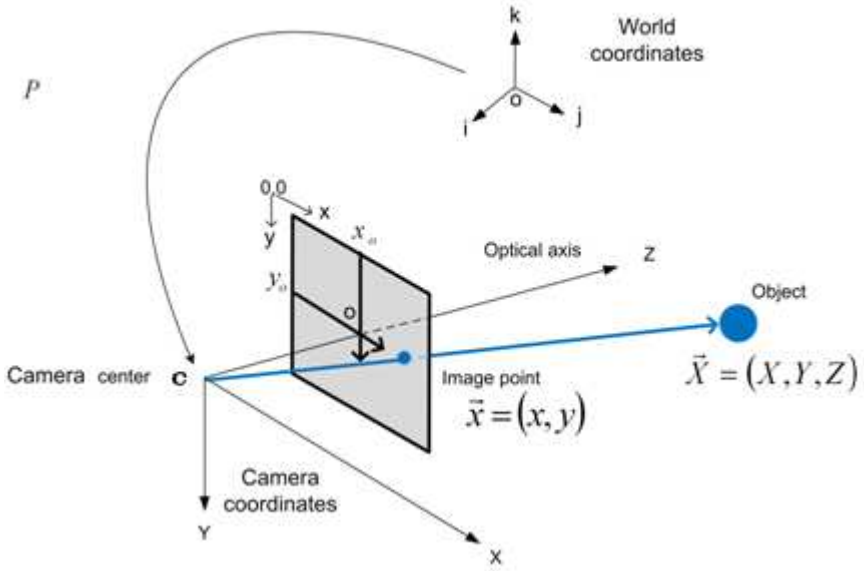


Figure 17: The blue line spanned by the camera center $\vec{c}$ and the image point $\vec{x}$ tranfered to world coordinates represent the possible 3D points in space. $\vec{X}$ is a random possible point on the line.

Using the two required coordinates we set up an equation of the line of possible points in 3D.

$$l = \vec{c} + (\vec{x}_{ijk} - \vec{c})t, t \in \mathbb{R}$$

for example if $t = 100$ then the point in the real world lies at 100 times the distance from the camera center to its retina. In Figure **??** t is about 4 (for point M).
Above calculations are done for each line segment endpoint. Now we know which possible points in 3D the skyline parts, we can use this to find the correspondence between a lines and the walls.

4.2.6 Associating line segments with building walls

Building wall appearance assumption:

We assume that every straight line segment of the skyline represent (a part of) the upper side of a specific wall of the building. Unfortunately we don't know which line is associated with which building wall. In this section we determine this association.
First we prepare the 3D model. Next we project the line segments to all planes of the 3D building by taking their endpoints and using the technique of the previous section (4.2.5) Finally we determine the most likely plane based on the largest line-wall overlap.

Preparing the 3D model

The 3D building model consists of different walls. A wall is described by two ground coordinates, a height, and a direction. The height is based on an average building height, the direction is always the y-direction (see *Gravity aligned walls assumption*).
First we transform the walls into (infinite) planes. This is done for two reasons: first this transformation is required to calculate the intersection properly. Second, because the 3D model is an estimate, the walls maybe just to small which could result in a missing intersection.

Intersect with all walls

Now we have the building walls transformed to planes we take the endpoints of the lines and project them to all the planes of the building.
Each 2D endpoint has a line of possible 3D points which we calculated in the previous section. This was the line spanned by the camera center and the image point in world coordinates. This line is intersected with all planes of the building walls. Every 2D endpoint is now associated with multiple intersections resulting in 2 x l x w points in 3D (grouped by the line segments), 2 means #endpoints of the line, l is the number of lines and w is the number of walls.

We now calculated every possible projection to (/intersection with) every plane. How do we determine which plane represents the most likely wall? We only calculated the projection to the infinite planes spanned by the walls hence we don't know which line lies on the wall and which falls outside the wall. To solve this problem let's zoom in to the situation: If we project l to the plane spanned by a wall W we get a line l_{proj_W} in $\mathbb{R}3$. If we assume l to come from the contour of wall W, then l_{proj_W} should have a large intersection with this wall W. Let's call this the line-wall overlap value, lwo. Note that the projection of l with the other walls should have a small lwo value.

Largest line-wall overlap assumption:

A line segment is associated with the wall with the largest projection overlap.
Having defined the assumptions, the situation and the idea behind the line-wall association, we can now explain the line-wall matching algorithm.

A line segment is projected to all walls and the amount of line-wall overlap, lwo is calculated. The wall with the largest overlap with the specific line segment is classified as the most likely wall for that line segment. Next the line segments are projected to their most likely wall and the algorithm outputs this set of lines in $\mathbb{R}3$.
This line-wall overlap is calculated in a different steps. First, different types of overlap are explained. Secondly the algorithm determines the *overlap type*, then the overlap amount is determined and finally the amount of overlap is normalized.

l_{proj_W} can overlap W in four different scenarios, this is explained in Figure 10. The wall W is spanned by $abcd$, and l_{proj_W} is spanned by vw.

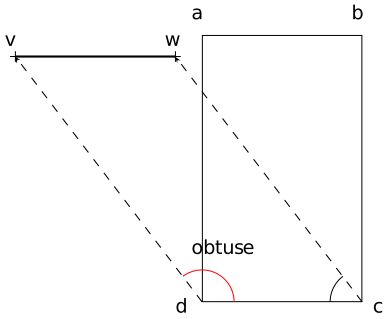


Figure 10a

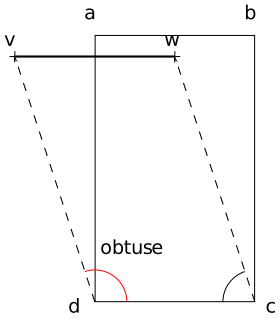


Figure 10b

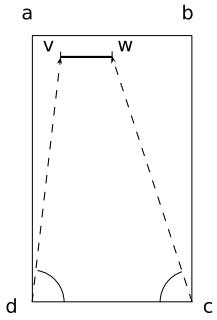


Figure 10c

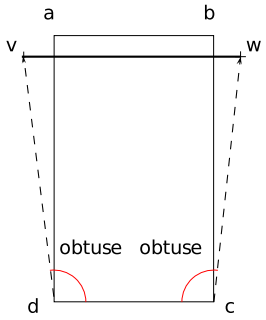


Figure 10d

The type of overlap is defined by exposing the endpoints of the line segments to an *in polygon* test, where the polygon represents a wall of the building (e.g. *abcd* in Figure 10).
Table 4 represents the types of overlap with the corresponding number of points that pass the *in polygon* test and their possible line-wall overlap value.

Table 4: Types of overlap with corresponding number of points in polygon

Type of line-wall overlap	Points in polygon	Line-wall overlap	Figure
No overlap	0	0	10a
Partial overlap	1	[0..1]	10b
Full overlap (included)	2	1	10c
Full overlap (overextended)	0	1	10d

No overlap

If the point in polygon test returns 0, the line-wall overlap calculation is skipped and 0 is returned. The remaining overlap types, partial and full, are treated individually:

Partial overlap

Let's first consider the partial overlap type (Figure 10b), the *in polygon* test returned 1, that means that one of the line segments endpoint lies inside and one lies outside the wall.
To calculate the amount of line-wall overlap, the line segment is cropped to the part that overlaps the wall and the length is measured.
The cropped line has two coordinates, first of course the point that passed the *in polygon* test and secondly the intersection of the line segment with one of the vertical wall sides (*da* or *cb* from Figure 10b).
We assume the walls to be of infinite height, therefore the partial overlapping line segment always intersects one of the vertical wall sides.
To determine which of the two vertical wall sides is crossed, we determine on which side the point that doesn't lie in the polygon (v) is on. This is done by an angle comparison.
First, two groups of two vectors are defined: dv, dc and cw, cd (see Figure 10b). We measure the angles between the vectors and call them $\angle d$, and $\angle c$. Because one of the line segment endpoints lies outside the wall $\angle d$ or $\angle c$ is obtuse, in this case $\angle d$ is obtuse. (Note that this holds because the walls are orthogonal to the basis which we assumed in the *Gravity aligned walls assumption*
To be more precise:

- If $\angle d$ is obtuse, the left vertical wall side da, is crossed.

- If $\angle c$ is obtuse, the right vertical wall side cb, is crossed.

The angles are acute or obtuse if the dot product of the vectors involved are respectively positive or negative. The advantage of this method is that it's simple and has low computational costs.

Line-wall overlap calculation
The amount of line-wall overlap is calculated by cutting of the point where l intersects the determined vertical wall side (da or cb) and measuring its remaining length.

Full or no overlap

Now let's consider the overlap types where the *in polygon* test returned 0. As you can see in Figure 10a and 10d this resulted in either full or no overlap. Again we analyze the vector angles to determine the remaining overlap-type. If only one of the angles is obtuse with no points in the polygon, like in Figure 10a, the whole line segment lies outside the wall: an overlap value of zero is returned.
Otherwise, if both angles $\angle d$ and $\angle c$ are obtuse or acute (Figure 10d), both endpoints lie on a different side of the wall, and they cross the wall somewhere in between. Full overlap is concluded here.
The amount of overlap is now calculated by measuring the length of the line segment which is cut down by his intersections with da and cb. In this case this is the same as line dc, but its easy to see that this is not the case when vw is not parallel to dc.

Line-wall overlap normalization

Finally the line-wall overlap is normalized by the line segments length:

$$\alpha_l = \frac{lwo}{|l|} \tag{1}$$

Where α_l is the normalized line-wall overlap, lwo is the unnormalized amount of line-wall overlap, and ($|l|$) is the total length of the line segment (uncut).
The intuition behind this is that line segments that are likely to present a wall not only have a large overlap but also have a small part that has no overlap, the missing overlap should have a negative effect. By calculating the relative overlap, both amounts of overlap and -missing overlap are taken into account.
The maximum of the normalized line-wall overlap is used to associate a line segment with its most likely wall. To summarize, the overlap type is defined by calculating the numbers of in polygon points and evaluating two dot products. Next the line segment is cut off depending on the overlap type and the line is

normalized.

Now we have determined the normalized line-wall overlap, we use this to search for the correct line-wall association. This is achieved by associating a line segment with the wall that has the largest line-wall overlap.

4.2.7 Improving the 3D model by wall height estimation

In the previous section we associated the line segments with their most likely wall. In this section this information is used to estimate the heights of the walls of the 3D model.
Now all line segments are associated with a certain wall, we re-project the line segment from the different views on their associated wall. The re-projection is done by intersecting both endpoints of the line segment to the plane that is spanned by the associated wall.
Next the 3D intersection points are collected and averaged, this gives us an average of the midpoints of the projected line segments. We do this for every wall separately, returning the average height of the line segments. These averages are then used as the new heights of the walls of the building. Note that this is only permitted in presence of the *flat roof assumption*.
The new individual heights are used in the 3D model by adjusting the location of the existing upper corner points of the walls. We copy the bottom left and right corner points and add the estimated height from the previous section to its y-value. The y-value is the direction of the gravity which is permitted by the *Gravity aligned walls assumption*.

4.3 Results

Figure 18: Three best ranked lines of the Hough transform on the skyline detector match the building walls

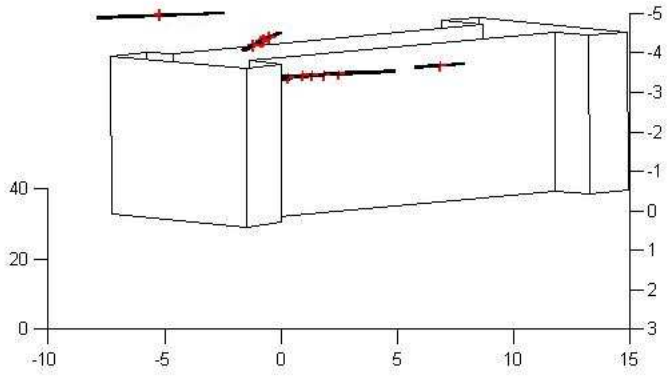


Figure 19: Houghlines projected on the most likely wall

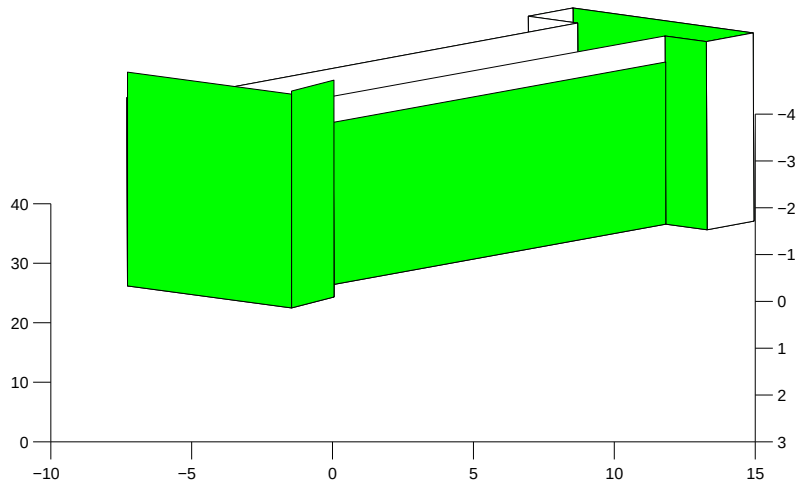


Figure 20: Improved 3D model

Figure 18 shows the top 3 longest Houghlines. The endpoints are denoted with a black and blue cross. All three line segments lie on the building contour. The left line segment covers only a part of the building wall. The middle line segment covers the full wall contour. The left and middle line segment are connected. The right line segment covers the wall until the tree occludes.

Figure 19 displays the line segments projected onto their associated walls. Note that the line segments are originated from different views.
For a clear view we have only selected the lines that were associated with three specific walls of the building of Figure 18. For each different view we draw a red cross that represents the average middle point for that view for each wall, some crosses overlap.

Figure 20 displays the updated 3D model. The corner points of the walls are adjusted according the calculated wall heights. The green plane displays the modified wall. The left and middle wall are extended whereas the right wall is shortened.

4.4 Discussion

As can be seen in Figure 18, the top three Houghlines correspond the the three most prominent building walls What also can be seen is that the left line segment doesn't cover the whole building wall. This is caused by the use of strict parameters in the Hough transform (like a small line thickness parameter). If some ascending skyline pixels fall just outside the Houghlines, a gap is created and the line segment is cut down at that point. This is however not a big problem because the lines are long enough to produce a good wall height estimate. Furthermore there are 5 other lines (originated from different views) that support the estimate for this wall.

4.5 Conclusion

Let's answer our research question.

Research question

Is it possible to use a set of (noisy) skyline points together with an aerial 2D model and a 3D pointcloud obtained by *FIT3D toolbox[?]* to generate a 3D model of the building?
Yes this is possible, we showed that a Houghline transform is a useful method to detect skyline outliers and find prominent structure in the contour of a building with a flat roof. We introduced a method to pair up line segments with their associated walls. This was used to produce new wall heights which were propagated to the 3D model. Existing and novel AI computer vision techniques were powerfully combined resulting in an significant improvement of a 3D model based on only a few calibrated 2D images.

4.6 Future research

4.6.1 Gravity aligned walls assumption

In this project we assumed the walls to align with the gravity. This means the camera must be up right: his parameter *roll* must be exactly zero when capturing the images. In practical use this is not true. We demonstrate this by plotting the 3D point cloud with the 3D model in Figure . Although this assumption let us focus on the important issues it would be nice to incorporate gravity estimation in future research. Costin Ionita wrote a part of his master thesis about gravity estimation in [?].

4.6.2 Double wall height influence

Sometimes two line segments appear on the same single wall. This means that they have a double influence on the average wall height, which is unjustified. A simple solution would be to add a normalization pre-process step, so each

view has only one wall height vote per wall. A more decent solution would be to merge the two (or more) line segments to a single line segment. Lines that are close and parallel could be merged and averaged. Lines that lie in each others direction could be merged by increasing the Hough transforms *FillGap* parameter. E.g. for the right wall of the building in Figure 19 the *FillGap* parameter needs to be at least as big as the occluding tree.

4.6.3 Complexity

In this thesis little is discussed about the computational costs. Because the computations are done efficiently (e.g. using matrix multiplications in MATLAB) and off line, the calculation are done in reasonable time. However, if we want to make the application real time, the next speedup would be useful.
To determine the best line-wall association the line segments are now projected to every wall and for every wall the amount of line-wall overlap is calculated. This is computational very expensive and looks a bit like an overkill.

It would be a significant speedup to reduce the set of walls to only the walls that contain the middle point of the line segments. To be more concrete the middle point needs to be calculated by averaging the line segments endpoints, this middle point is used in the *in polygon* test for every wall. Next the line-wall association algorithm only treats the walls that pass this test.
The downside of this method is that it makes the system less accurate because it will resulting in more false negatives. A line segment that overlaps the wall with only 1/3 could be an important candidate for the height estimation but because the speedup method it is discarded. What can be concluded is that there is a trade of in the accurateness of the height estimation and the computational costs.

4.6.4 Alternative roofs

We assumed a flat roof, this doesn't mean that our method is unusable if we discard this assumption. E.g. without adaptations the method could be used to determine the (maximum) building height which is a useful application.

If we discard the flat roof assumption the building is allowed to have any shape. In this situation it should also be possible to extract a full 3D model. We will now consider other roof types and discuss what adaptations the system should require to handle these. In Figure 21, 6 different roof shapes are displayed.

Consider the *Gable Roof*, it is a roof consisting of two planes which are not parallel with the facade of the building. This makes the problem of extracting the 3D model more complex, but not infeasible.
Because we assume that the roof images are taken from the ground, the skyline detector will always detect the top of the building. In case of a flat roof this is also the top of the building walls. In case of an alternative roof, this will be

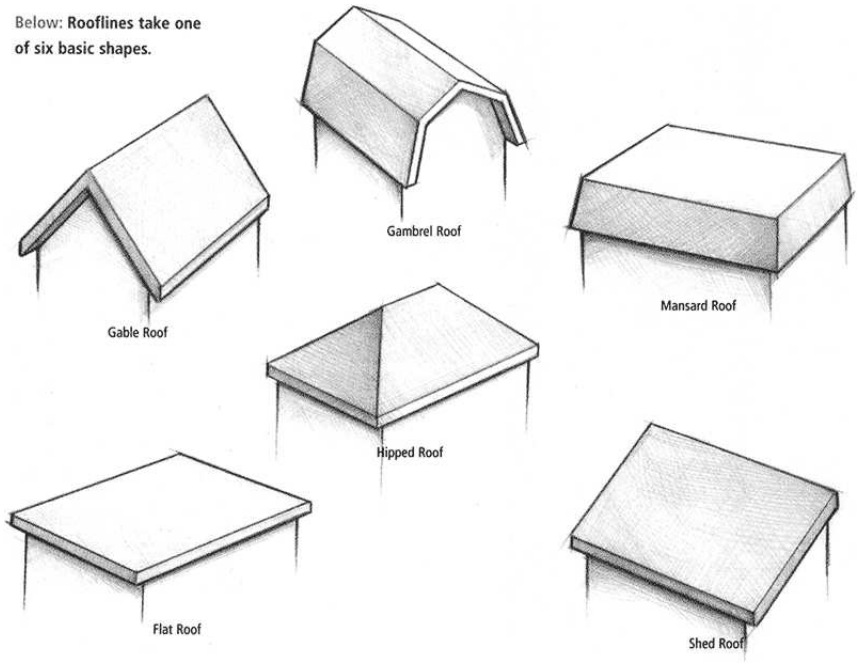


Figure 21: Different types of roofs

just the top of the building. The building walls however could lie a lot lower, therefore something else needs to be developed to find the wall heights. It would be useful to develop a method that can detect which roof type we are dealing with, what the wall heights are, and finally generate an entire 3D model.
Some ideas about this are now proposed:

- Use an object detector to detect doors, windows and dormers so the number of floors, the location of the wall-roof separation and the exclusion of some roof types (e.g. a dormer is never located on a flat roof) could be determined.

- Use the Hough transform to search for horizontal lines to detect the wall-roof separation, and use the the ground plane and the top roof line to guide the search. Some building have a gutter, because of this the number of horizontal lines on the wall-roof separation will be larger which could be of great use.

- Use geographic information (a database of roof types) with gps location to classify the roof type.

- The skyline detector detects the building height, if we could use predefined information about the ratio between the wall height and total height of the building, the wall heights could be estimated.

Assuming we determined the roof type,the building height and wall heights, the 3D model could easily be generated. For the *Gable* roof for example this will involve connecting two surfaces from the upper side of the walls with the high roof line (returned by the skyline detector). For the other roof types, the building height and wall height together with a template structure of the roof could be used to generate the 3D model.

5 Window detection

5.1 Introduction

In this chapter we deal with an important aspect of semantic urban scene interpretation: window detection. From the introduction we learned that window detection can play an important role in a variety of domains: semi automatic 3D reconstruction/modelling of city models, documenting historical buildings, analysis of old building deformation augmented reality, building recognition, etc.

This chapter is about our developed methods of window detection and it is organized as follows:
We start with related work and put our work in context. Next we describe our first window detection approach that is invariant to viewing direction. After this we present a facade rectification method. Next the rectification result is used for our second method that assumes orthogonal and aligned windows. Finally we show and discuss our results.

5.2 State of art window detection

A large amount of research is done on semantical interpretation of urban scenes. First we briefly discuss the research that is done on window detection using approaches that differ from our approach. After this, we discuss state of art window detection that has a big overlap with our approach in detail.

5.2.1 Alternative approaches on window detection

Muller et al. [**?**] detect regularity and symmetry in the building. The symmetry is detected in the vertical (floors) and horizontal (window rows) direction. They use shape grammars to divide the building wall in tiles, windows, doors etc. The results are used to derive a 3d model of high 3d visual quality. Although they achieved some interesting results their method has some disadvantages. As their method is fully based on detecting symmetry, they have to assume repeating and aligned windows: this constrains the variety of scenes the system can handle. Furthermore they match template window objects which they predefine. This constraints the variety of window types that could be matched. At last they use expensive algorithms that make it impossible for the system to run in real time.

Using a thermal camera, Sirmacek [**?**] detects heat leakage on building walls as an indicator for doors or windows. Windows are detected with L-shaped features as a set of *steerable filters*. The windows are grouped using *perceptual organization rules*: they search and group intersecting L-shapes to close a window shape. A shape is determined closed if it can separate an inside region from the outside, i.e. determine if it is a window.

(a) original texture (b) reconstruction

Figure 22: Results of Muller et al.

Ali et al. [**?**] describe the windows with *Haar* like features, the features are combined using a cascading classifier. The cascading classifier (which acts like a decision tree) is learned using the *Ada boost* algorithm. They also use window detection to determine the region of the facade.

Although this method is used a lot in computer vision, it is not the most promising approach to window detection because it has to be supervised. This means that it requires a large dataset in which every window must be accurately annotated. As the cascader uses a fixed swiping window it is sensitive to scale: all window sizes must be equal and a size range must be given to the system. Furthermore the system is always over fitted to the learning data , making it hard to generalize (detect windows that are not included in the dataset). A more general descriptor of the window that is size invariant is desirable.

To investigate this, we developed two methods on which one method does not require repeating windows nor aligned windows. One of the main targets of our research is a low requirements on the input data. First our system doesn't need a large annotated dataset. Furthermore we took the images with a mobile phone and decided to extract the windows from the image space only, this makes us independent of additional expensive data like heat or laser range images. It doesn't mean the previous work on window detection using laser or heat images isn't of good use. Instead we learned a lot from the previous research as they have to match the laser or heat data to the real image space. This matching process involves a description (semantical annotation) of the facade. Let's explain a method of that kind and discuss other approaches that are more similar to our approach.

5.2.2 Similar approaches

Pu and Vosselman [**?**] combine laser range images with ground images to reconstruct facade details. They solve inconsistency between laser and image data and improve the alignment of a 3d model with a matching algorithm. In one of the matching strategies they compare the edges of a 3d model to extracted Hough lines of both ground and laser range images. They match the lines by comparing the angle, location and length differences. These criteria are also used in our approach.

They also detect windows and use them to provide a significant better alignment of the 3d model. As windows have a high reflection, they form hole like shapes in the laser range images. These holes are directly used to extract the windows. Unfortunately due to bad laser range data, the results where far from accurate.

The work of Pu and Vosselman [**?**] provides an useful practical application of window detection. It amplifies the need for a robust window detection technique that is independent of laser range data.

Recky et al. [**?**] developed a window detector that is build on the primary work of Lee and Nevatia [**?**] (which is discussed next). In order to make clear orthogonal projections they rectify the facade. To determine the alignment of the windows the edges are projected into their orthogonal direction. For example the horizontal edges are projected in the vertical direction to establish the vertical division of the windows.
A function is developed that counts the amount of projected edges on each lo-

Figure 23: Projection profiles of Lee and Nevatias work

cation. This is the *Projection profile*, see Figure 23. A threshold is applied on the projection profile to indicate the window boundaries that are used for the

window alignment.

In the next step they use color to disambiguate the window areas from non-window areas. To be more precise, they convert the image to CIE-Lab color space and use k-means to classify the windows. Although this method is robust, both color transformation and k-means clustering are very computational expensive. Furthermore the classification based on color is sensitive to change in illumination conditions.

As in the work of Recky et al. [**?**], Lee et al. [**?**] perform orthogonal edge projection to find the window alignment. As different shapes of windows can exist in the same column/row, they only use the window alignment as a hypothesis. Then, using this hypothesis, they perform a refinement for each window independently. Although this comes with accurate results, the iterative refinement is a computational expensive procedure. As we want to run our system in real time this method is not suitable for our application.

5.2.3 Our work in context

The state of art window alignment procedure in [**?**] and [**?**] is very robust. Therefore we have decided to use this method as a basis and improved the alignment algorithm. Furthermore we have build a different window classification method. Our improvement on the alignment procedure is as follows. In the previous work [**?**] and [**?**] they use a single projection profile for each direction. We improved this process by fusing two (more advanced) projection profiles for each direction. E.g. for the determination of the horizontal division of the windows we fuse both horizontal and vertical projection profiles.

Furthermore we have build two alternative window classification procedures which are based on a higher level of shape interpretation of these projection profiles. As the classification is based on the projection profiles (edge information) we don't require expensive color transformations and we only apply (rectification) transformations on line segment endpoints. This makes our algorithm invariant to change in illumination and perform in real-time.

5.3 Method I: Connected corner approach

5.3.1 Situation and assumptions

A window consists of a complex structure involving a lot of connected horizontal and vertical lines, we use this property to detect the windows. We introduce the concept *connected corner*, this is a corner that is connected to a horizontal and vertical line. The search for these connected corners is based on edge information. The connected corners give a good indication of the position of the windows. In this approach the viewing direction is not required to be frontal.

The windows could be arbitrarily located and they don't need to be aligned to each other neither to the X and Y axis of the image. As such the windows are detected individually.

5.3.2 Edge detection and Houghline extraction

Edge detection is done using the Canny edge detector motivated earlier in section (3.3.3). From the edge images two groups of Hough lines are extracted. The groups fall in the two window directions horizontal and vertical. This is done by controlling the allowed angles, θ bin ranges, in the Hough transform. The horizontal group has a range of $\theta = [-30..0..30)$ degrees, where $\theta = 0$ presents a horizontal line. The vertical group has a range of $\theta = [80..90..100)$ degrees. We use these ranges because 1) the user hardly ever holds the camera exactly orthogonal. 2) we work with unrectified facades, meaning we deal with perspective distortion. To be more concrete, if the user takes a photo (Figure 24) with a certain Yaw $\bar{0}$, the horizontal lines become skew. The range of the vertical group is smaller then the horizontal group as the user often takes photos with a low pitch value and a high yaw.

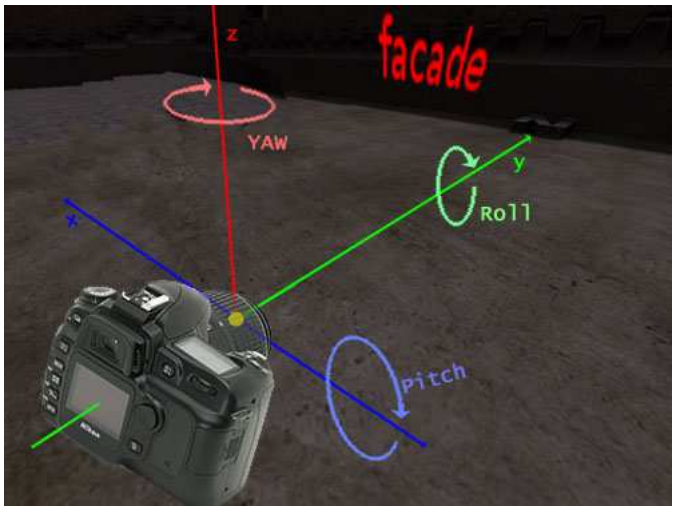


Figure 24: Pitch roll and yaw of the camera

The results of the edge detection and the Hough transform of two images can be seen in Figure 26 and 27.

5.3.3 Connected corners extraction

As windows contain complex structures the amount of horizontal and vertical Hough lines is large at these locations. A horizontal and a vertical line are often connected in a corner of a window. In this approach we pair up these horizontal and vertical lines to determine *connected corners* that indicate a window.

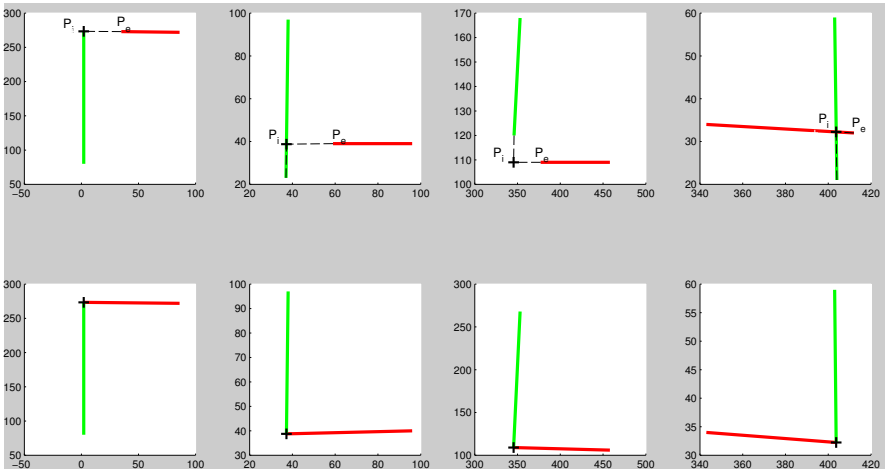


Figure 25: First row: different type of connected corner candidates. Second row: the result the clean connected corner

Often a connected corner contains a small gap or an extension which we tolerate, these cases are illustrated in Figure 25 in the top row. A horizontal gap, a vertical and horizontal gap and a vertical elongation. The cleaned up corners are given in the bottom row. When the horizontal and vertical lines intersect, the gap distance is $D = 0$. When the lines do not intersect, the distance D between the intersection point P_i and the endpoint P_e of the line is measured $D = ||P_i - P_e||$, this is illustrated as dotted lines in Figure 25. Next, D is compared to a *maximum intersection distance* threshold $midT$. And if $D <= midT$, the intersection is close enough to form a connected corner.

After two Hough lines are classified as a connected corner, they are extended or trimmed, depending on the situation. The results are shown in the second row in Figure 25. In Figure 25(I) the horizontal line is extended. Figure 25(II) shows that the vertical line is trimmed. In Figure 25(III) both lines are extended. At last, Figure 25(IV) shows how both lines are trimmed.

5.3.4 Window area extraction

To retrieve the actual windows, each connected corner is mirrored along its diagonal. The connected corner now contains four sides which form a quadrangle window area. All quadrangles are filled and displayed in Figure 29, this result is discussed in section 5.3.5.

5.3.5 Results

.

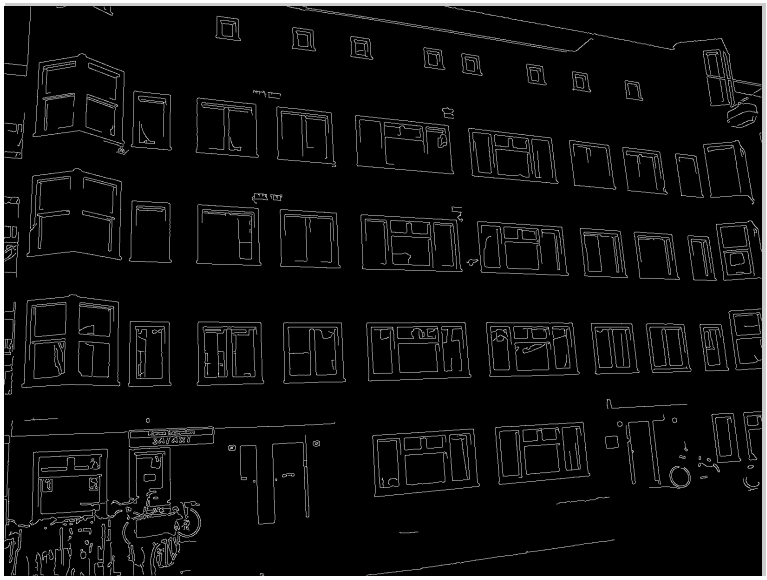


Figure 26: Edge detection

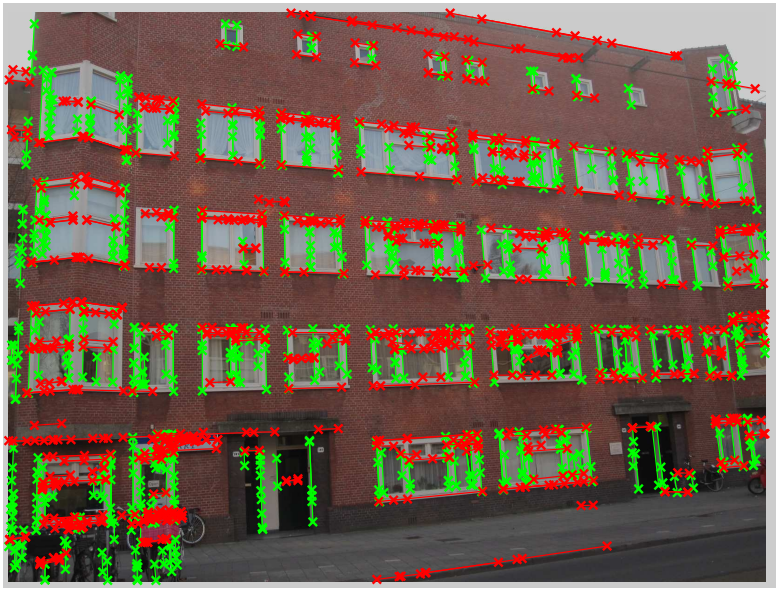


Figure 27: Result of θ constrained Hough transform

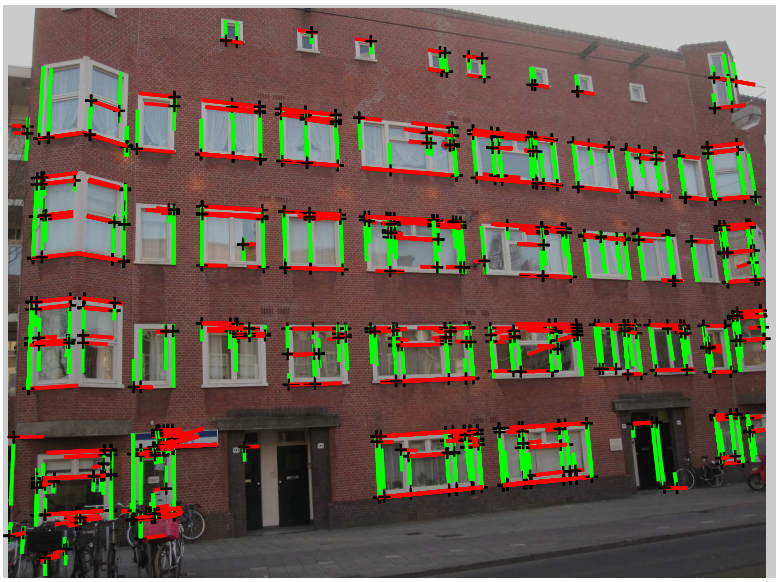


Figure 28: Found connected corners

Figure 29: Window regions

Figure 29 displays the result of the connected corner approach on an unrectified scene. It contains 110 windows of which are 109 detected, this is 99%. Furthermore there are some false positive areas, this is about 3 %. Sometimes a window is not detected, for example the window on the right top isn't detected, this is because its smaller then the minimum window width.

5.3.6 Conclusion

As can be concluded from the results, the connected corner method is suitable for, and robust to, scenes with a variation in window sizes and types. This makes the connected corner approach suitable for a wide range of window scenes where no or few prior information about the windows is known. Furthermore, because no image rectification is required, the system has low input requirements. It is independent of calibrated input data and 3d information about the building.

5.3.7 Future research

We only developed L-shaped connected corners. In future research we could connect more parts of the window. E.g. to form U shaped connected corners or even complete rectangular shapes. The latter is difficult because the edges are often incomplete due to for example occlusion.

Furthermore the next step in this study would be to group the connected corners to windows and sub windows. The big window that contains sub windows could be found by calculating the convex hull of the red areas in Figure 29. The sub windows could be found using a clustering algorithm that groups the connected corners to a window. For this method it would be useful to assume the window size as this correlates directly to the inter-cluster distance. We could also incorporate not only the center of the connected corner as a parameter of the cluster space but also the length and position of the connected corners' horizontal and vertical line parts. The inter cluster distance and the number of grouped connected corners could form a good source for the certainty of the sub window.

5.4 Facade rectification

5.4.1 Introduction

In order to apply our second method of window detection (5.6), we need the windows on the facade to be orthogonal and aligned. Therefore we rectify the facade, which can be achieved in a manual or in an automatic way.

The simple rectification method uses point to point correspondences. This requires annotation of the corner points of the facade that are mapped with the corners of a rectangle. This mapping is used to calculate a transformation matrix. The downside of this method is that it is not very accurate as it doesn't

take the width and height ratio of the facade into account. It also does not take the camera lens distortion into account. Another downside is that it requires (manual) annotation of the corner points. A more accurate and fully automatic method is desired.

The second method involves the extraction of a 3D plane of the facade. This method is more complex but gives more accurate results. It involves a comprehensive process and a large amount of research is done in this area. Given the related work, and our focus on the annotational part of facade interpretation, we used existing software to apply the window rectification. I. Estebans *FIT3D toolbox* [**?**] comes with an add-on which extracts a 3D model from a series of frames. The details of this process are explained in **??**.
Using this add-on of the *FIT3D toolbox* [**?**] we calculated the motion between a series of frames in order to extract a point cloud of matching features. This point cloud is used to extract a plane for every wall.

5.4.2 Efficient Projecting

Now we have extracted the 3D plane of the facade, the next challenge is to use this in order to rectify the facade.

It would be straight forward to rectify the full image. However this is computational very expensive as each pixel needs to be projected. To keep the computational cost to a minimum we project only the necessary data. Since we are using Hough lines we project only the coordinates of the endpoints of the found Hough lines. This is allowed because the projective transformation we apply preserves the straightness of the lines. Note that this means we apply the edge detection and Houghline extraction on the unrectified image.

If h is the number of Hough lines, the number of projections is $2h$. When we rectify the full image the number of projection is w x h, where w,h are the width and height of the image. To give an indication, for the *Anne1* dataset this means we apply 600 projections in stead of 1572864: a factor of almost 3k faster.

The Houghline endpoints are projected to the 3D plane we extracted in the same way as we explained in chapter 3. We have send rays from the camera center trough the Hough line endpoints and calculated the intersection with the 3D plane. The result is a 3D point cloud where each point is labeled to its corresponding Houghline.

The next step is to transform the facade (and therefore the Hough lines) to a frontal view. Instead of transforming the facade we rotate and translate the camera. This means the heading (z-axis) of the camera needs to be equivalent to the normal of the facade plane. We determine the rotation matrix R by calculating the angle between the camera's heading an the normal of the facade. This process involves calcuting 1) the orthogonal rotation vector $\vec{n}$ and 2) the

angle θ, see Figure 30.

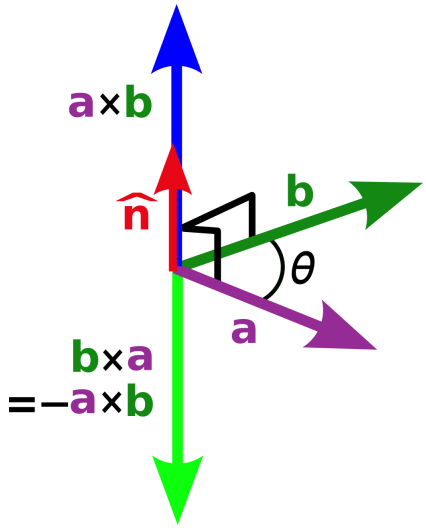


Figure 30: $\vec{a}$ is the camera's heading and $\vec{b}$ is the normal of the facade plane, $\vec{n}$ and θ are used to generate rotation matrix A [**?**]

Next, R is applied to the 3D point cloud resulting in a set of rectified 3D points that are grouped to their Hough lines.

5.4.3 Results

We show the result of two datasets, Anne1 and Dirk. Note that we also rectified the image pixels itself, this is only for the purpose of displaying the projected Hough lines in context.

Figure 31: Dataset: Anne1, Original, (unrectified) image

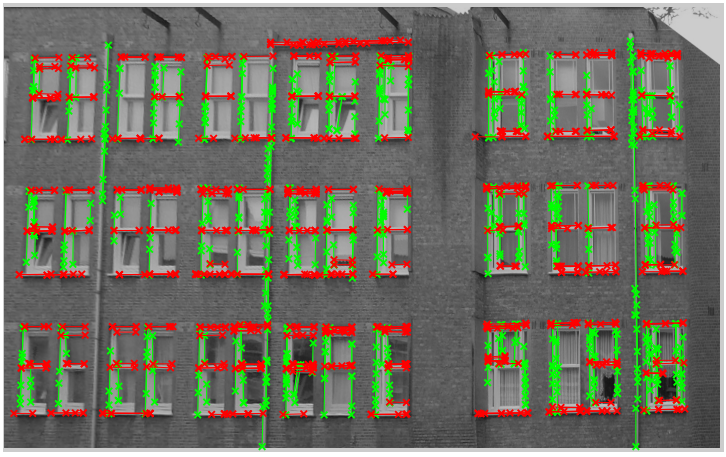


Figure 32: Dataset: Anne1, (Projected) Hough lines on rectified image

Figure 33: Dataset: Dirk, Original, (unrectified) image,

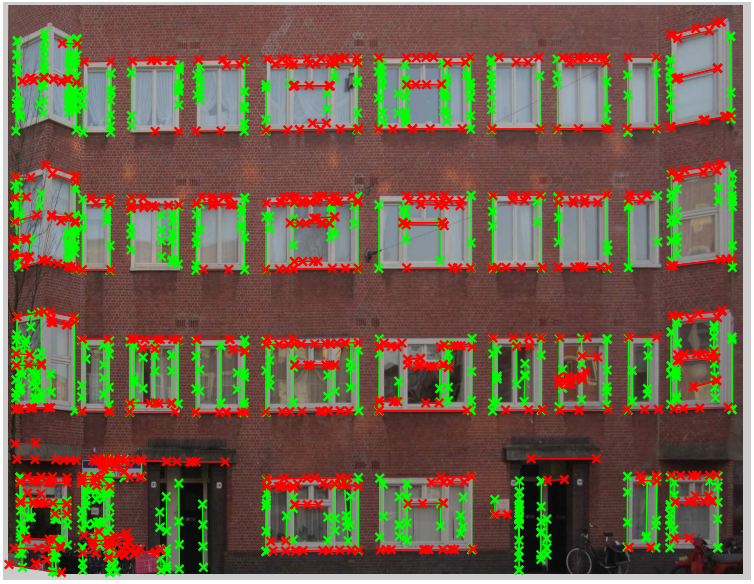


Figure 34: Dataset: Dirk, (Projected) Hough lines on rectified image

5.5 Datasets

To avoid over fitting we used multiple datasets that have a large propertie variety. Furthermore every dataset has a challenge/handicap. We used three datasets that are recorded in the suburban area 'de Baarsjes' of Amsterdam. All images have a original resolution of 3072x2304 px and are downscaled (using trilinear interpolation) to 1280x1024 px.

5.5.1 Anne1

The challenge of this dataset is that it suffers from rectification errors. This makes the window alignment a challenging task as it assumes the windows to be aligned with the x-axis and y-axis of the image. The rectification error can be seen as the skew window alignment (and skew drainpipe) on the left of the image. Furthermore the yaw value of the camera (horizontal viewing angle) was (relative to the other datasets) high, making parts of the windows occluded. The height of the windows on the right side of the image is 698 px, at the left side this is 320 px: a resolution of more then 2 times smaller making it hard to detect the left windows. Trilinear interpolation is used to minimize this loss. To reduce the number of handicaps (and to focus on the rectification and occlusion error) we cut off the bottom of the image which included cars, unaligned doors and windows.

Figure 35: Dataset: Anne1, Rectified image

5.5.2 Anne2

This dataset contains images of the same scene as Anne1. It has zero rectification error: the windows are perfect aligned, although the resolution on the left is as in the Anne1 dataset two times smaller. The challenge of this dataset are the occlusion artefacts and the bottom area of the image (cars, unaligned doors and windows).

Figure 36: Dataset: Anne2

5.5.3 Dirk

The Dirk dataset represents an everyday scene as it contains light spots on the facade, bicycles and an occluding tree. It contains zero rectification error but the windows are partially aligned. The windows are very closely located (making it hard to detect non-window areas between them), furthermore they differ in shape, size and in type. The yaw of the camera is relatively low, implicating little or no occlusion artefacts.

Figure 37: Dataset: Dirk

5.6 Method II: Histogram based approach

5.6.1 Introduction

From the previous section we saw that from a series of images, a 3D model of a building can be extracted. Furthermore we saw that using this 3D model the scene could be converted to a frontal view of a building, where a building wall appears orthogonal. This frontal view enables us to assume orthogonality and alignment of the windows. We exploit this properties to build a robust window detector as follows: First we rectify the image as described in the previous section. Then the alignment of the windows is determined. This is based on a histogram of the Hough lines. We use this alignment to divide the image in window or not window regions. Finally these regions are classified and combined which gives us the windows. We present a regular and alternative window alignment method followed by two different kind of window classifications.

Situation and assumptions

To be more precise in our assumptions, we assume the windows have orthogonal sides. Furthermore we assume that the windows are aligned. This means that a row of windows share the same height and y position. For a column of windows the width and x position has to be equal. Note that this doesn't mean that all windows have the share the same size.

5.6.2 Extracting the window alignment

Regular window alignment

We introduce the concept alignment line. We define this as a horizontal or vertical line that aligns multiple windows. In Figure 38 we show the alignment lines as two groups, horizontal (red) and vertical (green) alignment lines. The combination of both groups give a grid of blocks that we classify as window or non-window areas.

How do we determine this alignment lines? We make use of the fact that among a horizontal alignment line a lot of horizontal Hough lines are present, see Figure 32. For the vertical alignment lines the number of vertical Hough lines is high, see green lines in Figure 32.

We begin by extracting the pixel coordinates of Hough transformed line segments. We store them in two groups, horizontal and vertical.We discard the dimension that is least informative by projecting the coordinates to the axis that is orthogonal to its group. This means that for each horizontal Houghline the coordinates on the line are projected to the X axis and for each vertical Houghline the coordinates are projected to the Y axis. We have now transformed the data in two groups of 1 dimensional coordinates which represent the projected position of the Hough lines.

Next we calculate two histograms H(orizontal) and V(ertical), containing respectively w and h bins where w x h is the dimension of the image. The histograms are presented as small yellow bars in Figure 38.
The peaks are located at the positions where an increased number of Hough lines start or end. These are the interesting positions as they are highly correlated to the alignment lines of the windows.
It is easy to see that the number of peaks is far more then the desired number of alignment lines. Therefore we smooth the values using a moving average filter. The result, red lines in Figure 38 , is a smooth *projection profile* which contains the right number of peaks. The peaks are located at the average positions of the window edges. Next step is to calculate the peak areas and after this the peak positions.
Before we find the peak positions we extract the peak *areas* by thresholding the function. To make the threshold invariant to the values, we set the threshold to $0.5 \cdot$ max Peak. (This value works for most datasets but is a parameter that can be changed). Next we create a binary list of peaks P, P returns 1 for positions that are contained in a peak, i.e. are above the threshold, and 0 otherwise. We detect the peak areas by searching for the positions where P = 1 (where the function passes the threshold line). If we loop through the values of P we detect a peak-start on position s if $P(s-1), P(s) = 0, 1$ and a peak-end on e if $P(e-1), P(e) = 1, 0$. I.e. if P = 0011000011100, then two peaks are present. The first peak covers positions $(3, 4)$, the second peak covers $(9, 10, 11)$.

Having segmented the peak areas, the next step is to extract the peak positions. Each peak area has only one peak, and since we used an average smoothing filter, the shape of the peaks are often concave. Therefore we extract the peaks by locating the max of each peak area. These locations are used to draw the window alignment lines, they can be seen as dotted red lines and dotted green lines in Figure 38.

Alternative window alignment

As you can see in Figure 39 a few window alignment lines are not found and a few lines are found at wrong locations. The right side of the window frame of the first 4 columns of windows is not found. This means we have to find another way to detect the window alignment on these positions.

For the vertical alignment we only took vertical lines into account. In this method we examine the projection profile of the *horizontal* Hough lines projected on the X axis, X_h, Figure 40. On the positions of the desired vertical alignment lines there appears to be a big decrease or increase of X_h at the window frame. This is because on these positions a window containing (a large amount of horizontal lines) starts or end.
We detect these big decreases or increases by creating a new pseudo peak profile

D that takes the absolute of the derivative of X_h, Figure 40.

$$D = abs(X'_h)$$

Next we extract the locations of the peaks as the previous method.

Fusing the window alignment methods

We have presented two window alignment methods, next we fuse the methods to gain a robust window alignment. The target is to have as few as possible false positives while detecting all alignment locations.

If a window alignment position is found by both methods, often the peaks are located very close to each other, see Figure 41. If this is the case we want to fuse the results by grouping the peaks. Most of the times the peaks indicate the same window alignment but have some disparity. This is often the case when horizontal lines stop at the *inside* of a window frame while the vertical edges are located at the *outer* side of the window frame (this is supported by the fact that the disparity is often the size of the window frame part). Yet, in other cases close peaks indicate different windows that are just happen to be located closely. To apply a proper grouping of the peaks the challenge is to distinguish these two cases.

First we decrease the total number of found window alignment locations by increasing the individual thresholds (from 0.3*max peak to 0.5*max peak). Note that this has a positive side effect that the peaks that are found are more certain. After this we group the peaks as follows:
First we calculate the average of the maximum window frame part and the minimum window distance and call this the maximum peak group distance G. Next we compare all peaks and if the distance between two peaks is lower than G, we discard the peak with the least evidence (lowest peak). The result, a set of unique peaks, can be seen in Figure 47.

The advanced peak grouping is only required for the vertical alignment of the windows: The horizontal inter window distance is often big enough to not be mistaken by a window frame part.

Results

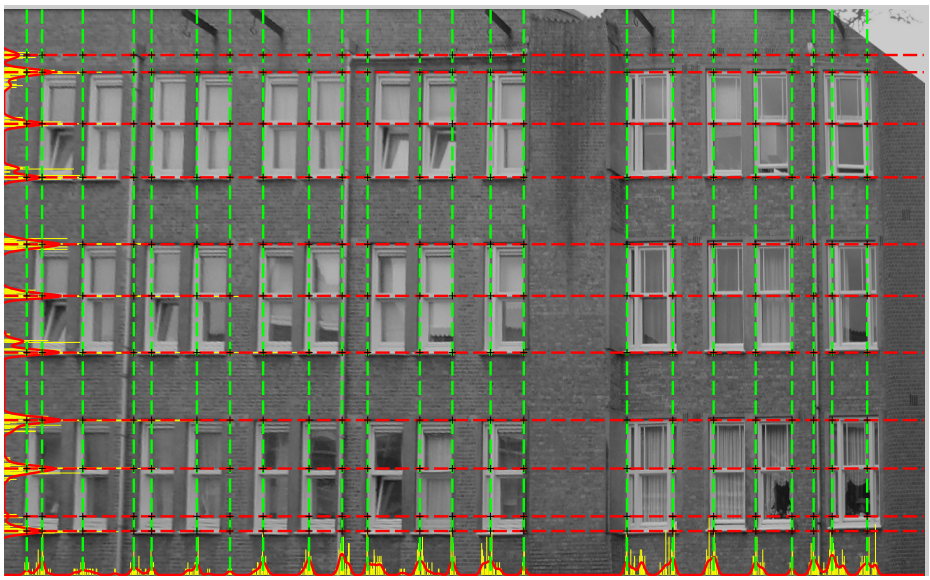


Figure 38: Dataset: Anne1, Regular window alignment (parallel projection): Based on a smoothed histogram (red line) that displays the amount of overlapping Hough lines, for the column division the horizontal Hough lines are counted (at each y position), for the row division the vertical Hough lines are counted (at each x position)

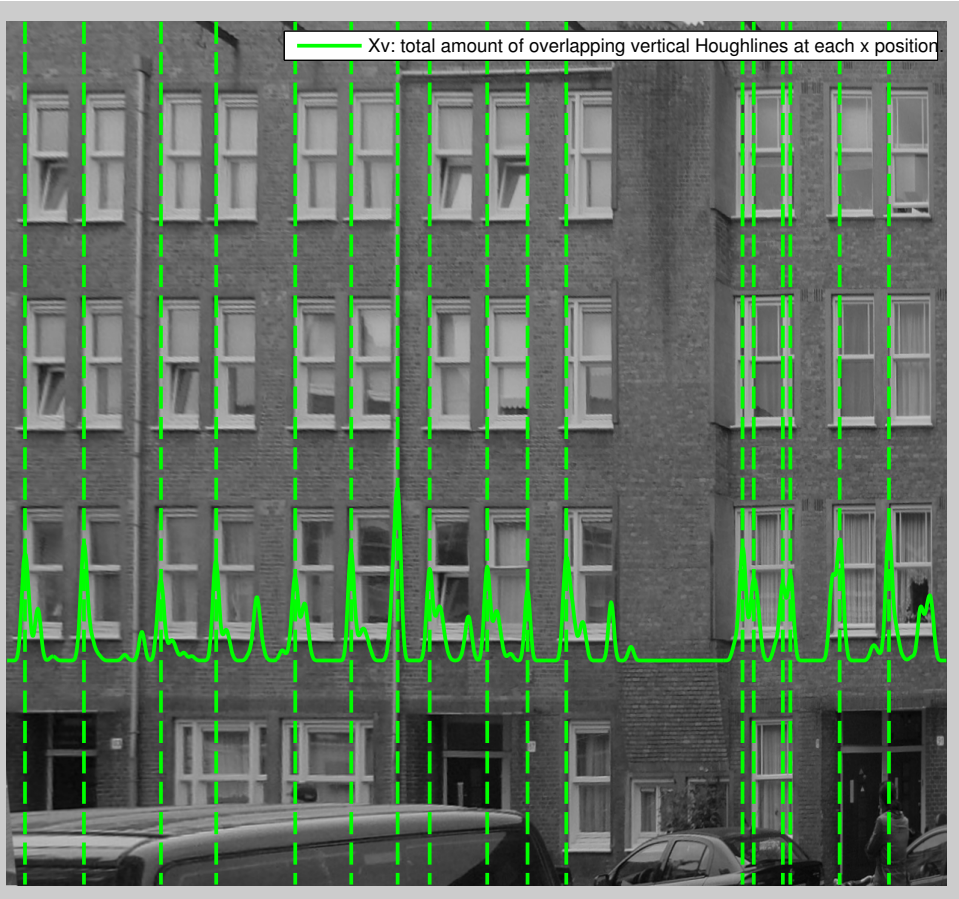


Figure 39: Dataset: Anne2, Regular window alignment: Based on a histogram that displays amount of overlapping vertical Hough lines at each x position

Figure 40: Dataset: Anne2, Alternative window alignment (orthogonal projection): Based on the shape of the smoothed histogram function. For the column division, the number of *horizontal* Hough lines is counted (Note that this is the orthogonal opposite of the regular window alignment method). Peaks (that represent a big decrease or increase of the histogram function) are used for the alignment.

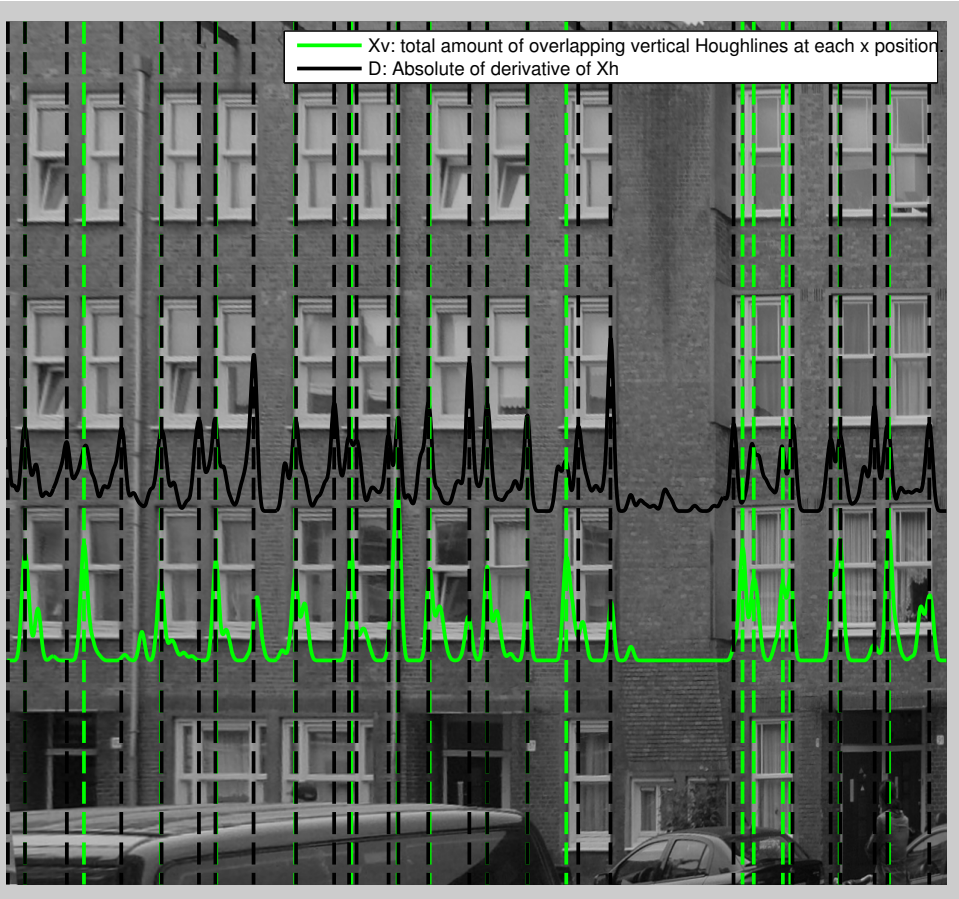


Figure 41: Dataset: Anne2, Regular+Alternative window alignment combined

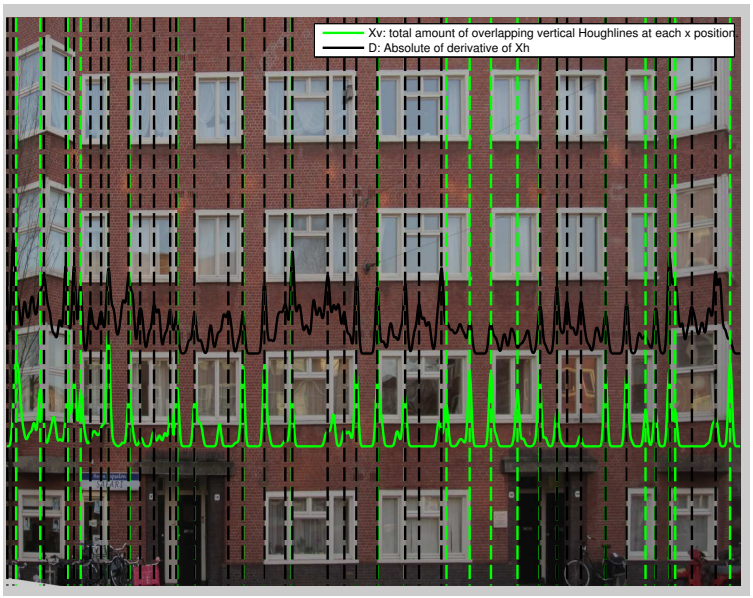


Figure 42: Dataset: Dirk, Regular+Alternative window alignment combined

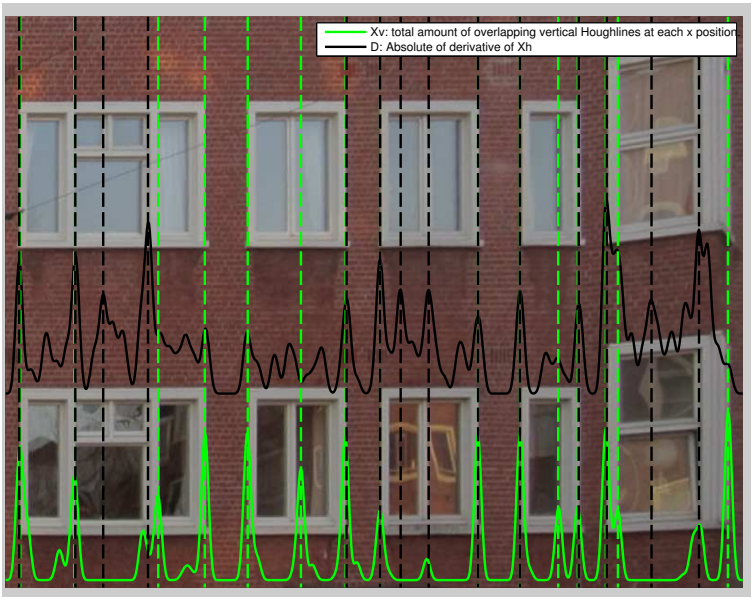


Figure 43: Dataset: Dirk, Regular+Alternative window alignment combined, zoomed

Discussion

The results are promising, as for both datasets 100 % of the alignment lines are positioned either at the boundary of a window area or inside a window area.

Discussion:Many alignment lines

In Figure 42, too many alignment lines detected and not every alignment line is placed correctly. This is mainly because the windows are partially aligned: the two front doors contain many vertical edges that are not aligned with the upper windows therefore a double alignment is found (doors and windows). Another cause of this artefact is that the bicycles on the left (see Figure 60) cause a large amount of found Hough lines which implies a (false) alignment line. Although the causes are clear, we don't have to worry about this additional window alignment lines that lie inside a window area as adjacent window areas are grouped by the classification module.
Fortunately no alignment lines are found at non-window areas.

Discussion:Fusing the methods

If we evaluate the independent window detection results on the Anne2 dataset we see (Figure 40, 39) that in both datasets neither the regular window alignment method nor the improved window alignment detect 100% of the window alignment lines. The effect of fusing the methods is high as, after fusing, 100% of the window alignment is found on both datasets: each window alignment line is detected by at least one method. This is basically explained by the fact that both methods are based on a different Houghline direction. We now discuss the interesting cases, where only one method succeeds in detecting the window alignment.

If we take a look at the regular window alignment in Figure 39 we see that for the first 4 window columns the right side of the window is not detected. This is because the original (unrectified) image (Figure 31) is not a frontal image. This makes building wall extensions (middle of Figure 31) or drainpipes occlude parts of the window. In this case the windows are countersunked into the wall making the building wall itself occluding the right window frame (Figure 39). The color of the reflection of the window is very similar to the bricks and because the window and wall are not separated by the window frame, the edge detector doesn't find a strong edge. This means that on all positions where the window frame is missing, no vertical Hough lines will be detected and (as the regular window alignment is based on the amount of vertical Hough lines) no window alignment will be found. This artefact can be studied in the edge image Figure 58: few or no edges are present, and at the low height of the peaks in Figure 39 at these positions.

However, the alternative window alignment does find a window alignment on this positions. This is because the method is based on the opposite Houghline

direction : for the vertical window alignment the horizontal Houghline direction is taken into account. This occlusion artefact has no effect on the horizontal Hough lines which makes the alternative window alignment method a strong alternative for the alignment of (partially) occluded windows.

In general, the alternative window alignment performs better then the regular window alignment method. This is because this method takes the horizontal window frame parts into account which is a priori stronger as there are more horizontal window parts then vertical window parts present. E.g. Figure 32 every window has (as it has two vertical sub windows) 3 horizontal window frame parts but only 2 vertical window parts). Furthermore the method is more robust because it relies on a higher level of histogram interpretation (by using the derivative). However, in a few cases alternative window alignment method is outperformed by the regular window alignment . For example, in Figure 43, the left side of the second window column is not detected. This is because this window type has no horizontal divisions. Only the top and bottom of the window frame produce an edge, therefore the derivative of the amount of Hough lines will return a small peak on this position (which is too small to survive the threshold). The threshold is a priori hard to survive because it has a high value as it is determined by taking a fraction of the maximum peak which is located at the windows that do have horizontal divisions (for example the most left or most right window column)).

Conclusion

We can conclude that developing histogram functions of the amount of Hough lines is a strong approach towards determination of the window alignment as the results in this work and in previous work a very promising.
We proposed two window detection methods and can conclude that the alternative approach performs better because it is based on a horizontal window division and because it uses a higher level interpretation of the histogram function.
We showed that window alignment becomes a challenging task if the windows are partially occluded. One of the main solutions we proposed is to use multiple window detection approaches. To have a 100% detection rate one need to be certain that if one approach fails at least one other approach must succeed. We proposed the strong combination of two methods where the first method filled the gab of the second method and the other way around.
We can conclude that a method that fills the gab of the occluded alignment locations must rely on Hough lines that lie in the orthogonal direction of the occlusion. In our case the occluded vertical window parts are supported by a method that detects horizontal window parts.

Conclusion:Relative thresholding

Furthermore we discussed the implication of the use of different window types. Applying a threshold that is relative to a maximum peak doesn't work well on this window types that differ (in for example horizontal divisions) as window alignment lines are missing (Figure 43). This means that if we want to stick with the relative thresholding method, we have to assume a certain equality of the window types.

Future research

Future research:Determine window alignment of different window types

A solution of the missing window alignment lines on a scene with different window types (Figure 43) would be to decrease the threshold if multiple window types are found. One could design a measure of variety of the window types. This could be done by taking the variation of the derivative of the amount of Hough lines. This amount of variation will determine the amount of decrease in threshold. Let's explain this by two examples: If there is just a few variation the maximum peak is very representative for a window so the threshold could be for example 0.7*max peak. However if there is a large variation is found (which means multiple window types are present), the threshold should be lowered to 0.3*max peak to detect the hard window types (with few horizontal divisions). Another method would be to cluster the amount of Hough lines in n+1 values where n is the number of window types (the 1 is for the non-window area). Areas that transcend from a window area cluster to a non-window area cluster (or vice versa) are determined as the window alignment locations. For both methods the challenge is again to detect the window alignment with unknown window types but keep the number of false positives (e.g. a drain pipe) zero.

Future research:Peak grouping

We fused the result of the horizontal and vertical Hough lines and discarded peaks that where close. A more accurate result would be achieved if close peaks where averaged, the height of the peak could be used as a weight. We used a manual maximum peak group distance (G), this value could also be automatically derived from the image by for example taking a percentage of the window, or by detecting the size of the window frame parts.
It is challenging to handle multiple close peaks, e.g. if 4 peaks are close, then peak 1 could indicate the same window as peak 2 but indicate a different window then peak 4. The location of the close peaks: inside, at the border, or outside the window could add important evidence, some methods of the window classification could be used to detect these values.

Future research:Window alignment refinement

To get more accurate result or to handle scenes with poor window alignment a refinement procedure could be applied. As mentioned in the related work, Lee et all [**?**] applied window refinement. Although this comes with accurate results, the iterative refinement is a computational expensive procedure. It would be nice to have a dynamic system that is aware of this accuracy and computational time trade of. A system that only refines the results when the recourses are available. For example if a car is driving and uses window detection for building recognition the refinement is disabled. But if the car is lowering speed the refinement procedure could be activated. Resulting in accurate building recognition which opens the door for augmented reality.

Both window refinement and window alignment steps could use some additional evidence which could be provided by feature based methods. For example a *multi scale Harris corner detector* could help an accurate alignment or refinement of the windows.

5.6.3 Basic window classification (based on line amount)

The image is now divided in a new grid of blocks based on these alignment. The next challenge is to classify the blocks as window and non-window areas: the window classification, we developed two different methods for this.
Instead of classifying each block independently, we classify full rows and columns of blocks as window or non-window areas. This approach results in a accurate classification as it combines a full blockrow and blockcolumn as evidence for a singular window.
The method exploits the fact that the windows are assumed to be aligned. A blockrow that contains windows will have a high amount of vertical Hough lines, Figure 32 (green). For the blockcolumns the number of horizontal Hough lines (red) is high at window areas. We use this property to classify the blockrows/blockcolumns.
For each blockrow the overlap of all vertical Hough lines are summed up. (Remark that with this method we take both the length of the Hough lines and amount of Hough lines implicitly into account.)
To prevent the effect that the size of the blockrow influences the outcome, this total value is normalized by the size of the blockrow.

$$\forall Ri \in \{1..numRows\} : R_i = \frac{HoughlinePxCount}{R_i^{width} \cdot R_i^{height}}$$

Leaving us with $||R||$ (number of blockrows) scalar values that give a rank of a blockrow begin a window area or not. This is also done for each blockcolumn (using the normalized horizontal amount of Hough lines pixels) which leaves us with C.
If we examine the distribution of R and C, we see two clusters appear: one with high values (the blockrows/blockcolumns that contain windows) and one

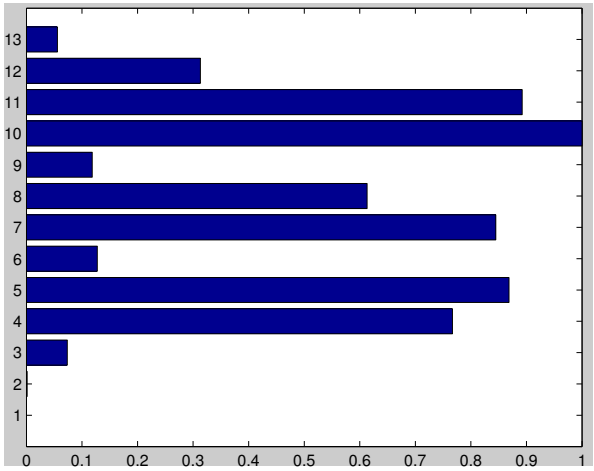


Figure 44: Classification values for window block rows representing the normalized vertical Houghline pixel count of (R)

with low values (non-window blockrows/blockcolumns). For a specific example we displayed the values of R in Figure 44. Its easy to see that the high values, blockrow 4,5,7,8,10 and 11, correspond to the six window blockrows in Figure **??**.
How do we determine which value is classified as high? A straight forward approach would be to apply a threshold, for example 0.5 would work fine. However, as the variation of the values depend on (unknown) properties like the number of windows, window types etc., the threshold maybe classify insufficient in another scene. Hence working with the threshold wouldn't be robust.
Instead we use the fact that a blockrow is either filled with windows or not, hence there should always be two clusters. We use *k-means* clustering (with $k = 2$) as the classification procedure. This results in a set of Rows and Columns that are classified as window an non-window areas.
The next step is to determine the actual windows W. A block $w \in W$ that is crossed by R_j and C_k is classified as a window iff *k-means* classified both R_j and C_k as window areas. These are displayed in Figure **??** as green rectangles.
The last step is to group a set of windows that belong to each other. This is done by grouping adjacent positively classified blocks. These are displayed as red rectangles in Figure **??**.
As the figure gives a binary representation of the windows it is not possible to see how certain a block is classification. To get insight about this we developed a measure of certainty function.

$$P(R_i) = \frac{R_i}{max(R)}$$

$$P(C_i) = \frac{C_i}{max(C)}$$

$$C(w) = \frac{P(w^{R_i}) + P(w^{C_i})}{2}$$

As you can see C is normalized, this is to ensure the value of the maximum certainty is exactly 1. The results can now be relatively interpreted, e.g. if the rectangle's $P = 0.5$ then the system nows for 50 % sure it is a window, compared to its best window ($P = 1$). And, as the normalization implies this, there is at least one window with $P = 1$.
The visualization of the measure of certainty is shown in Figure 45, the whiter the area the higher the measure of certainty.

Results

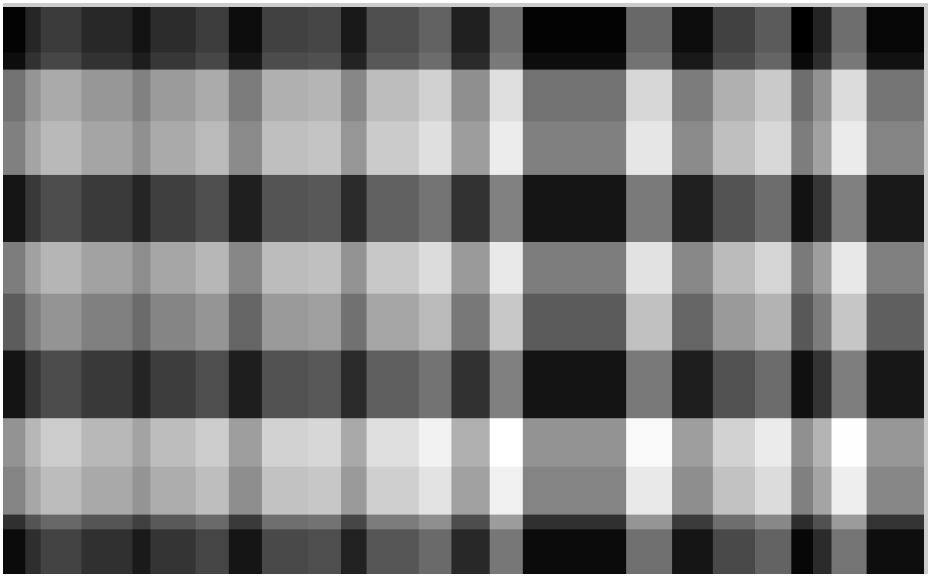


Figure 45: Dataset: Anne1, Basic window classification method:Measure of certainty, the whiter the area the higher the measure of certainty.

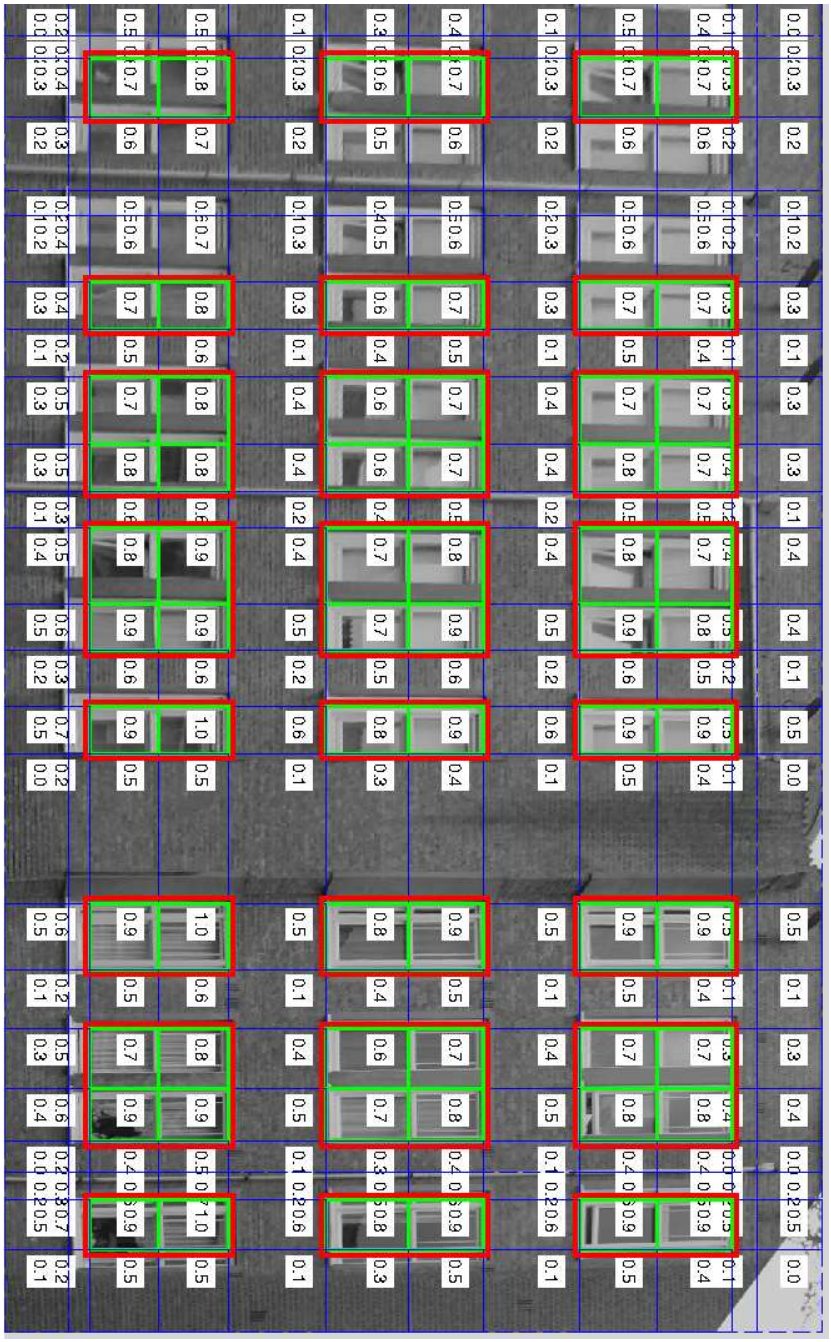


Figure 46: Dataset: Anne1, Basic window classification method: The extracted windows, red:the grouping

The bright rows and columns in Figure 45, indicate window blockrows and blockcolumns, whereas the dark rows and columns indicate non-window areas. The stripe patterns support that the classification process exist of individual row and column classification. The area of window positions is particularly white, as it intersects a bright (positively classified) row and column. One can compare this result to the windows in the original image (Figure 35).

Discussion

The outcome of this method is non-deterministic, as it depends on to the random initialization of the cluster centers. This means that our results could be correct by coincidence. To exclude this artefact, we ran the cluster algorithm 10 times on all datasets. Unfortunately for the Dirk dataset, 2 of 10 times it resulted in a bad result. This result can be found in Appendix Figures 63, 64, 65 and 66. This result can be explained as follows. The Dirk dataset has window types in the middle of that contain a horizontal subdivision while the others don't. The windows in the middle will cause k-means to drag the cluster center to a value that is too high making windows classified as non-window areas.

Future research

A solution to the bad luck on the initialization of the cluster centers is to increase the number of cluster centers to n+1, where n is the number of window types (the 1 is for the non-window area).

Conclusion

We can conclude that this certainty based classification works quite good on this dataset. However, it requires a very well alignment of the windows which is a disadvantage. As it is based on the number of found Hough lines, the errors in the window alignment will propagate to the classification. This makes this classification method inappropriate for a system where one cannot fully rely on the window alignment.
This conclusion amplifies the need for a robust classification method (that is independent, (or at least less sensitive)) to window alignment errors.

5.6.4 Improved window classification (based on shape of the histogram function)

If we take a look at Figure 47 we see that X_h (the amount of horizontal Hough lines) has two shapes that repeat: At the location where a window is present X_h is concave whereas at non-window areas X_h is convex. This is because lots of horizontal lines are found at certain window positions (the window centers) and few are found at positions that are far away from these certain positions, which are the centers of non-window areas (that lie between windows). The shape type of X_h is used as a cue for our second window classifier.

This shape type of X_h is detected as follows: As in **improved window alignment**, the first step is to examine the derivative of X_h, blue line in Figure 47. We investigate the positions where $D = X'_h$ changes from sign, these are the peaks or valleys of X_h. X_h is concave at the sign changes from positive to negative (+,-) and X_h is convex if the sign changes (-,+).

We expect one sign change per block, however it is possible that multiple sign changes occur. In this case we smooth X_h again and repeat the algorithm until for each block a maximum of one sign change is found.

Now we have detected the shape type (concave or convex), we can directly classify the blockrows and blockcolumns as window areas and non-window areas. The windows are determined as the previous classifier by combining the (positively classified) blockrows and blockcolumns.

For the sake of representation, only blockcolumns are presented. The method for the blockrows is the same: the projection profiles are projected to the Y-axis.

Results

Results:Dirk dataset

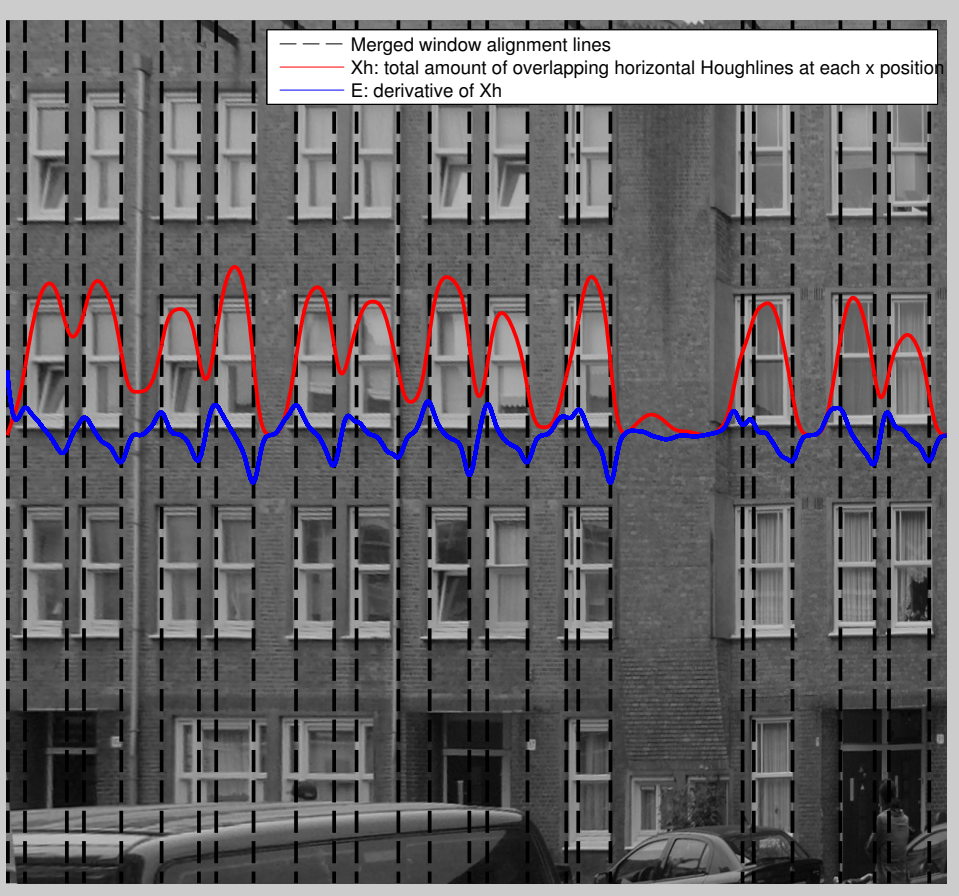


Figure 47: Dataset: Anne2, Improved window classification method: The red line shows concave shapes at window locations and convex shapes at non-window locations

Although the scene contains a lot occlusion artefacts and suffers resolution loss, especially on the left side of the image, the results for the Anne2 dataset, a 100% detection and a 100% true positive rate are very well. This is mainly due the high interpretation level of the Histogram function. We used the fact that the histogram function has a very consistent pattern, at every window area its shape is convex and every non-window its shape is concave.

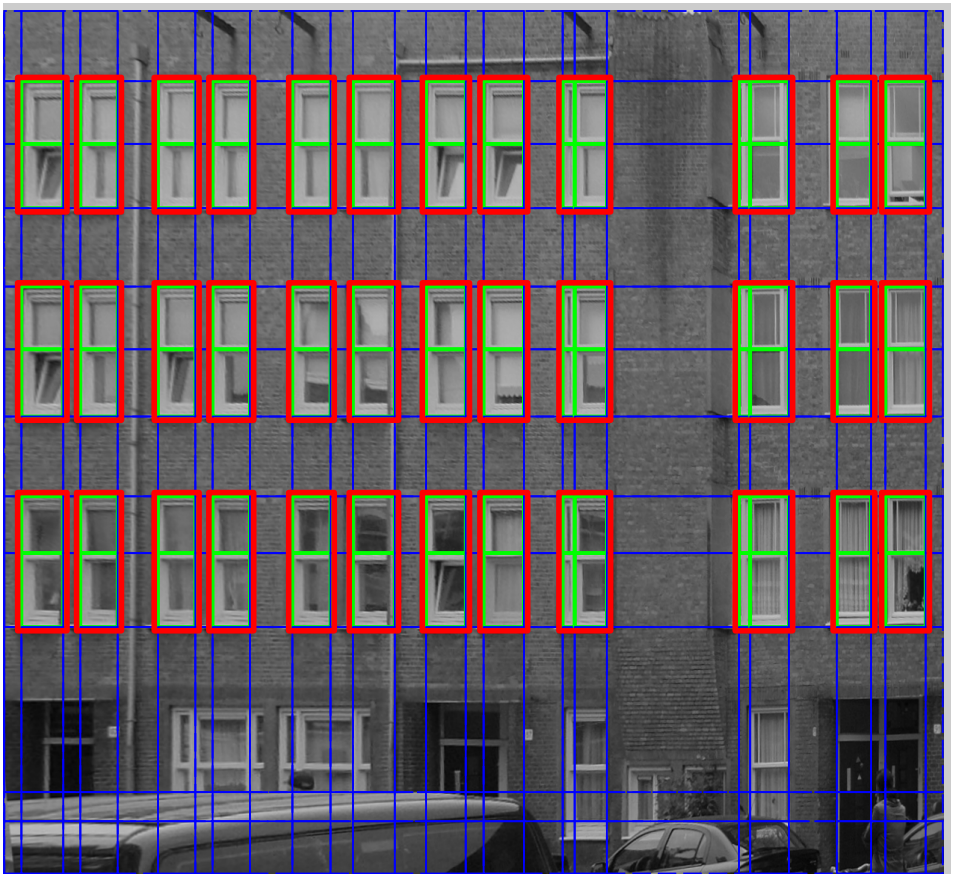


Figure 48: Dataset: Anne2, Improved window classification method: The extracted windows, red:the grouping

Table 5: Improved window classification results on Anne2 and Dirk dataset

Type	Anne2 dataset (Fig 48)	Dirk dataset (Fig 51)
Detection rate	100 %	100 %
True positive rate	100 %	87 %
False positive rate	0 %	11 %
False negative rate	0 %	9 %

Results:Anne2 dataset

Figure 49: Dataset: Dirk, Rectified image of a realistic scene which is realistic as it contains light spots, bicycles. Note that the windows are partially aligned and the differ in size and type

Figure 50: Dataset: Dirk, Improved window classification method: The red line shows concave shapes at window locations and convex shapes at non-window locations

Table 6: Grouping results on Anne2 and Dirk dataset

Type	Anne2 dataset (Fig 48)	Dirk dataset (Fig 51)
Grouped correct	100 %	90%
Grouped incorrect	0 %	10%

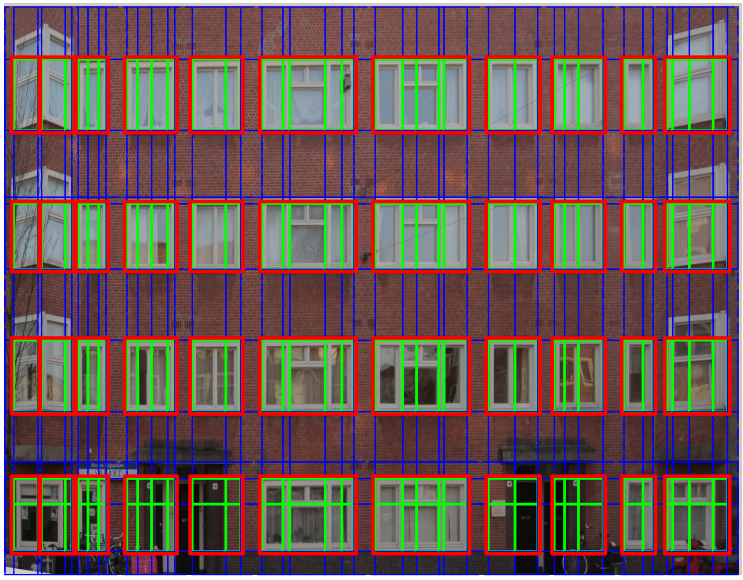


Figure 51: Dataset: Dirk, Improved window classification method: The extracted windows, red:the grouping

We discussed in the section about window alignment that the classification module would handle the redundant window alignment lines. This can be seen in Figure 51 where the classification took advantage of the fact that all alignment lines are located at a boundary of a window or in a window. The peak fusing module discarded many close alignment lines and the residual redundant alignment lines create small green adjacent sub windows which where grouped to clear red big windows.

This grouping result is promising as 90% of the grouping went well. This is a quite good result given the scene contains a large variety in window shape, window size and window type. This result is mostly due an almost perfect window alignment and classification (as the grouping only groups adjacent positively classified sub windows). The first two window rows in Figure 51 however are classified as two groups but this should be one. Normally the peak merging step would handle this problem, but with this dataset we could not use a large *maximum peak distance* value because the windows in the image (especially the first and last columns) are very close. More on this in *Future research on window classification (5.6.4)*.

Conclusion

It is a typically artificial intelligent approach, to look for a high interpretation of the data. We can conclude that the results become very robust if one takes a high level interpretation of the Histogram function.
Furthermore we can conclude that:

- Redundant lines found by the window alignment module cause no problems as long as they are located at a boundary of a window or in a window.
- The grouping module contained a small weakness, a fix is provided in the next section.
- The improved window classification outperformed the basic window classification.

Future research

Future work:Extensive evaluation methods

As the classification worked very well it would be interesting to find out on which point it will fail? This could be investigated by using low quality images (which are taken for example with a cell phone), furthermore we could downscale the images and/or add Gaussian noise to the data.

The results of the classification module depend heavily on the result of the window alignment. It would be nice feature research to make an independent evaluation of the classification modules. This could be done with a random

window alignment generator. Some evaluation should be developed and it would help if the windows are annotated.

Future work:Grouping error minimization

Although the grouping module gave promising results it can be optimized. The grouping module now groups adjacent positively classified window areas. Some window in Figure areas are false negatively classified, see the left side of Figure 51. This is caused by a small area between the windows that is classified as a non-window area. This could be solved by by adding a minimum size constraint of a area to be threaded as a non-window area. In this way small negatively classified areas cannot interrupt the adjacent windows.

Performance measure for extreme viewing angles

It would be nice to investigate the effect of the occlusion and to examine the robustness of the window detector under extreme viewing angles. For example the viewing angle could be plotted against the percentage of correct detected windows.

5.7 Conclusion

6 Conclusion

In the introduction we emphasized the ease with which humans semantically interpret a scene. We also denoted that for a computer system this task is far from trivial. Having completed this research we can conclude that this holds even more.

In this research quite some useful semantics of an urban scene was derived: a full 3D construction of the building and the extraction of his windows. Although this is very little in comparison to what humans perceive, we made a valuable start towards human-like interpretation of urban scenes.

The applications grow in the number of correct derived semantics. With just the extracted 3D model and the extracted windows we can now recognize buildings, build 3D city models, monitor building deformation and add augmented reality to scenes. Imagine what would be possible if we add more semantics like house numbers, doors, trees, cars or even (your stolen?) bicycle...

A Appendices

A.1 Edge detection results

Figure 52: Edge detection results. Method: Laplacian of Gaussian

Figure 53: Edge detection results. Method: Prewitt

Figure 54: Edge detection results. Method: Roberts

Figure 55: Edge detection results. Method: Sobel

Figure 56: Edge detection results. Method: Zerocross

Figure 57: Edge detection results. Method: Canny

A.2 Detailed window detection images

Figure 58: Dataset: Anne1, Result edge detection

Figure 59: Dataset Dirk, Edge detection result

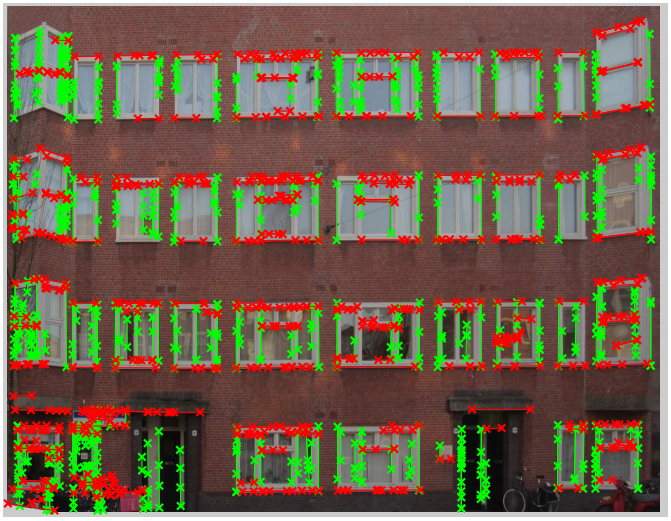


Figure 60: θ-constrained Hough transform

Figure 61: Dataset: Dirk, Regular window alignment: Based on a histogram that displays amount of overlapping vertical Houghlines at each x position

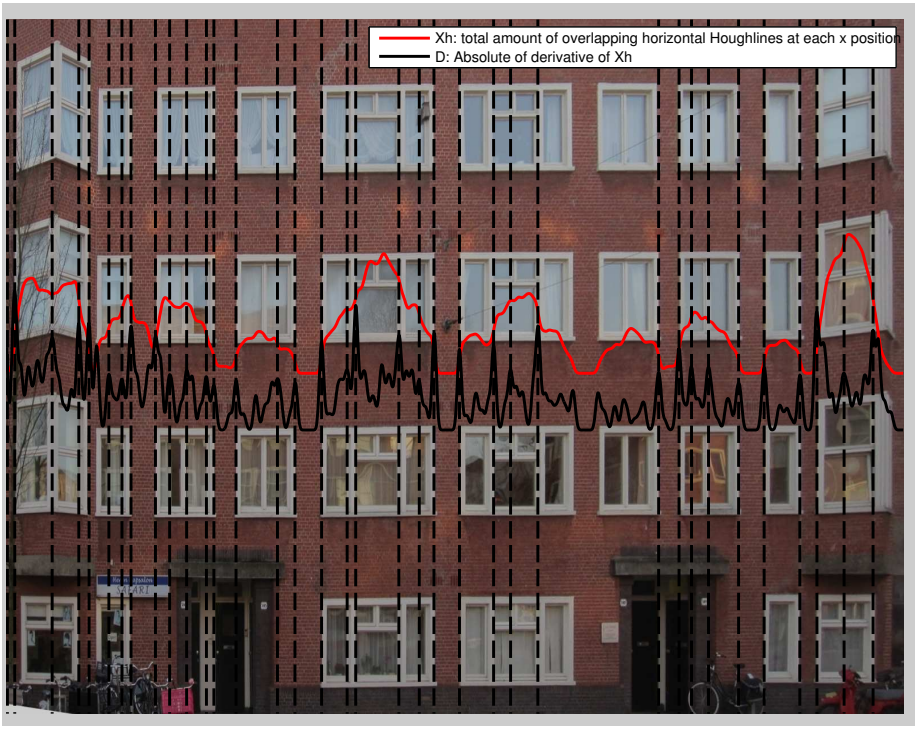


Figure 62: Dataset: Dirk, Alternative window alignment (orthogonal projection): Based on the shape of the smoothed histogram function. For the column division, the number of *horizontal* Houghlines is counted. Peaks (that represent a big decrease or increase of the histogram function) are used for the alignment.

A.2.1 K-means bad luck

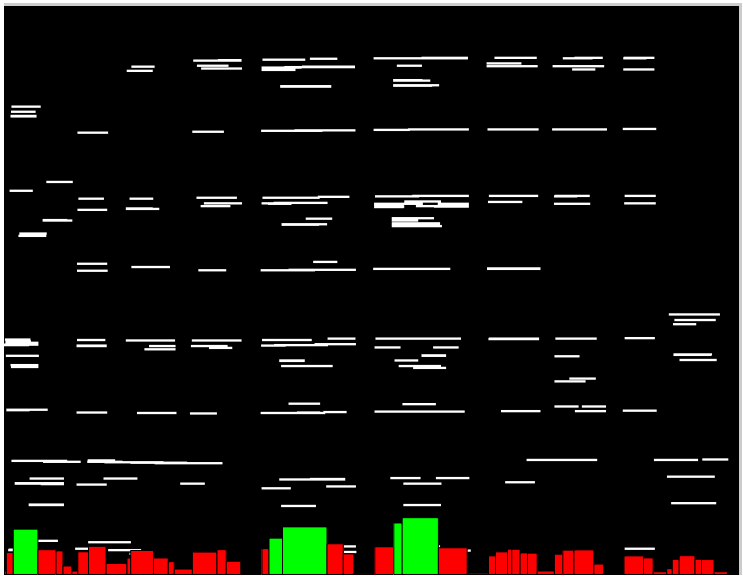


Figure 63: Dataset: Dirk, Horizontal Houghlines

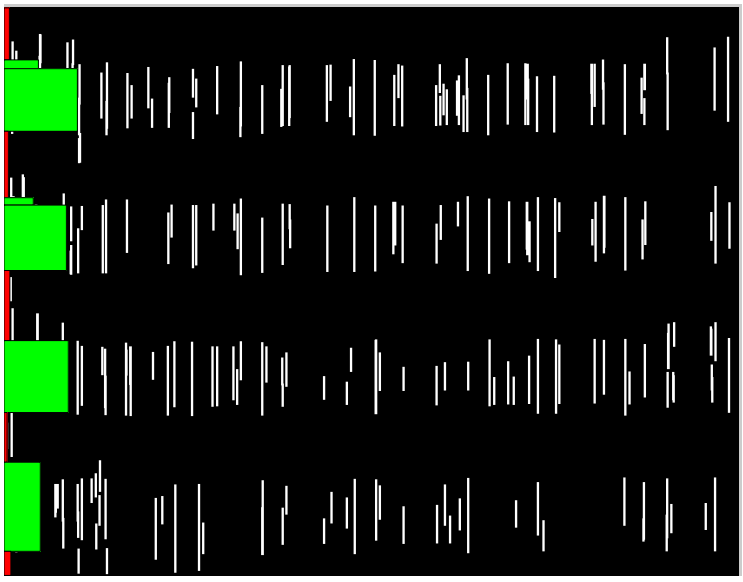


Figure 64: Dataset: Dirk, Vertical Houghlines

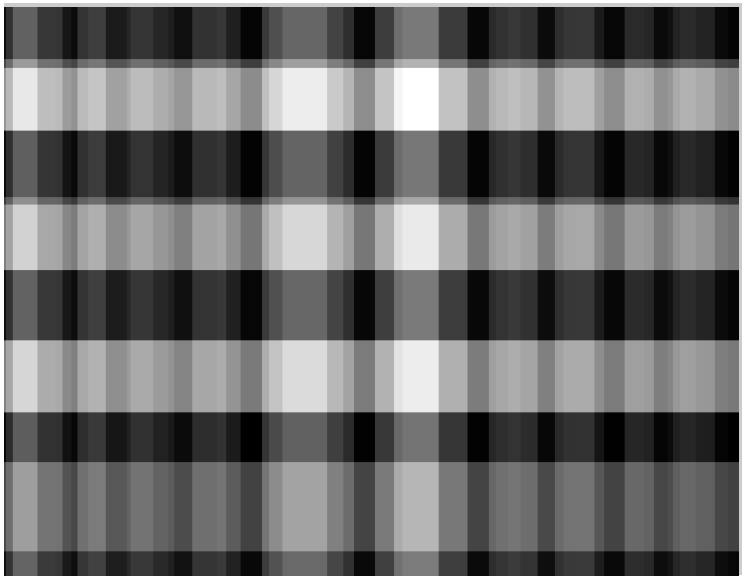


Figure 65: Dataset: Dirk, Basic window classification method:Measure of certainty, the whiter the area the higher the measure of certainty. The values of the window areas in the middle are relative large, this is due the horizontal sub windows.

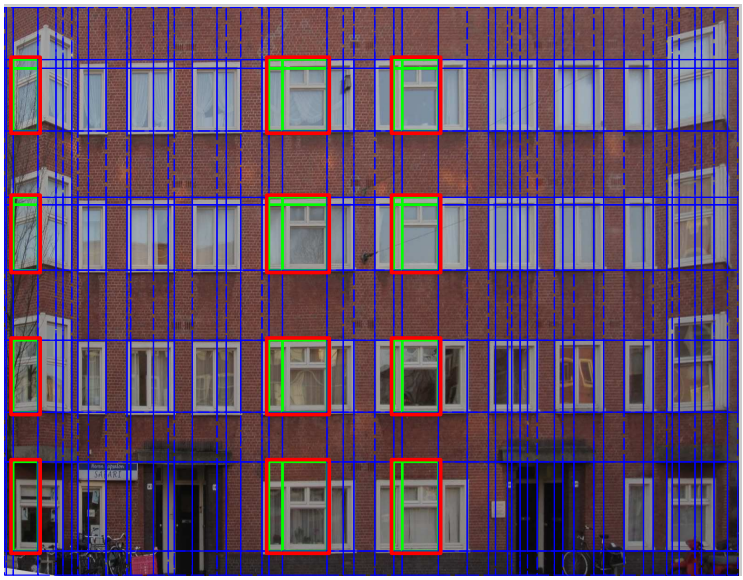


Figure 66: Dataset: Dirk, Extracted windows