Ready for detection

Stair-detecting in RGB-D images using OpenCV and an Adaboosting algorithm

F.D. Andriessen
1517619
Space Engineering

Draft Report
MSc. Thesis

Delft University of Technology

Aerospace Engineering

July 23, 2017

Ready for detection

A coming of age story

Frerik Andriessen
University supervisor: Prem Sundaramoorthy
Company supervisor: Anne van Rossum

July 23, 2017

Contents

List of Figures

1 Introduction

Space exploration knows many faces, and ranges from being physically at a location to retrieve samples, to looking at images from an orbiter, to piecing small chunks of information together about a planetary-like body you did not know existed but that should exist based on the data you have found. Now, the most concrete form of getting data, by physically visiting extraterrestrial bodies to do research, is often not feasible. Landing a piece of equipment in such a manner that it is still able to do research after landing is extremely difficult [13]. A more feasible solution is then to use satellites with imaging capabilities to form a picture of the world under investigation.

These images need to be scanned for interesting parts. A computer vision algorithm that is able to detect certain structures is one of the ways the data can be filtered. However, gathering knowledge is not the only criteria for space exploration. The other criterion is a necessary evil, keeping to a budget. Space travel is expensive. For instance, getting a payload to Low Earth Orbit from Europe costs around 10000€per kg [14], although SpaceX is working hard on bringing this number down[15]. Because of these large costs and tight budget, development for certain technologies becomes unfeasible. A computer vision algorithm that needs to detect the one feature of interest in thousands upon thousands of images will be very complex and therefore costly to develop and maintain.

An often seen solution to this problem is the practice of spin-in [16]. In this practice, an already existing technique for terrestrial application is adapted for usage in extra-terrestrial situations. This drastically reduces the costs required to get the TIL-level to the required level. This could be an interesting route for object detection. By maturing the technology on Earth, it can afterwards be implemented on space exploration missions for a fraction of the cost. This is of course only possible if there is a business-case on Earth for the technology, to propel the development. A case for the further development of object detection will therefore be made in the next paragraph. The plan is to start of with detection of simple structures, which can gradually be expanded to increasingly more complex objects and structures.

Case for object detection The aging of the population in Western countries is a serious problem. This problem finds its roots at the Second World War. More specifically, at the end of the Second World War. Due to the euphoria and relief of the war being over, our ancestors could not contain their joy, amongst other things. This resulted in a significant spike in birthrate for several years, see Fig. 1.1. Since the birthrate returned to a more normal rate after some years, there is now a generation (called the babyboomers) that is significantly larger than other generations, and it is slowly moving through life. Since the babyboom is more than 60 years ago, these large group of babies have now turned into a large group of elderly people, see Fig. 1.2 and 1.3.

These elderly people are now in need of care, and this will only increase as in the following years their age group shifts slowly towards 70-80 years. Because of this, there is a lot of pressure on the medical sector, who have too many patients and not enough people in the work-force to take good care of every patient or give them the attention, medical and social, that they require [17].

With not enough people in the medical work-force to treat the ill, a logical idea would be to look in the direction of prevention. It has been shown that keeping up a healthy diet and exercising regularly, both mentally and physically, aid in the process of staying healthy longer [18]. Indeed for an elderly person it is not often easy to keep doing gymnastics up to their retirement age, nor is that necessary. It is shown that only small amounts of exercise done during a normal day can show significant increases for a seniors health [19]. One of those activities is taking the stairs.

Unfortunately, at some point this becomes hard to combine with their everyday activity of taking a long walk towards the local supermarket and buying their groceries. Because they can with some effort climb the stairs, but it becomes very difficult when also lugging around a big bag of groceries. They are fine climbing stairs or carrying a bag of groceries, but not both at the same time. This is these days solved with elevators, stair-lifts or even worse, delivery of groceries at home. This robs the elderly person of a good daily exercise in the first two scenarios, but in the third scenario it also robs him of the social interaction with the outside world. Something that elderly people these days are in desperate need of [20].

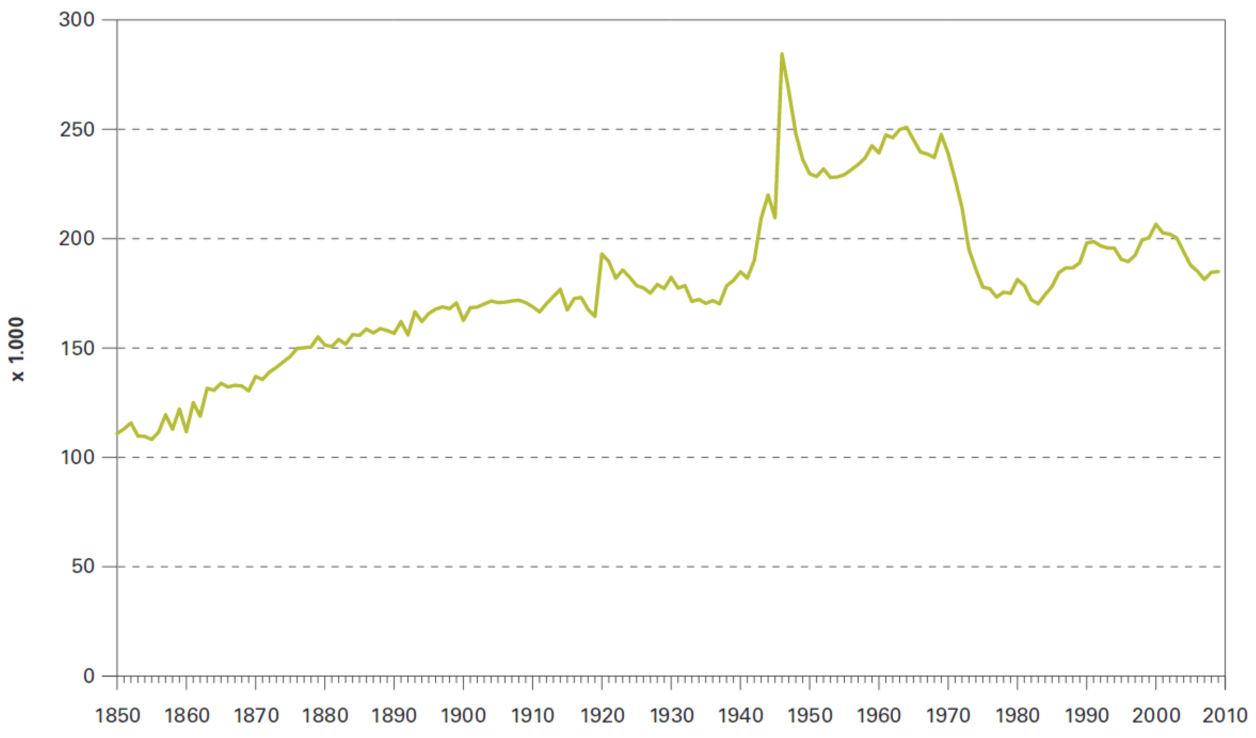


Figure 1.1: Birthrate of people in the Netherlands between 1850 and 2010 [1].

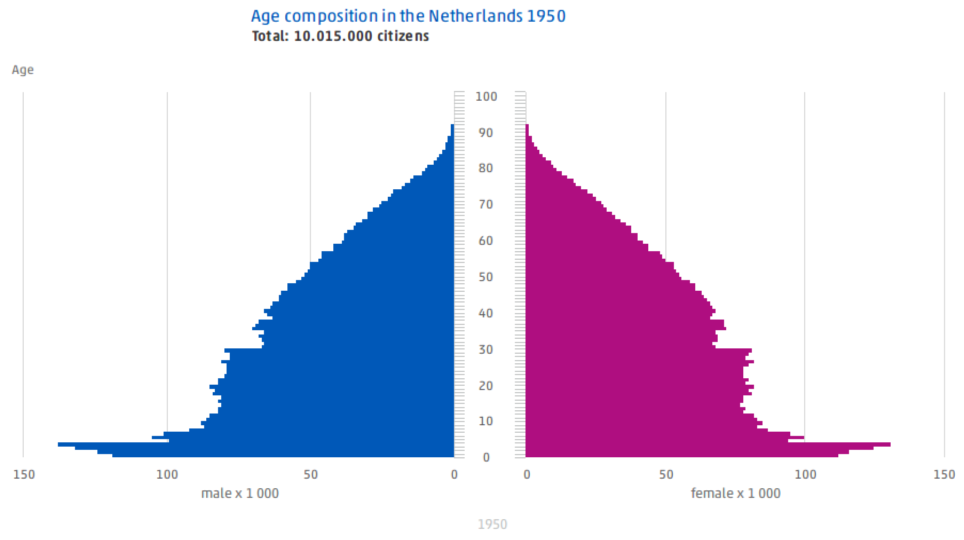


Figure 1.2: Age distribution of the population in the Netherlands in 1950 [2].

1.1 Thesis topic

This thesis was done under the supervision of DoBots. Dobots is a robotics company in Rotterdam that has experience in working with all sizes of robots. They worked with large robots in the form of a robot vaccuum cleaner, and with robot swarms like FireSwarms, a project they did in cooperation with

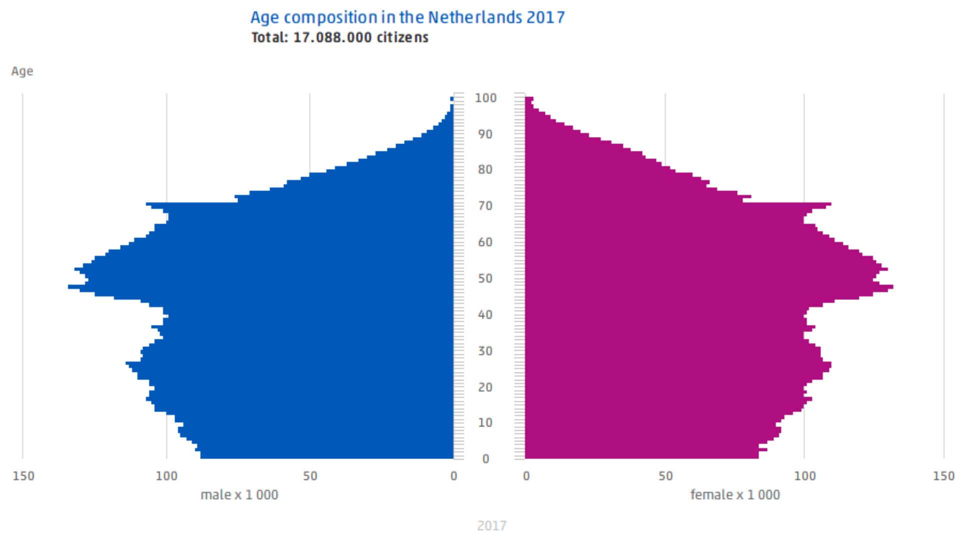


Figure 1.3: Age distribution of the population in the Netherlands in 2017 [2].

(amongst others) TU Delft. They are currently working on making smart homes an affordable reality, visit http://crownstone.rocks for more information. What DoBots proposes is a robot to help with the act of climbing stairs and carrying groceries. In an internship under DoBots a prototype of a stair-climbing robot was built by Frerik Andriessen[21]. The robot, built out of technical Lego, managed to successfully climb parts of the Groothandelstrap. There was an article about it on AD.nl which can be read in Appendix B. Meanwhile, another intern, Reka Hajnovic, is working on a robot that will carry the groceries for the elderly. A very nice touch of her robot is that it also allows the elderly person to use it for support. Combining that robot with stair-climbing capabilities will aid massively for elderly people that just need a bit of support. One of the things a stair-climbing robot needs is the ability to sense if there are stairs or not, and thus if it should climb the staircase or not. That is what this thesis aims to provide: a starting point for staircase navigation, in the form of a stair-detection algorithm.

The algorithm should be presented with a depth image from one of its own camera modules, and the algorithm should then decide that it sees a staircase or not, and send a signal accordingly.

1.2 Research questions

Developing a stair-detection algorithm is a long and difficult process. Carefully crafted research questions aid in keeping the right focus and give something to verify you are going in the right direction. For the stair-detection algorithm these are as follows:

- What are the recurring physical traits for a staircase?
- How can these traits be recognized?
- How can these traits be combined to accurately classify a staircase?
- Can an algorithm made up of simple rules accurately detect a staircase?

1.3 Outline of thesis

1.3.1 Methodology

The following methodology will be adapted:

- Think about how we as humans are able to determine something to be a staircase. Do not just focus on the most obvious ways, but also look into subtle cues that subconsciously alert us that something might be a staircase.
- Find and read papers on staircase detection, and what features they use as indications of a staircase. See how they correspond with own ideas, what did they not address; what did they address that was missed in the previous step?
- Synthesize papers with own ideas into a set of rules/features to detect a staircase.
- Read papers to see what is currently used to collect data on the aforementioned features.
- Read papers on combining the set of rules and features into a tool for staircase or object detection.
- Develop classification algorithm based on these rules and features.
- Combine datasets from earlier mentioned papers, and other sources into a big test data-set.
- Implement algorithm in a programming language and test on data-set.
- Compare results with papers, most importantly compare the FP/TN and FN/TP rate. What was different in the results and what might have caused this.
- Update algorithm and compare again, until good results have been met, or a reasonable amount of iterations have taken place.

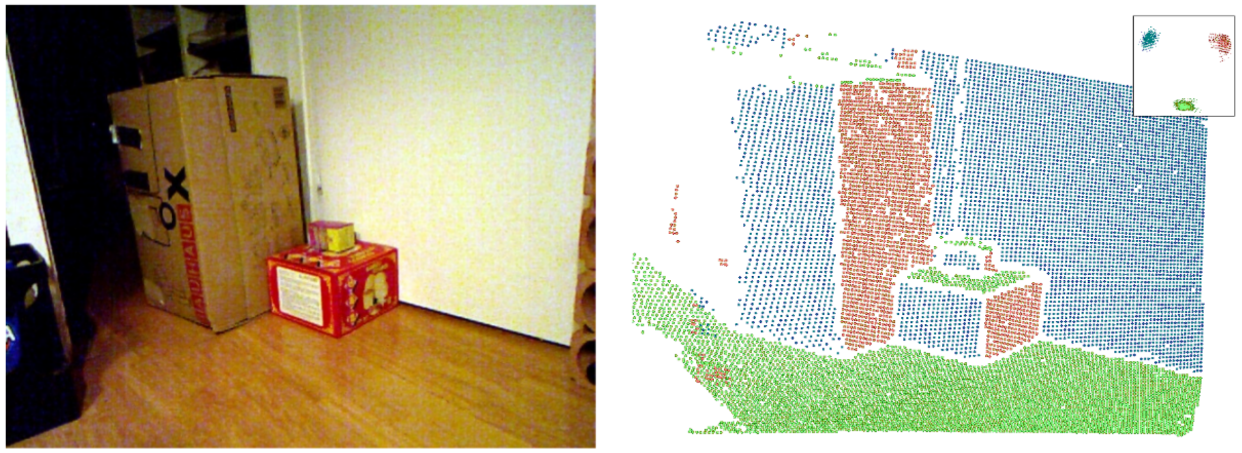
In chapter 2 the current state of the art in object and stair-case recognition is discussed. It shows which methods are seldom used and which are overly used and why. Chapter 4 will walk through the preprocessing algorithm step by step, and show the effects on an example RGB-D input image. It starts off with using the depth-info of the RGB-D image to create surfaces. It then differentiates between the surfaces and segments them. Chapter 5 then continues with the detection algorithm. First 20 random line coordinate pairs are chosen, and used to draw lines through the image. The underlying pixel values on these lines are stored in a vector, and checked for changes in pixel value ("transitions"). A chirp-signal is fitted to these transitions. Finally the results of all lines are weighted and summed using the AdaBoosting method. Chapter 7 shows the results of the algorithm on several datasets. Finally chapter 8 gives a short summary, draws conclusions, and gives several recommendations for future work.

2 Literature review

This chapter discusses the state of the art of staircase detection. It also reviews techniques relevant for staircase detection, like surface segmentation and the detection of objects in general. Section 2.1 discusses the method of surface segmentation, which allows uniform sections to be extracted from images. Section 2.2 discusses object detection using simple features, where Sect. 2.2.1 in particular discusses the current state of stair-detection.

2.1 Surface segmentation

Stairs consists mostly of a collection of flat surfaces (steps, risers, walls). A core concept of detecting a staircase therefor consists of first distilling the different surfaces from the depth image. These manipulations need to be done in real-time for a robot to have practical usage. The algorithm of S. Holz et al. allows for manipulation while dealing with images on a rate of 30Hz [3]. It allows detection of objects and obstacles while also grasping the geometry of the scene. Surfaces are detected using surface normals. There are two ways of doing so. One way is to fit a plane through the spatial neighbourhoods of a 3D point cloud, using either k-neighbours or a radius of r to delimit the neighbourhood, see Fig. 2.1. The second, computationally cheaper, way is to take the pixel structure of the input image and use those as pixel-based neighbourhood. The surface normal is obtained by calculating two tangential vectors using the left and right neighbour and top and bottom neighbour for the x and y vector respectively. If there is a large amount of noise (small spots that do not correspond to the rest of the surface) in the data, an average of multiple neighbours is used. Both techniques have to deal with blurring/smoothing when the neighbourhood is chosen too large, while they have to deal with the significant effect of noise in the depth image input when the neighbourhood is chosen to be too small. However, this effect is lesser for the pixel-based neighbourhood [3]. Objects and obstacles can be detected based on the fact that these should be situated on a groundplane or horizontal surface like a table. This simple algorithm had a 93% detection rate for objects, and 100% detection rate for obstacles.



(a) Input camera image (b) Segmented cloud

Figure 2.1: Segmented surfaces in the point cloud using plane-fitting through spatial neighbourhoods of a 3d point cloud by Holz et al. [3]. The different orientations of the surfaces can be easily distinguished based on their color.

2.2 Object detection

When detecting objects it is very difficult using just one or two characteristic traits to accurately determine whether or not it is the object you are looking for. Sometimes the lighting is off or the angle is such that an important trait is covered up. To deal with these problems and get a more robust detection algorithm, P. Viola and M. Jones [4] used an Adaboosted set of Haar-like features to detect faces, see Fig. 2.2. These features are called Haar-like because of their resemblance to Haar Wavelets, as can be seen in Fig. 2.3. These Haar Wavelets are used, amongst other applications, in image compression [22]. These Haar-like features are combined into a cascaded classifier of 38 stages and applied to 24x24 resolution images to detect whether or not a face was shown in the picture. The structuring of the classifier level was chosen in such a way that a large number of false images were discarded in the first levels, speeding up the process drastically. With each added level, the false positive rate decreases, while the detection rate decreases as well. While the goal is to have the false positive rate drop below a certain threshold, the detection rate should stay above a certain threshold. Therefore, it will add levels until the false positive rate has dropped to the required value, while the detection rate is still at an acceptable value.

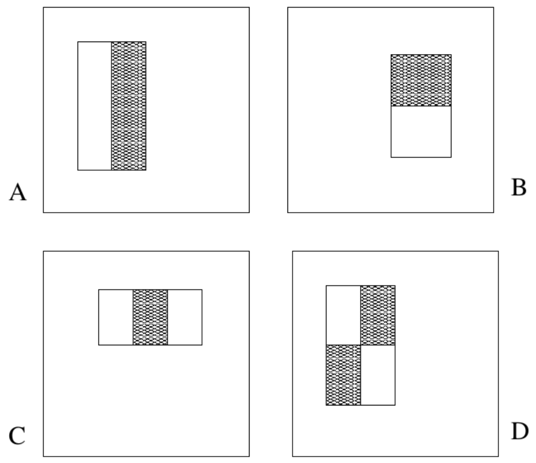


Figure 2.2: Examples of the Haar-like features used by Viola and Jones [4].

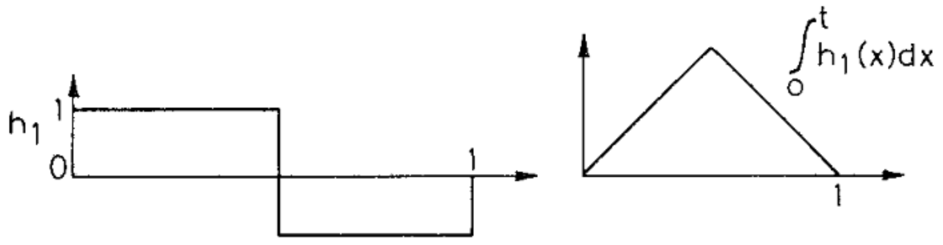


Figure 2.3: Example of Haar Wavelet [5]. Note the resemblance to the Haar-like features from Fig. 2.2.

2.2.1 Stair-detection

Although there have been many object and scene detection algorithms studied before, staircases are often not in the standard lists of everyday objects or scenes. See for instance the Microsoft COCO database[23]. While incredibly extensive, stairs are not included. Therefore, many research papers focusing on that dataset do not work on detecting staircases, and not many papers have dealt outright with detecting staircases. That said, there have been some smaller papers that looked into it.

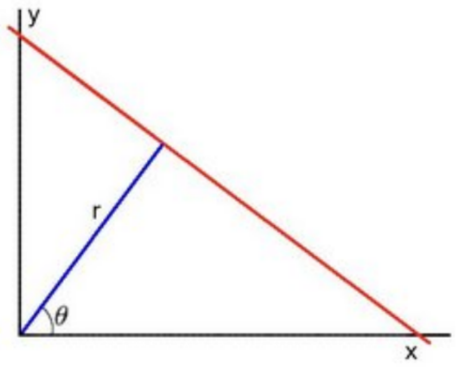


Figure 2.4: Parametric representation of line in (x,y) space [6].

S. Wang et al. worked on both stair-detection and crosswalk-detection using both normal RGB images and RGB-D images [8]. First, they used a Sobel edge detector[24] to retreive all the edges in the image. A sobel edge detector works on the fact that a relative large difference in pixel intensity often accompanies an edge. It takes an approximate derivative and searches for local maxima; this is implemented using two kernels in the following form (as an example, using a kernel of 3), which are afterwards convolved with the image:

$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, G_y = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, G = \sqrt{G_x^2 + G_y^2} \tag{2.1}$$

This returned an image with a collection of pixels where an edge was detected. They then converted the loose pixels into lines using the Hough Transform. The Hough transform takes a point that was determined to be an edge, and fits a line through it. This line could be represented using $y = ax + b$, but there is a major downside to this representation. The value of the slope a goes to infinity as the line becomes vertical. This makes it impractical for storing and plotting of the (a,b) space. That is why these lines are converted to polar coordinates, using

$$y = \left(-\frac{\cos\theta}{\sin\theta}\right) x + \left(\frac{r}{\sin\theta}\right) \tag{2.2}$$

where r is the length of the perpendicular line from the origin to the line being converted to polar coordinates, see Fig. 2.4, and θ is the angle in degrees, which can range from (0-180).

This equation can be rearranged to

$$r = y\sin\theta + x\cos\theta \tag{2.3}$$

Now, for every edge-point (x,y) there are many combinations of (r,θ), and thus many lines, that go through this point. Plotting these combinations of r and θ would give a sinusoid, where each point on the sinusoid would represent a line that goes through the edge-point. Doing this for all edge-points in the image gives a different sinusoid for each point. The points where some of these sinusoids intersect signify the lines that go through multiple edge-points. The more sinusoids intersect, the more points can be found on that line. The Hough Transform has a certain threshold of these intersections. If the amount of intersections is above the threshold it will draw the line through these points. Of course, the lower the threshold, the more lines are detected, but it also increases the amount of noise and bad line-fits.

Wang et al. filtered out all lines that were not at least somewhat horizontal. They used the same representation for their lines as was used in the Hough Transform, e.g. $r = y\sin\theta + x\cos\theta$, where the origin is the starting point of the line. Now, the lines that are found need to be parallel, which is the case of their θ is equal. If this is the case the algorithm checks if the lines have a certain minimum length, and if there is a large enough amount of individual lines. In their case they chose a length of 60 pixels and a minimum of 5 lines. If the image scores low on either one of them it is discarded as a negative image. Otherwise, it is classified as either a staircase or a crosswalk. Once this is determined, it will check its RGB-D counterpart and they will create a depth-graph by drawing a line through the stairs/crosswalk and collecting the values along this line, see Fig. 2.6. They then use a hierarchical Support Vector Machine based Radial Basis Function to classify either a crosswalk, an ascending staircase or a descending staircase [7]. This Radial Basis Function uses the training data to demarcate areas for each classification group, see Fig. 2.5. The area that every new data point falls in, is then designated as the data-point's classification group. Note that the example in the figure is 2-D for visual purposes but the RBF supports higher dimensionality.

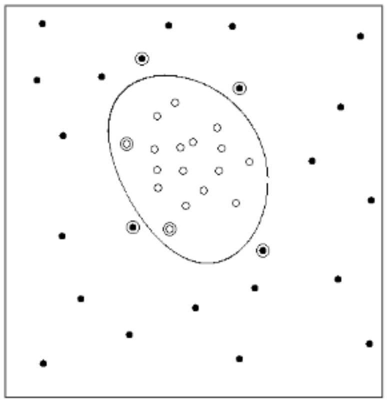


Figure 2.5: Example of demarcation done by a Radial Basis Function[7]. Any new datapoint falling within the circle will be designated white, while any point falling outside the circle will be designated black. Note that higher dimensions are possible.

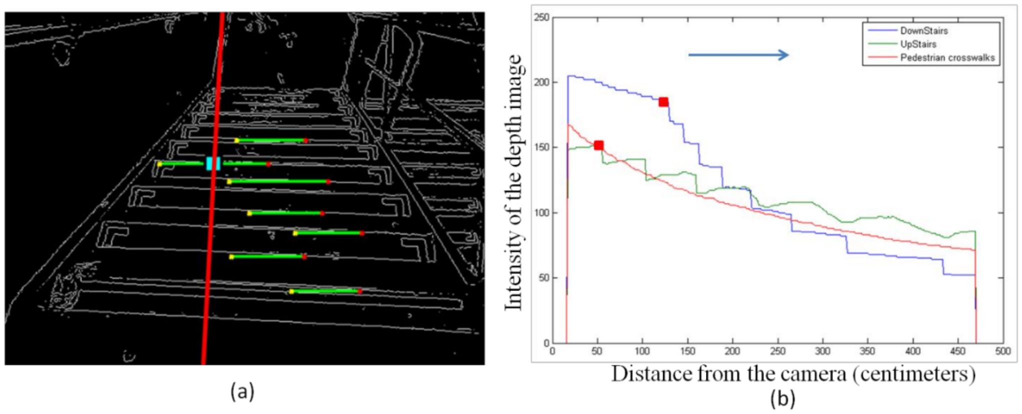


Figure 2.6: Example of depth-graph of crosswalk and staircases from Wang et al.[8]. The crosswalk decreases in intensity as the line moves further away from the camera. The upstairs and downstairs graphs also show clear characteristics. The downstairs graph is more steep and has straight corners, while the upstairs graph slowly rotates as the steps move up.

Using a training data-set of 60 stair images, 30 crosswalk images and 30 negative images, they tested on a test data-set of 106 staircase images, 70 negative images and 52 crosswalk images. A True Positive rate of 97.2% and a True Negative rate of 100% was obtained.

Lee et al. devised a portable stair-detection vest for the visually impaired [9]. It uses a combination of Haar-like features with adaBoosting, ground plane measurements and temporal consistency, see Fig. 2.7. For the first part of the algorithm, the classifier is trained with Discrete AdaBoosting for 18 rounds on 210 positive images with 25 distorted variants each and 7000 negative images all with a resolution of 40x40. At the end of this training, the True Positive rate on the training data-set was 0.983. The classifier scans the image and divides it up in 40x40 subwindows. To weed out False Positives, multiple detection subwindows need to overlap for the classifier to consider it. Using stereo vision it determines the groundplane and gives a weight to detected staircases. As the staircase is closer to the groundplane it gets a higher weight and vice versa. Finally, the affinity between two consecutive frames with detection needs to be above a certain threshold for it to be ultimately determined to be a staircase. It is shown that a combination of the three (Cascaded Classifier, Ground Plane, Temporal Consistency) has the best result, but also that Ground Plane has very little effect on the precision of the algorithm. They attribute this sub-par performance to the not-yet matured ground-plane detector technique. They are able to detect stairs under a varying number of conditions, like differences in sunlight and angle. This gives them an advantage over the technique used by S. Wang et al [8] that is only able to detect stairs when directly facing them.

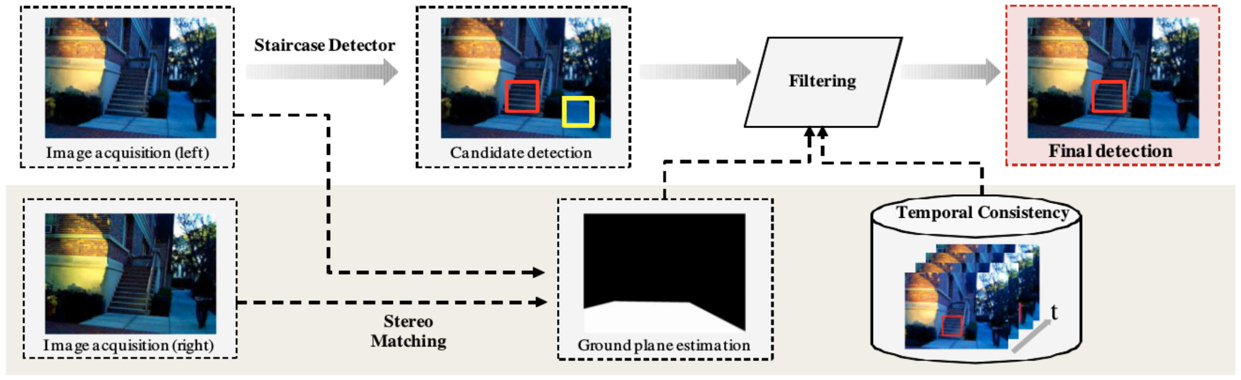


Figure 2.7: Process of staircase detection by Lee et al. [9]

Stair-navigation

Y. Xiong & L. Matthies used low-threshold edge detection to determine their relative position and attitude on the stairs. [10]. Due to its low threshold it will register a lot of edges that are not staircase edges. They incorporated the following rules to filter the large amount of edges obtained:

- Stairs edges are straight lines
- When looking straight at a staircase, the edges are relatively horizontal and mostly parallel to each other
- If there are several small edge segments that lie on the same line, they can be merged to one
- Stair edges in general, are long

Using these rules the non-relevant edges obtained with the low-threshold technique get filtered out, namely by filtering out non-straight edges, filtering out edges whose orientation is too different from the dominant orientation, linking together edges that are close and parallel, and filtering out edges that are small, since staircase edges are long. The results can be seen in Fig. 2.8. This was then used by them to determine the robots center position and heading, but it could also be used as input for a stair-detection algorithm.

2.3 Conclusion

Object detection and scene recognition is a widely-studied subject that is well-represented in the academic world. However, the subset of stair-detection is significantly less studied. The papers that addressed stair-detection both used a combination of a simple classification algorithm and machine learning to train the classifier. They have relied on Haar-like features or parallel lines for their simple classifier. They do not take advantage of the converging repeatability of staircases which seems to be an interesting way to make the classifications more robust.

This thesis will develop an algorithm for stair-detection based on the repeatability of the structure of staircases. It will use a simple classification rule which will then be trained using a varied data-set. The goal is to expand the options that are currently available for stair-detection.

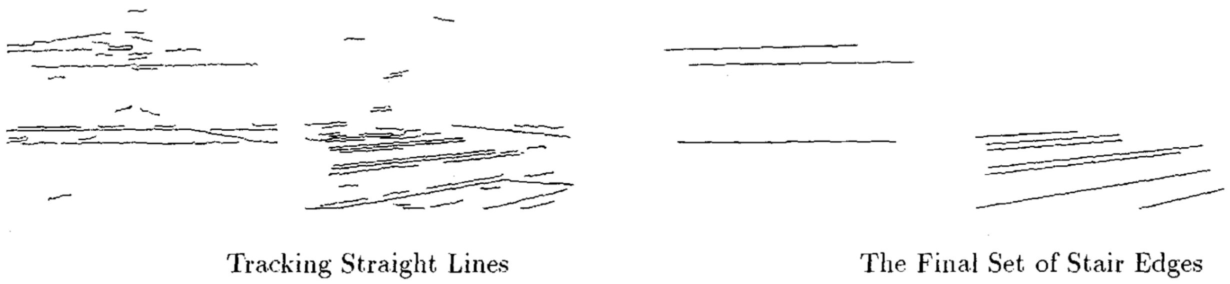


Figure 2.8: Start and finish of filtering stair line-segments [10]

3 Tools and Techniques

This thesis builds on several algorithms and libraries that already exist. This chapters aims to clarify which parts were original work and which parts were partly or completely taken from previous research. First, in Sect. 3.1 the already existing set of tools and techniques are mentioned, which will be discussed in more detail in Chapters 4 and 5. In Sect. 3.2 the adaptions to the techniques from Sect. 3.1 are discussed. These will also be discussed in more detail in the main body of the thesis.

3.1 Existing tools and techniques

The already existing tools and techniques used in this thesis are, in order of usage:

- The math for the extraction of surface normals
- The C++ implementation of a Gaussian Blur, from the OpenCV library[25]
- Bitmask manipulation to divide each pixel in bins based on their combined channel-value
- The C++ implementation of the Floodfill algorithm, from the OpenCV library[25]
- The Bresenham line-drawing algorithm for one octant[11]
- The AdaBoosting algorithm [26]

3.2 Adaptions and novel contributions

The following contributions, adaptions or novel ideas were implemented in this thesis, in order of usage:

- An implementation of surface normal extraction in C++ using OpenCV
- An implementation of bitmask manipulation to divide pixels between bins in C++
- An implementation of the calculation of random coordinate pairs for the drawing of random lines in C++
- An implementation of Bresenham's line-drawing algorithm in C++
- Updating the Bresenham line-drawing algorithm to deal with all cases in an octant
- An algorithm for fitting a chirp-function to a repeating structure
- An implementation of AdaBoosting in C++
- An adaption of the AdaBoosting algorithm to use chirpfitting as a rule

4 Stair-case RGB-D preprocessing

As discussed in the literature review in chapter 2, the input will be a dataset of 16-bit RGB-D images. The image processing software is based on the C++ libraries of OpenCV[1].

The stair-detection algorithm consists of two parts. The first part consists of processing the input-images to prepare them for the stair-detection methods in the second part. An overview of the first part can be seen in Fig. 4.1. First, a depth image is loaded. Then, the surface normals are calculated using the depth gradients in Sect. 4.1. To remove most of the noise a Gaussian Blur is applied to the images in Subsect. 4.1.1. In Sect. 4.2 the different surface orientations are divided into 8 bins based on binary addition. Finally, in Subsect. 4.2.2 the small specks of different orientations are smoothed out using a Floodfill algorithm, that recolors sections that have an area below a certain threshold. This is all discussed in more detail in the following sections.

4.1 Extracting surfaces

With the depth information provided by the RGB-D images, it is possible to see the image as quasi-3D, using the pixel values to visualize the position on the z-axis. The image that is loaded up in the stair-detection program is the depth image, using grayscale intensities to represent the depth of the objects in the scene. An example of the loaded input can be seen in Fig. 4.2.

Every pixel has a depth value based on its distance to the camera. We cannot directly extract steps from this information. To create coherent surfaces, we rely on the surface normals of the image. The gradient of each pixel is calculated based on its direct horizontal and vertical neighbours. Taking the cross product of the derivatives in x and y direction gives us the normal vector, see Eq. 4.1. Taking p as a vector with x, y and z coordinates as vector elements, so $p = [x, y, z]^T$, allows us to easily relate and manipulate nearby pixel values. Here x and y represent the coordinates of the pixel and z is the intensity value.

$$n = \frac{\partial p}{\partial x} \times \frac{\partial p}{\partial y} \tag{4.1}$$

where $\frac{\partial p}{\partial x} = [1, 0, \frac{\partial z}{\partial x}]$ and $\frac{\partial p}{\partial y} = [0, 1, \frac{\partial z}{\partial y}]$. In our example $\frac{\partial z}{\partial x}$ is calculated taking the difference between the depth value at (x-1) and (x+1) and dividing by 2. More neighbours could be incorporated in both directions, which would give a smoothing effect as the number of neighbours are increased. This is discussed in Subsect. 4.1.1. The same principle is applied to $\frac{\partial z}{\partial y}$. The surface normal following from this calculation is a vector with 3 elements. These are stored in a new image container using the RGB channels. This leads to the following output, see Fig. 4.3.

As can be seen, there is quite some noise in the resulting image, as visualized by the lines and specks scattered throughout the different surfaces. This stems from the noise in the input image, where a different value means a different depth, which translates to a wildly different surface normal at those neighbourhoods. Not only that, there will also be round-off errors. Say that for each pixel moved in y-direction, z increases with 0.8, thus $\frac{\partial z}{\partial y} = 0.9$. The pixel values only work with integer values from 0-255, any decimal value is truncated. So 0.8 turns into 0. The next pixel has a value of 1.6, thus it is 1. Next has a value of 2.4, so it becomes 2. 3.2 turns into 3 and 4.0 is actually 4. Then, the next value is 4.8, but due to the truncation this is also turned into 4. The previous pixel transitions all showed $\frac{\partial z}{\partial y} = 1$ but this pixel transition is now shown as $\frac{\partial z}{\partial y} = 0$ even though the pixels are uniformly increasing in value. This shows up in the image clearly at the vertical part of the steps, the risers, where the normal value of red is sometimes interjected by a black line, which is the point where the truncation caused two succeeding pixels to be the same value.

[1]An important note: OpenCV uses BGR values instead of the commonly used RGB values. This has no effect on the calculations but when referring to, for instance, the blue channel, we are talking about the first element of the vector, followed by the green channel, and then the red. Similarly, pixel coordinates are given in the form of (y,x) instead of the normally seen (x,y) format.

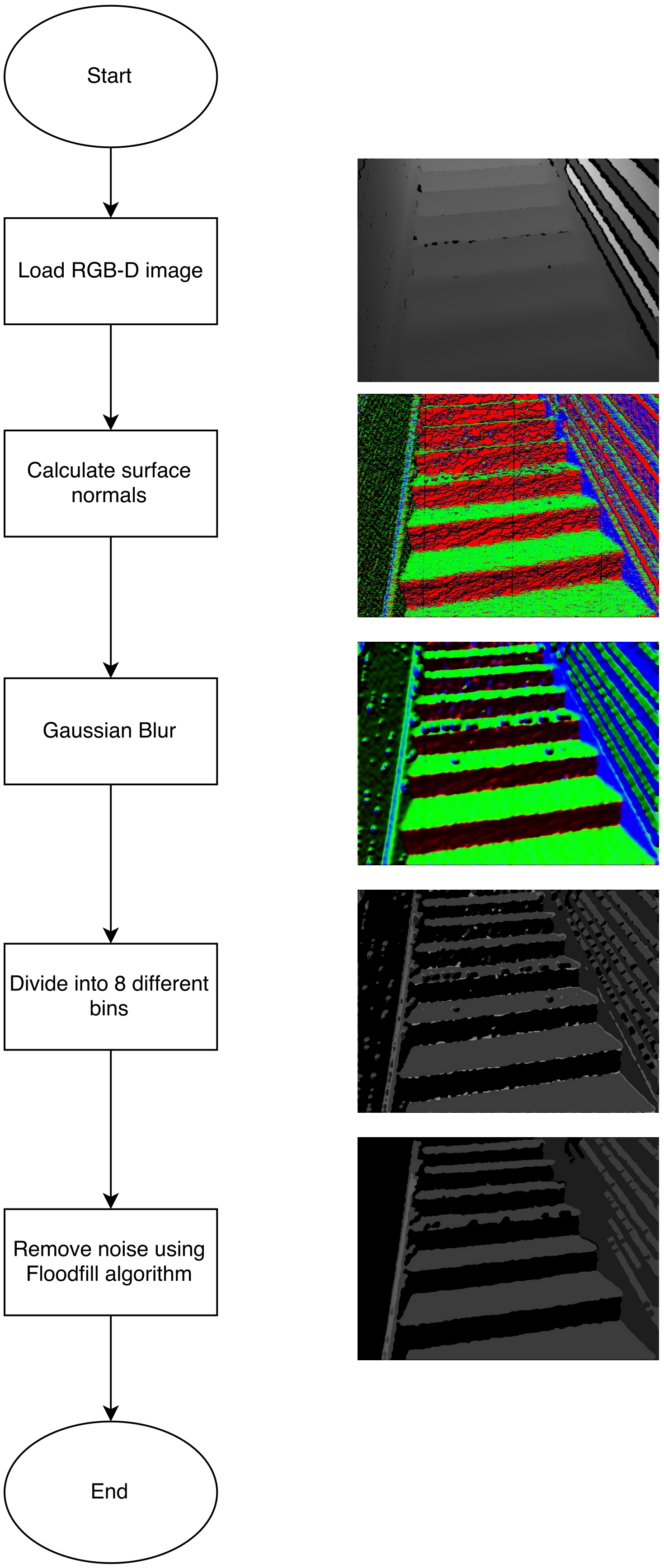

Start
Load RGB-D image
Calculate surface normals
Gaussian Blur
Divide into 8 different bins
Remove noise using Floodfill algorithm
End

Figure 4.1: The steps taken in pre-processing part. Outputs after each step can be seen next to the step. The transformation of a noisy multi-channel image to a smooth one-channel image should be clear.

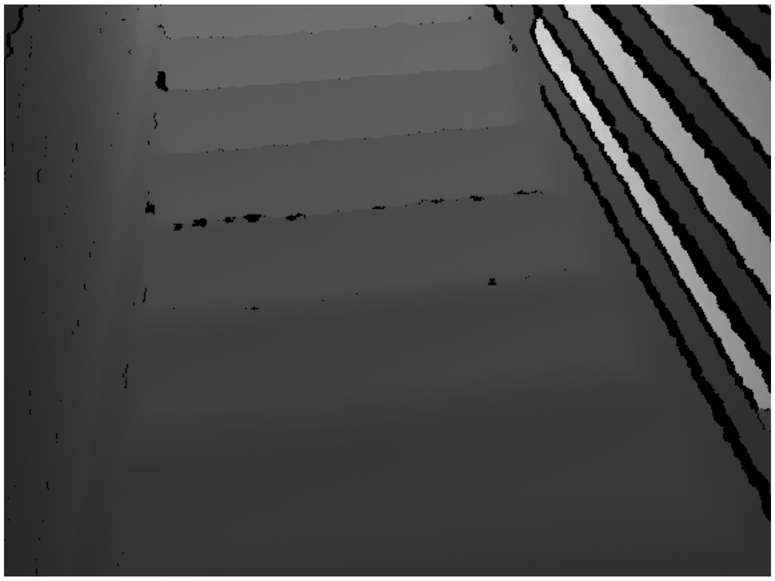


Figure 4.2: Loaded raw input, before doing any processing on it. Already some noise can be seen at the edges of some steps.

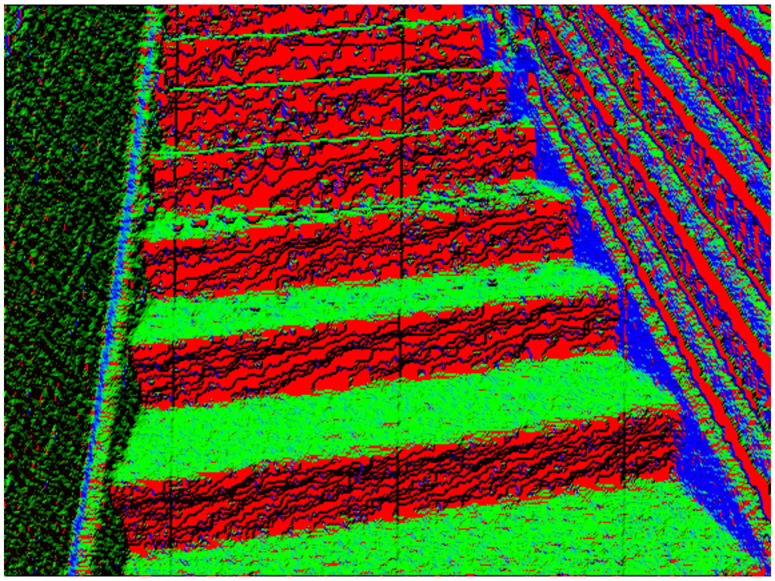


Figure 4.3: Surface normals without Gaussian Blur. There is noise due to the discrete (values of 0-255 in steps of 1) nature of the pixel values as well as the lesser quality of the input data.

4.1.1 Gaussian Blur

To correct for the noise as seen in Fig. 4.3, a Gaussian Blur filter is applied. By incorporating values of the neighbouring pixels, large differences in pixel values between pixels are smoothed out, causing a blur. This is a double-edged sword since it removes unwanted noise, but it also smooths out edges, if applied over too many neighbours per pixel. Depending on the situation, this can merge distinct sections that should not be merged. The weights of the pixels used to average the pixel that is the target of the Gaussian Blur follow a Gaussian distribution. This means the center pixels own value has the most weight, after which the neighbouring pixels have lower weight the further they are away from the center pixel, according to the 2D

Gaussian distribution. The filter is implemented via

$$p^*(i,j) = \sum_{k,l} p(i+k, j+l)h(k,l) \tag{4.2}$$

where p^* is the new value of pixel p, and $h(k,l)$ is the kernel for the 2D Gaussian distribution. This kernel is a 2D matrix of [k x l] dimensions, storing the value of the weights for each cell in the kernel. The kernel is calculated with

$$h(k,l) = A \exp^{\frac{-(k-\mu_k)^2}{2\sigma_k^2} + \frac{-(l-\mu_l)^2}{2\sigma_l^2}} \tag{4.3}$$

where μ is the mean and σ is the standard deviation of the kernel. A is the amplitude, which is chosen such that $\sum_{k,l} h(k,l) = 1$ to not add or remove any energy from the image, and since the probability of each value combined should be equal to 1. Now the sum of a 1-dimensional Gaussian distribution G(k) can be calculated by taking its discrete integral:

$$\sum_k G(k) = \int_{-\infty}^{\infty} A \exp^{\frac{-(k-\mu_k)^2}{2\sigma_k^2}} dx \tag{4.4}$$

Recall that for an arbitrary Gaussian integral

$$\int_{-\infty}^{\infty} \exp^{a(x+b)^2} dx = \sqrt{\frac{\pi}{a}} \tag{4.5}$$

which means that, substituting $a = \frac{-1}{2\sigma_k^2}$ and adding constant A (which has no effect on the integral):

$$\int_{-\infty}^{\infty} A \exp^{\frac{-(k-\mu_k)^2}{2\sigma_k^2}} dx = A\sigma_k\sqrt{2\pi} \tag{4.6}$$

This can only be 1 if $A = \frac{1}{\sigma_k\sqrt{2\pi}}$. So

$$G(k) = \frac{1}{\sigma_k\sqrt{2\pi}} \exp^{\frac{-(k-\mu_k)^2}{2\sigma_k^2}} \tag{4.7}$$

Naturally, the same holds for the 1D Gaussian in the other direction, G(l). To transform from a 1D Gaussian distribution to a 2D Gaussian distrbitution we simply multiply both 1D Gaussian with each other, which leads to

$$h(k,l) = G(k)G(l) = \frac{1}{2\pi\sigma_k\sigma_l} \exp^{\frac{-(k-\mu_k)^2}{2\sigma_k^2} + \frac{-(l-\mu_l)^2}{2\sigma_l^2}} \tag{4.8}$$

Now, since we want the distribution to have the center pixel as the focus point, the mean is 0, thus $\mu_k = \mu_l = 0$. Furthermore, if we assume both standard deviations to be equal to each other, thus $\sigma_k = \sigma_l = \sigma$, we get

$$h(k,l) = \frac{1}{2\pi\sigma^2} \exp^{\frac{-(k^2+l^2)}{2\sigma^2}} \tag{4.9}$$

Sigma can be chosen freely, dependent on what spread you want the distribution to have. In OpenCV, it is normally calculated based on the size of the kernel, K, with

$$\sigma = 0.3\left(\frac{K-1}{2} - 1\right) + 0.8 \tag{4.10}$$

In this case, the kernel size was chosen to be 17 in both directions, to smoothen out all noise. Since the images need to focus on showing the difference in surfaces and not have a focus on having the edge at exactly the right point, this was a sacrifice that could be made, in return for a very smooth image.

The result of extracting the surface normals after Gaussian Blur has been applied can be seen in Fig. 4.4.

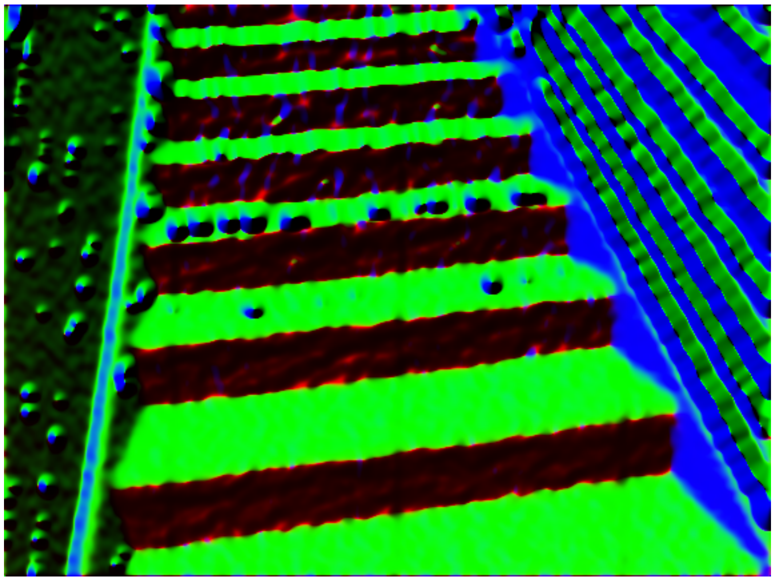


Figure 4.4: Surface normals after Gaussian Blur

4.2 Clustering pixels

We now have an image that looks very clear when looked at from a humans perspective. The steps are easy to distinguish from each other and are somewhat uniformly colored. However, for a computer the image still consists of constantly varying pixels. This is evident especially in the risers (vertical parts of the step). Furthermore, the value of each pixel is a combination of each individual channel, which does not allow for easy comparison. Therefore we start off with dividing the pixels into bins.

4.2.1 Histogram or bins

The pixels will be divided in eight different bins, dependent on their combined RGB values using binary arithmetic. For each channel, the pixel is classified in either the lower half of the value spectrum, or the upper half. Zero meaning it has none of this color channel, 255 meaning it has 100% intensity for this particular color channel.

When the channel falls in the upper half of the spectrum it gets a '1', otherwise it gets a '0'. The BGR channels, represented by 2^0, 2^1, and 2^2 respectively, can form a binary number from 000 to 111. This binary value is then converted to a decimal value and applied to a new image container. For example, a pixel with BGR values of Red(230): '1', Green(150): '1' and Blue(90): '0', gets a binary value of '110' or 6 in decimals. This is done for every pixel in the image.

First, an empty 1-channel matrix ('clust') of the same size is initialized with zeros. The algorithm then loops through the surface normal matrix, called 'nor' from now on, and checks each channel. If the value of the first, second or third channel exceeds 127, a value of +1, +2, or +4 is added to the corresponding pixel in clust. See Fig. 4.5 below.

The results can be seen in the following figure, see Fig. 4.6.

The steps have now mostly become a uniform surface, both the horizontal parts and the vertical parts. However, especially at the edges there are a lot of noisy patches. To have them not interfere with the stairdetection algorithm, these need to be removed.

4.2.2 Floodfill algorithm

To get rid of these noisy patches, we want to find areas of the same value. Any area that has a size below a certain threshold needs to be given the same value as a neighbouring area, such that it becomes one. To do this a Floodfill algorithm is used. Floodfill is comparable to the bucket-function in most Paint-like

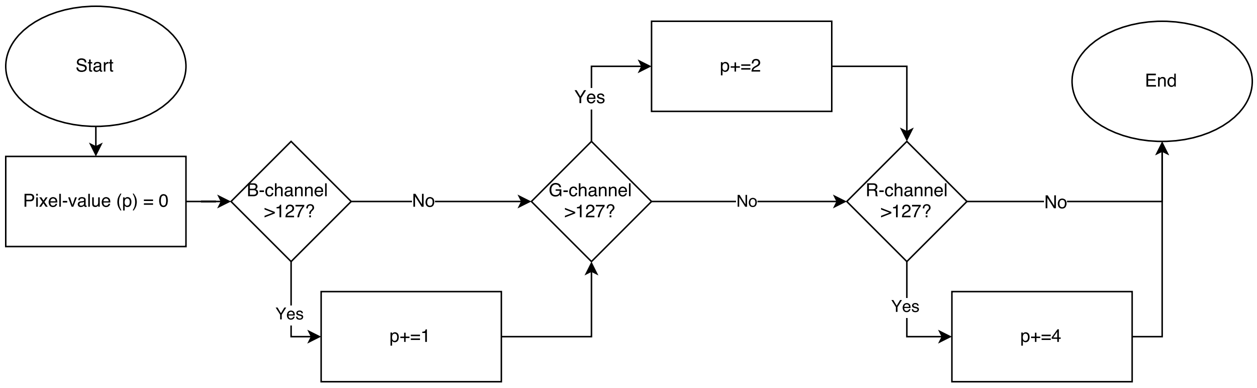


Figure 4.5: Flow Diagram for segmenting pixels based on their individual channel value.

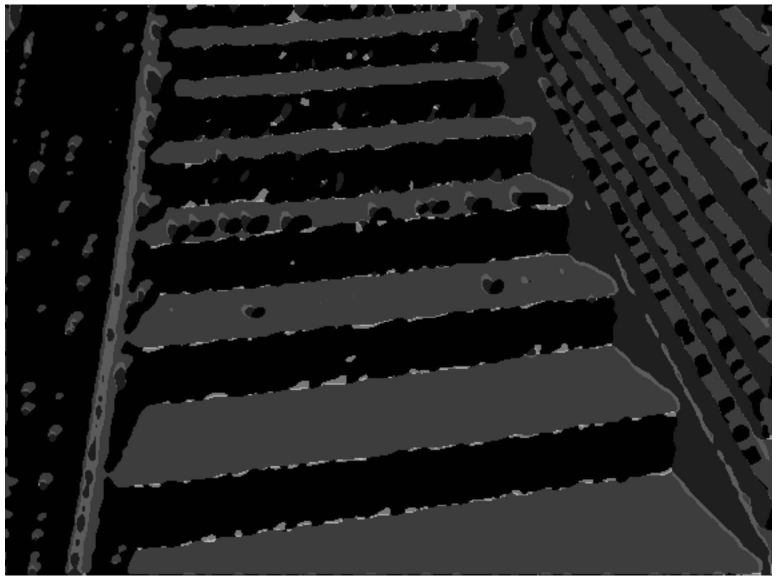


Figure 4.6: Clustered staircase using 8 different bins

programs. It uses a recursive algorithm where it checks each neighbour (using either a Von Neumann or Moore neighbourhood) for the same value, if this is the case it will go to that neighbour and do the same thing, until done, see Alg. below. In the implementation of the Step-1 algorithm a Moore neighbourhood was used.

Sections with an area below 300 pixels were deemed noise, and were floodfilled with the value of the left-neighbour, $(y, x-1)$, unless it was the first column of the matrix, then the neighbour above, $(y-1, x)$, was used. The result can be seen in Fig. 4.7.

Now that the input has been processed, it is ready for the stair-detection algorithms.

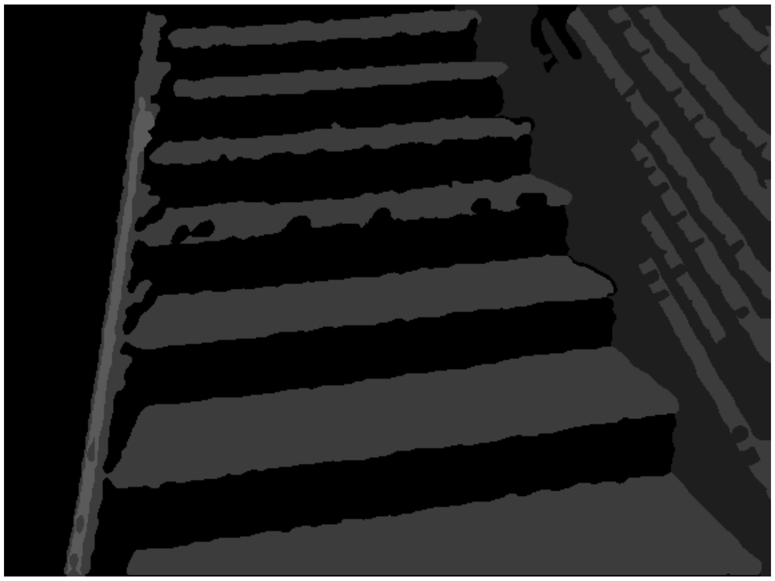
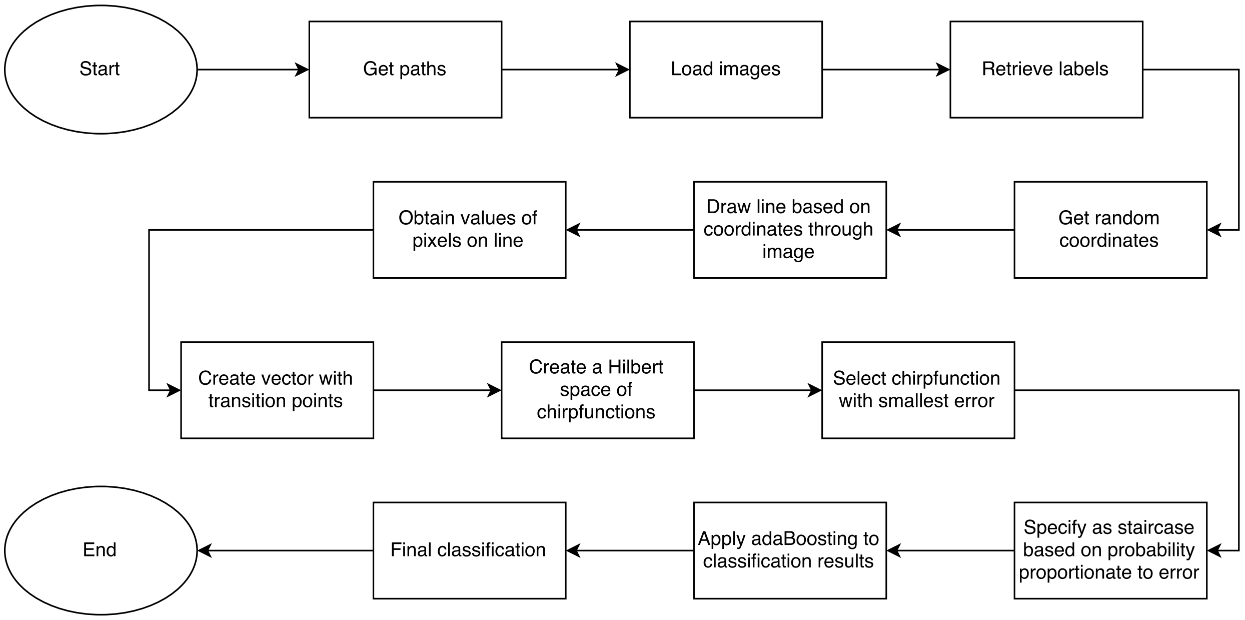


Figure 4.7: Noisy areas removed with Floodfill algorithm

5 Stair-detection algorithm

The stair-detection process consists of several steps, the overview of which can be found in Fig. 5.1. First the images are loaded again and the correct labels are retreived from the filenames. Using a random number generator an n-number of random line-pair coordinates (consisting of a horizontal $[x_0 - x_{max}]$ and a vertical $[y_0 - y_{max}]$ line) are generated. These coordinates are used with Bresenham's line drawing algorithm to collect the values of the pixels between these points [11]. This vector of values is then searched for transitions, which are noted with a '1'. Every pixel that is the same value as the pixel before it gets a value of '0'. There is a certain spacing between these transition points that tend to get smaller or larger over time. These gaps are then fitted with a sine-wave with increasing/decreasing period. A Hilbert space is created of these different chirpfunctions, and the chirpfunction with the smallest error is selected. Its error is used as an input for a probabilistic classification. This is done for every coordinate-pair. AdaBoost is used to get a final classification based on the sum of the weighted results of each coordinate-pair. Every image is trained with the same coordinates.



Start
Get paths
Load images
Retrieve labels
Get random coordinates
Draw line based on coordinates through image
Obtain values of pixels on line
Create vector with transition points
Create a Hilbert space of chirpfunctions
Select chirpfunction with smallest error
Specify as staircase based on probability proportionate to error
Apply adaBoosting to classification results
Final classification
End

Figure 5.1: Overview of stair-detection algorithm

5.1 Drawing random lines

Since the images that are presented to the algorithm do not have a fixed angle, it needs to be robust against deviations in perspective or angle, on a per-picture basis. To aid in this, randomly drawn lines will be used, ensuring that the orientation of the input does not matter since the lines deviate amongst themselves anyway.

5.1.1 Generating random coordinate-pairs

A very simple algorithm for drawing random lines was used, consisting of three steps:

1. The program selects the first column as starting point for the horizontal line, and the first row as the starting point for the vertical line. Similarly, it selects the last column as endpoint for the horizontal line and the last row as endpoint for the vertical line.

2. It then chooses two random values for the other start and end points of the line.

3. It draws a line using Bresenham's line drawing algorithm using these coordinates.

5.1.2 Bresenham's line drawing algorithm

Bresenham's line drawing algorithm uses two points as input, the starting point (X_0, Y_0) and the end point (X_1, Y_1). The basic version of the algorithm is as follows: In the situation that $\Delta X > \Delta Y$, the line will travel over the x-values, and depending on the buildup error in y, travel one pixel in y-direction everytime the error has build up to far. The error is calculated with

$$\Delta\epsilon = \frac{\Delta Y}{\Delta X} \tag{5.1}$$

This means that, for every pixel the line travels in X-direction, it builds up an error of $\Delta\epsilon$. Once the error becomes larger than one, i.e. $\epsilon > 1$, the line being drawn should shift one pixel in the Y-direction. The value of ϵ is then subtracted by 1 (since the error has now been decreased by 1 pixel) and the algorithm continues to the end point of X. An example can be seen in Fig. 5.2.

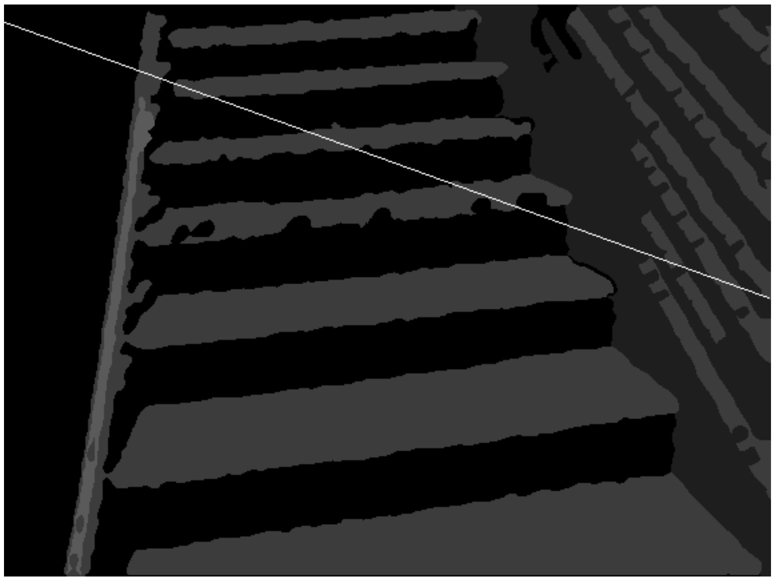


Figure 5.2: Example of Bresenham's line drawing algorithm applied to clustered input

There are some limitations to the current implementation:

- If the error increases with more than 1 for each pixel it travels, it will not reach the end point of X
- If either ΔY or ΔX are negative, the error will be negative as well and never adjust.
- It can only travel from top to bottom and from left to right

$\Delta\epsilon$ is of course the slope of the line, so what this means is that the current implementation does not allow for slopes larger than 1, and the line can only be drawn from a top-left position to a bottom-right position. This only addresses one of the possible eight line-orientations, see Fig. 5.4. In this drawing, (x_1, y_1) is the starting point, (x_2, y_2) is the end point, and m is the slope, i.e. $m = \Delta\epsilon = \frac{\Delta Y}{\Delta X}$.

So, some adjustments need to be made. This can be done using logical statements that check for the conditions in the octants and change the code to accompany those conditions. For example, if $1 < m < \infty$ the error fraction needs to be reversed, such that the slope now falls between 0 and 1, and the leading direction changes from an x-direction to a y-direction. Similarly, if the error is negative, then every time a

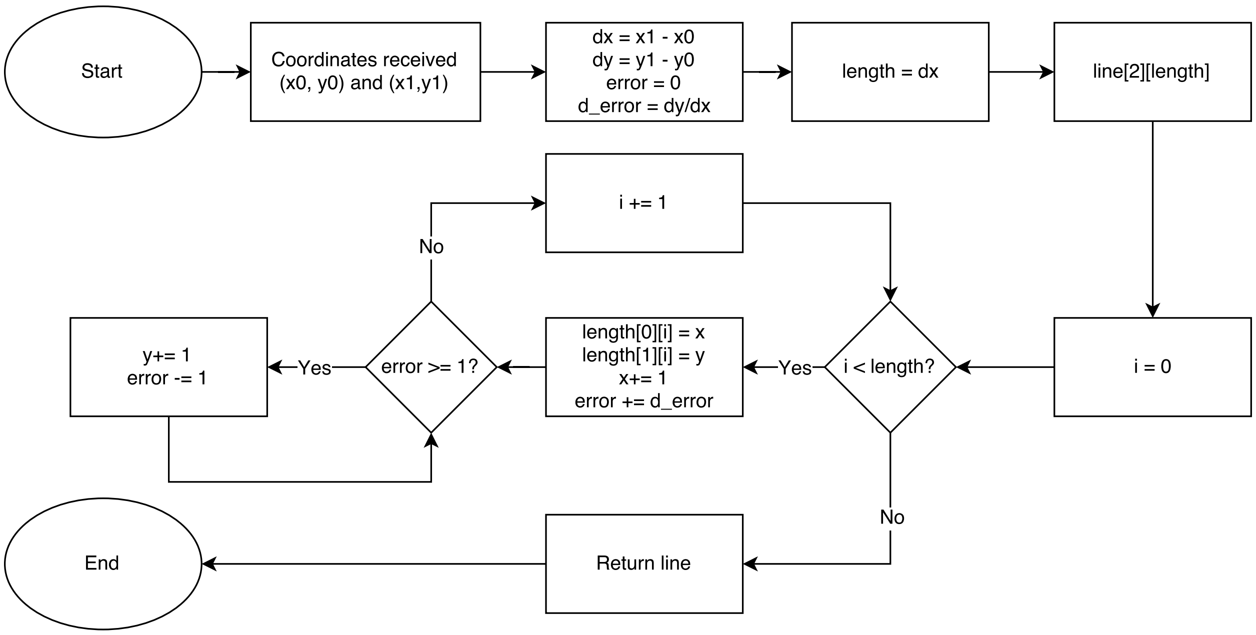


Figure 5.3: Flow Diagram of the original Bresenham line drawing algorithm. Note that it does not allow for negative dx or dy, or for dy to be bigger than dx.

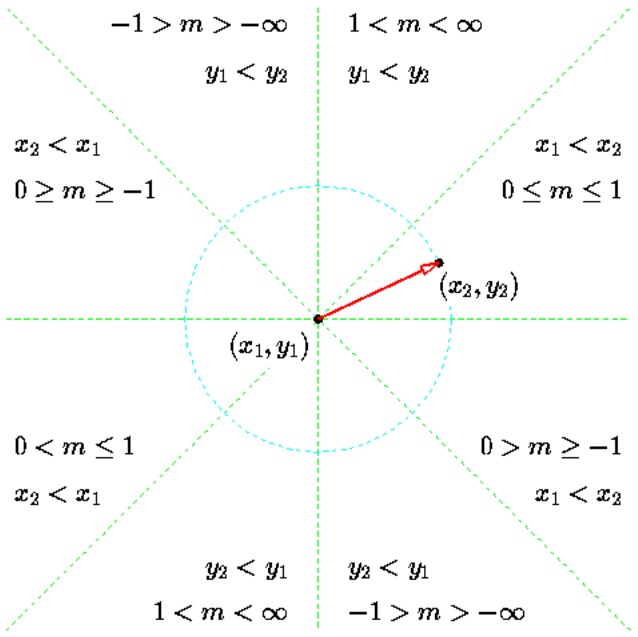


Figure 5.4: The eight possible orientations for drawing a random line[11]

pixel is jumped and the error needs to be decreased in absolute terms, you increase it by 1, i.e. from -1.5 to -0.5.

To prevent having to write out the code 8 times, a condensed version was used where the sign of ΔY and ΔX determines in what direction the next pixel moves. See below.

With these adjustments made, the line-drawing algorithm is now able to draw any line on the image, see Fig. 5.6.

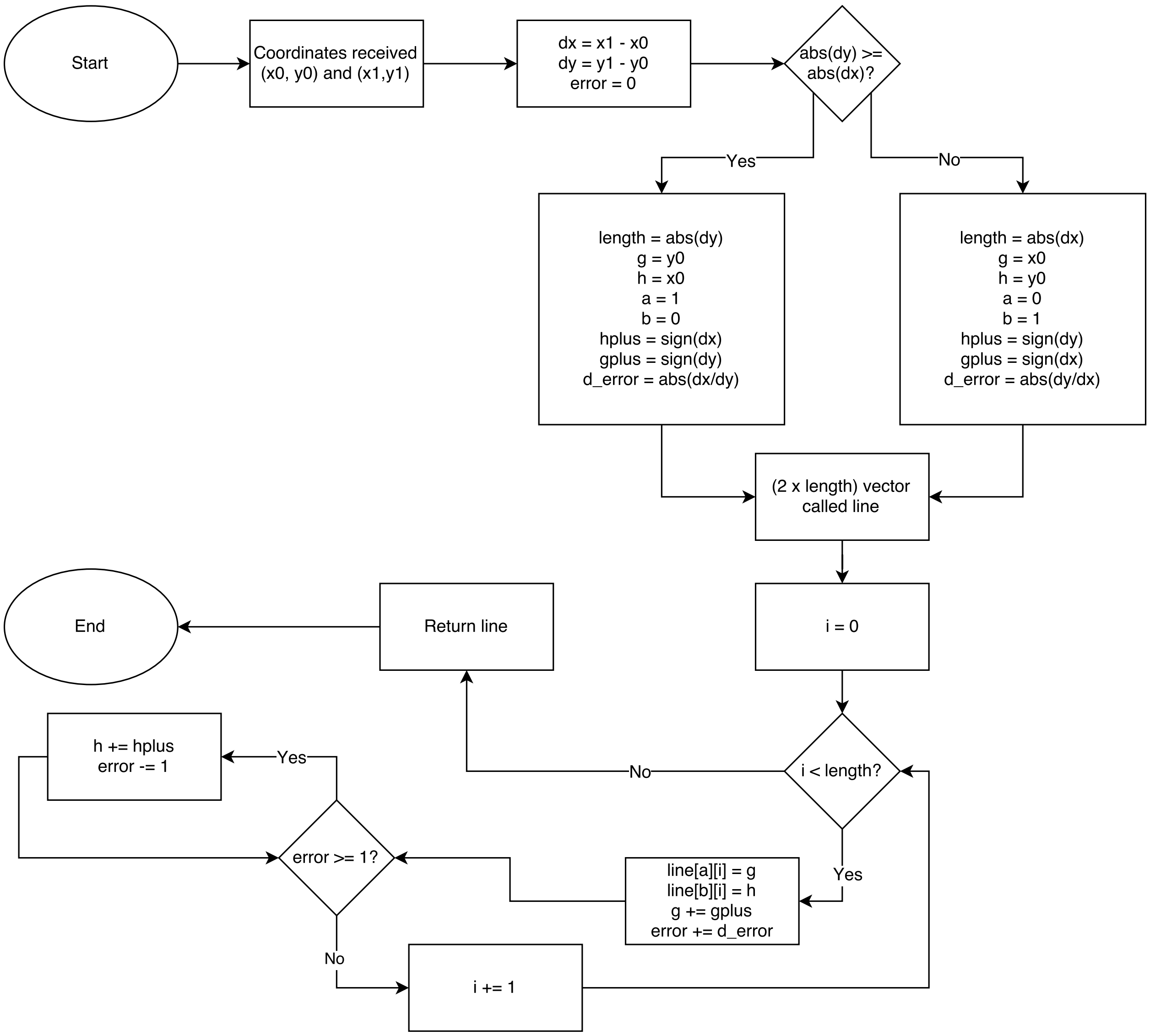


Figure 5.5: Flow Diagram of updated Bresenham Line Drawing algorithm. It is now able to deal with any value of dy and dx.

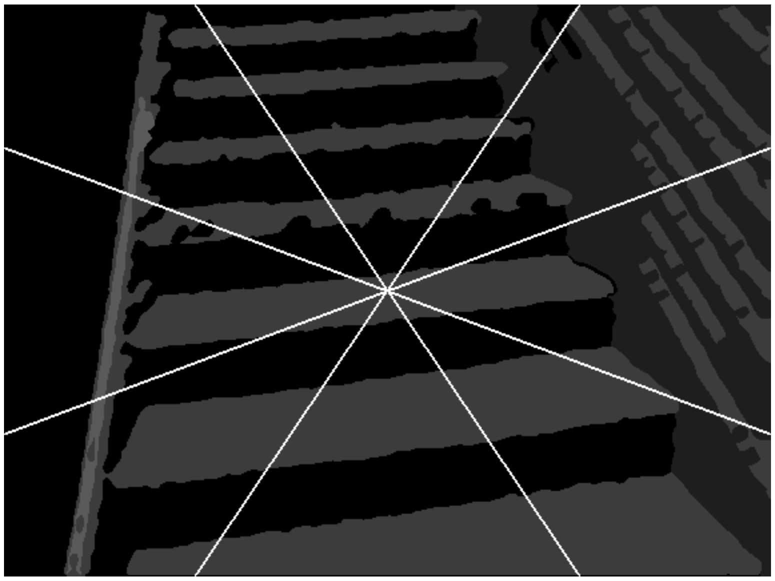


Figure 5.6: Showing all 8 possibilities for drawing a line

5.2 Classifying as chirp

Finally, it is time to start detecting staircases. For this, you need several identifiers. What constitutes what a staircase is, but also, what is a dead giveaway that something is not a staircase? The first part of that question deals with the possibility of type II error, or False Negatives (rejecting an image containing a staircase) and the second part deals with type I error, or False Positives (accepting an image that does not contain a staircase).

There are several of these identification rules to be thought of for stairs. For one, a staircase always has several horizontal surfaces with a similar orientation. The same can be said for closets or boxes stacked together, so this is not enough. Another phenomena that can be observed in repeating structures is the occurrence of a chirp-like repeatability, see Fig. 5.7. The chirp name refers to the waveform of a chirp from a bird, which has a similar shape. A more nuanced chirp could be projected on Fig. 4.7. Such a chirpwave can exist in the form of

$$f(x) = \sin ax^b \tag{5.2}$$

which is a simplified and adapted version of the standard sine-wave form in which it is ensured to have an increasing frequency for any $b > 1$, creating the wanted chirp-effect.

Now, how can we determine if there is a chirp-like spacing between the steps? If it is possible to fit a chirp-wave function to the spacing between the steps that would be sufficient proof. To do so we define a 2-dimensional function space called a Hilbert space.

5.2.1 Hilbert space

Hilbert space is in its simplest form an infinite-dimensional vector space of which the vectors satisfy several conditions: but that is not really interesting at this point. Although I should probably rewrite this to be more professional.

In our scenario we have two vectors, $\vec{a}$ and $\vec{b}$, that have finite length. For computational purposes, the domain of $\vec{a}$ is chosen to be $0 < \vec{a} <= 0.500$ in steps of 0.001, and the domain of $\vec{b}$ is chosen as $1 < \vec{b} <= 6$ in steps of 0.01. Since the vectors are not really infinitely dimensional, we should actually be referring to it as pseudo-Hilbert space. However, for the sake of simplification we will keep referencing towards it as Hilbert space. The Hilbert space $H_{a,b}$ can be seen in Eq.5.4. It is filled using

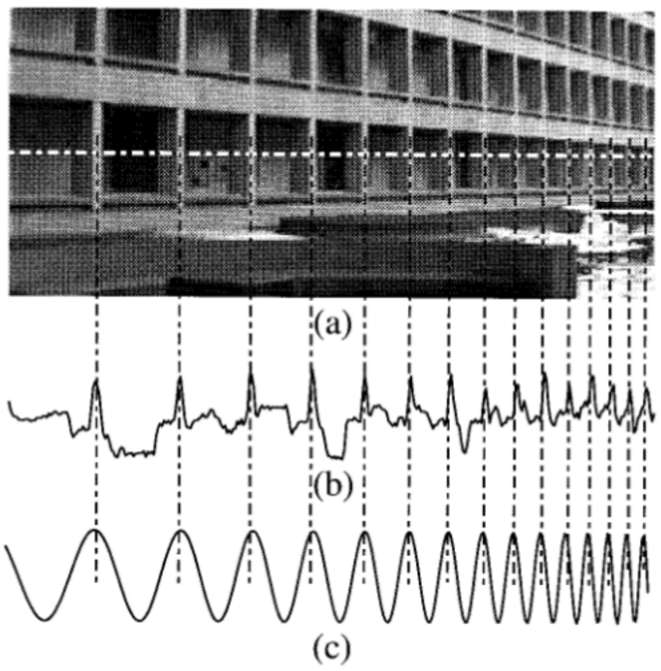


Figure 5.7: Example of a chirp-like repeatability in a real world setting[12]

$$H_{i,j} = \{0.001ix^{1+0.1j} \mid 1 \leq i \leq 500, 1 \leq j \leq 500\} \tag{5.3}$$

$$H_{a,b} = \begin{pmatrix} \sin 0.001x^{1.01} & \sin 0.001x^{1.02} & \cdots & \sin 0.001x^{6.00} \\ \sin 0.002x^{1.01} & \sin 0.002x^{1.02} & \cdots & \sin 0.002x^{6.00} \\ \sin 0.003x^{1.01} & \sin 0.003x^{1.02} & \cdots & \sin 0.003x^{6.00} \\ \vdots & \vdots & \ddots & \vdots \\ \sin 0.500x^{1.01} & \sin 0.500x^{1.02} & \cdots & \sin 0.500x^{6.00} \end{pmatrix} \tag{5.4}$$

Least Squares Error

Using an exhaustive search on the Hilbert space, the least squares error is computed for each point in the Hilbert space. The best fit is found for the point that shows the smallest error. If this error is below a certain threshold, it will be determined there is a chirp in the picture and thus a staircase. The fit starts at the first transition x_f, and ends with the last transition x_l, such that $f(x_f) = 1$. This means

$$f(x) = \sin\left(a\left(x - x_f\right)^b + \frac{\pi}{2}\right) \tag{5.5}$$

The error is calculated for every transition from a horizontal surface to a non-horizontal surface, and every point between two transitions. These transitions get a value of "1", and the points between two transitions get the value of "-1", since that is what we expect the sine-wave to look like as well. This also prevents a simple line of value "1" to hit all the transition points without actually showing a chirp. Each one of these individual errors are squared and summed together, then the square root is taken to give the total fit error E_f, see Eq. 5.6.

$$E_f = \sqrt{\Sigma\varepsilon_i^2} \tag{5.6}$$

Also, to ensure that the period of the fitted function and the period of the transition points is not far off, the difference in periods E_p is added to the total error. Otherwise, fitted functions of a frequency that is more than ten times the frequency of the transition points occur.

$$E_p = \left| \left(\frac{a(x_l - x_f)^b}{2\pi} \right) - (S-1) \right| \tag{5.7}$$

where S is the amount of transition points (or "steps") and it is subtracted by 1 since you should have one period between 2 steps. Two periods between three steps, etc.

Aside from this, there should also be a difference of one period between each peak. Otherwise you get situations where 3 peaks follow each other so rapidly they still fall in the same peak for the fitted function, followed by 2 empty fitted peaks. The algorithm then sees that there are 3 peaks and 3 fitted peaks, and the peaks all correspond with a value of '1' for the fitted function. Thus: no error. But the reality is of course that these peaks have been terribly mis-fitted.

The total error E_T is then found with

$$E_T = E_f + E_p \tag{5.8}$$

Visualization Hilbert Space

The total error E_T is calculated for each element in the Hilbert Space. These errors can be visualized using the a and b index as x and y, where the errors are represented by the values of the pixels. One would expect a well-fitted function to be increasingly worse-fitted the further it gets away from its perfect-fit point. It is expected to see a gradual change from black/dark grey to light grey as the distance from the best-fitted pixel increases. If the changes in values are very random, the fit is not very solid in the first place. An example of this visualization can be seen in Fig. 5.8. It can be seen that there is a certain region through which the combination of a and b result in a small error, as displayed via the darker color of the pixels. It can also be seen that as the pixels travel further away from the correct fit, they become lighter, as expected. At some point, the errors become larger very quickly, due to the x^b relationship, which then shows in a complete white region, which means $E_T > 255$.

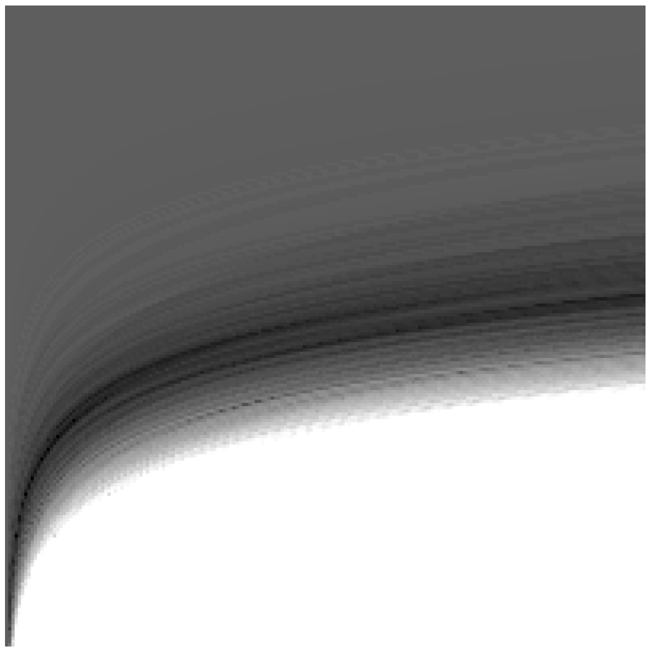


Figure 5.8: Hilbert visualization for one random line

To find the best chirp-fit, we run a minimization algorithm on the Hilbert Space, such that

$$E_T(x, y) = \min(E_T(a, b)) \tag{5.9}$$

The chirp function based on these coordinates is then deemed the best fit. An example of a fitted function can be seen in Fig. 5.9.

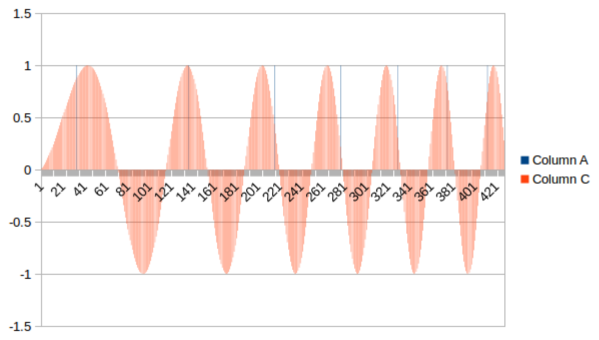


Figure 5.9: Example of a chirp function fitted to the transition points of a staircase

The error is then used to classify the image based on probability. A classification-value between 0 and 1 is given based on the error, via

$$P(x) = \frac{1}{\exp(\frac{E_T}{v})} \tag{5.10}$$

where v is a parameter that can be tweaked for a better fit on all training samples. Currently the value is $v = 5$. It can be seen that as the total error E_T decreases, the probability that the image is classified as a staircase $P(x)$ increases. Then, a random number is obtained between 0 and 1 as well. If this random number is smaller than the classification-value, the image is deemed to be a staircase.

5.3 AdaBoosting for final classification

Now the random line that gave us the transition points might not be a good choice for the particular image. If it comes in from the side it will only see one step and will not be able to fit a stair-like chirp. This cannot be solved by hard-coding the line to travel from the top to the bottom of the image, since you do not know the orientation of the image. To account for this, we draw multiple random lines and rely on a technique called AdaBoosting, where rules for classification get weights based on their effectiveness in correctly classifying the object or scene[26]. In this case, each line is a rule. A training dataset of equal size for positive and negative examples is used to train the rules and give each rule a weight. The weight for each rule gets multiplied with the classification outcome, which can be either 1 or -1, and the final classification is based on the total sum of these weighted rules. If the sum is positive it is a staircase, if it is negative it is not. The mathematical equation looks like

$$H(x) = sign\left(\sum_{t=1}^{T} a_t h_t(x)\right) \tag{5.11}$$

and

$$\text{Image}(x) \begin{cases} \text{staircase}, & \text{if } H(x) \geq 0 \\ \text{not a staircase}, & \text{if } H(x) < 0 \end{cases} \tag{5.12}$$

where $H(x)$ is the total weighted sum for image $x \mid x = 1, .., m$, a_t is the weight for rule t, $t = 1, 2, ..T$ and the $h_t(x)$ is the classification for image x using rule t. The weight for each rule is inversely proportional to the error rate in classification using this rule, see Eq. 5.13.

$$a_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right) \tag{5.13}$$

where ϵ_t is the error rate. It can be seen that as the error rate reaches zero, the weight goes to a relatively large positive number. As the error rate reaches 0.5 the weight is 0. Since the results of this rule were exactly the same as random chance, it gets no weight. Finally, as the error rate goes below 0.5 (worse than random chance) the weight becomes negative. The rule is then so bad at classifying things correctly you can get correct classifications doing the opposite of what it says. The error rate for the first round is obtained with

$$\epsilon_1 = \frac{1}{m}\sum_{x=1}^{m}\eta_1(x)\begin{cases}\eta_1(x) = 0, & \text{if } h_1(x) = y(x)\\ \eta_1(x) = 1, & \text{if } h_1(x) \neq y(x)\end{cases} \tag{5.14}$$

Some images will be easy to classify, and maybe 1 or 2 rules already give a clear classification. Others will be more difficult. The algorithm is designed such that it will spend more time fitting the harder images. For this, a distribution is used and updated between each round. At the start, the distribution is equal for every image, which is

$$D_1(x) = \frac{1}{m} \tag{5.15}$$

for every x, so that

$$\sum_{x=1}^{m} D_t(x) = 1 \tag{5.16}$$

The error rate for every round is then found with

$$\epsilon_t = \sum_{x=1}^{m} D_t(x)\eta_t(x)\begin{cases}\eta_t(x) = 0, & \text{if } h_t(x) = y(x)\\ \eta_t(x) = 1, & \text{if } h_t(x) \neq y(x)\end{cases} \tag{5.17}$$

where $D_t(x)$ is based on the error rate of the previous round via

$$D_{t+1}(x) = \frac{D_t(x)\exp(-a_t y(x) h_t(x))}{Z_t} \tag{5.18}$$

Z_t is a normalization factor which ensures $\sum_{x=1}^{m} D_{t+1}(x) = 1$ e.g. that D_{t+1} is a distribution. Thus

$$Z_t = \sum_{x=1}^{m} D_t(x)\exp(-a_t y(x) h_t(x)) \tag{5.19}$$

en $D_{t+1}(x)$ becomes

$$D_{t+1}(x) = \frac{D_t(x)\exp(-a_t y(x) h_t(x))}{\sum\limits_{x=1}^{m} D_t(x)\exp(-a_t y(x) h_t(x))} \tag{5.20}$$

The AdaBoosting algorithm is thus as follows:

- Set distribution of first round $D_1(x) = \frac{1}{m}$
- Use simple rule to classify images
- Calculate error rate ϵ_t using distribution, classification and labels
- Calculate rule-weight a_t using error rate
- Calculate new distribution $D_{t+1}(x)$ using rule-weight, classification and labels
- Repeat for each rule
- Finally, come to a total classification using the sum of the weighted rules

These steps are visualized in the Flow Diagram in Fig. 5.10.

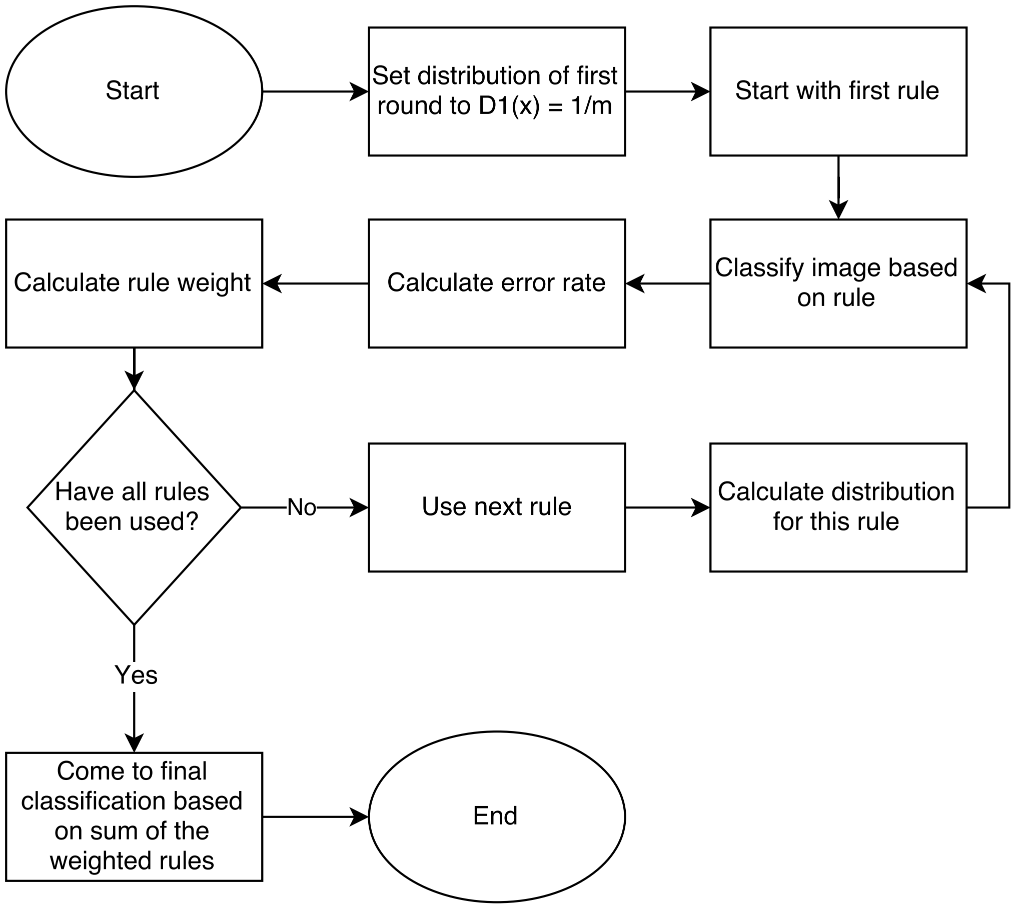

Start
Set distribution of first round to D1(x) = 1/m
Start with first rule
Classify image based on rule
Calculate error rate
Calculate rule weight
Have all rules been used?
No
Use next rule
Calculate distribution for this rule
Yes
Come to final classification based on sum of the weighted rules
End

Figure 5.10: Flow Diagram of the AdaBoost part of the final algorithm.

6 Different phases of algorithm development

This section discusses the changes the algorithm went through, from the start to the last update. It illustrates how the algorithm has logically progressed over the course of the project, and shows what is still to be done.

6.1 A simple Stepcounter algorithm, without chirpfitting or boosting

The first version of the algorithm did not incorporate chirpfitting or a boosting method. It would draw a random line through the image and read out the values. Whenever a change in value occurs, either positive or negative, it counts as a transition. Once more than a certain number of transitions have been counted, the algorithm declares that it has found a staircase. Let's call this number of transitions t. This happens for a number of lines. Naturally, from an accuracy point of view this should be done with as many lines as possible. But from a time-perspective the lowest amount of lines possible are preferable. To find the most optimal balance for the number of lines, a test was done using 20 lines and 100 lines, varying the number of detections. Since it followed from the results in Fig. 6.1 and Fig. 6.2 that the best score is found at a relatively similar detection threshold for both 20 and 100 lines, it was decided that 20 lines should be a large enough sample-base for a correct detection. Calculation time was not yet an issue at this stage, but it was projected that it could become a bottleneck later on as the algorithm development progressed.

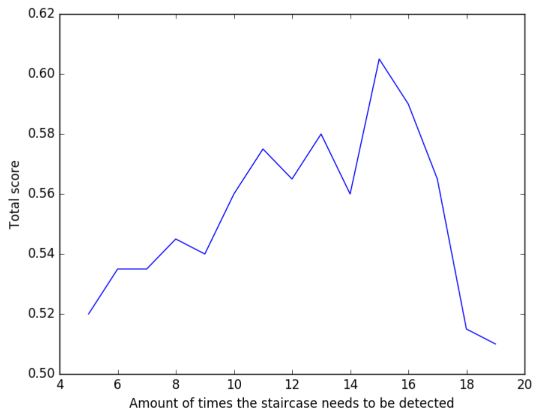


Figure 6.1: The effect of varying the detection threshold on the total score for 20 lines.

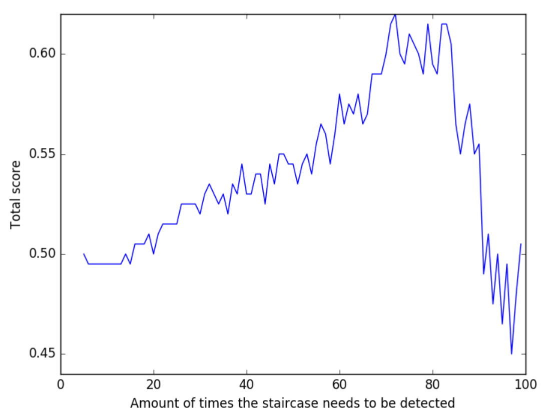


Figure 6.2: The effect of varying the detection threshold on the total score for 100 lines.

From these 20 lines, there need to be at least a certain number of detections to show a staircase is in the image for the classifier to classify the image as a staircase. Let us call this number of detections d. To find out what the best combination of transitions and detections thresholds was, the code was run while varying the number for both the required transitions and detections between 5 and 20. The result can be seen plotted as a 3D-plot in Fig. 6.3 and as a heatmap in Fig. 6.4. The best total score can be found at (10,15), so a threshold of 10 for d and 15 for t. Using these values as input for the algorithm, the following results are obtained, see Table 6.1.

This was just slightly better than random guessing, with a total score of 0.62. There were too many negative images that incorporated random transitions for the algorithm to accurately predict a staircase based on just the number of transitions. A more robust solution would incorporate the spatial aspect of the

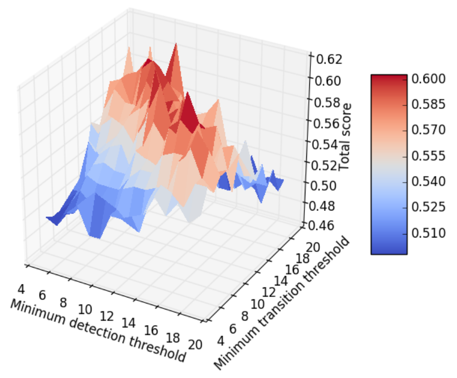


Figure 6.3: The 3d graph of the effect of the step and detection threshold on the stepCounter algorithm. Best score is 0.62 which is found at minimum detection threshold = 10 and minimum transition threshold = 15.

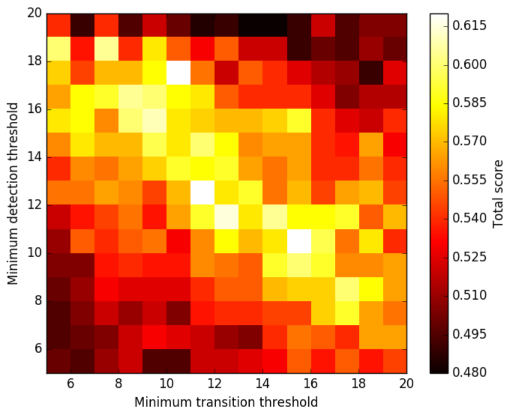


Figure 6.4: Heatmap of the effect of the step and detection threshold on the stepCounter algorithm. Best score is 0.62 which is found at minimum detection threshold = 10 and minimum transition threshold = 15.

Table 6.1: Results of using the stepCounter algorithm, without adjustments

FP	0.5
FN	0.26
TP	0.74
TN	0.5
total score	0.62
time spent	0m47s

transitions in the algorithm. That is why a chirp was fitted for more accurate results.

6.2 Chirpfitting without boosting

The fitted chirpfunction should incorporate both the number of transitions as well as the spatial information of these transitions. See Sect. 5.2 for a more detailed overview of what chirpfunctions are and why it is hypothesized to work in this scenario. The chirpfunction that needed to be fit is of the form

$$f(x) = \sin a(x-c)^b \tag{6.1}$$

The best fit will be found by iterating over the a, b, and c values. To save on computation time, it was assumed that the best starting point for the fitting would be at the first peak, and thus at the first transition point. This means that f(x) should be equal to 1 at the first transition point x_f. This is ensured by adapting the equation to

$$f(x) = \sin\left(a\left(x-x_f\right)^b + \frac{\pi}{2}\right) \tag{6.2}$$

Now iteration only needs to be done for a and b. It is assumed that there will be a maximum of 21 transitions that need to be fitted, since that translates to around 10 steps because every step should have two transitions, at the front edge and the rear edge. This leads to 20 periods that need to be covered. The maximum value of x is around 640 (pixels). Thus the range of a and b should be able to satisfy at least

$$a640^b = 40\pi \tag{6.3}$$

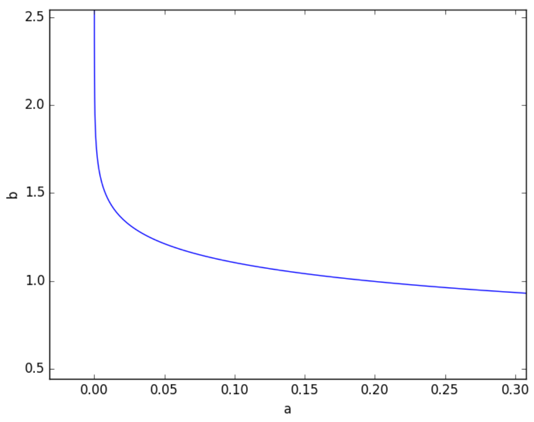


Figure 6.5: Plot of possible combinations of a and b to satisfy $a640^b = 40\pi$. Based on this plot a range of a = (0, 0.2) and b = (0.5,2.0) was chosen for the iteration.

Rewriting to $a = \frac{40\pi}{640^b}$ and plotting the result gives Fig. 6.5. Based on this plot a range of a = (0, 0.2) and b = (0.5, 2.0) was chosen for the iteration, using steps of 0.001 and 0.01 respectively.

After running the iteration the following figures were obtained, see Figs. 6.9, 6.10, and 6.11.

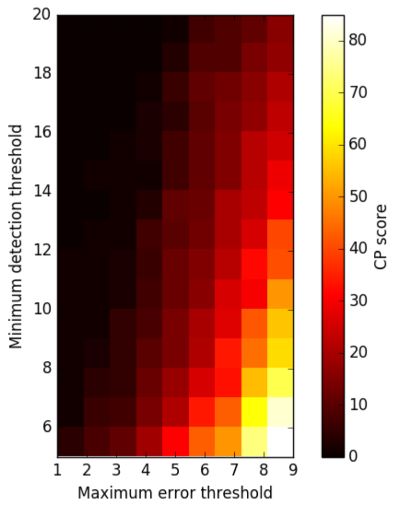


Figure 6.6: Heatmap of the effect of the step and detection threshold on the TP score of the chirpNoBoost algorithm. Best score is 85 which is found at minimum detection threshold = 5 and maximum error threshold = 9.

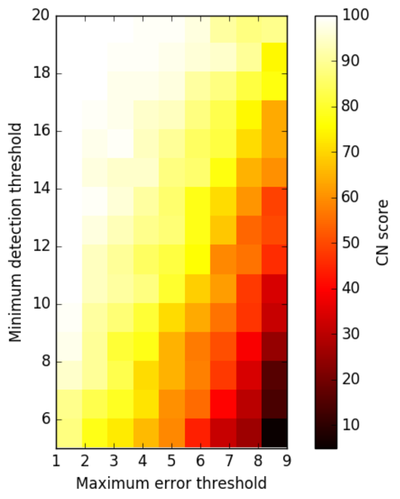


Figure 6.7: Heatmap of the effect of the step and detection threshold on the TN score of the chirpNoBoost algorithm. Best score is 100 which is found at minimum detection threshold = 10 and maximum error threshold = 1.

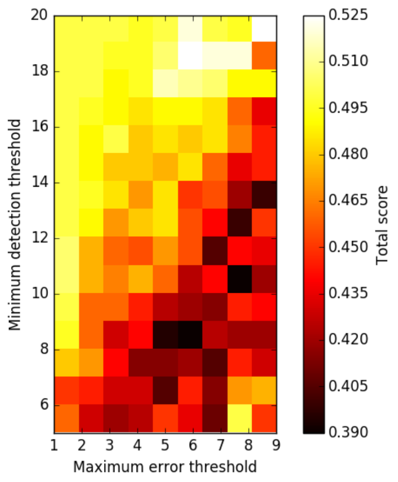


Figure 6.8: Heatmap of the effect of the step and detection threshold on the total score of the chirpNoBoost algorithm. Best score is 0.525 which is found at minimum detection threshold = 18 and maximum error threshold = 16. The corresponding values at that point are TP = 0.09 and TN = 0.96

As can be seen, the best total score is found at a point where the amount of correctly identified positive images is only 0.09 and the amount of correctly identified negative images are 0.96. This is not a well balanced result and it is clear the algorithm needs more tweaking. As it currently is, it is overfitted for negative images. The total results can be seen in Table 6.2.

Now, the first thing that needs to be addressed is the large computation time. Although a considerable

Table 6.2: Results of using the chirpNoBoost algorithm, without adjustments

FP	0.04
FN	0.91
TP	0.09
TN	0.97
total score	0.525
time spent	856m41s

amount of time is spent looking for the right combination of the error and detection threshold, it should be looked into if this can be improved. A proposed first improvement is to change the stepsize for the fitting of a from 0.001 to 0.01, and the stepsize of b from 0.01 to 0.1.

As can be seen by comparing the Tables. 6.2 and 6.3, using larger steps for the fitting process gives a significant improvement in running time (100 times faster), while having a negligible effect on the performance.

Table 6.3: Results of using the sped-up chirpNoBoost algorithm, without further adjustments

FP	0.03
FN	0.93
TP	0.07
TN	0.97
total score	0.52
time spent	8m58s

6.3 Chirpfitting using line-pairs

In the next version of the algorithm, a combination of line-pairs is used. This means, a random horizontal and a random vertical oriented line is drawn across the image. Naturally a good chirpfit should be found in only one of these lines, since they are perpendicular. Thus, they will be combined into line-pairs, such that if one of the lines finds a staircase, it counts as a staircase detection. This means that 40 random lines will be drawn resulting in 20 line-pairs. A further change in the algorithm is that the lines will be read out both ways. In the previous iterations of the algorithm, if the orientation of the image was such that the chirp would be found from the top going down, it would not be able to be fitted if the line was only read from the bottom going up. That is now prevented by reading the lines out both ways. The result is found in Figs. 6.9, 6.10 and 6.11, and Table 6.4.

Table 6.4: Results of using the sped-up chirpNoBoost algorithm, using horizontal and vertical line-pairs, reading the line out both ways

FP	0.84
FN	0.11
TP	0.89
TN	0.16
total score	0.525
time spent	32m20s

Unfortunately the results are again not great. The improvements on the previous iteration did not seem to have a great effect on the results, even though it logically should at least have some impact. Apparently, the way the algorithm is implemented, the factors that are thought of to have effect on the stairdetection might not actually be as relevant in its current form. The next section will look deeper into what is actually fitted based on the algorithm so far.

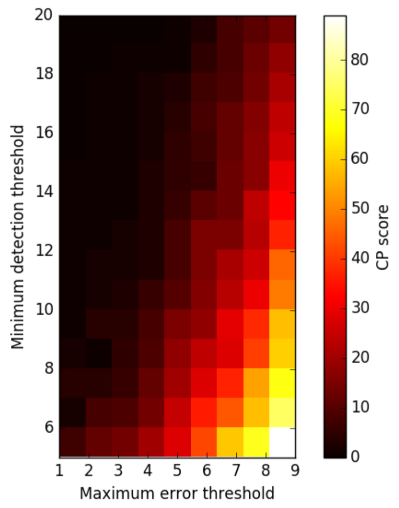


Figure 6.9: Heatmap of the effect of the step and detection threshold on the CP score of the chirpNoBoost algorithm. Best score is 85 which is found at minimum detection threshold = 5 and minimum transition threshold = 9.

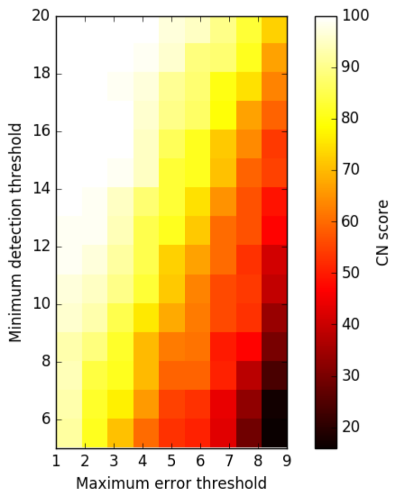


Figure 6.10: Heatmap of the effect of the step and detection threshold on the CN score of the chirpNoBoost algorithm. Best score is 100 which is found at minimum detection threshold = 10 and minimum transition threshold = 1.

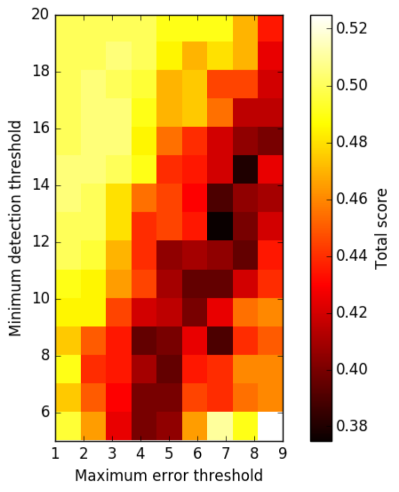


Figure 6.11: Heatmap of the effect of the step and detection threshold on the total score of the chirpNoBoost algorithm. Best score is 0.525 which is found at minimum detection threshold = 5 and maximum error threshold = 16. The corresponding values at that point are CP = 0.09 and CN = 0.96

6.4 Chirpfitting with penalties

The algorithm was clearly not performing as intended. A look into the the graphs of the chosen fitted function shows what went wrong, see Fig. 6.12. It shows a very large amount of periods in the plot; there are in fact so many periods that the plot has difficulty displaying them all. This is a reason to give penalties for large differences between the supposed number of periods (the amount of transitions minus one) and the actual total number of periods, as found by

$$P = \frac{a(x_l - x_f)^b}{2\pi} \tag{6.4}$$

where P is the number of periods, x_l is the x-coordinate of the last peak and x_f is the x-coordinate of the first peak.

However, only changing the algorithm to account for this is not enough, because there is a second phenomenon that can occur, which can be seen in Fig. 6.13.

As can be seen from the figure, the fit is off in this example. However, the algorithm in its current form sees nothing wrong with this. It looks at the peaks, and the valleys in between the peaks, and sees that they correspond to the fitted wave. It counts the number of periods and compares it to the number of peaks and sees that they are very close. Thus, the error is small and this is deemed a good fit. To prevent this, there should also be a penalty for the difference in actual period advancement between each peak and the projected period advancement (which is 1). This is calculated with

$$E_a = \sum_{n=1}^{t} \left(2\pi - [a(x_n - x_f)^b - a(x_{n-1} - x_f)^b]\right)^2 \tag{6.5}$$

where x_n is one of the transition points on the line. The errors are summed and added together to get the error of transition spacing, E_a.

The total error is then gotten with

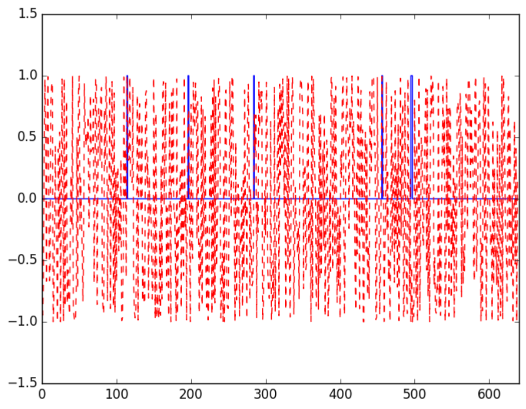


Figure 6.12: Example of a chirpfitting with a very large amount of periods, erroneously determined to be the best fit.

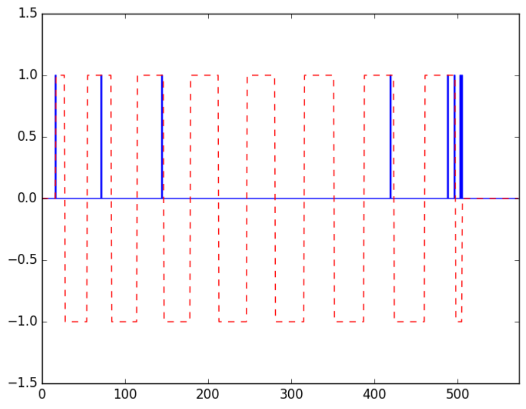


Figure 6.13: Example of a bad fit being classified as good, due to bad spacing between each period.

$$E_T = \sqrt{E_f^2 + E_p^2 + E_a^2} \tag{6.6}$$

Implementing this in the algorithm gives the following results, see Table 6.5.

Table 6.5: Results of using the sped-up chirpNoBoost algorithm, using horizontal and vertical line-pairs, reading the line out both ways, adjusting for the periods.

FP	0.02
FN	0.92
TP	0.08
TN	0.98
total score	0.53
time spent	73m19s

The results are still not up to standards. As can be seen from the results, the algorithm is currently way too strict. It rejects most of the images, resulting in many True Negatives, but many False Negatives as well. At least it rejects more Negatives than Positives, so perhaps if it is taught to be more lenient it allows more positives to pass while still catching the negatives.

6.5 Chirpfitting with normalized error

As the algorithm is currently functioning, it favors images with smaller number of transitions. This is because every transition is a possibility for an error to accumulate. However, in reality you would want many transitions since that is quite typical for a staircase. To therefore remove the penalties that many transitions give, the error is normalized by dividing the error over the number of transitions, see Eq. 6.7.

$$E_T = \frac{\sqrt{E_f^2 + E_p^2 + E_a^2}}{S} \tag{6.7}$$

where S is the number of transitions.

See Table 6.6 for the results. There is a clear improvement but it still has difficulty correctly identifying positive images.

Table 6.6: Results of using the sped-up chirpNoBoost algorithm, using horizontal and vertical line-pairs, reading the line out both ways, adjusting for the periods and normalizing the error.

FP	0.03
FN	0.83
TP	0.17
TN	0.97
total score	0.57
time spent	76m02s

6.6 Chirpfitting with boosting

The results mentioned in Sect. 6.5 indicate a weak classifier for staircase detection. Luckily, this weak classifier can be boosted and trained to perform better. Using the AdaBoosting method discussed in Sect. 5.3 every line-pair used functioned as a classification rule, totalling to 20 rules to be boosted. The results can be seen in Table 6.7.

Table 6.7: Results of using the chirpBoosting algorithm, using horizontal and vertical line-pairs, reading the line out both ways, adjusting for the periods and normalizing the error.

FP	0.62
FN	0.11
TP	0.89
TN	0.38
total score	0.64
time spent	9m52s

The results are better than they were before, but they are still far from acceptable when it comes to accurately detecting staircases. It seems the algorithm does not have an issue with detecting positive images anymore, in fact it looks like it is now slightly skewed to being too positive when classifying images.

7 Results

The algorithm as discussed in Sect. 5.3 outputs a classification for each image. This is compared with the actual classification using the labels of the images. The weights for each rule has been trained using a training data-set of 100 positive images (stairs) and 100 negative images (no stairs). After the training is completed, the weighted rules are then used to classify a testing data-set of 20 positive and 20 negative images. The errors for the different phases of the stair-detection algorithm can be seen in Table 7.1.

Table 7.1: Results of the different phases of the stair-detection algorithm compared with other stair-detection algorithms. The acronyms are explained in Table. 7.2.

	sC	cF	cFLP	cFLP3	cFLP3N	cFLP3NB	Wang[8]	Lee[9]
FP	0.50	0.03	0.84	0.02	0.03	0.62	0.00	0.23*
FN	0.26	0.93	0.11	0.92	0.83	0.11	0.03	0.30*
TP	0.74	0.07	0.89	0.08	0.17	0.89	0.97	0.70*
TN	0.50	0.97	0.16	0.98	0.97	0.38	1.00	0.77*

* Approximated from recall vs precision plot

Table 7.2: Abbreviations of algorithms explained.

sC	stepCounter
cF	chirpFit
cFLP	chirpFit with line-pairs
cFLP3	chirpFit with line-pairs and period penalties
cFLP3N	chirpFit with line-pairs and period penalties, with a normalized error
cFLP3NB	boosted chirpFit with line-pairs and period penalties, with a normalized error

As can be seen the algorithm under-performance when compared with similar algorithms. It is slightly outperformed by Lee et al, but dwarved by the results of Wang et al. It is interesting to see that the first version of the algorithm, which was not reliant on chirpfitting and boosting, is not that far off in performance compared to the final algorithm. This probably has to do with the fact that the first algorithm is a much simpler algorithm which has already been optimized, while the final algorithm still requires more development to obtain good results.

8 Conclusion & recommendations

In this thesis we looked into detecting staircases using recurring physical traits in depth images. Using RGB-D images as input, these images were preprocessed by calculating surface normals to extract surfaces. These surfaces where blurred to remove most of the noise and divided in 8 bins based on their RGB values. They were subsequently floodfilled to remove the remaining 'lone spots' in the image. Using the processed images, 20 random lines were drawn, using the same 20 coordinate-pairs for every image. A vector of transition points was collected along the pixels of the line, and a chirp-signal based on a sine-wave was fitted to this vector. If this was possible with a small error, a staircase would be declared detected. Using adaBoost, every line was evaluated for its effectiveness on the stair-detection, which was finally added together for a final classification.

Doing so, it was able to get a score of 62.3%. This is not a good performance compared to the other algorithms in Table. 7.1. Intuitively, the algorithm seems solid. A staircase is in essence just a collection of simple horizontal lines, and adaBoosting is very effective in taking simple rules and creating a well-performing algorithm. There are a number of possibilities that could explain the bad performance:

1. AdaBoost is not actually that good in boosting simple rules to a high performance classification
2. Staircases are not as easily defined by simple rules as initially thought
3. When implementing the algorithm, some vital details were not handled adequately.

Possibility 1 is not very likely. The following papers show excellent results using adaBoosting to increase the performance of simple classification rules, Schapire [26], Schapire et al [27], Schwenk and Bengio [28]. Error rates converge to near-zero after only a few rounds.

Possibility 2 could have some merit. A staircase is in fact more than just a collection of horizontal lines. It also contains the relative position and orientation of those lines, a certain number of those lines, but also contextual clues that help us humans recognize a staircase as a staircase. However, Wang et. al [8] and Lee et. al [9] did manage to obtain reasonable results using simple classification rules; Wang used a line-detector, filtering out lines that were not parallel or that were not long enough. Lee used Haar-like features in a cascaded adaBoosting trainig, and combined that with a temporal consistency-check to filter out False Positives.

The algorithm discussed in this thesis consists of finding transition points, or edges, and drawing a random line through them. The transition points on this line should have a positional spacing similar to that seen in a staircase, where the distance between the steps decreases as you look further away from your own point of view. These lines are then boosted with the adaBoost algorithm.

The algorithm actually performs quite well on the True Positives part: 89% versus 97.2% from Wang [8] and $\tilde{7}0\%$ from Lee [9]. However, it has a problem with False Positives, scoring 62% vs 0% from Wang and 23% from Lee. As discussed before, both their algorithms incorporated a form of filtering out False Positives (minimal line length, temporal consistency) which seems to be the main reason for them performing so well. The random line drawing technique relies only on non-related points, which means there is room for noise to affect the outcome. The idea was that by adaBoosting the multitude of lines, the randomness would be filtered out or rendered ineffective, but that does not seem to be the case.

Possibility 3 is also likely to be a factor. There are a couple of aspects that could be handled better by a future researcher:

- In the algorithm, a certain threshold is used for the error. Below this threshold the image is determined to be a staircase, and above the threshold it is determined to not be a staircase. However, the whole algorithm is based on robustness against small deviations. This is why many random lines are drawn and why these lines are boosted, such that a small change in angle or spots on the image should not matter. However, the error threshold reduces the outcome at the lower level back to a boolean

True/False result. This could lead to round-off errors in the total result. It would be better to use the errors to indicate a likelihood of being a staircase or not. The lower the error, the more likely it is to be a staircase and vice versa.

- As mentioned in Possibility 2, the method as it currently is might also be more susceptible to noise. The training data was difficult to obtain and there was not a wide variety of choice, both with respect to quality and sources. There were a large amount of images that preferably would not be used but that would leave a data-set that is too small for training anything effectively. A possibility could be for a future researcher to collect his own RGB-D images such that he can guarantee its quality and diversity.
- The algorithm currently uses every transition-point to fit a chirp. However, every step has two transition-points, at the front and at the back. This makes it difficult for the chirp to be fitted since there are actually two chirps taking place at the same time. These should be separated from each other for better results.

Ultimately, it seems to be a combination of both 2 and 3 that cause the algorithm to under-perform on the False Positive aspect.

The main question of this thesis was: is it possible to detect a staircase using simple rules? This question was guided by the following sub-questions, which have been answered during this thesis.

- **What are the recurring physical traits for a staircase?** Edges, represented by long lines or points that are related in position and orientation.
- **How can these traits be recognized?** There are a multitude of options, surface segmentation, Sobel Edge detector, Haar-like features, etc.
- **How can these traits be combined to accurately classify a staircase?** Using a boosting algorithm that favors strong traits and penalizes weak traits.

Which leaves the main question: is it possible to detect a staircase using simple rules?
As it currently stands, this is not possible in a reliable manner. The current detection rate of 62.3% is not great, but there is clear room for improvement. The main reasons for this have been outlined above, being a combination of not filtering the False Positives adequately and forcing a boolean classification on a low level, instead of a floating classification.

Furthermore, a more clear and larger data-set should be obtained, or created yourself. There was a lot of noise on the data-set used, which meant there were a lot of transitions not accounted for in the correct chirp-fits.

It would probably also need some improvements to make it robust against similar structures, like book-cases, since the detection makes use of the repeatability in the structure of staircases. It might also have problems when confronted with structures outside like fences, large flats and train tracks or cross walks.

Aside from using this method for detecting stairs, this method can also be used for detecting other objects based on simple rules and a certain repeatability of structure, like the similar structures mentioned before. It could also be used to detect man-made structures from satellite images of uninhabited places, since man-made structures often adhere to a certain repeatability due to its efficient nature.

A C++ implementations

In this chapter, the various implementations of algorithms in C++ are shown, for completeness.

A.1 Bitmask for segmenting

```
Mat clust(nor.size());
int bin = 127;
for (int x = 0; x < clust.cols; x++){
        for (int y = 0; y < clust.rows; y++){
                clust.at(y,x) = 0;

                if (nor.at(y,x)[0] > bin) {
                        clust.at(y,x) += 1;
                }
                if (nor.at(y,x)[1] > bin) {
                        clust.at(y,x) += 2;
                }
                if (nor.at(y,x)[2] > bin) {
                    clust.at(y,x) += 4;
                }
        }
}
```

A.2 Floodfill algorithm

```
Floodfill (segment, oldColor, newColor, x, y){

    if (segment[y,x] == oldColor){
        segment[y,x] = newColor;
        for (int j = -1; j < 2; j++){
            for (int k = -1; k < 2; k++){
                if (j != 0 || k != 0){
                    Floodfill(segment, oldColor, newColor, x+j, y+k);
                    }
                }
            }
        }
    return;
}
```

A similar algorithm is used for calculating the area of these sections but instead of coloring the pixel it adds 1 to a counting variable, such as

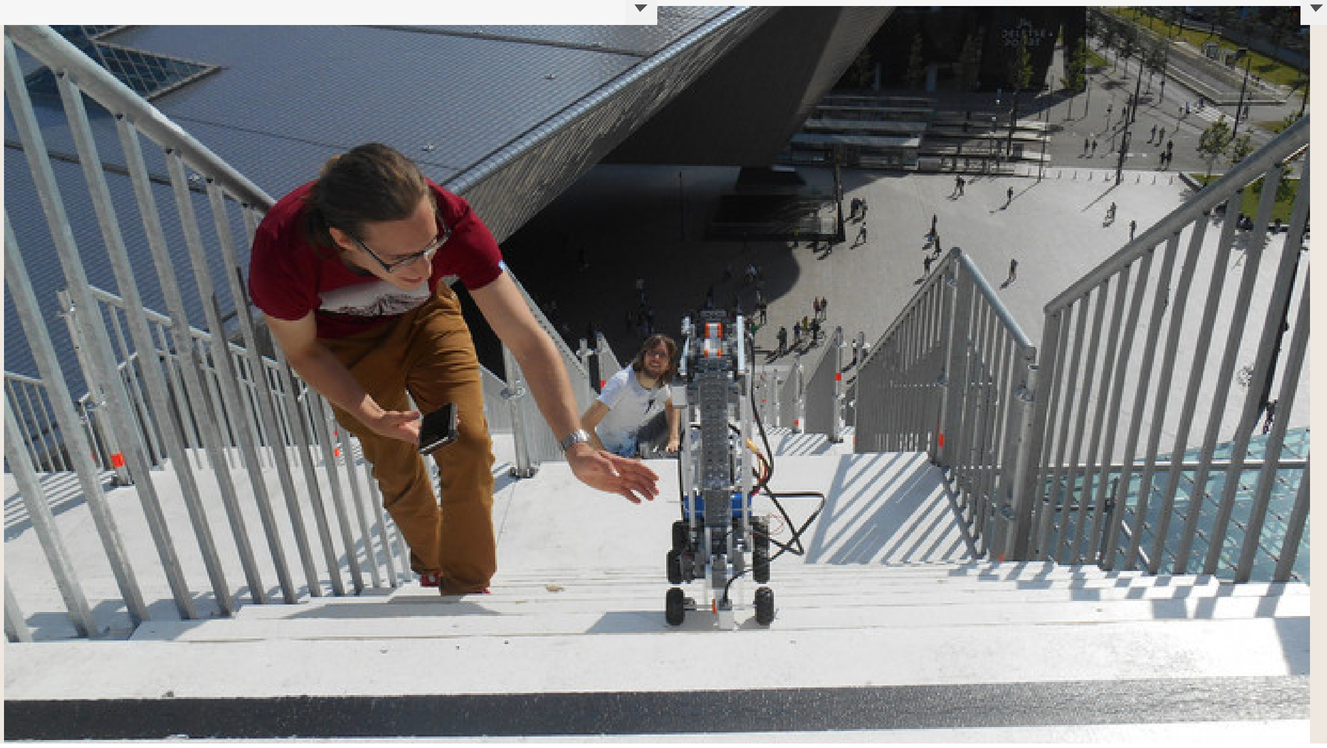
```
Floodfill (segment, color, x, y, i){

    if (segment[y,x] == oldColor){
        i++;
        for (int j = -1; j < 2; j++){
            for (int k = -1; k < 2; k++){
                if (j != 0 || k != 0){
                    Floodfill(segment, oldColor, newColor, x+j, y+k);
                    }
                }
            }
        }
    return i;
}
```

A.3 Updated Bresenham line drawing algorithm

```
vector<vector<int> > random_line_all_points (Mat matrix,
int x0, int x1, int y0, int y1){
        comment("Running random_line_all_points function..");
        cout<<"x0 = "<<x0<<" | x1 = "<<x1<<" | y0 = "<<y0<<" | y1 = "<<y1<<endl;

        // catching any coordinate that is too large for the image
        y0 = min(y0, matrix.rows - 1);
        y1 = min(y1, matrix.rows - 1);
        x0 = min(x0, matrix.cols - 1);
        x1 = min(x1, matrix.cols - 1);

        int dx = x1 - x0;
        int dy = y1 - y0;
        float error = 0.0;
        float d_error = 0.0;
        int length, g, h, a, b, hplusser, gplusser;
        if ( abs(dy) >= abs(dx) ){
                length = abs(dy);
                g = y0;
                h = x0;
                a = 1;
                b = 0;
                hplusser = sign(dx);
                gplusser = sign(dy);
                d_error = abs((float)dx / (float)dy);
        }
        else {
                length = abs(dx);
                g = x0;
                h = y0;
```

```
                    a = 0;
                    b = 1;
                    hplusser = sign(dy);
                    gplusser = sign(dx);
                    d_error = abs((float)dy / (float)dx);
            }
            //vector that will be returned
            vector<vector<int> > line(2, vector<int>(length) );

            //Actual line plotting
            for (int i=0; i < length; i++){
                    line[a][i] = g;
                    line[b][i] = h;
                    g = g + gplusser;
                    error += d_error;
                    while (error >= 1.0){
                            h = h + hplusser;
                            error--;
                    }
            }
            return line;
    }
```

B Article in the AD about stairclimbing robot


Student Frerik leidt de robot naar de top van de trap. © AD

Robot steelt de show op Rotterdamse supertrap

Robot Freddy heeft geen haast. De dag is nog jong, en de supertrap lang. Heel erg lang. Onder toeziend oog van zijn 'baasje' Frerik (24), student ruimtevaarttechniek aan de TU Delft, gaat het wieltje voor wieltje naar boven, waarbij het uit Lego opgebouwde apparaatje sfeervolle, snorrende geluidjes maakt. Eenmaal boven aangekomen, klinkt luid applaus. ,,Hij heeft het gehaald", roept iemand.

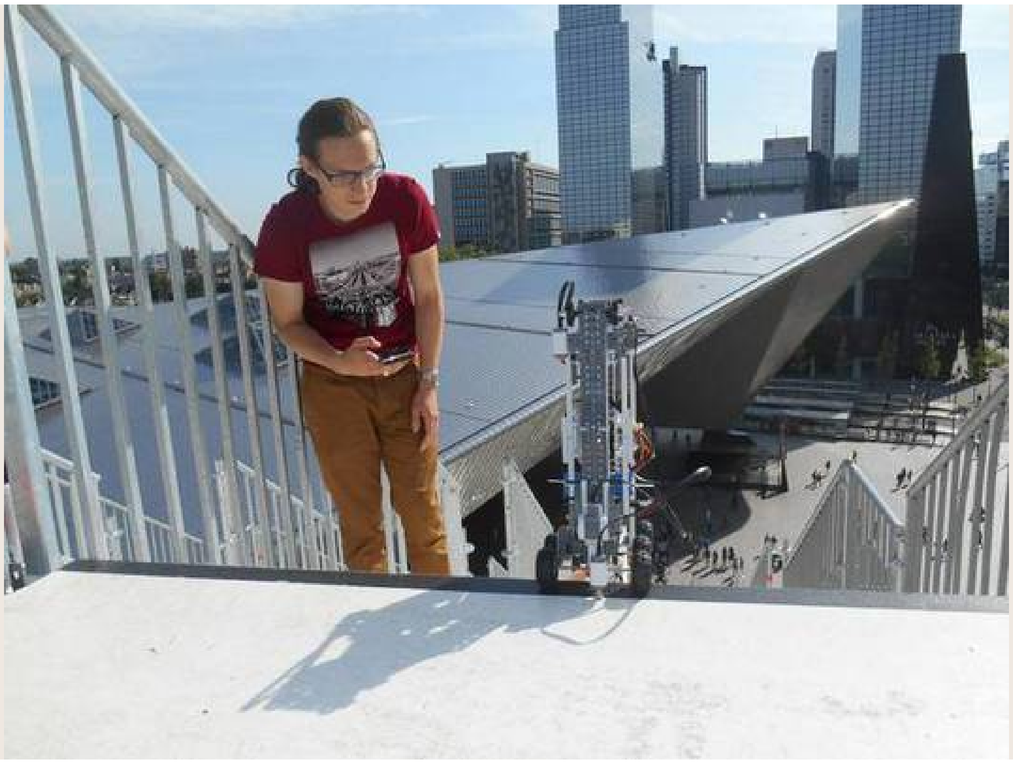
Marcel Potters 10-06-16, 11:29 **Laatste update:** 11-06-16, 09:02

22 13 2

Eerst heb ik 'm thuis uitgeprobeerd, maar toen ik hoorde van de megatrap in Rotterdam, heb ik

Feedback

meteen contact opgenomen



▲ © AD

Frerik is trots. Alsof hij zojuist een langharige teckel 'n boeiend kunstje heeft geleerd, showt hij zijn geavanceerde machientje aan het publiek. ,,Dit is mijn afstudeerproject", legt hij uit. ,,Hier ben ik al een jaar mee bezig. Ik heb Freddy in ongeveer twee maanden gebouwd. Eerst heb ik 'm thuis uitgeprobeerd, maar toen ik hoorde van de megatrap in Rotterdam, heb ik meteen contact opgenomen. Deze kans werd me op een presenteerblaadje aangeboden."

Ingewikkeld

De werking is niet zo ingewikkeld, benadrukt de techneut in spe. ,,De robot is zo geconstrueerd, dat hij zichzelf als het ware optrekt. Net zoals een rups zich voortbeweegt", klinkt het. En dit is nog niet het einde van het project. ,,Nu help ik 'm nog een handje, bestuur ik hem. Maar op den duur moet 'ie zélf naar boven kunnen."

Zijn droom? ,,Dat Freddy ouderen kan helpen, bijvoorbeeld met boodschappen doen. Hij moet dan wel een stuk groter worden en zeker 10 kilo aankunnen. Dát concept ga ik de komende tijd verder uitwerken."

Lees ook toon alles(4) +

Lingeriemodellen schitteren op de Rotterdamse Trap

Lees meer

Feedback

Bibliography

[1] GIJS BEETS. De geboortepiek van 1946. *Demos*, 27(2):6–8, 2011.

[2] CBS. Bevolkingspiramide.

[3] Dirk Holz, Stefan Holzer, Radu Bogdan Rusu, and Sven Behnke. Real-time plane segmentation using RGB-D cameras. In *Robot Soccer World Cup*, pages 306–317. Springer, 2011.

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

[5] C. F. Chen and C. H. Hsiao. Haar wavelet method for solving lumped and distributed-parameter systems. *IEE Proceedings-Control Theory and Applications*, 144(1):87–94, 1997.

[6] Hough Line Transform — OpenCV 2.4.13.2 documentation.

[7] Bernhard Scholkopf, Kah-Kay Sung, Christopher JC Burges, Federico Girosi, Partha Niyogi, Tomaso Poggio, and Vladimir Vapnik. Comparing support vector machines with Gaussian kernels to radial basis function classifiers. *IEEE transactions on Signal Processing*, 45(11):2758–2765, 1997.

[8] Shuihua Wang, Hangrong Pan, Chenyang Zhang, and Yingli Tian. RGB-D image-based detection of stairs, pedestrian crosswalks and traffic signs. *Journal of Visual Communication and Image Representation*, 25(2):263–272, 2014.

[9] Young Hoon Lee, Tung-Sing Leung, and Gérard Medioni. Real-time staircase detection from a wearable stereo system. In *Pattern Recognition (ICPR), 2012 21st International Conference On*, pages 3770–3773. IEEE, 2012.

[10] Yalin Xiong and Larry Matthies. Vision-guided autonomous stair climbing. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 1842–1847. IEEE, 2000.

[11] The Bresenham Line-Drawing Algorithm.

[12] Steve Mann and Rosalind W. Picard. Video orbits of the projective group a simple approach to featureless estimation of parameters. *IEEE Transactions on Image Processing*, 6(9):1281–1295, 1997.

[13] Robert M. Manning and Mark Adler. Landing on Mars. 2005.

[14] Wiley J. Larson and James Richard Wertz. Space mission analysis and design. Technical report, Microcosm, Inc., Torrance, CA (US), 1992.

[15] S. Alan Stern. The low-cost ticket to space. *Scientific American*, 308(4):68–73, 2013.

[16] Michael Simpson. Spin-Out and Spin-In in the Newest Space Age. 2010.

[17] Jeanie Kayser-Jones, Ellen Schell, William Lyons, Alison E. Kris, Joyce Chan, and Renee L. Beard. Factors that influence end-of-life care in nursing homes: the physical environment, inadequate staffing, and lack of supervision. *The Gerontologist*, 43(suppl_2):76–84, 2003.

[18] Tiia Ngandu, Jenni Lehtisalo, Alina Solomon, Esko Levälahti, Satu Ahtiluoto, Riitta Antikainen, Lars Bäckman, Tuomo Hänninen, Antti Jula, Tiina Laatikainen, and others. A 2 year multidomain intervention of diet, exercise, cognitive training, and vascular risk monitoring versus control to prevent cognitive decline in at-risk elderly people (FINGER): a randomised controlled trial. *The Lancet*, 385(9984):2255–2263, 2015.

[19] Miriam E. Nelson, Jennifer E. Layne, Melissa J. Bernstein, Andrea Nuernberger, Carmen Castaneda, David Kaliton, Jeffrey Hausdorff, James O. Judge, David M. Buchner, Ronenn Roubenoff, and others. The effects of multidimensional home-based exercise on functional performance in elderly people. *The Journals of Gerontology Series A: Biological Sciences and Medical Sciences*, 59(2):M154–M160, 2004.

[20] Shu-Chuan Jennifer Yeh and Sing Kai Lo. Living alone, social support, and feeling lonely among the elderly. *Social Behavior and Personality: an international journal*, 32(2):129–138, 2004.

[21] Frerik Andriessen. Internship Report Stairclimbing Robot. Technical report, Delft University of Technology, Delft, 2016.

[22] Yung-Kai Lai and C.-C. Jay Kuo. A Haar wavelet approach to compressed image quality measurement. *Journal of Visual Communication and Image Representation*, 11(1):17–40, 2000.

[23] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft coco: Common objects in context. In *European Conference on Computer Vision*, pages 740–755. Springer, 2014.

[24] G. T. Shrivakshan, C. Chandrasekar, and others. A comparison of various edge detection techniques used in image processing. *IJCSI International Journal of Computer Science Issues*, 9(5):272–276, 2012.

[25] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[26] Robert E. Schapire. The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification*, pages 149–171. Springer, 2003.

[27] Robert E. Schapire and Yoram Singer. Improved boosting algorithms using confidence-rated predictions. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 80–91. ACM, 1998.

[28] Holger Schwenk and Yoshua Bengio. AdaBoosting neural networks: Application to on-line character recognition. In *International Conference on Artificial Neural Networks*, pages 967–972. Springer, 1997.