Magic Canvas

Art installation at Hjørring Library 2012

Gustav Dahl, 20113263
Johannes Møjen, 20113238
Maximilian Müller, 20113279
Marco Winther, 20080791
Marta Botella, 20116050
Simon Jakobsen, 20113295

Medialogy 3rd semester
Supervisor: Thomas B. Moeslund
Co-supervisor: Andreas Møgelmose
Aalborg University - December 19, 2012

AALBORG UNIVERSITET
STUDENTERRAPPORT

Medialogy Aalborg
Niels Jernes Vej 14 - 9220
Aalborg Øst
http://www.medialogi-

Title: Magic Canvas

Theme: Visual Computing

Project period: September-December 2012

Project group: MTA 12338

Participants:

Gustav Dahl

_____ _____

Johannes Møjen

_____ _____

Maximilian Müller

_____ _____

Marco Winther

_____ _____

Marta Botella

_____ _____

Simon Jakobsen

_____ _____

Supervisors:

Thomas B. Moeslund
Andreas Møgelmose

Abstract:

The theme for this project is Visual Computing. The group has been in collaboration with Hjørring Library in developing an installation called Magic Canvas. The goals are to make an entertaining installation that can be enjoyed by people of all ages. The Magic Canvas utilizes an infrared webcam to capture video of people casually passing by, displaying them on a big canvas as a Christmas-related character. To make the camera get a clear contrast, infrared LED strips are placed near the canvas. The software is mainly written in C++, using the OpenCV library to extract video information. The Unity game engine is then used for visualization. To detect people with the camera, various image processing techniques have been used, such as thresholding, background subtraction, region of interest and BLOB analysis. The program is working without any physical devices, and it does not require any external maintenance. The program ran throughout December 2012.

Circulation number: 4
Pages: 87
Attachments: 4
Finalized date: December 19, 2012

Table of content

Foreword

Images and videos have acquired a big prevalence in our daily lives. From the creation of the worlds first camera, the *Camera obscura*, to the creation of one of NASA's most successful science missions, the *Hubble Space Telescope* - it is clear that humans have always had the urge to capture images and use them for scientific and creative purposes. Being able to create machines that can see and differentiate objects in a similar fashion to humans, has revolutionized our lives. Something as simple as recycling bottles in a supermarket is made possible by programming and utilizing techniques from image processing. Doing this enables the bottle refund machine to recognize characteristics of the bottle, rejecting or accepting it depending on the information extracted. This happens all the time wherever we go, but we rarely notice it.

Magic Canvas is a 3^{rd} semester project at Medialogy, Aalborg University, that ran from September to December 2012.

Group MTA 12338 would like to thank Martin Jørgensen and Tone Lunden from Hjørring Library. We also want to thank our two supervisors: Thomas B. Moeslund and Andreas Møgelmose.

Introduction 1

One aspect of Medialogy is building systems that react accordingly to humans, to create a better human-computer confluence. For this semester project, we were asked to focus less on the problem analysis and more on producing an entertaining product or program in which to apply knowledge taught in the Image Processing and Procedural Programming courses.

Eleven topics were presented as project proposals:

- Image Processing for Fun utilizing an Industrial Robot
- Image Processing for Ambient Intelligent Robots
- Interactive Floor
- Interactive Book
- Interactive Drawing Game
- Interactive Arcade Game
- Emergency System for Old People that have fallen
- Thermal Sock Puppet Show
- Body Motion Controlled Exercise Game
- Mobile App for recognizing Electric Components
- Hjørring Library

After reading into the different project proposals, the group decided on the following three topics:

1. Hjørring Library
2. Interactive Floor
3. Image Processing for Ambient Intelligent Robots

To great delight, our group was chosen to work with Hjørring Library, which was the first choice.

During the following months, the group worked on the project, which later became the **Magic Canvas**. The name is inspired by the magic of Christmas and the fact that the final project is a canvas that reflects visitors walking and magically turns them into Christmas characters.

1.1 Establishing a collaboration with Hjørring Library

During the first visit to Hjørring Library, a short informational meeting was held by the staff, outlining the possibilities for the groups working at the library. Although there were

no predefined projects, the staff at the library expected to get an installation that the visitors could spend some time interacting with.

The first idea that emerged included scanning barcodes from books in the library. The idea was that people would explore the library and depending on the books gathered and scanned, some specific graphics would appear on a canvas as the visitors passed by. This could be a hat or other types of clothes related to the genre of the book that was scanned. An example could be scanning a fairy tale written by H. C. Andersen which would then produce a top hat on the visitors head, or maybe a book with Sherlock Holmes would spawn his famous hat and pipe, etc. This would allow people that casually walk past the shelves to see something interesting on the canvas. If they want to they could participate further by going around the library to find the correct books, scan them, and thereby change what would be projected on the canvas the next time. For instance, they could be told to find a book about Harry Potter and if they succeeded in scanning a book from the series, everybody walking past the canvas from now on would see themselves with a Harry Potter costume on (see figure 1.1). It seemed like a concept with endless possibilities.

Figure 1.1. Scanning a Harry Potter book would change the theme to magic.

After conferencing with the supervisors, it was decided that the concept needed to be narrowed down to something more specific. There were some uncertainties whether it would be possible to make a program that could track people and display graphics on top of them, as well as scan barcodes from books and extract useful data. Therefore it was suggested to only focus on the first part of the concept, so that instead of trying to do multiple themes we should focus mainly on one.

Another aspect that the group did not consider at first, was that the library has its own topics running for 8 to 12 weeks. The project should ideally fit into the current theme. By the time the project had to be delivered, the theme at the library was Christmas. To make the project fit, it was decided to work towards a Christmas theme.

Two members of the group then travelled to Hjørring to find a proper location for the

project. The priority was to find a spot where people would pass through naturally, instead of having to go look for the canvas. It was also necessary to check the light conditions and what equipment would be provided by the library. It was finally decided to position the installation on the walkway from the reception desk to the core of the library (see figure 1.2). This seemed to be an ideal position for the project as the lights were controllable. Also, the group was told that people tend to use that walkway either to enter or exit the library, as it is the shortest route.

Figure 1.2. A central location for the canvas was chosen at Hjørring Library.

1.2 Prototyping

The first step into developing the software was to write some initial code that would run and produce a useful output. This was done by dividing the group into two smaller groups of three persons, with the goal of coming up with a solution of how to track an object in a video. The two sub groups competed over a course of three days. The winner would be the group that came up with the best program. And so the winner's code, or parts from both projects, would be incorporated in a program that would be expanded upon in the following months.

At the end of the week, two programs were ready to be tested. Both programs were able to do the same things, only the code differed. Some of the techniques used were: median filtering, morphology (opening/closing), background subtraction, conversion from a color image to a grayscale image and a draft of the BLOB analysis to find the head. All of these concepts are going to be explained more in-depth in chapter 3.

The basic idea is to have a canvas on which the silhouettes of the visitors are projected. To achieve this it would be necessary to have the following:

- A device to gather information about people's position while walking past
- A computer to run the program code

- A projector to display the output from the computer
- A canvas on which the output is projected
- Controllable light conditions

1.2.1 Running the first test

As the first prototype of the program was completed, it was decided to test the code and the equipment. The test was performed to investigate the video output, the frame rate of the video and the threshold values. (see figure 1.3).

Figure 1.3. A prototype test using temporary equipment.

The group was provided with two Logitech Pro 9000 web cameras, one with an infrared filter and one without. Both were tested and the camera with the filter gave the best result. In addition to the two web cameras, four light bulbs were used to illuminate the scene. In order to achieve the biggest contrast between the background and the person, two tests were initiated. First the person was illuminated by the light bulbs and then it was tested to illuminate the background. The results proved that illuminating the background provided the greatest contrast, (see figure 1.4). One problem still remained: once the test person faced the screen, the head was too small and was not very precise on the output, producing a misplacement of the Christmas hat.

1.3 Changes on the main concept

After the first short testing session it seemed clear that some things needed to be clarified. For this reason, the group asked for a brief meeting with the supervisors to receive some valuable feedback on the prototype test. Further, the group wanted help with some

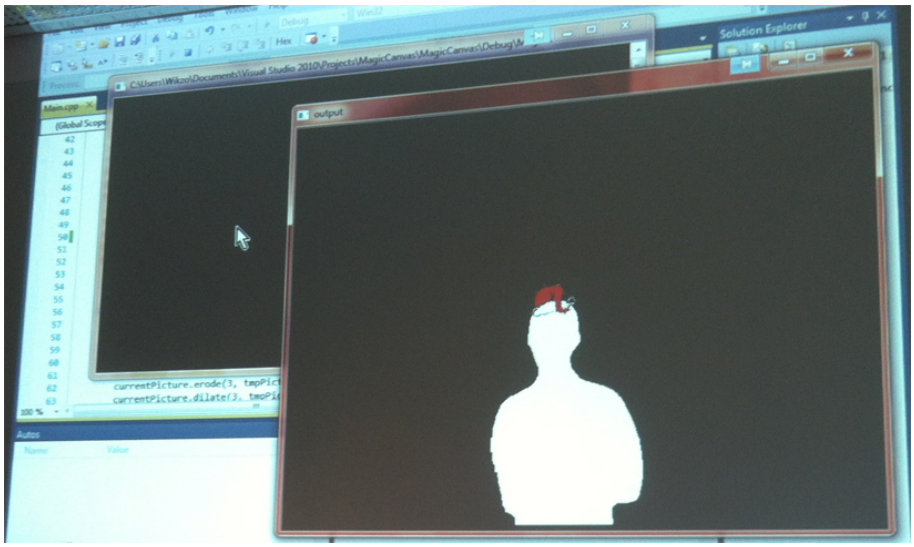


Figure 1.4. An image showing output from the prototype test projected onto a wall.

programming dilemmas regarding speed and efficiency looping through a large quantity of pixels in each frame. At first the program code was designed to look for the top-most row of pixels, resulting in several different problems. The persons standing in front of the screen could end up wearing the hat on his hand just by raising it. Another problem could be that somebody taller came into the picture, resulting in the taller person "stealing" the hat from the previous owner.

Although the group imagined the program to display the whole person in the output with a graphical hat on top, it was suggested to not show the person directly, but instead show a silhouette. Both approaches seemed to suffer from the same problem: the program would have to loop through the entire image to find the person and then do further computations. Even with only one person, as in the prototype test, this seemed to be a somewhat slow approach. Since the group wanted to track multiple persons, it would not work as intended.

This dilemma was brought up and various suggestions were given, such as looking only for the head instead of the full body, adjusting the BLOB analysis to look for a more specific shape, as well as some tips on how to optimize the background subtraction.

1.3.1 Using Christmas characters instead of showing the actual person

Instead of outputting the entire person and performing several image processing operations, it was chosen to limit the amount of data used to find the person. Instead of looking at the whole shape, the program should only look for the position of the person. Then it would display some pre-drawn characters underneath the head, i.e. a visitor at the library does not see himself, but a funny Christmas-themed character running around on the screen.

This would save a lot of calculations, as only the position of the person is needed to be found. This is expected to make the program react faster, which is important in order to create the proper immersion. That is why it is also relevant that the camera is quick enough to pick up and recognize new people stepping into the scene.

Vision and goals 2

2.1 Hjørring Library: a place for experiences

In this chapter we will investigate the target group for the project. A preliminary assessment will be made to gather information that will help towards making the program suit the target group best.

Hjørring Library aims to distinguish itself from most other Danish libraries. The first distinguishing factor is the location of the library: a big shopping mall. Inside, visitors find themselves in a big open hall with modern architecture. The different sections are divided by bookshelves and a red stripe that runs throughout the library (called "den røde tråd"). While the library first and foremost is a library as any other, one of the prevalent feelings that visitors get is that Hjørring Library is a place to "hang out" and relax (see figure 2.1), but also a place to have fun or do some work. There are many unique pieces of furniture, creating a great environment for all purposes. Especially kids can have a fun time playing games (physical or digital) and recording videos in the children zone (see figures 2.2 and 2.3).

At Hjørring Library the staff has made a big effort to create an environment as interactive as possible, with many places for people to entertain themselves with other things than just traditional books. Even the cafeteria appears to be part of the library's ordinary environment where people can show up, eat their lunch, drink coffee or just have a beer after work.

Figure 2.1. Lounge.

Figure 2.2. Playing games.

Figure 2.3. Physical Angry Birds game

Hjørring Library is popular for giving new and exiting experiences to its visitors. One way to achieve this is by having various themes that run throughout the whole library. These includes topics such as: Arabia, birds, fairy tales or simply the color brown. In the period of this project, Hjørring library has chosen the topic of Christmas, and it will run from the second week of November until the last week of December, 2012. At the time of the first

visit to the library, a bird-theme was running. All kinds of birds were exhibited around the library (see figures 2.4 and 2.5), and bird songs were played on several hidden speakers.

Figure 2.4. Birds.

Figure 2.5. More birds.

These themes are an example of how Hjørring Library tries to work with the concept of *serendipity*, which basically describes a happy accident; something you did not expect but turned out to be a pleasant surprise. Even though you go to the library to find a book, you might also experience and learn something that you originally did not expect.

Hjørring Library is always looking for new projects to involve its visitors in new and engaging ways. Since some Medialogy students previously came to the library to do projects and they turned out to be successful, the staff is eager to try it out again. This is where our group comes into the picture. To us, this is a great opportunity to collaborate with an institution on a project in a real-life setting. Instead of only testing under artificial circumstances, the library will allow us to test on visitors at the library throughout December.

Generally the atmosphere at the library is the same as that of any other library, as such the visitors (mainly students and adult visitors) naturally expect a quiet and calm working area.

2.1.1 Considerations

The goal for the program is to fit into the library without disturbing visitors, who do not wish to interact with the installation. This means that the program should not emit loud sounds that disturb visitors.

In order for the installation to be a fun experience for both a passive and an active user, the program will function on more than one level. An active user is a user that actively interacts with the canvas, and a passive is a someone who just walks by. This means that you should be able to pass by and admire the installation and also be able to spend more time in front of it.

2.1.2 Visitor data and placement of the canvas

The library is divided into several parts, each aiming to give a different experience. It is important for the group to get a central location for the placement of the canvas. This ensures that people will have the chance to interact with the program. In fact, the bookshelves chosen are just in the middle of the library, so it is hard not to walk past it at some point (see figure 2.6).

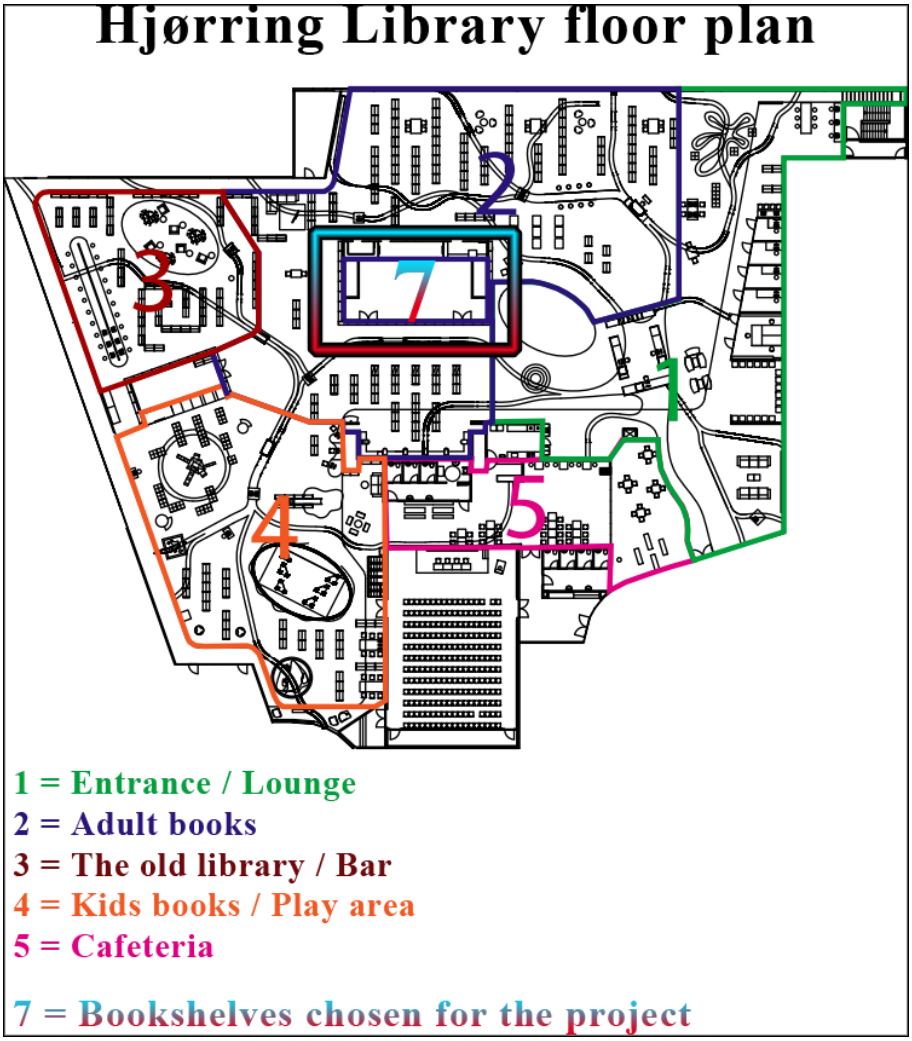


Figure 2.6. Hjørring Library is divided into multiple sections. We chose to use a big four-sided bookshelf in the core of the library. Image inspired by [Suenson et al., 2010].

In 2010 [Suenson et al., 2010] did a survey to investigate people's habits and movement patterns at Hjørring Library. They found that an average visitor spends about half an hour at the library. Those in the age group between 0-10 years and 21-30 years spent the most time at the library, 34 and 44 minutes, while those between 41-50 and 61-70 years spent 19 and 26 minutes during an average visit.

Several cylinder and flow maps were made to visualize the gathered data. Two are shown in figures 2.7 and 2.8.

Figure 2.7. Cylinder map showing multiple visitor's accumulated visiting time at Hjørring Library Tuesday November 24, 2010. Image created by [Suenson et al., 2010].

Figure 2.8. Flow map showing a single visitor's movement between 10.20-11.15, Saturday November 21, 2010. Image created by [Suenson et al., 2010].

2.2 Vision

The vision for the project is to make a simple, yet working program that is fun and engaging to use. It should be something you can just walk past once and enjoy without actively engaging in the installation. Simultaneously it should be possible to engage on a deeper level; you should also be able to interact with the installation. It is not meant as a game per se, but more like an interactive playground where you can move around and have fun. Another interesting aspect would be that the program should be fairly self-maintained and independent.

Based on the above-mentioned goals of engaging people at the library, the initial vision of this semester project is to engage visitors, both passively and actively. Ultimately, the final problem statement ended up being as shown (see figure 2.2.1).

The group decided to make everything from scratch, i.e. not using any existing functionality. However, since the OpenCV library was taught in the semester courses, as well as the C++ programming language, it seemed appropriate to use OpenCV to extract basic pixel information from video input. OpenCV provides a lot of useful functionalities that could easily be used for this project, such as applying filters and looking for BLOBs. It was decided not to use any of those functions and write the code ourselves. This means that even though the program is going to be slower in overall terms, in the end the group will have a deeper understanding of image processing implemented in the program.

2.2.1 Problem statement

The group decided that in order to succeed, the program should meet the following set of requirements:

- Low maintenance: The program should be easy to use for the library staff.
- The installation should fit the overall theme of Hjørring Library during December.
- It should be entertaining for both active and passive users.
- Multiple people should be able to use it simultaneously.
- All of the image processing functionality should be written by the group.

Theory 3

This chapter covers the various techniques used in image processing, explained in theory but not in practice. Later, in chapter 5, we will return to the Magic Canvas program and explain how those theories were applied and implemented.

3.1 A general framework for image processing

During this third semester project, image processing has been used to achieve the goals of the project. Image processing is an umbrella term that covers a set of tools used to process images and/or video. [Moeslund, 2012] writes that image processing comes from the general field of signal processing, and that it covers ways to segment objects of interest in a digital image. This is achieved by using multiple steps. It should be noted that the order of the steps sometimes change and there might be more focus on some steps than others, depending on the specific goal. (see figure 3.1) it shows a general framework that can be used as a good starting point [Moeslund, 2012].

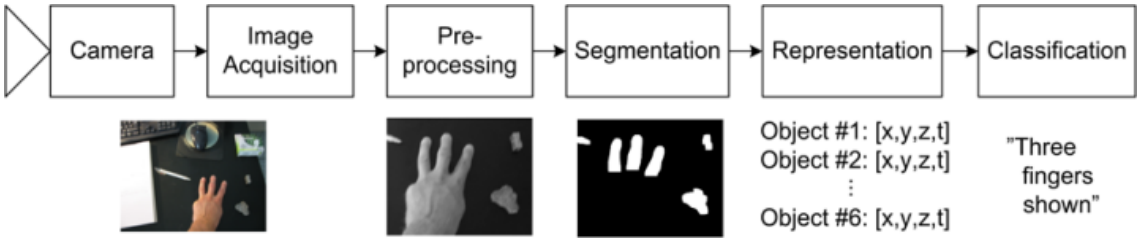


Figure 3.1. Image Processing framework, from [Moeslund, 2012].

Image Acquisition Before anything can be processed, data needs to be captured, typically using a camera. This step is all about the setup, as well as the environment, setting, lighting and so on.

Pre-processing Here the initial arrangement is completed, e.g. converting the image from color to grayscale.

Segmentation To be able to work with a specific object, it needs to be separated from the rest of the picture. This is done using segmentation to remove noise and background elements, so that only the object of interest is seen. Thresholding is often applied to make the object stand out, i.e. make the object appear white and the background black.

Representation The object needs to be represented in an intuitive manner.

Classification For the system to actually know if an object is a hand or not, it has to

do some features extraction and classifications to compare the data to some predefined database. This can be done with template matching and BLOB classification.

The combination of these different techniques makes it possible to get a functional program, but before using them, it is necessary to know how they work. The following sections will explain the different techniques in theory, starting with a basic information about digital image acquisition.

3.2 Image acquisition

An image captured through a camera is the result of reflected light being detected on a camera sensor, (see figure 3.2). This process is know as *image acquisition*. In this section the basics of image acquisition using a digital camera will be explained. Though it is necessary to have a basic understanding of the physics behind light to get through this process.

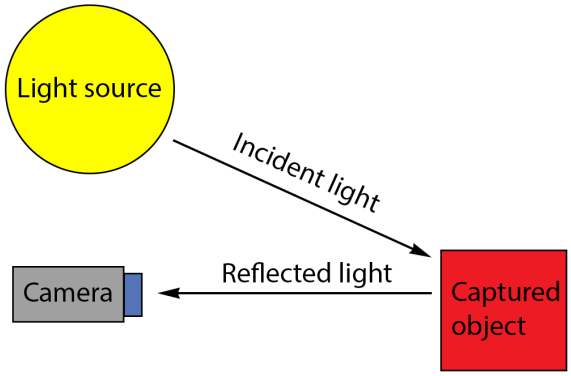


Figure 3.2. Light as captured by a camera

Light is a form of electromagnetic radiation that can be viewed as both waves and particles. This duality is however not something that will be covered within this section, as the wave model is sufficient to build the foundation of the understanding we need. A light wave is a small packet of energy travelling through space. These energy packets are known as photons and they can be described by three properties:

- **Wavelenght** - Measured in meters from wave top to wave top and denoted as λ.
- **Frequency** - Measured in oscillations per second, Hz, denoted f.
- **Energy** - Measured in electronvolts, eV, denoted E.

To derive the wavelength or the frequency, formula 3.1 is applied:

$$\lambda = \frac{C}{f} \tag{3.1}$$

where C is the speed of light.

The wavelength of the photon determines what color will be perceived. As the speed of light is constant, changing the frequency will alter the wavelength. When photons impact

objects, the material provokes a refraction that changes the frequency of the wavelength and therefore, a color can be perceived. The visible spectrum of light is only a fraction of the full spectrum of electromagnetic radiation, (see figure 3.3). Light interacts additively, with a mix of equal parts of each wavelength resulting in white light. The light from the sun can be broken down into its component wavelengths by refracting the light in a prism. This yields the full spectrum of visible light, as seen in a rainbow.

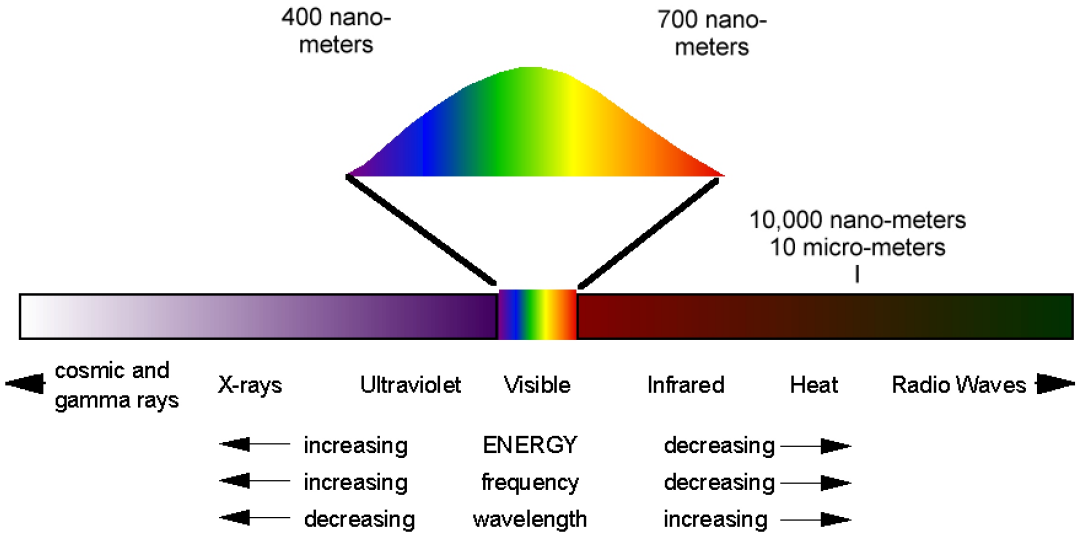


Figure 3.3. Figure illustrating the full spectrum of electromagnetic radiation. Image from [LPI].

3.2.1 Digital image acquisition

When capturing an image using a digital camera, light passes through a lens onto the sensor. This acts in much the same way as the human eye. In place of photoreceptors sensitive to specific wavelengths, a camera sensor has a physical matrix of pixel sensors (one for each pixel in the output image). A camera that can capture color images has three types of sensors in each pixel[Moeslund, 2012].

3.3 Digital images

One does perhaps not realize the technical aspects happening when capturing an image with a camera. In fact, several different computations are happening simultaneously. To be able to get some useful information from images and use it for further purposes, such as altering the image in an image editing program, it is necessary to have some fundamental knowledge about how images work on a computer. (see figure 3.4) it shows what this chapter is all about: working with color, grayscale, and binary images, as well as the different attributes of each type.

3.3.1 Pixels

When a digital camera takes a picture, it uses an image sensor consisting of an array of interconnected cells. All cells hold a filter and a sensor. As light hits the cells, the energy

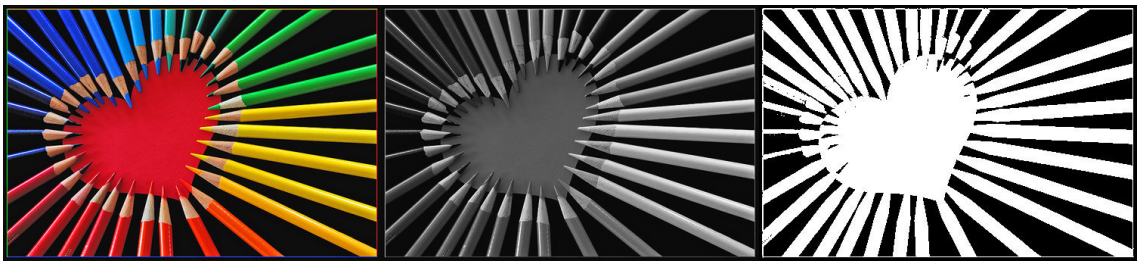


Figure 3.4. Image illustrating a conversion from a color image to grayscale and finally binary. Inspired by [Visualizeus.com].

is converted into a digital number via an analog-to-digital converter (ADC). This value is denoted a *pixel* and describes how bright a specific part of the image is [Moeslund, 2012].

3.3.2 Bits and bytes

The way computers store data is using bits and bytes. One bit can be thought of as a switch that can be turned on or off. However, instead of on/off buttons, a computer uses digits in form of 0's and 1's. A byte consist of 8 bits and is a row of eight "switches" capable of holding the values 0 and 1. This provides one byte with the ability to store 256 different values, since 2^8 is equal to 256.

3.3.3 8-bit images and grayscale images

In an 8-bit image, the 8-bit prefix describes the *bit depth* of the image. The bit depth tells us about the amount of information that can be stored in a single pixel.

An image with a single channel of information for each pixel in the X and Y axis is typically a grayscale image. The information stored in each pixel is the gray-level value of the particular pixel [Moeslund, 2012]. Considering an 8-bit image, each pixel can hold 256 different levels of gray. In the case of a zero-based computer system, this equates to a pixel holding a value between 0-255 (see figure 3.5). While 8-bit images are widely-used as a format for images, it is also possible to create images capable of holding more than 256 gray-level values for each individual pixel.

Figure 3.5. Image illustrating the 256 different gray-level values of an 8 bit picture.

3.3.4 Indexing an image

When working with images on a computer and performing image processing, one often has to look at the individual pixels within an image.

Working with pixels in an image is similar to working with a coordinate system. The image is indexed from the top-left corner. Both X- and Y-axes are indexed beginning with (0,0). Going to the right the X value increases, going down the Y value increases. A single pixel can be described as a coordinate, e.g. (see figure 3.6) the pixel (1191,665) is the bottom-most corner to the right.

Figure 3.6. The pixel values are stored in a coordinate system. Note that the Y-axis is reversed.

3.3.5 Working with color images

Now that a basic knowledge regarding images has been established, the next step into using images in calculations is to understand the basics of a color image.

There are several formats for handling color images. In this report we will only describe RGB (red-green-blue). The main difference when talking about color and grayscale images, is that color images have three channels, compared to the single channel in grayscale images. Each of the three channels holds the value of a specific color for each pixel in red, green and blue, which are the primary colors our eyes are sensitive to. In addition, equal amounts of red, green and blue will produce a gradient of grey. A pixel with each color channels holding a value of 255 produces a white pixel, and a pixel with the color channel values of 0 will produce a black pixel. This is explained by the additive mixing of colors in RGB images [Moeslund, 2012].

3.3.6 Binary images

In a binary image a pixel is either white or black (true or false). Binary pictures are less resource-demanding when performing mathematical operations as each pixel can take only one of two values.

3.4 Segmentation

Segmenting an image or video is the process of extracting the information you are interested in. Segmentation often consists of different sub operations, all with the same goal of getting the information you want from the input image [Moeslund, 2012]. Before segmentation is applied, an image undergoes preprocessing. This might be converting from a color image to a grayscale image to ease the computations required. (see figure 3.4) It serves as an illustration of preprocessing and segmentation. First the image is converted to a grayscale image, this step is part of the preprocessing. Then the image is thresholded, which yields a binary image with only the pens and the heart.

Two possible problems can arise while segmenting an image. If the segmentation algorithms are too strict, and do not include enough in their extraction, the object meant to be extracted might not be complete. Therefore it might not be recognized correctly. This case is called *under-segmentation* [Moeslund, 2012].

In the opposite case, the algorithm is too sloppily designed. The result is that too much of the input image is included in the segmentation. The final segmented image might contain too much noise or objects might be connected, even though they were separate entities. This is called *over-segmentation* [Moeslund, 2012].

3.5 Histograms

A commonly-used tool when working with digital images is the histogram. Essentially a histogram provides data distribution, which is an easy way to get an overview of the frequency of events. An image histogram is a graphical representation of how many times a specific pixel value appears. This can be used to show whether an image is too dark or too bright, as well as how good the contrast is [Moeslund, 2012]. (see figure 3.7) It shows a histogram with pixel values from 0-255. The more often a specific pixel value is present on the image, the taller the bin representing what the particular pixel value will be.

Histograms can also be used to describe the contrast in an image. The contrast is a measure of the difference of intensity levels between the dark and light areas on the picture [Cambridgeincolour.com]. This means that a histogram with a broader distribution of values will represent an image with good contrast. On the other hand, if the histogram is very narrow, the contrast in the image is smaller, and it will appear more flat and dull (see figure 3.9).

3.6 Thresholding

Thresholding is one of the most fundamental operations when working with images. It is performed to convert an image to a binary image. This is useful in programs where you

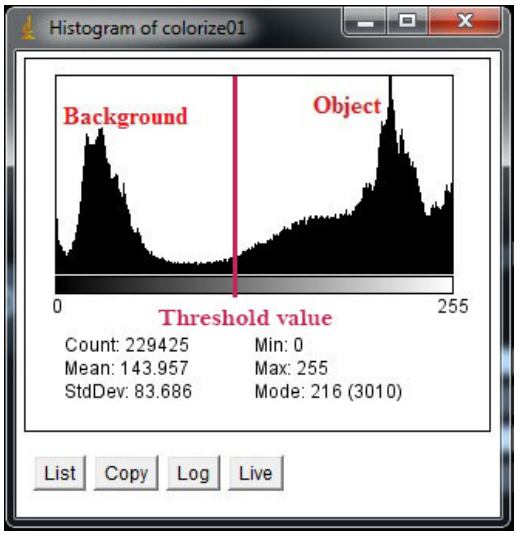


Figure 3.7. Ideal histogram with two "mountains".

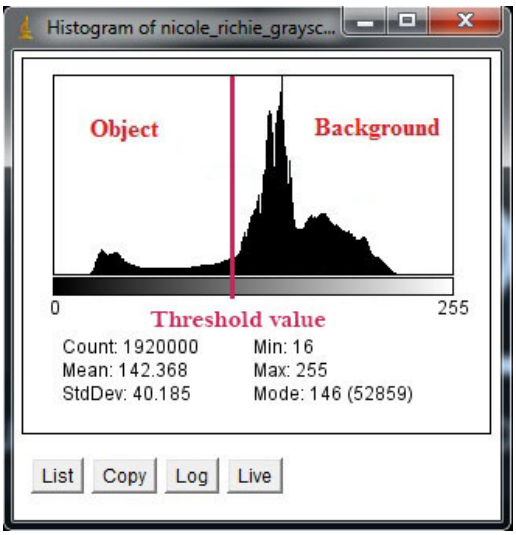


Figure 3.8. Problematic histogram.

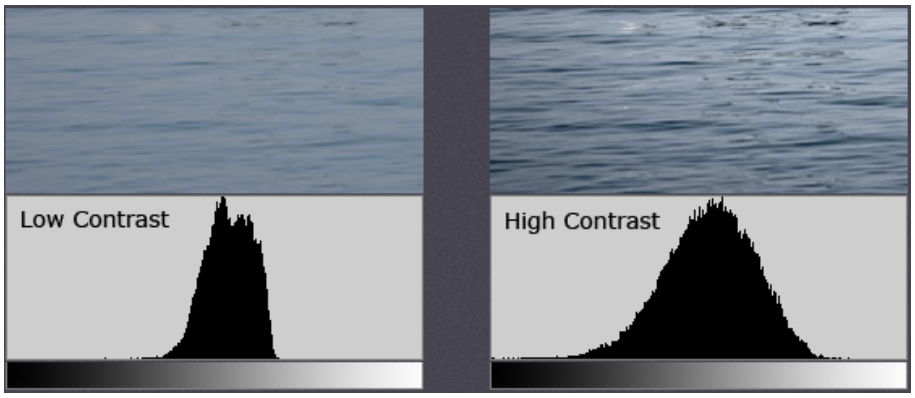


Figure 3.9. Histograms can be used to describe the contrast in an image. Picture from [Cambridgeincolour.com].

need to find silhouettes, such as tracking a person, where smaller details are not important.

To determine which pixels should be completely black and which should be completely white, a *threshold value* is required. Using a metaphor, the threshold value can be compared to a gatekeeper that lets everyone who is 18 years old or older inside the club, but denies access to people who are 17 or younger. Using this analogy, pixels that are "old" (bright) enough are let in, while "younger" pixels (dark) are denied. The value used to classify which persons (pixels) can pass and which have to stay out, have to be decided by the gatekeeper (or the programmer, in this case).

In practice this means that pixels with values greater than a threshold is set to true (or white), which typically is 255 when talking in bytes. On the other hand, if a pixel is less than the threshold value, it is set to false (black) or 0. The formula for calculating a

threshold is shown in equation 3.2.

$$\begin{aligned} &\text{if } f(x,y) \leq T \quad \text{then } g(x,y) = 0 \\ &\text{if } f(x,y) > T \quad \text{then } g(x,y) = 255 \end{aligned} \tag{3.2}$$

Where T is the threshold value, $f(x,y)$ is the input pixel and $g(x,y)$ is the output pixel.

When setting the threshold value it is important to consider what you want as the output. The effect of thresholding will differ from image to image, depending on the contrast between the object and the background. If the background and the object have a high contrast or are very different in color, it will be easy to distinguish them and choose an effective threshold value. However, if the object and background are very similar, it will be hard to choose a value to separate them properly. At that point it will be necessary to choose between losing information associated to the object of interest, or to gather some background noise to preserve it.

Figures 3.7 and 3.8 show the threshold value used on the images 3.10 and 3.11. Looking to the histograms, it is easy to tell that the leftmost is more ideal to threshold because the object and background are very different. That is shown graphically with two "mountains" in the histogram (see figure 3.7), also called a *bi-modal histogram* [Moeslund, 2012]. The effect of a good threshold value can be seen on picture 3.10, where the output gives a clear outline of the silhouette of the woman. On the other hand, the histogram to the right does not have two clear mountains, but a big one and two smaller. This means that the color values are distributed more equally and therefore, it will be hard to define a good threshold value. The result in this case is a poor silhouette of the woman (see figure 3.11), due to the fact that the background and the object are too similar.

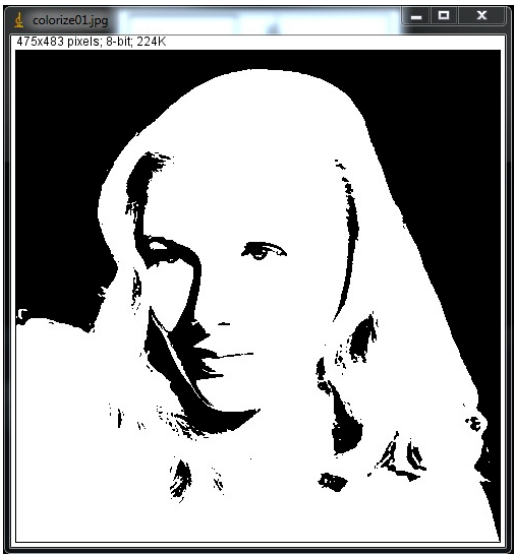


Figure 3.10. Good silhouette after thresholding.

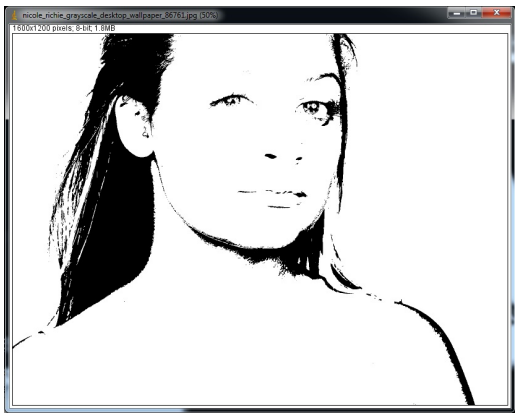


Figure 3.11. Poor silhouette after thresholding.

3.7 Morphology

Morphology is a collection of operations used to analyze and process structures. Although these techniques are commonly used on binary images, it is possible to apply them to grayscale images as well. Morphology on binary images can be used to remove noise produced by thresholding an image, as well as defining the contours of the objects in order to achieve a proper BLOB analysis [Moeslund, 2012]. BLOBs are described in 3.12.

Morphology is a type of neighborhood processing. In neighborhood processing the surrounding pixels of the input pixel contributes to the operation that is performed on the pixel itself. For this purpose a kernel is used, as shown in figure 3.12. A kernel, also called *structuring element*, is a grid containing numbers, denoted *kernel coefficients*. These coefficients differ depending on the image processing applied.

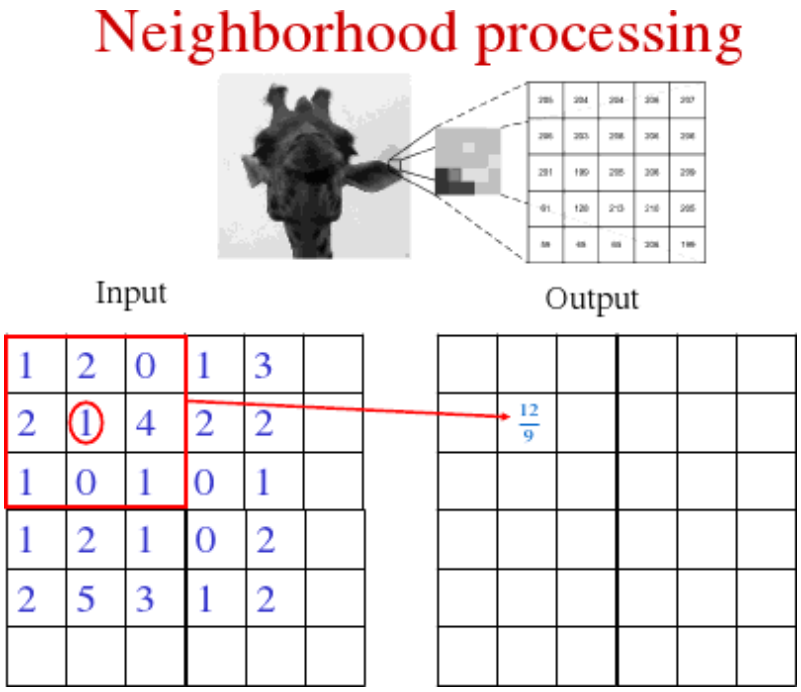


Figure 3.12. Neighborhood processing uses a kernel. Image from [Moeslund, 2012].

Hit and fit

Whenever one decides to apply a kernel and *hit* or *fit* the pixels, what happens is that the algorithm will compare the values of the pixels on the input image to the kernel. Depending on these values and the image's pixel values, different output occur.

The process of hitting the pixels starts by positioning the kernel on the image, so that it operates on one pixel at a time. If one of the pixels on the input image is a '1' where the kernel is '1' as well, then it is said to hit. This means that the pixel in the center of the kernel will become a '1' (white) in the output. If none of the pixels inside the kernel "hits", then the pixel in the middle of the kernel becomes a '0' (black) in the output.

The opposite operation is fitting the pixels. But in this case, all the values of the pixels on the area of the kernel have to be the same both on the kernel and the input image. If

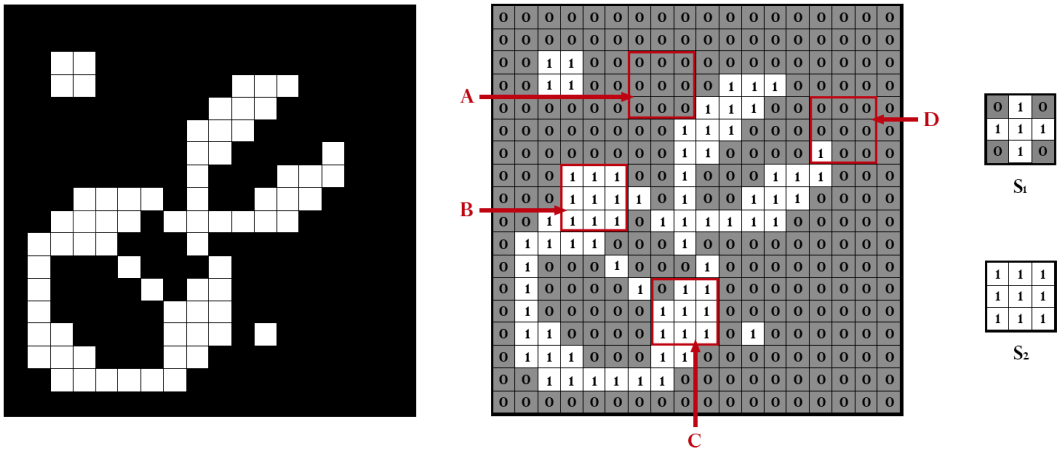


Figure 3.13. Binary image and structuring elements. Inspired by [Moeslund, 2012].

one of the pixels on the input image have a different value to the corresponding 1's of the kernel, the pixel in which the kernel is centred will be a '0' on the output image. On table 3.1 an example image, figure 3.13, is being represented with two different types of kernel.

Position	SE	Hit	Fit
A	S_1	No	No
A	S_2	No	No
B	S_1	Yes	Yes
B	S_2	Yes	Yes
C	S_1	Yes	Yes
C	S_2	Yes	No
D	S_1	No	No
D	S_2	No	No

Table 3.1. Results of hitting and fitting with two different structuring elements. Inspired by [Moeslund, 2012].

3.7.1 Dilation

Dilation is the process of applying the hit process to an entire image, and it refers to the expansion of an object on an image (see equation 3.3 for a mathematical definition). The result of this method implies filling small holes and merging objects. As mentioned before, the final effect on these objects will depend on the kernel, how big it is and which shape it has. This can be seen on figure 3.14.

$$g(x, y) = f(x, y) \oplus SE \tag{3.3}$$

A small kernel applied several times will have the same effect as a big kernel applied once. This can be seen on equation 3.4 where dilating twice with the element S_1 has the same effect on the object as dilating one time with S_2, even though the only difference between

those two kernels is that SE_2 has a radius twice times bigger than the radius of S_1.

$$f(x,y) \oplus S_2 \approx (f(x,y) \oplus S_1) \oplus S_1 \tag{3.4}$$

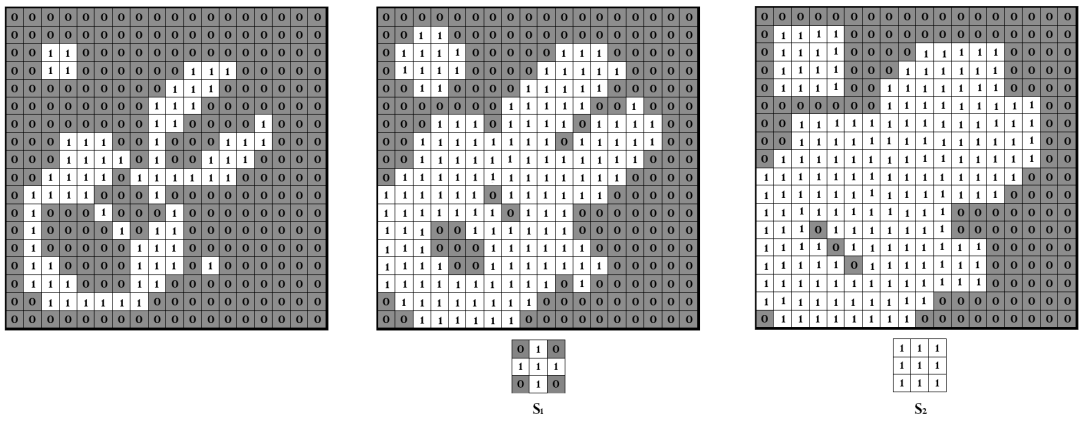


Figure 3.14. Dilation produced by two different kernel. Inspired by [Moeslund, 2012].

3.7.2 Erosion

Erosion is the process of applying fit to an entire image and refers to the reduction of the size of an object in an image (the mathematical definition is provided in equation 3.5). As the fit method is applied, small objects will also disappear and larger objects will be broken down into smaller ones. As it happens with dilation, the effects depend on the size and shape of the kernel.

$$g(x,y) = f(x,y) \ominus SE \tag{3.5}$$

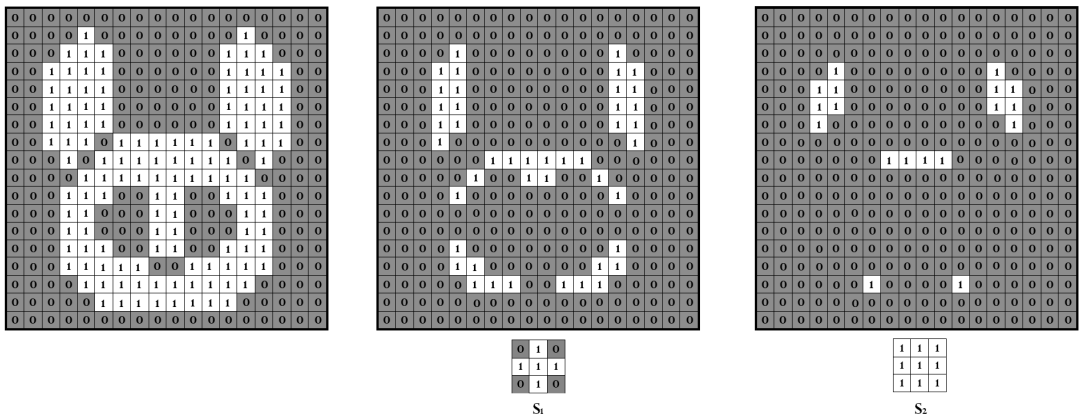


Figure 3.15. Erosion produced by two different kernels. Inspired by [Moeslund, 2012].

3.7.3 Compound operations

The term *compound operations* refers to the combination of dilating and eroding objects on an image and can therefore imply the *union* or *intersection* of the objects, but also other operations like *closing*, *opening* or doing edge detection, also know as *boundary detection*.

Opening

When eroding images to erase noisy objects or split parts, it often occurs that the object of interest has decreased its size. The solution to this problem is to dilate the eroded image. This operation is denoted *opening*, as shown in equation 3.6.

$$g(x,y) = f(x,y) \circ SE = (f(x,y) \ominus SE) \oplus SE \tag{3.6}$$

The effect of opening can be seen on figure 3.16 where a circular kernel is applied to the image. Even though the object still maintains its original size, some information has been lost due to the effect of eroding and dilating the image. Although the same structuring element is used along the process.

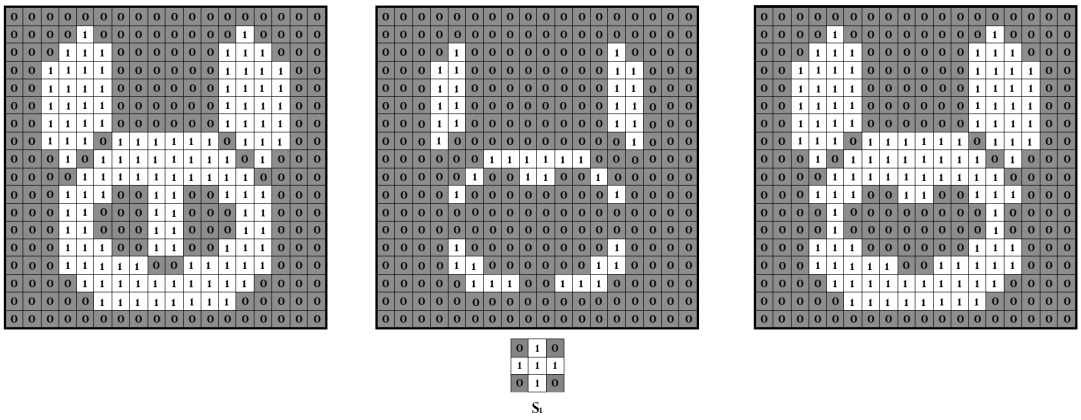


Figure 3.16. Opening produced by a circular structured element. Inspired by [Moeslund, 2012].

Closing

When dilating, the sizes of the objects are increased. When just wanting to fill the holes it is important to constrain the measure. A good solution is to erode after the dilation. This is denoted *closing* and uses equation 3.7. As with the opening method, the size and shape of the kernel must be the same in order to obtain the desired result.

The effect of this operation can be seen in figure 3.17: even though the holes are filled and the object maintains its original size, the noise of the background is still there. Therefore, it will be necessary to apply either closing or a BLOB analysis to delete those small objects.

$$g(x,y) = f(x,y) \bullet SE = (f(x,y) \oplus SE) \ominus SE \tag{3.7}$$

When applying closing, most of the holes of the image will be filled, but the size of the object will be the original one.

It should be noted that using closing and/or opening combination multiple times will not achieve a better result than doing it only once on the image. However if the same operation is applied a second time, the size of the final image will be decreased and increased respectively.

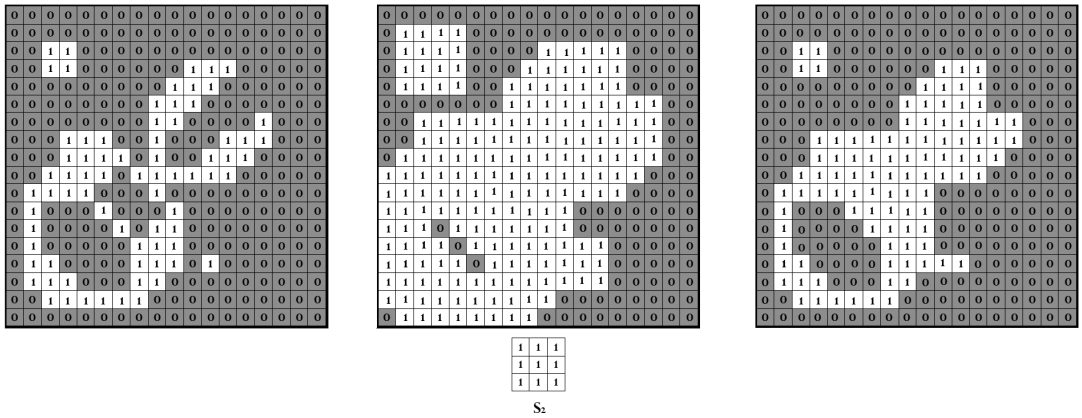


Figure 3.17. Closing produced by a squared structured element. Inspired by [Moeslund, 2012].

3.8 Region of interest (ROI)

When dealing with images some might think that more pixels are always better. However this is not always true. Since there are more pixels to loop through, it takes longer time for the computer to analyze and process the image data. The more pixels to process, the more time is required for the program to do the necessary calculations.

When working with video, it is important to maintain a stable frame rate. The more information needed to be processed, the slower the program will be. One way to ease the computational requirements is to use what is called a *region of interest* (ROI). As the name suggests, the point is to choose a specific region of the image that is of interest. This could for instance be the upper part of an image where heads are expected to be. This region is usually enclosed in a rectangle and only the pixels inside this rectangle are being processed, ignoring all the pixels outside of the ROI. This prevents unnecessary processing and optimizes the speed of the program by decreasing the amount of data processed.

An example of how this could be used is taking a photo where only the head is of interest (see picture 3.18).

3.9 Filters

A filter can be applied to an image to either remove unwanted noise or to blur the image. In most of the cases, when thresholding an image, noise appears. Noise can be in form of small dots in the background or small holes in the object of interest. By using filters on the image it is possible to remove noise. Two techniques will be described in this section: the mean and median filters.

Mean filter

One method of image filtering is the *mean filter*. As the name implies, a mean filter takes an input image and calculates the mean value of a given pixel and outputs this value. This type of neighborhood processing takes the average of the input pixel and its surrounding pixels and thereby decides the value of the output pixel (see figure 3.19).

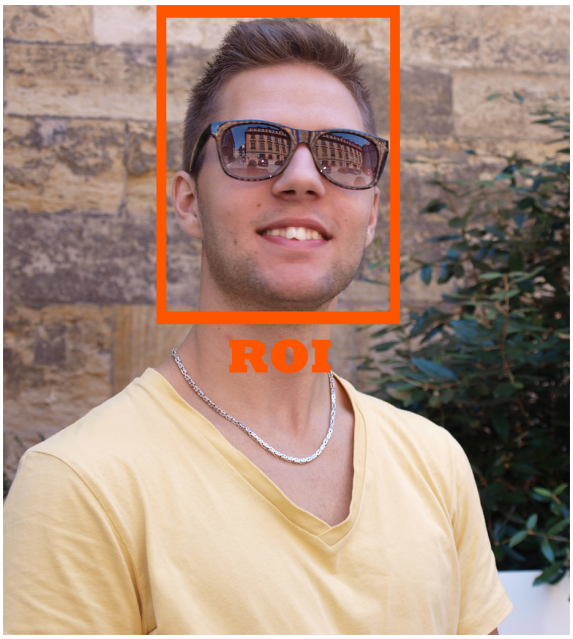


Figure 3.18. Illustrating the principle of region of interest.

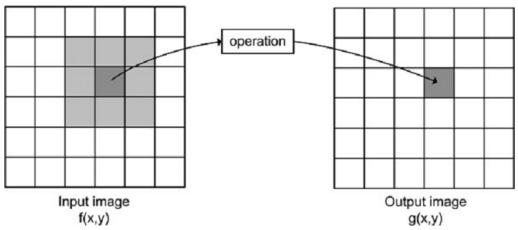


Figure 3.19. Practical neighborhood processing. All the surrounding pixels contribute to the value of the output. From [Moeslund, 2012].

The calculations behind the average is shown in equation 3.8.

$$\text{Mean} = \frac{\text{Sum of input and neighboring pixels}}{\text{Number of pixels}} \tag{3.8}$$

Mean filters are often used to blur an image, e.g. when a person's face should stay anonymous. The amount of blur depends on the kernel's size [Moeslund, 2012].

Median filter

The second method is the *median filter*. This works by taking the value from a pixel, as well as its surrounding neighbors and ordering them so that the lowest value appears to the most-left and the highest value to the far-right. The method outputs the value in the very middle of the list. (see 3.9).

$$\text{Ordered list:}[1, 4, 17, 24, \mathbf{42}, 43, 52, 108, 235] \tag{3.9}$$

Median filters are good for removing noise, especially the so-called "salt and pepper noise", as shown on figure 3.20 [Moeslund, 2012].

Figure 3.20. A median filter can be used to remove salt and pepper noise similar to this.

3.10 Border problem

In neighborhood processing a kernel is applied to an image. The problem is that the kernel will always, when placed at the edge of the image, try to access pixels outside of it. Three different solutions exist to this problem, increasing the size of the output image after the neighborhood processing, increasing the size of the input image before neighborhood processing or by using special kernels for the edge-pixels.

3.11 Background subtraction

A way to detect changes in a scene and extract an object is using *background subtraction*. As the name implies, it is done by subtracting the background from the scenery, so that only changes are shown.

In order to be able to use background subtraction efficiently, a controlled setup is required. The optimal setup is an indoor environment with controllable lights. This is important as the background should be static. Imagine that sunshine is let inside a room. The light will change together with the sun's position. The difference in illumination would mean that changes will be seen everywhere, which gives an inaccurate result. When doing background subtraction, it is important that the background stays the same.

3.12 BLOB analysis

A common task when dealing with images is determining if the image contains a particular object or shape. The term *BLOB* is an acronym for Binary Large Object and refers to a region of connected pixels in a binary image [Moeslund, 2012]. This technique can be used to extract meaningful information from images. It can be achieved by separating the pixels in points or regions that differ in properties, like brightness, color or differences compared to the background.

The process of BLOB analysis will be split in three main steps: *extraction* of the BLOBs, *representation* of the BLOBs and lastly, *classification* of the BLOBs [Moeslund, 2012].

3.12.1 BLOB extraction

To isolate BLOBs in a binary image, we need to know if two pixels are connected or not. This is done by applying algorithms that will help to determine the connectivity of the pixels, but also the number of BLOBs contained in an image. The most commonly-used kernels in BLOB extraction are the 8-connectivity and the 4-connectivity kernels [Moeslund, 2012]. As the 8-connectivity kernel checks for more connectivity, it requires more computations.

Grass-fire algorithm

One of these connected component labelling algorithms is the *recursive grass-fire algorithm*, used to label the regions of foreground pixels to separate the BLOBs from the background. To explain this algorithm, we will use both 8-connectivity and 4-connectivity kernels to illustrate how these choices can affect to the final result. The algorithm is denoted as recursive because it is contained within a function that calls itself Moeslund [2012].

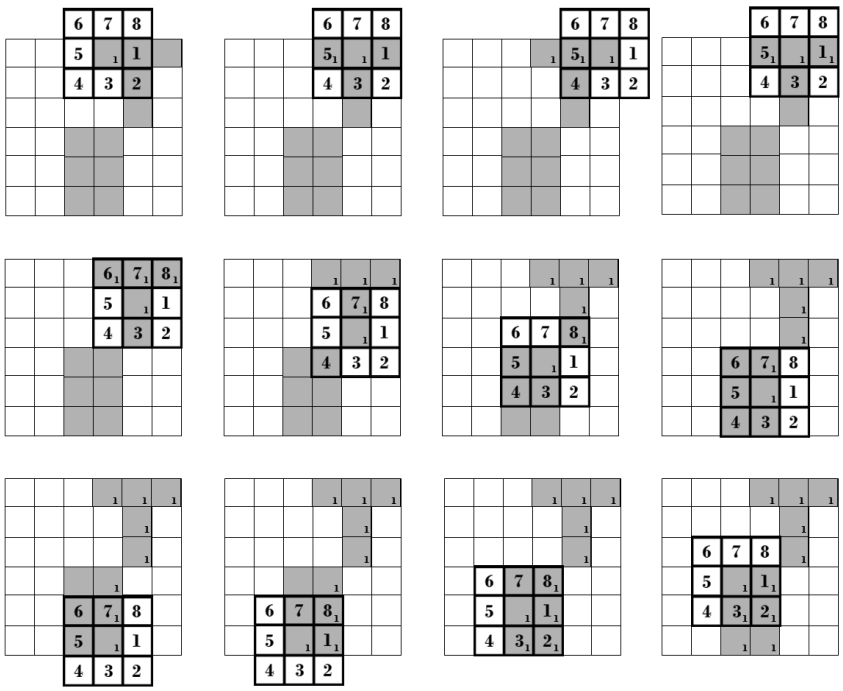


Figure 3.21. 8-connectivity labelling kernel detects a single object. Inspired by [Moeslund, 2012].

As shown in figure 3.21, the grass-fire algorithm scans the entire image from the upper-left corner to the bottom right, row by row. When the kernel encounters a non-zero pixel value, it centers on it and "burns" the center pixel. Then the algorithm looks in the possible

directions around that pixel to check if the surrounding pixels are connected to the burned pixel. The way the algorithm will do this will depend on the connectivity used (for a 8-connected kernel, the algorithm will look in 8 directions, but for the 4-connected kernel, the algorithm will look in 4 directions).

Whenever it finds an object, the algorithm labels the pixel in the output image and then burns that pixel on the input image, in order to turn it into a zero and ensure that the pixel will not be part of a new grassfire.

The algorithm now looks to the possible directions around the new pixel, starting with the pixel on the right. This process is represented on figure 3.21. Once it finds a non-zero pixel value, the algorithm once again centers its attention on it and repeats the process, labelling the pixel with the number of the previous one. At the end of that row the algorithm will check the surrounding pixels on the row below. If a non-zero pixel value is found, the algorithm will continue the process of burning and labelling pixels, otherwise the algorithm will start its way back to the beginning checking the surrounding pixels one by one to verify that it has checked all the possible directions and non-zero pixels values.

Comparing figures 3.21 and 3.22, we can realize how the choice of a certain kernel will affect the final result of the algorithm using the same picture. The 8-connectivity kernel is unable to separate the different objects as intended in this particular case, whereas the 4-connectivity kernel finds the different objects performing less operations.

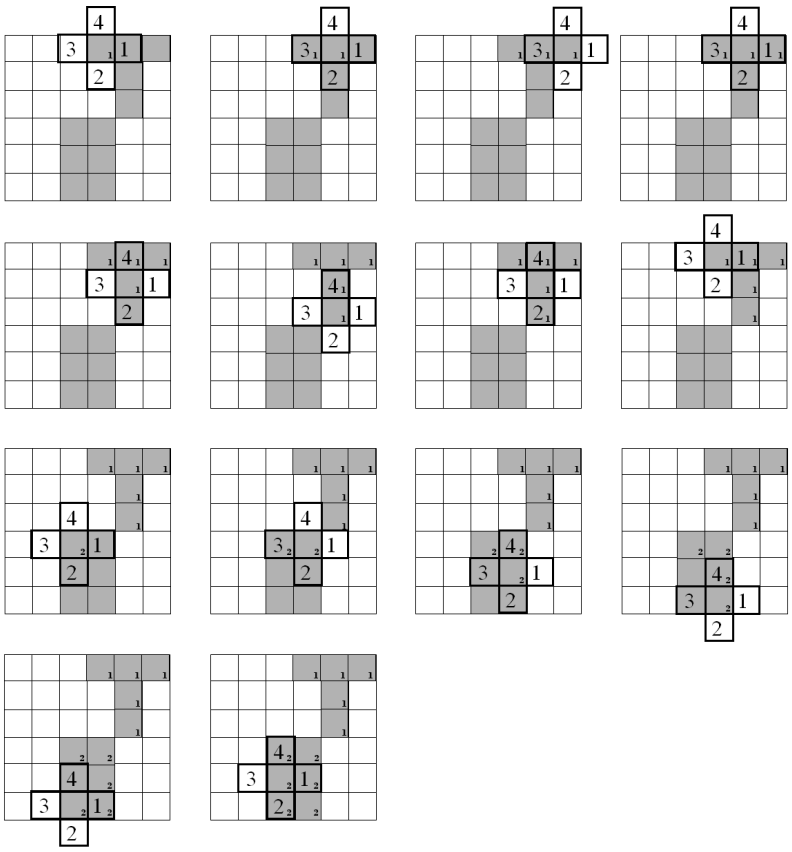


Figure 3.22. 4-connectivity labelling kernel detects two objects. Inspired by [Moeslund, 2012].

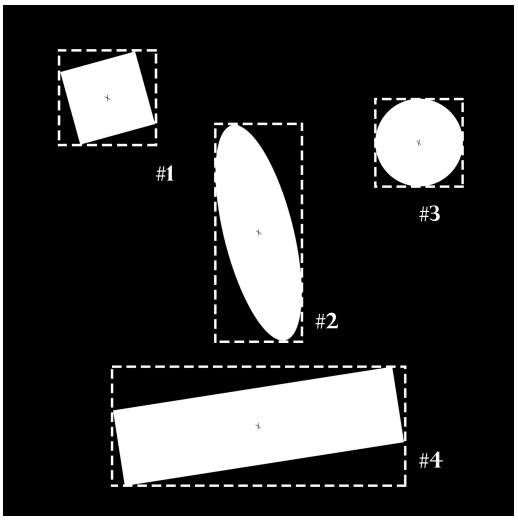
3.12.2 BLOB representation

When looking for several different features, the classification of the BLOBs can be made by creating a prototype model of the object that we are looking for. This will allow to state the features that the BLOBs should contain, and the deviations that would be acceptable. This process consists of two steps: determining the features of the BLOBs; and matching the features with the prototype to determine if they belong to the expect ones. Still, a few features like the area, the perimeter and the circularity can already help classifying the BLOBs [Moeslund, 2012].

The features

- **Area** The importance of calculating features like the area of a BLOB is well understood since one of the first steps while classifying the detected objects is to delete those that are bigger or smaller than the prototype. One way of doing this is by calculating the area of the BLOB.
- **Perimeter** Scanning along the border of the BLOB and summing the pixels, we obtain the length of the contour of the BLOB. In image processing, this could be done by using morphology techniques like erosion to get a smaller version of the object and then subtracting this to the input image in order to get the edges.
- **Circularity** Circularity is a common shape-factor that depends on the perimeter and the area. There are several ways to define how circular an object can be, but usually applying the ratio to get a value lower or equal to 1 will indicate how circular the BLOB is, where 1 is a complete circle.

After all the features are extracted, a binary object can be defined by its features' values that would be stored into a feature vector in order to have a list where to start the BLOB classification.



BLOB number	Circu-larity	Area (pixels)
1	0.13	65536
2	0.32	144011
3	1	72584
4	0.38	258456

Figure 3.23. Binary image with a bounding box and the center-of-mass. The table shows two other features that can be calculated: the perimeter and the area. Inspired by [Moeslund, 2012].

3.12.3 BLOB classification

Once the BLOBs are extracted, their features will be used to determine if they are the BLOBs looked for. Further more, one can use the found features to build a *prototype model*. This prototype model is used to keep track of more complex features like circularity or combined features. As some extracted features will not fit perfectly the prototype model features, a deviation will still be acceptable.

Setup 4

4.1 Image acquisition in the project

For image acquisition in the project, two cameras were considered. Both were the same model, a Logitech Pro 9000. One was an RGB web camera, the other camera had an infrared (IR) filter installed. The filter consists of a small strip of an analogue film. The film strip allows only infrared light to pass, making the only light that reaches the sensor infrared light. Infrared light lies right next to visible light in the electromagnetic spectrum, see figure 3.3. It can be used to illuminate things without having a visible component to the light, but this requires a device able to capture IR. This enables a larger degree of control over the illumination in the scene. Normally in a scene, controlling the lighting can prove a challenge, and in order to operate without error an image processing algorithm would have to take into account the variance in lighting that occurs naturally during a day cycle [Moeslund, 2012]. Working with infrared light helps to avoid this problem.

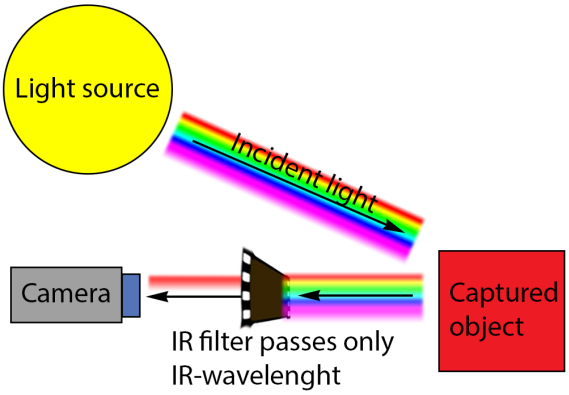


Figure 4.1. Figure illustrating how the IR filter only allows infrared light to pass.

4.1.1 Illuminating the scene

It is important that the lighting in the scene is constant. To avoid running into issues with variances in the illumination, it was decided to illuminate the scene artificially. Different options were evaluated for the illumination, all under the consideration of having the ability to filter the camera input, allowing only infrared light through. The following options were considered:

- Using the library's lighting.
- Making adjustments to the lighting at the library.

- Bringing normal lights to illuminate the scene.
- Building our own light system to illuminate the scene.

Using the library's lighting

The library's lighting consists of two different types of lamps: Fluorescent lights and very bright halogen spots. Using the infrared camera, the library's normal lighting was investigated. The investigation proved that too much light was emitted from the halogen spots, preventing a decent contrast between the background and the subject.

Making adjustments to the lighting at the library

It was not possible to adjust the fluorescent lights, but the halogen spots could be pointed in a different direction. Using cardboard to direct the cone of light from the spots, as to create the largest contrast between background and subject was considered. Two options emerged, pointing the lights at the background, or pointing the lights at the subjects. These two cases offered two different problems. Limiting the cone of light to only hit the subjects was not possible due to the placement of the spots. On the other hand limiting the spots to illuminate the canvas would interfere with the projection of images, as the spots are very bright.

Bringing normal lights to the scene

By bringing normal lights that emit a high amount of IR light, the subject could be illuminated in a sufficient manner, but this would require a setup to be constructed. One pitfall was the high amount of visible light emitted. This would once again interfere with the image projected on the canvas.

Building our own light system to light the scene

Building a lighting system from scratch to light the scene would allow a tailored solution for the installation, removing the problems offered by the other solutions. By having a strip of bright infrared lights, and using this to light up the background. It should be possible to create a high contrast between the background and the subject of interest. Furthermore this provides us with the benefit of already having defined a region of interest, thus easing the workload of the image processing.

4.1.2 Concluding on the lighting

Using the default setup at the library did not provide a decent scene to be captured and processed by the program. The lights at the library were too diffuse, making adjustments to the lighting non-viable. While the light could be controlled to only illuminate the background, the visible light emitted would shine too bright, making the image from the projector hard to see. Bringing normal lights would result in the same problem. This meant that the final decision was to create a custom infrared lighting setup for the project.

4.2 Building the LED illumination

For illuminating the scene, IR LEDs were chosen because of their good illumination outside of the visible spectrum. For the project two different LEDs were tested, both emitting at roughly 880nm [SIEMENS, 2004b] [SIEMENS, 2004a], a 5mm with a cone of light of 16 degrees and a 3mm with a cone of light of 130 degrees.

During initial testing the light from the 5mm LED looked as though it might be too focussed, making it difficult to create a bright line of unbroken illumination. For that reason a series of experiments were conducted: A setup that tested the light that the two different LEDs offered were constructed and it was apparent that the broader LEDs did not focus the light enough, and thus did not create a high enough illumination, see figure 4.2 and 4.3. Therefore the choice was made to go with the 5mm, 16 degree LEDs.

Figure 4.2. 5mm LED illuminates a sheet.

Figure 4.3. 3mm LED illuminates the same sheet.

The 5mm LEDs were tested further to see whether is was possible to improve their lighting properties. A test was conducted where the LEDs had their lens sanded down in order to make the light less focussed. Figure 4.4 shows an image where the LEDs in progression from left to right get more and more sanded down. As the figure illustrates the intensity of the light decreases as the LED-lens is sanded down. After conducting these tests, it became clear that an unaltered 5mm LED performed better than the modified ones.

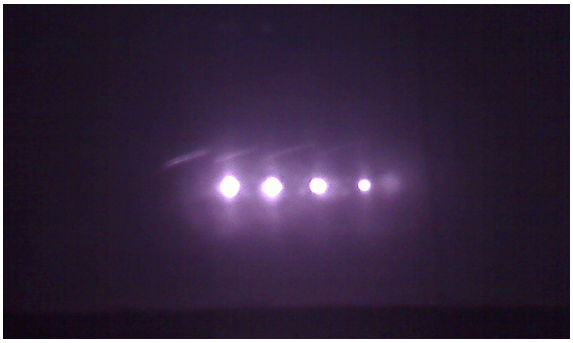


Figure 4.4. The LEDs have been sanded down in progressive order from left to right.

For ease of construction the LEDs were arranged into arrays of eight LEDs connected in series, which in turn are connected in parallel. The LEDs have a forward voltage of 1.5V,

which means that the power supply has to be able to supply at least 12V to power all eight LEDs in a series. Initially a small AC-DC power supply capable of supplying 12V at 600mA was used for testing purposes. However, the total draw of a single LED array is 500mA and for the project a total of four arrays were planned. The current far exceeded what could be supplied by the small AC-DC supply. Luckily the group managed to salvage a 300 watt computer power supply capable of supplying 12V at 8A. After getting the power supply to work without being in a computer, terminals were soldered to LED-strips and the power supply, allowing the LED strips to plug into the power supply.

4.3 Setup at Hjørring Library

The initial steps of the setup was to determine the location at the library where the installation would be located. The library offered a series of different locations. The chosen location is in the middle of the library. The main reason for this choice was that the area had artificial lighting and is also the most direct path from the entrance to the working area and the children's area. In figure 2.6 the layout of the library is shown along with the designated project area. The area was also provides a good place for the projector and canvas. The bookshelf is a sturdy and relatively unreachable place for the projector, so no visitor will accidentally touch or mess with it.

Behind the canvas a strip of LEDs are placed. Between the projector and the canvas there is a long red bookshelf (dubbed "den røde tråd"); this will also be used to mount a strip of infrared LEDs. In figure 4.5 the setup is illustrated in figure4.5.

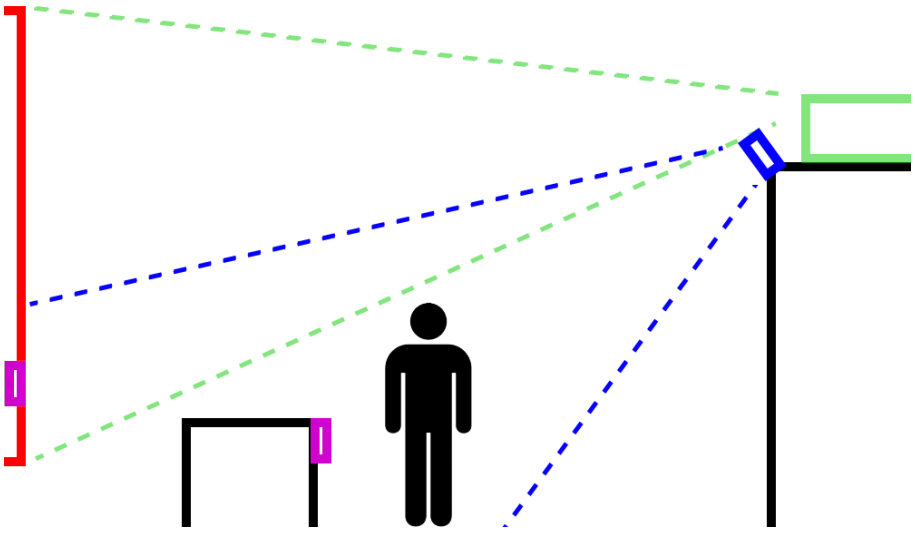


Figure 4.5. Sketch of the setup: Canvas is red; LED strips are purple; camera with IR filter is blue; projector is faded green; den røde tråd is black.

The feasibility of the model was tested during several visits at the library. The projector's

and camera's placements were tested, and proved to be suitable for implementation.

However, there proved to be a problem with one of the two LED strips: the strip mounted on the wall did not, due to the angle of and distance to the camera, covering the intended ROI. Therefore adjustments were made to the installation, and one row of LEDs were disabled.

4.3.1 LED strips

For displaying the output it was decided to use a sheet. Originally it was planned to build a custom wood box for the sheets, but due to time constraints it was instead chosen to use velcro tape to hang it.

The first step in the construction of the canvas was to position the LED strips in order to get the desired illumination for the infrared camera. As the placement of this project is a corridor divided by a red bookshelf that crosses the whole library, it was decided to add one strip on the lower part of this shelve and another LED stripes on the background covered by the canvas (see figure 4.6).

The mounting of these LED strips was made as simple in order to make it as light as possible. Therefore the strips were taped to the shelves with duct tape. The solutions can be seen on figure 4.8 and 4.7.

Figure 4.6. Canvas placement.

Figure 4.7. LED strips construction.

Figure 4.8. LED strips construction.

Figure 4.9. Light as captured by a camera.

Graphical design 5

Several decisions were made regarding the graphical style of the program. The drawings should be easy to distinguish, funny-looking and related to the Christmas theme.

The first step was to draw a suitable background that ultimately would create a Christmas mood. The original idea was to have a few different sets to choose from, depending on the time of the day or week. Due to time constrains only one background was drawn in the end. As shown on figure 5.1, it consists of some mountain hills and a blue sky. It was important not to have too many details, since the focus should be on the characters themselves.

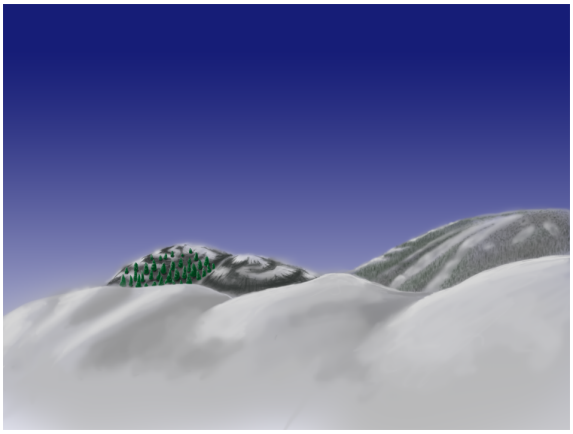


Figure 5.1. The background image used for the program.

In addition, due to light conditions at the library, the colors chosen for the background are bright (see figure 5.2), in order to get an usable output from the projector. If the colors were too dark, it would be difficult to see them properly on the canvas.

To make the scene feel more alive, Christmas trees were added, as well as falling snowflakes.

5.0.2 Drawing the characters

In addition to the background a group of Christmas characters were drawn. It was decided to have a small selection of characters that were visually different from each other. The different characters are: a snowman, a pixie boy, a pixie girl, an angle, and a squirrel.

It was decided to draw the characters by hand, instead of using pre-made drawings from the Internet, to achieve a personal art style. In the beginning, outlines of the characters were drawn on sheets of paper (see figure 5.3). When the drafts were satisfying, they were imported to the computer and re-drawn in Photoshop using drawing tablets. The drawing

Figure 5.2. The colors should be bright enough so they are clear on the canvas.

tablets have pressure sensors, which make the line's thickness change according to pressure of the pen. Having different strokes makes the characters spring more to life. The last step was then to color the characters, keeping a contrast to the background.

Figure 5.3. The characters were first drawn on paper, then imported to and colored in Photoshop.

It was decided to give the characters a cartoon-like look, to appeal to a broader target group. This was done by using a variety of techniques. One of which was oversized body parts, e.g. oversized heads. Two of the characters, the snowman and the angel, are shown on figures 5.4 and 5.5.

Figure 5.4. The snowman character.

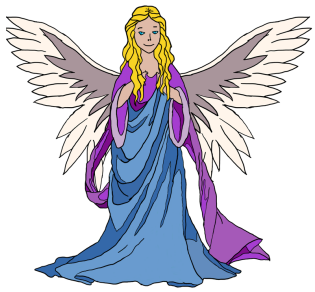


Figure 5.5. The angle character.

5.0.3 Making animations

It was decided to give three animations for each character, in order to achieve more vivid characters. This creates a sense of motion, and makes the characters appear more life-like. The primary animation is an idle stance that will play whenever a visitor is standing still in front of the camera (see figure 5.6). In addition, there are two walking animations: one for each direction (left and right).

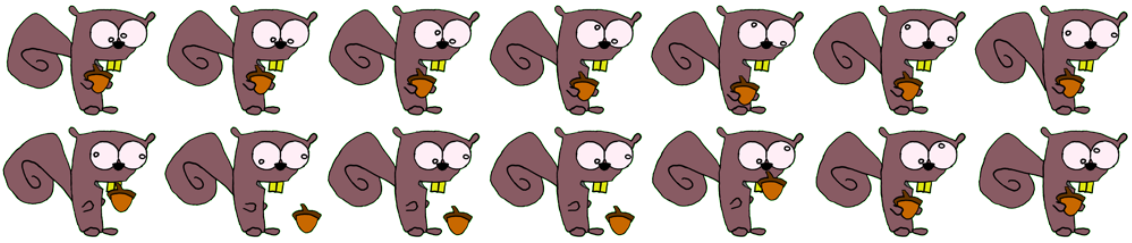


Figure 5.6. Idle animation for the squirrel.

The animations were created as sprite sheets, which is a single image containing multiple frames. This approach was decided for practical purposes. In the program, the sprite sheet is looped through, showing each frame for a specific amount of time. Figure 5.7 shows an example of the pixie boy's walk cycle that will play whenever a visitor is moving.

Figure 5.7. The pixie boy walk cycle is looped through to create a sense of motion.

Implementation 6

In this chapter we will describe how the program is implemented. As mentioned earlier, the C++ programming language was used, along with the open source library OpenCV to work with image data. While C++ was used to do the actual image processing part, it was decided to use the Unity game engine to display the actual result in an intuitive manner. This approach will be described in 6.8.

6.1 Writing the program from scratch

When the group started working on the image processing, they were encouraged by the supervisors to avoid using built-in algorithms from OpenCV. Even if the program would be slower, in the end the group would walk away with more knowledge and experience if they programmed everything from scratch.

The group decided to agree to this approach. Therefore the OpenCV library was only used for getting access to the pixel values, and to show a result on the screen. From there they could write all the algorithms by themselves.

In the following we will not explain all the code written for this project, but instead focus on essential code snippets (the complete code is available via the appendix 10.1). When we want to refer to a specific line of code, this will be done using a number inside parentheses like this: (0). Function names, classes, variables, etc. will have their names written in bold font, such as: the **Picture** class.

6.1.1 The pixel system

In OpenCV a pixel value can be accessed by the function **at()**:

```
1 myOpenCVPicture.at<Vec3b>(y,x)[0]; // blue
2 myOpenCVPicture.at<Vec3b>(y,x)[1]; // green
3 myOpenCVPicture.at<Vec3b>(y,x)[2]; // red
```

This piece of code returns a pixel value of a picture called **myOpenCVPicture** at the position **(X, Y)**. In particular, the value that is returned is a 8-bit value that describes the color channels of the pixel. The index number inside the square brackets identify the channel accessed.

The question was now how the group should order, structure, and store the values they would get from OpenCV. One thing the group didn't like was the fact that pixels in OpenCV have to be accessed reversed; meaning that instead of the common RGB format

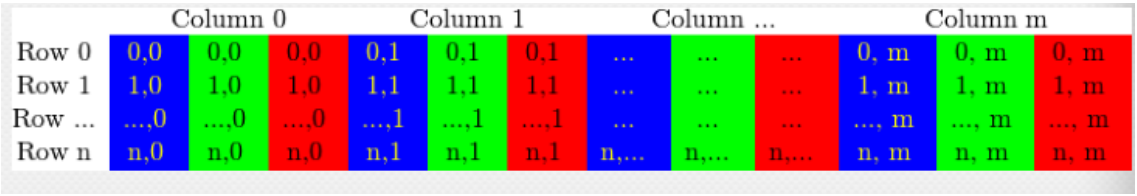


Figure 6.1. OpenCV stores the values of the color channels reversed in a matrix system. Picture from [Tsesmelis, 2012].

(red, green and blue), OpenCV uses BGR (blue, green and red; see figure 6.1). The same thing applies to to the coordinates X and Y that have to be written in the reversed order as arguments for the function **at()**.

The group decided that they wanted to write their own class for pictures. This class should work in a similar way as the OpenCV matrix class called **Mat**. It should hold the values of the pixels in a way that is easy to understand, and it should contain the functions that can perform different image processing algorithms. This class was named **Picture** and used throughout the whole program.

The group decided that it would be easiest to store each color channel in a two-dimensional array of integers. These values should be accessed in the right order, in contrast to the the way OpenCV stores them.

```
1  void Picture::initialize(VideoCapture captureToStoreCamra)
2  {
3    captureToStoreCamra >> tmp; // (1)
4    width = tmp.cols; //(2)
5    height = tmp.rows; //(2)
6
7    pixelR = new int*[width];
8    for(int x = 0; x < width; x++)
9    {
10     pixelR[x] = new int[height];
11     for(int y = 0; y < height; y++)
12       pixelR[x][y] = tmp.at<Vec3b>(y,x)[2];
13   }
14
15   pixelG = new int*[width];
16   for(int x = 0; x < width; x++)
17   {
18     pixelG[x] = new int[height];
19     for(int y = 0; y < height; y++)
20       pixelG[x][y] = tmp.at<Vec3b>(y,x)[1];
21   }
22
23   pixelB = new int*[width];
24   for(int x = 0; x < width; x++)
25   {
26     pixelB[x] = new int[height];
```

```
27      for(int y = 0; y < height; y++)
28        pixelB[x][y] = tmp.at<Vec3b>(y,x)[0]; // (3)
29    }
30  }
```

This code is one of the first functions that was written. The purpose of the function is to open an image with OpenCV, as well as writing the values of the opened picture to three two-dimensional arrays that from there can be processed. These two-dimensional arrays are member variables of an object of type **Picture**, as well as the function is a member function.

The first thing this function does is that it streams a picture from a chosen camera. Streams provide a character-based interface to read/write data from one place to another, for instance a text file or OpenCV's **VideoCapture** that allows for the use of webcams [Mukherjee]. We store a temporary object of type **Mat** and the name **tmp** (1). This **tmp** object is also a member of the class **Picture**. It is never used for image processing, just for reading either from an image file, or a capture for outputting the picture to the screen.

The next thing the function does is that it saves the width and the height of the **Mat** object in the **Picture** object (2). From there the function creates the arrays of integers that hold the pixel values in all three channels (3).

When one wants to access a pixel, one first has to write the picture's name, e.g. **myCustomPicture**. The next thing is to specify the color channel, for instance **pixelG**, which is a member variable of **myCustomPicture**. Lastly, the position of the pixel has to be chosen, by writing the position in the array, as shown in the following code:

```
1  myCustomPicture.pixelG[x][y] // getting pixel access using the custom↩
       -written Picture class
```

When one wants to see the picture again, one performs a call to the **output()** function that does the opposite of the **initialize()**; it takes in the pixels and puts them in a temporary **Mat** instance that is used to show the picture.

```
1  void Picture::output(string windowName)
2  {
3    if(height == 0||width == 0) //(1)
4      cout << "No picture is loaded";//(2)
5      waitKey(0);//(2)
6      exit(0);//(2)
7    }
8    Mat out(height, width, CV_8UC3);//(3)
9
10   for(int x = 0; x < width; x++)
11     for(int y = 0; y < height; y++)
12       out.at<Vec3b>(y,x)[2] = pixelR[x][y]; //(4)
13   for(int x = 0; x < width; x++)
14     for(int y = 0; y < height; y++)
15       out.at<Vec3b>(y,x)[1] = pixelG[x][y]; //(4)
```

```
16    for(int x = 0; x < width; x++)
17      for(int y = 0; y < height; y++)
18        out.at<Vec3b>(y,x)[0] = pixelB[x][y]; //(4)
19    imshow(windowName, out); //(5)
20  }
```

The check in the beginning checks if the programmer tries to output a picture that does not exist, by checking if the picture's size is zero (1). If this is the case, the programmer gets an error message with the text *No picture is loaded*. Pressing a key will then close down the program (2). If, on the other hand, a picture is loaded correctly, the program will create a **Mat** object (3), write pixel values to it (4), and show the picture on the screen (5) in an output window.

6.1.2 Structuring the functions

Another big advantage in having our own **Picture** class is that each group member can work on a different function in the **Picture** class without disturbing somebody else who is working on another function. Even if a function does not work, it does not break the code as long as it is not called.

6.2 The structure of the program

The main structure of the program is simple. It is basically divided into two parts: the setup and the main loop. The setup is everything that just needs to happen one time when the program starts.

Then there is the main loop which is a while loop that constantly runs while the program is open. The condition that is given in the main loop is **true**. This means that the loop is never exited, unless something in the loop makes it stop, in this case by hitting the Escape key. This makes it possible to exit what would otherwise be an infinite loop.

Besides the main loop there is also a while loop in the setup, but this will be described a later in 6.5.1.

It is important to note that it is bad programming practice to make the program allocate memory in the loop, since it will use more and more memory, unless the memory gets released again. That is why it would run out of memory very fast if the function **initialize()** would be called in the main loop. Since we still need to get new input from the camera, a function named **refresh()** was implemented. It does almost the same as **initialize()**, but instead of creating a new integer for each pixel value, it assigns the old integer a new pixel value, as shown in the following code snippet:

```
1  void Picture::refresh(VideoCapture captureToStoreCamra)
2  {
3    captureToStoreCamra >> tmp;
4    width = tmp.cols;
5    height = tmp.rows;
6
7    for(int x = 0; x < width; x++)
```

```
8   {
9     for(int y = 0; y < height; y++)
10      pixelR[x][y] = tmp.at<Vec3b>(y,x)[2];
11  }
12
13  for(int x = 0; x < width; x++)
14  {
15    for(int y = 0; y < height; y++)
16      pixelG[x][y] = tmp.at<Vec3b>(y,x)[1];
17  }
18
19  for(int x = 0; x < width; x++)
20  {
21    for(int y = 0; y < height; y++)
22      pixelB[x][y] = tmp.at<Vec3b>(y,x)[0];
23  }
24 }
```

6.2.1 The downside of the custom-written pixel system

The decision to have all pixels in our own class might sound like a neat way of accessing the pixels, but on the other hand it also makes the program slower, since it has to go through all the color channels of each and every pixel in each frame. To circumvent this problem, and thereby achieve a faster frame rate for the program, another system was implemented. This will be described later in 6.3.3.

6.3 The preprocessing

6.3.1 Region of Interest

Like mentioned in 4.1.1, the group was only interested in the horizontal position of the person that would step into the camera's field of view. The region of interest was illuminated by a narrow strip of LEDs where people were detected. That also means that just this part of the picture had to be analyzed. The result is a faster-running program, since only a part of the picture has to be analyzed.

The ROI was implemented by initializing the picture in **setup()**, setting all the pixel values to 0, and in the main loop just refreshing the parts that are in the region of interest.

6.3.2 Background subtraction

When the group started to make the program, it seemed like a good idea to have a reference picture of what the camera sees without people in the picture. This would then be used to background subtract in every frame. After that the group wanted to threshold the result to get a binary picture.

This approach worked fine, however, it was also quite slow, since the computer had to do the calculations for every pixel value it received. That is why the group had to look into which parts of this process could be left out for optimizing the program.

The thing the group knew was that the person would be darker than the background, since the background would be illuminated with external lights. That meant that the program was not interested in what had become brighter in comparison to the reference picture. The other thing the group knew was that the camera just took infrared light, since the camera contained an IR filter, making most of the color information disappear (see figure 6.2). That is why it was chosen to use a single color channel.

Figure 6.2. Image seen from the IR camera.

Another possibility would have been to put all the color channels together with different weight factors, like in the conversion from a color image to a grayscale image. It was chosen not to do this, because the color channels were so similar that it wouldn't have improved the final result, and it would have meant that we had to read all color channels, making the program slower.

6.3.3 Optimizing the code

Having the previous sections in mind, we are now interested in the pixels inside the ROI that have become darker in comparison to the reference picture. Also, we would like to have the steps of reading all the pixels using OpenCV removed. This can all be written in code in a simple way, using fewer steps than needed before:

```
1 void Picture::refreshDiscradBGSubtractAndThreshholdForBnW(↩
    VideoCapture captureToStoreCamra, Picture refPicture, int ↩
    threshhold, int procentOfScreenUsed, int heightOfLowerROI)
2 {
3   captureToStoreCamra >> tmp; //(1)
4   for(int x = 0; x < width; x++) //(2)
5   {
6     for(int y = heightOfLowerROI; y > heightOfLowerROI - (int)(((↩
        float)(procentOfScreenUsed/2)/100)*height); y--)//(2)
7     {
```

```
8       if((int)tmp.at<Vec3b>(y,x)[2] - refPicture.pixelR[x][y] < -1*↩
            threshhold) //(3)
9       {
10        pixelR[x][y] = 255; //(4)
11      }
12      else
13      {
14        pixelR[x][y] = 0; //(5)
15      }
16    }
17  }
18 }
```

This function does four things, as described previously: reading in pixel values with help from OpenCV; using the region of interest to discard unnecessary data; performing background subtraction; and performing thresholding.

The first thing the function does is that it streams the current picture from the camera to a temporary **Mat** object called **tmp** (1). The next thing it does is that it loops through all the X values of the picture and through all the Y values that we are interested in (2). The Y values we are interested in are the ones that go from **heightOfLowerROI**, which is a parameter to the function, up to a specific percentage of the screen. For each of these coordinates the program checks if the red channel of the of the new picture is darker than the threshold value (3). If this is the case, the current pixel gets the maximum value, to show that this pixel has changed in comparison to the reference picture (4). If it hasn't changed enough, the pixel's value is set to to zero (5).

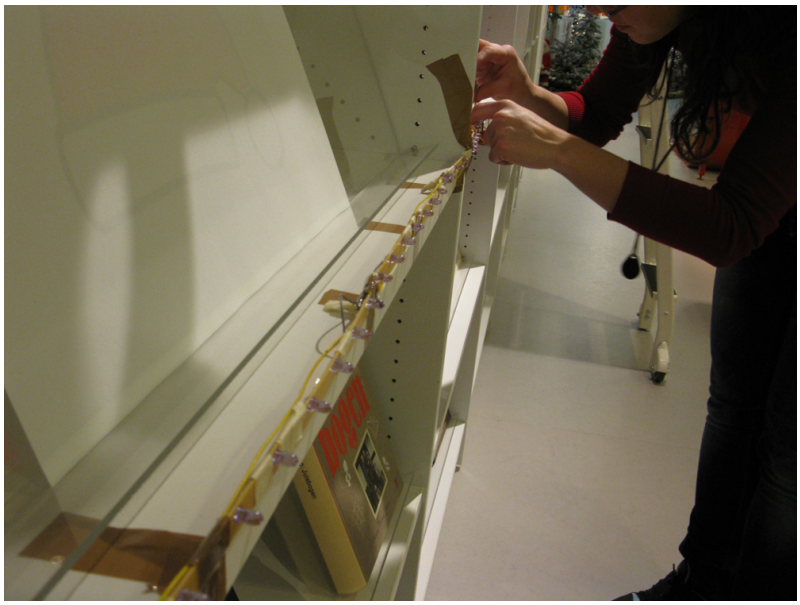


Figure 6.3. The LEDs are put on a long strip at 4 centimetres intervals.

6.4 Morphology

Because the group arranged the LEDs on a long strip in an interval of 4 centimetres (see figure 6.3), there were some light gaps on the captured image. This meant that some of the pixels that actually should have been detected as changed were detected as unchanged. This caused the program to detect some persons doubled, because they were separated, even though there was in fact only a single person in front of the camera.

The group solved this by applying morphology to the picture. Closing was applied to close the gaps within the persons. As mentioned in 3.7.3, closing consists of first dilating the picture and then eroding it. This is shown in the following code:

```
1  void Picture::dilate(int radius, Picture &tmpPicture)
2  {
3    for(int x = radius; x < width-radius; x++) //(1)
4    {
5      for(int y = radius; y < height-radius; y++) //(1)
6      {
7        bool pixelIsaccepted = false;
8        for(int filterX = x - radius; !pixelIsaccepted && filterX <= x ↩
             + radius; filterX++) //(2)
9        {
10         for(int filterY = y - radius; !pixelIsaccepted && filterY <= ↩
               y + radius; filterY++) //(2)
11         {
12           if (pixelR[filterX][filterY] == 255)(3)
13           {
14             pixelIsaccepted = true;
15           }
16         }
17       }
18       if (pixelIsaccepted == true)
19         tmpPicture.pixelR[x][y] = 255; //(5)
20       else
21         tmpPicture.pixelR[x][y] = 0;
22     }
23   }
24   for(int x = 0; x < width; x++)
25     for(int y = 0; y < height; y++){
26       pixelR[x][y] = tmpPicture.pixelR[x][y];
27       pixelG[x][y] = tmpPicture.pixelR[x][y];
28       pixelB[x][y] = tmpPicture.pixelR[x][y];
29     } //(6)
30 }
31 void Picture::erode(int radius, Picture &tmpPicture)
32 {
33   for(int x = radius; x < width-radius; x++) //(1)
34   {
35     for(int y = radius; y < height-radius; y++) //(1)
36     {
37       bool pixelIsaccepted = true;
```

```
38        for(int filterX = x - radius; pixelIsaccepted && filterX <= x +↩
              radius; filterX++) //(2)
39        {
40          for(int filterY = y - radius; pixelIsaccepted && filterY <= y↩
                + radius; filterY++) //(2)
41          {
42            if (pixelR[filterX][filterY] == 0)(4)
43            {
44              pixelIsaccepted = false;
45            }
46          }
47        }
48        if (pixelIsaccepted == true)
49          tmpPicture.pixelR[x][y] = 255; //(5)
50        else
51          tmpPicture.pixelR[x][y] = 0;
52      }
53    }
54    for(int x = 0; x < width; x++)
55      for(int y = 0; y < height; y++)
56      {
57        pixelR[x][y] = tmpPicture.pixelR[x][y];
58        pixelG[x][y] = tmpPicture.pixelR[x][y];
59        pixelB[x][y] = tmpPicture.pixelR[x][y];
60      } //(6)
61  }
```

Both the erosion and dilation go through almost all the pixels in the picture. The pixels that are not analyzed are the ones that are too close to the edge of the picture, so the kernel would be empty for some of the values (1). There are various ways to circumvent this edge problem, but it was decided not to pursue these, since the pixel edges are not important for the program.

At every pixel, the functions look around the given pixel with the chosen radius (2). The difference between **erosion()** and **dilation()** is that for a pixel to be accepted in dilation, just a single pixel in the radius has to be **true** (3). With erosion, there can't be any pixels in the radius that are **false** (4). Other than that the two functions work the same way. While the program is still looking through the image, the output has to be stored in a buffer picture. This buffer picture holds the output, so the the original input picture is left intact (5). When the functions are finished with analyzing the picture, the output from the buffer picture gets written to the picture that the function is called from (6).

6.5 The setup loop

6.5.1 Take a background picture

Like mentioned before in 6.2, there is also a loop in the setup. This loop contains a while loop that has **true** as its condition. The loop is not supposed to end when the Escape key is pressed, but when it has figured out that there are no people in the picture. The reason

for this is that it has to take a reference picture, which is the one used in the background subtraction.

The function works under the assumption that people in front of the camera always move a little and are not standing 100% still. The function has three thresholds. The first threshold is called **threshholdPixelChange** and is the value from which a pixel can be counted as changed.

The second is **threshholdPixelsChanged** which is the threshold for how many pixels count as a person, since we are not interested if just a single pixel has changed.

The third threshold, **threshholdFramesChanged**, is set in case that a person should accidentally stand still for one frame. This threshold sets how many pictures in a row that needs to be captured without changes, before assuming that nobody is on the picture.

```
1  void configBG(Picture &BG, VideoCapture &camera1, int ↩
       threshholdPixelChange, int threshholdPixelsChanged, int ↩
       threshholdFramesChanged, int lroi)
2  {
3    Picture test1;
4    Picture test2;
5    test1.initialize(camera1);
6    test2.initialize(camera1); // (1)
7
8    double framesUnchangeged = 0; //(2)
9    while(true)
10   {
11     test1.refresh(camera1);
12     waitKey(30);
13     test2.refresh(camera1); //(3)
14     double pixelChange = 0; // (4)
15     double pixelsChanged = 0; // (5)
16
17
18     for(int x = 0; x < test1.width; x++)
19     {
20       for(int y = 0; y < test1.height; y++) // (6)
21       {
22         pixelChange = 0;
23         if(test1.pixelR[x][y] - test2.pixelR[x][y] > 0)
24           pixelChange += test1.pixelR[x][y] - test2.pixelR[x][y];
25         else
26           pixelChange += test2.pixelR[x][y] - test1.pixelR[x][y];
27
28         if(test1.pixelG[x][y] - test2.pixelG[x][y] > 0)
29           pixelChange += test1.pixelG[x][y] - test2.pixelG[x][y];
30         else
31           pixelChange += test2.pixelG[x][y] - test1.pixelG[x][y];
32
33         if(test1.pixelB[x][y] - test2.pixelB[x][y] > 0)
34           pixelChange += test1.pixelB[x][y] - test2.pixelB[x][y];
```

```
35          else
36            pixelChange += test2.pixelB[x][y] - test1.pixelB[x][y]; // ↩
                (7)
37
38          if(pixelChange > threshholdPixelChange)
39            pixelsChanged++; // (8)
40        }
41      }
42      if(pixelsChanged < threshholdPixelsChanged)
43        framesUnchangeged++; // (9)
44      else
45        framesUnchangeged = 0; // (10)
46
47      if(framesUnchangeged >= threshholdFramesChanged) //(11)
48      {
49        BG.refresh(camera1); //(12)
50        int brightestYProduct = 0;
51        for(int x = 0; x < BG.width; x++) //(13)
52        {
53          int brightestVal = 0;
54          for(int y = 0; y < BG.height; y++) //(13)
55          {
56            if(BG.pixelR[x][y] > brightestVal)
57            {
58              brightestVal = BG.pixelR[x][y];
59              brightestYatX[x] = y; //(14)
60            }
61          }
62          brightestYProduct += brightestYatX[x]; //(15)
63        }
64        startHeightOfROI = brightestYProduct/BG.width + BG.height*((↩
              float)procentOfTheScreenUsed/100)/4; //(16)
65
66        if(startHeightOfROI > BG.height-1)
67          startHeightOfROI = BG.height-1; //(17)
68        return;
69      }
70    }
71  }
```

Before starting the while loop, **configBG()** configures the background; initializes two pictures (1); as well as a variable that will hold the number of frames that are unchanged in a row (2).

In the while loop it refreshes the pictures with a time delay of 30 milliseconds (3), and it initializes a counter that holds how much each pixel is changed (4). It also has a variable that knows how many pixels that have been changed (5). After this the program goes through all pixel coordinates (6) and saves the difference between the pixel value of each channel in the variable **pixelChange** (7).

When **pixelChange** is bigger than the threshold that classifies a pixel as being changed,

the counter **pixelsChanged** gets raised by one (8). When all the pixels have been looked through, the program knows how many pixels have been changed. When sufficiently few pixels have changed, which is set by the parameter **threshholdPixelsChanged**, the counter for frames that haven't changed gets counted up (9). When too many pixels have changed the counter gets set to zero (10).

When enough pictures have been unchanged, the last part of the loop is entered (11). Here, first the **Picture** object called **BG** gets refreshed (12). When this is done, we analyze two things on the picture. The first one is finding the brightest pixel's position in each column of the picture. The second is placing the region of interest. We find the brightest pixel by going through each pixel of each column (13). The brightest pixels and their values are then stored (14). This gets saved in the integer array **brightestYatX[X]**. When feeding in an X value, it returns the Y position of the brightest pixel at that X position.

To find out where the region of interest has to be placed, the program adds all Y positions of the brightest pixels (15) together and finds the average of them in the end (16). In the end there is a check that makes sure that the region of interest is in the frame (17).

6.5.2 The enter zone

Now the question might be why we are interested in the position of the brightest pixel in each column on the background picture. The answer to that is that when a person goes into the picture and produces pixels that have been changed in comparison to the background, the person will do that in front of the LEDs. If we are looking for this person, we don't want to look through all the pixels but only the ones where the LEDs are on the background picture. We are looking for the brightest pixel, because we hope to find the pixel at the position of each LED. We will come back to this in 6.7.2.

6.6 Finding and analysing the BLOBs

Let us assume that we have examined a pixel that is found to be true or in other words changed in comparison to the background. The next thing would be to find the pixels that are connected to this single pixel, categorizing them as one object, also known as a BLOB (Binary Large Object). One method that is often used is recursion. That's why a recursive function, when called, could go from pixel to pixel and call itself:

```
1  void Picture::startFire(point startingPoint, Picture &tmpPicture)
2  {
3    tmpPicture.makeBlack();
4    findNextPoint(startingPoint, tmpPicture);
5    for(int x = 0; x < width; x++){
6      for(int y = 0; y < height; y++){
7        if(tmpPicture.pixelR[x][y] == 255){
8          //pixelR[x][y] = 0;
9          pixelG[x][y] = 150;
10         //pixelB[x][y] = 0;
11       }
12     }
```

```
13    }
14  }
15  void Picture::findNextPoint(point currentPosition, Picture &↩
        tmpPicture)
16  {
17    tmpPicture.pixelR[currentPosition.x][currentPosition.y] = 255;
18    //if right is available, is free and not burned
19    if(currentPosition.x+1 < width && pixelR[currentPosition.x+1][↩
          currentPosition.y] == 255 && tmpPicture.pixelR[currentPosition.↩
          x+1][currentPosition.y] != 255)
20      {
21        //cout << "right\n";
22        findNextPoint(point(currentPosition.x+1,currentPosition.y), ↩
              tmpPicture);
23      }
24    //if down is available, is free and not burned
25    if(currentPosition.y+1 < height && pixelR[currentPosition.x][↩
          currentPosition.y+1] == 255 && tmpPicture.pixelR[↩
          currentPosition.x][currentPosition.y+1] != 255)
26      {
27        //cout << "down\n";
28        findNextPoint(point(currentPosition.x,currentPosition.y+1), ↩
              tmpPicture);
29      }
30    //if left is available, is free and not burned
31    if(currentPosition.x-1 > 0 && pixelR[currentPosition.x-1][↩
          currentPosition.y] == 255 && tmpPicture.pixelR[currentPosition.↩
          x-1][currentPosition.y] != 255)
32      {
33        //cout << "left\n";
34        findNextPoint(point(currentPosition.x-1,currentPosition.y), ↩
              tmpPicture);
35      }
36    //if up is available, is free and not burned
37    if(currentPosition.y-1 > 0 && pixelR[currentPosition.x][↩
          currentPosition.y-1] == 255 && tmpPicture.pixelR[↩
          currentPosition.x][currentPosition.y-1] != 255)
38      {
39        //cout << "up\n";
40        findNextPoint(point(currentPosition.x,currentPosition.y-1), ↩
              tmpPicture);
41      }
42  }
```

This function works only on smaller BLOBs. The problem was that for each pixel a new function was called causing a stack overflow. A stack overflow occurs when too much memory is used on the call stack and often results in the program crashing.

That is why there was a need to write a new function that wouldn't call itself multiple times, since it would make the program unstable and in the end stop working. To get a better overview, the logic behind behind the recursive function was written down on paper

to be analyzed.

This lead to a reformulation of the logic, making the function non-recursive. However, it kept the same functionality, just in a while loop with eight cases for where the function should look next.

The first four cases are the ones where the function goes forward to the next pixel, with a priority for each direction. The priorities five to eight are for when the function already has tried to move forward and the only possibility is to go back. When none of these cases occur, the function is done with the analysis and the while loop is exited, as it is shown in the following code:

```
1  void Picture::startFireLoggingPersons(point startingPoint)
2  {
3    if(pixelR[startingPoint.x][startingPoint.y] != 255) //(7)
4      return;
5    //R is for Input - 255 = blob not classified, 0 = nothing found or ↩
       to small
6    //G is for way back
7
8    resetChannelsExcept('R'); //function that sets all values pixel ↩
       values to zero besides the one from the red channel
9    point currentPosition = startingPoint; // (8)
10   int pixelCount = 0; //(1)
11
12   p[currentPersonId].minX = height + 100;
13   p[currentPersonId].maxX = -100;
14
15   while(true)
16   {
17     if(currentPosition.x > p[currentPersonId].maxX)
18       p[currentPersonId].maxX = currentPosition.x;
19     if(currentPosition.x < p[currentPersonId].minX)
20       p[currentPersonId].minX = currentPosition.x; //(11)
21
22     pixelR[currentPosition.x][currentPosition.y] = 200 + ↩
         currentPersonId;
23
24     //if right is available, is free and not burned
25     if(currentPosition.x+1 < width && pixelR[currentPosition.x+1][↩
         currentPosition.y] == 255) //(3)
26     {
27       pixelCount++; //(9)
28       pixelG[currentPosition.x][currentPosition.y] = pixelCount; //↩
           (2)
29       currentPosition.x++; //(6)
30
31     }
32     //if down is available, is free and not burned
33     else if(currentPosition.y+1 < height && pixelR[currentPosition.x↩
         ][currentPosition.y+1] == 255) //(3)
```

```
34      {
35        pixelCount++; //(9)
36        pixelG[currentPosition.x][currentPosition.y] = pixelCount; //↩
              (2)
37        currentPosition.y++; //(6)
38      }
39      //if left is available, is free and not burned
40      else if(currentPosition.x-1 >= 0 && pixelR[currentPosition.x-1][↩
            currentPosition.y] == 255) //(3)
41      {
42        pixelCount++; //(9)
43        pixelG[currentPosition.x][currentPosition.y] = pixelCount; //↩
              (2)
44        currentPosition.x--; //(6)
45      }
46      //if up is available, is free and not burned
47      else if(currentPosition.y-1 >= 0 && pixelR[currentPosition.x][↩
            currentPosition.y-1] == 255) //(3)
48      {
49        pixelCount++; //(9)
50        pixelG[currentPosition.x][currentPosition.y] = pixelCount; //↩
              (2)
51        currentPosition.y--; //(6)
52      }
53      //steps back:
54      //if right is available, burned and has the biggest count(also ↩
            check if the pixel, that is compared to is available) -> we ↩
            have been there last -> go back
55      else if(currentPosition.x+1 < width && (currentPosition.y+1 >= ↩
            height || pixelG[currentPosition.x+1][currentPosition.y] > ↩
            pixelG[currentPosition.x][currentPosition.y+1]) && (↩
            currentPosition.x-1 < 0 || pixelG[currentPosition.x+1][↩
            currentPosition.y] > pixelG[currentPosition.x-1][↩
            currentPosition.y]) && (currentPosition.y-1 < 0 || pixelG[↩
            currentPosition.x+1][currentPosition.y] > pixelG[↩
            currentPosition.x][currentPosition.y-1])) //(4)
56      {
57        pixelG[currentPosition.x][currentPosition.y] = 0; //(5)
58        currentPosition.x++; //(6)
59      }
60      //if down is available, burned and has the biggest count -> we ↩
            have been there last
61      else if(currentPosition.y+1 < height && (currentPosition.x+1 >= ↩
            width || pixelG[currentPosition.x][currentPosition.y+1] > ↩
            pixelG[currentPosition.x+1][currentPosition.y]) && (↩
            currentPosition.x-1 < 0 || pixelG[currentPosition.x][↩
            currentPosition.y+1] > pixelG[currentPosition.x-1][↩
            currentPosition.y]) && (currentPosition.y-1 < 0||pixelG[↩
            currentPosition.x][currentPosition.y+1] > pixelG[↩
            currentPosition.x][currentPosition.y-1])) //(4)
62      {
63        pixelG[currentPosition.x][currentPosition.y] = 0; //(5)
```

```
64        currentPosition.y++; //(6)
65      }
66      //if left is available, burned and has the biggest count -> we ↩
          have been there last
67      else if(currentPosition.x-1 >= 0 && (currentPosition.y+1 >= ↩
          height || pixelG[currentPosition.x-1][currentPosition.y] > ↩
          pixelG[currentPosition.x][currentPosition.y+1]) && (↩
          currentPosition.x+1 >= width || pixelG[currentPosition.x-1][↩
          currentPosition.y] > pixelG[currentPosition.x+1][↩
          currentPosition.y]) && (currentPosition.y-1 < 0 || pixelG[↩
          currentPosition.x-1][currentPosition.y] > pixelG[↩
          currentPosition.x][currentPosition.y-1])) //(4)
68      {
69        pixelG[currentPosition.x][currentPosition.y] = 0; //(5)
70        currentPosition.x--; //(6)
71      }
72      //if up is available, burned and has the biggest count -> we have↩
            been there last
73      else if(currentPosition.y-1 >= 0 && (currentPosition.x+1 >= width↩
            || pixelG[currentPosition.x][currentPosition.y-1] > pixelG[↩
          currentPosition.x+1][currentPosition.y]) && (currentPosition.↩
          x-1 < 0||pixelG[currentPosition.x][currentPosition.y-1] > ↩
          pixelG[currentPosition.x-1][currentPosition.y]) && (↩
          currentPosition.y+1 >= height||pixelG[currentPosition.x][↩
          currentPosition.y-1] > pixelG[currentPosition.x][↩
          currentPosition.y+1])) //(4)
74      {
75        pixelG[currentPosition.x][currentPosition.y] = 0; //(5)
76        currentPosition.y--; //(6)
77      }
78      else
79        break;
80    }
81
82    if(pixelCount > minPixelToBeAPerson) //(10)
83    {
84      float average = ((p[currentPersonId].minX + p[currentPersonId].↩
            maxX)/2); //(12)
85      float zeroToOne = average/width;
86      p[currentPersonId].posX = zeroToOne;
87    }
88    else //(10)
89      for(int y = 0; y < height; y++)
90        for(int x = 0; x < width; x++)
91          if(pixelR[x][y] == 200 + currentPersonId;)
92            pixelR[x][y] = 0;
93      p[currentPersonId].posX = -1;
94    }
95  }
```

This is the function that replaced our previous recursive function.

startFireLoggingPersons() somehow violates the pixel system we had until then. The different channels are not used for saving the color channels of the picture any longer. Instead, the states of the analysis of the pixels are saved in the color channels. When the red channel of the pixel has a value of 255, the pixel is found but not yet analyzed by the function. When it is 0, it means that nothing is found, and when it is another number, it is going to be analyzed. One can find the BLOB index number, as well as which pixel it belongs to, by subtracting 200 from its value.

The blue channel isn't used. The green channel is used to save the way the function goes with the four first forward-cases, so that it in the following four cases knows which way the way back is. The function does that by counting the pixels with the **pixelCount** (1). The count is assigned to the green channel of every pixel that is found forwards (2). When non of the four if statements (3) have become true, the function compares the values in the green channel around itself and goes to the biggest; in other words it finds the position where it has been last (4). In order to not let the function go there again the green channel is set to zero, when going backwards (5).

The function moves through the pixels in all eight cases and updates the object's **currentPosition**, which is a struct of type **point**. This struct contains two integers called **X** and **Y**. When the object moves in a direction, X or Y are either increased or reduced (6).

After that, the while loop runs with the new given position. When calling the function, one has to put in the point where the function will start to analyze the BLOB, in this case called **startingPoint**. The starting point will in the beginning of the function be checked, if it is found to be 255 in the red channel (7) the function is allowed to continue. As the next operation **currentPosition** is set equal to the **startingPoint** (8). While going through the BLOB, the function also saves some information about it: it counts how many pixels the BLOB consists of (9). We use this information to discard BLOBs that are too small to be considered as a person (10). If this is the case, the **personCounter** gets reset to what it was before the function was called, and the pixel values are set to zero. The function also saves the biggest and smallest X positions of the BLOB (11). In case the **personCounter** is big enough, the average of these two number is stored in the **posX** and is later used to output where the person is positioned in front of the screen (12).

6.7 Persons

Now it is possible to categorize the pixels in BLOBs and assign them a number. The next question was how to keep track of where each person went from frame to frame. When we just would go through the whole picture in every frame, the BLOBs would always be numbered in the order they were found, from the upper left corner to the lower right corner.

It was decided that the persons found in the picture should be stored in an array of found persons. From frame to frame, the persons would then have to be re-found, so there was a connection between the person from the previous frame to the current frame. It was important that the program first would try to re-find the persons, and then it would look for new persons among the BLOBs that have not been touched yet.

Another thing to be considered was when people should be allowed to enter the picture, and what should happen if a person from one frame to the next disappeared. Was it likely that the person reached the edge of the picture and then walked out of it? There was also a possibility that the person was occluded by another person. How to separate this case from the case of exiting the picture?

Here we will at first describe how a person is stored in the program, secondly how it is found in the beginning, and then how it is re-found. We structure it like this, even though in the main loop we first try to re-find the person and then look for new persons.

6.7.1 The person class

When choosing how to keep track of the person that has already been found, the group decided to make a new class called **person**. This person class was decided to be a member of the **Picture** class.

```
1  class Picture
2  {
3  public:
4
5    class person
6    {
7    public:
8
9      float posX; //(1)
10     int minX; //(2)
11     int maxX; //(3)
12     float moveVector; //(4)
13     int heightOfROI; //(5)
14     double id; //(6)
15     int pId; //(7)
16
17     bool notAddedToTheNewInitialMoveVectorProductYet; //(8)
18
19     bool refind(Picture& parent); //(9)
20     bool refindOccluded(Picture& parent);//(10)
21
22   }p[50]; //(8)
23   {...}
24 }
```

The **moveVector** (4) is the vector with which the person has moved from the last frame to the current frame. This vector has just one dimension. The **heightOfROI** (5) is a variable that should hold the height of the region of interest where the person was found. The **id** (6) is the id tag that describes where the person stepped into the picture. For instance, the person number 221 to step into the picture after the program had been launched would get the id 221.

Then there is **pId** (7), which is each person's id number in their own array. This array is also a member of the **Picture** class (8). The array allows for a maximum of 50 people,

and since it is zero-based, **pId** is always between 0 and 49.

6.7.2 Finding new persons

When trying to find new persons, the group limited the search to looking at the left-most and the right-most pixels, since nobody could come from above or from underneath the camera.

Since the image processing was limited to the region of interest, looking for new persons is also limited to this area. It was decided to only look at the brightest pixel where, hopefully, the LED strip was positioned.

```
1  void Picture::lookForNewPersons(int ↩
       procentOfScreenUsedForEnterAndExit, int brightestYatX[])
2  {
3      for(int x = 0; x < width * (procentOfScreenUsedForEnterAndExit/2)↩
           / 100; x++) //(1)
4      {
5        if(pixelR[x][brightestYatX[x]] == 255)  //(2)
6        {
7          point currentPoint;
8          currentPoint.x = x;
9          currentPoint.y = brightestYatX[x];
10
11         int j = 0;
12         do
13         {
14           currentPersonId = j;
15           j++;
16         }
17         while (j < 51 && p[j-1].posX != -1); //(3)
18
19         startFireLoggingPersons(currentPoint); //(4)
20
21         if(p[currentPersonId].posX != -1) //(5)
22         {
23           personCount++; //(6)
24           p[currentPersonId].id = personCount; //(7)
25
26           p[currentPersonId].↩
                 notAddedToTheNewInitialMoveVectorProductYet = true;
27           p[currentPersonId].moveVector = initialMoveVector;
28           p[currentPersonId].heightOfROI = brightestYatX[x]; //(8)
29         }
30       }
31     }
32     for(int x = width-1; x > width - ((width * (↩
           procentOfScreenUsedForEnterAndExit/2)) / 100); x--) //(1)
33     {
34       if(pixelR[x][brightestYatX[x]] == 255) //(2)
35       {
```

```
36          point currentPoint;
37          currentPoint.x = x;
38          currentPoint.y = brightestYatX[x];
39
40          int j = 0;
41          do
42          {
43            currentPersonId = j;
44            j++;
45          }
46          while (j < 51 && p[j-1].posX != -1); //(3)
47
48          startFireLoggingPersons(currentPoint); //(4)
49
50          if(p[currentPersonId].posX != -1) //(5)
51          {
52            personCount++; //(6)
53            p[currentPersonId].id = personCount; //(7)
54
55            p[currentPersonId].↩
                notAddedToTheNewInitialMoveVectorProductYet = true;
56            p[currentPersonId].moveVector = -1*initialMoveVector;
57            p[currentPersonId].heightOfROI = brightestYatX[x]; //(8)
58          }
59        }
60      }
61  }
```

When calling this function, one has to input how big a percentage of the screen one wants to use for the enter zone (the zone where new persons are allowed to enter the picture). This percentage refers to the percent of the X axis that should be used.

The program then loops through the relevant X values in two for loops (1). The array of the brightest Y values is used in combination with the X value to find the position where we want to look for a new person. When the red channel of the pixel is 255, meaning that it is found in the preprocessing part, but not yet analyzed, the program starts to analyze the BLOB (2). This is done by first finding a **person** object in the array that is empty, by looking for one which X position is -1 (3). -1 is the value that is given to all **person** objects initially, and when they disappear. After that the function **startFireLoggingPersons()** is called (4). Since an empty **person** object was found, the **startFireLoggingPersons()** function fills the empty **person**. unless the BLOB is too small. When the BLOB is too small, the **person** gets the status of being empty; this is done by setting its **posX** to -1 again.

When the function **startFireLoggingPersons()** is called, and the **posX** is not -1 (5), the **person** counter gets increased by one (6), and the person's id is set to the corresponding value (7). Furthermore, **notAddedToTheNewInitialMoveVectorProductYet** is set to true, and **moveVector** is set to an initial value, since the person did not exist one frame ago.

In the next sections we will describe how we find the pre-found persons, as well as describing **moveVector**. Finally, the Y value in which the BLOB was found gets saved in the **heightOfROI** (8).

6.7.3 Re-finding a previous-found person

When trying to find a BLOB again, there are different approaches one can use. An often-used method is to analyze the BLOB and save specific data from it, so it can be recognized in the next frame again. Here, one could for example use the color or the shape of the object. In the case of the chosen setup, this wasn't really possible, due to the fact that the group chose to filter all the visible light out and just collect the infrared light.

This makes all the color informations disappear. Also, almost all the information about the shape get discarded, since the camera just looks at a small strip of LEDs. This does just tell us the position of the BLOB in the X axis. By combining the current X position with one or more previous X positions, one can predict the movement that the BLOB will take on the X axis, and by that establish a connection between the BLOBs from frame to frame.

The following approach of predicting movement is inspired by [Yilmaz et al., 2006].

The methods are highly dependent on the continuity of movement. The movement of BLOBs are predicted by looking at how the BLOB moved from the previous frame to the current frame. This is saved in a move vector. The theory is that when the move vector is added to the current position again, then we will get the new position.

If the program can not find anything at this point, it was assumed that the object would have stopped moving. So this is the next thing that is checked. If this isn't the case, the program takes the midpoint of these two points and move to the edges until a maximum distance to move is reached.

```
1  bool Picture::person::refind(Picture& parent)
2  {
3    point currentPoint;
4    bool found = false;
5    int maxAmountToMove = parent.maxAmountToMove;
6    parent.currentPersonId = pId;
7
8    if(posX+moveVector < 1 && posX+moveVector >= 0 && parent.pixelR[(↩
       int)((posX+moveVector)*parent.width)][heightOfROI] == 255) //↩
       (1)
9    {
10     //first priority to look is the position+the move vector (has ↩
         moved normal)
11     currentPoint.x = (int)((posX+moveVector)*parent.width);
12     currentPoint.y = heightOfROI;
13     parent.startFireLoggingPersons(currentPoint); //(6)
14     if(parent.p[parent.currentPersonId].posX != -1) //(7)
15     {
16       cout << "found first try \n";
```

```
17        found = true; //(8)
18      }
19    }
20    if(posX < 1 && posX >= 0 && parent.pixelR[(int)(posX*parent.width)↩
          ][heightOfROI] == 255 && !found) //(2)
21    {
22      //second priority to look is the position (has stoped)
23      currentPoint.x = (int)(posX*parent.width);
24      currentPoint.y = heightOfROI;
25      parent.startFireLoggingPersons(currentPoint); //(6)
26      if(parent.p[parent.currentPersonId].posX != -1) //(7)
27      {
28        cout << "found second try \n";
29        found = true; //(8)
30      }
31    }
32    //third priority is to look around the most likely point until the ↩
          max amount to move is reached
33    for(int i = 0; (int)((moveVector/2)*parent.width)+i <= ↩
          maxAmountToMove && (int)((moveVector/2)*parent.width)-i >= -↩
          maxAmountToMove && !found; i++) //(3)
34    {
35      if((int)((posX+moveVector/2)*parent.width)+i < parent.width && (↩
            int)((posX+moveVector/2)*parent.width)+i >= 0 && parent.↩
            pixelR[(int)((posX+moveVector/2)*parent.width)+i][heightOfROI↩
            ] == 255) //(4)
36      {
37        currentPoint.x = (int)((posX+moveVector/2)*parent.width)+i;
38        currentPoint.y = heightOfROI;
39        parent.startFireLoggingPersons(currentPoint); //(6)
40        if(parent.p[parent.currentPersonId].posX != -1) //(7)
41        {
42          cout << "found normal \n";
43          found = true; //(8)
44          break; //(9)
45        }
46      }
47      if((int)((posX+moveVector/2)*parent.width)-i < parent.width && (↩
            int)((posX+moveVector/2)*parent.width)-i >= 0 && parent.↩
            pixelR[(int)((posX+moveVector/2)*parent.width)-i][heightOfROI↩
            ] == 255) //(5)
48      {
49        currentPoint.x = (int)((posX+moveVector/2)*parent.width)-i;
50        currentPoint.y = heightOfROI;
51        parent.startFireLoggingPersons(currentPoint); //(6)
52        if(parent.p[parent.currentPersonId].posX != -1) //(7)
53        {
54          cout << "found normal \n";
55          found = true; //(8)
56          break; //(9)
57        }
58      }
```

```
59    }
60    return found; //(10)
61  }
```

This function contains the three priorities for where we would like to look for the BLOB

First the position is checked for where the BLOB should be when it would move with a constant speed (1). The next priority is when the object has stopped abruptly (2).

The last priority is looking several places: starting from the position between the two previous points, the function goes to both sides until it on one of the side hits the border for the maximum distance to move. This is done by having a for loop that iterates from 0 and up (3). This number is then added to (4) and subtracted (5) from the midpoint of the two first points. Like when looking for new persons, the program starts the **startFireLoggingPersons()** with the point the function has found to be true (red channel is 255) (6). If this function has set the person's position to something that is not -1 (7), it will end itself by setting the boolean variable **found** to true (8), breaking the loop (9). In the end of the function it will return whether the person was found or not (10).

6.7.4 Occluded or exited

The last thing that can happen now is that the person can disappear. When a person disappears, it is because one of three reasons: the person's BLOB is occluded by another BLOB (see figure 6.4); the person has exited (see figure 6.5) the picture; or the system has made an error. This logic is implemented in the main loop.

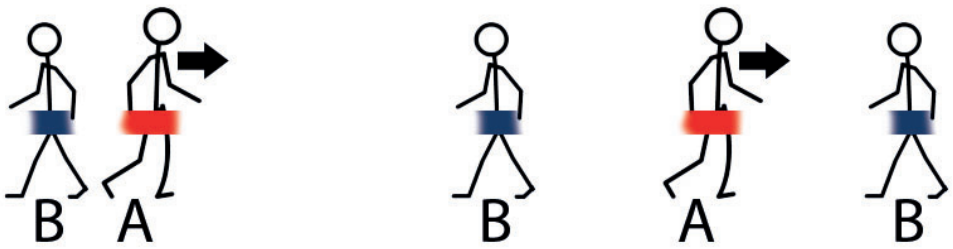


Figure 6.4. Illustration of how person B stays in the picture, while person A exits it. The red and the blue bar show their detected BLOBs.

```
1  for(int i = 0; i < maxNumberOfPersons; i++) //(1)
2      {
3        if(currentPicture.p[i].posX != -1) //(2)
4        //the person does exist
5        {
6        float prePos = currentPicture.p[i].posX;  //(3)
7          if(!currentPicture.p[i].refind(currentPicture))  //(4)
8          {
9          //new position of the person not found
10         //either person is exited or occluded
```

```
11          if(((prePos*currentPicture.width) - currentPicture.↩
                maxAmountToMove > 0) && ((prePos*currentPicture.width) ↩
                + currentPicture.maxAmountToMove < currentPicture.width↩
                )) //(5)
12          // check if the person is NOT close enogth to the edge of ↩
                the picture to exit
13          {
14            //is not close enough to the border of the picture to ↩
                  exit -> it must be occluded
15            if(!currentPicture.p[i].refindOccluded(currentPicture)) ↩
                  //(7)
16            {
17            //the person is not found, not occluded and not exited
18              currentPicture.p[i].posX = -1; //(8)
19              cout << "Person is magical disappeared"
20            }
21          }
22          else
23          {
24          //the person is exited normally
25            currentPicture.p[i].posX = -1; //(6)
26          }
27
28
29          if(currentPicture.p[i].↩
                notAddedToTheNewInitialMoveVectorProductYet) //(9)
30          {
31          //when the initial movevector isn't measured yet
32
33            currentPicture.newInitialMoveVectorProduct = ↩
                  currentPicture.initialMoveVector; //(11)
34            //since there is nothing to take the average take the ↩
                  initial move vector from this run
35          }
36
37
38        }
39        currentPicture.p[i].moveVector = currentPicture.p[i].posX - ↩
              prePos;
40        if(currentPicture.p[i].↩
              notAddedToTheNewInitialMoveVectorProductYet) //(9)
41        {
42          if(currentPicture.p[i].moveVector < 0)
43            currentPicture.newInitialMoveVectorProduct += -↩
                  currentPicture.p[i].moveVector; //(10)
44          else
45            currentPicture.newInitialMoveVectorProduct += ↩
                  currentPicture.p[i].moveVector; //(10)
46        }
47      }
48      currentPicture.p[i].notAddedToTheNewInitialMoveVectorProductYet↩
            = false; // <- and make shure that it isn't added again
```

```
49      }
```

This logic is looping through all the **person** objects of the current picture (1). At every iteration of the main loop the program first finds out if the person was previously found, by asking if its position is not equal to -1 (2). When the person was found in the previous frame, the program saves its **posX** in a float called **prePos** (3). After that, the function that is supposed to find the previous (old) person is called in an if statement (4). If the person can't be found again, the function checks if the person is close enough to the border that it could have exited (5). In this case the person gets removed out of the **person** array, by setting its **posX** to -1 (6).

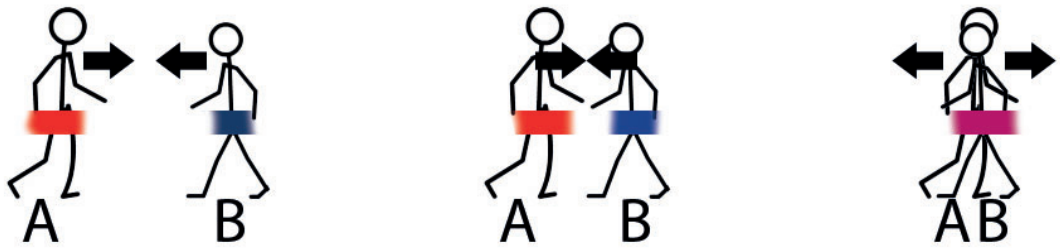


Figure 6.5. Illustration of how person B occludes Person A. The red and the blue bar show their detected BLOBs, and on the last frame their purple occluded BLOB.

If the person on the other hand is so far away that it can't exit the picture, the only logical explanation is that the person is occluded by another person. This is tested by the function **refindOccluded()**, which is positioned in an if statement like the **refind()** function (7). The **refindOccluded()** function is similar to **refind()**, it differs in that it looks for values that are not 0 instead of some that are 255. It also sets the **xPos** value of the occluded BLOBs equal to each other, when an occlusion is detected.

```
1  bool Picture::person::refindOccluded(Picture& parent)
2  {
3    point currentPoint;
4    bool found = false;
5    int maxAmountToMove = parent.maxAmountToMove;
6    parent.currentPersonId = pId;
7
8    if(posX+moveVector < 1 && posX+moveVector >= 0 && parent.pixelR[(↩
       int)((posX+moveVector)*parent.width)][heightOfROI] != 0)
9    {
10     //first priority to look is the position+the move vector (has ↩
         moved normal)
11     posX = parent.p[parent.pixelR[(int)((posX+moveVector)*parent.↩
         width)][heightOfROI]-200].posX;
12     found = true;
13   }
14   if(posX < 1 && posX >= 0 && parent.pixelR[(int)(posX*parent.width)↩
       ][heightOfROI] != 0 && !found)
15   {
```

```
16      //second priority to look is the position (has stoped)
17      posX = parent.p[parent.pixelR[(int)(posX*parent.width)][heightOfROI]-200].posX;
18      found = true;
19    }
20    for(int i = 0; (int)((moveVector/2)*parent.width)+i <= maxAmountToMove && (int)((moveVector/2)*parent.width)-i >= -maxAmountToMove && !found; i++/*check if this works*/)
21    {
22      if((int)((posX+moveVector/2)*parent.width)+i < parent.width && (int)((posX+moveVector/2)*parent.width)+i >= 0 && parent.pixelR[(int)((posX+moveVector/2)*parent.width)+i][heightOfROI] != 0)
23      {
24        posX = parent.p[parent.pixelR[(int)((posX+moveVector/2)*parent.width)+i][heightOfROI]-200].posX;
25        found = true;
26        break;
27      }
28      if((int)((posX+moveVector/2)*parent.width)-i < parent.width && (int)((posX+moveVector/2)*parent.width)-i >= 0 && parent.pixelR[(int)((posX+moveVector/2)*parent.width)-i][heightOfROI] != 0)
29      {
30        posX = parent.p[parent.pixelR[(int)((posX+moveVector/2)*parent.width)-i][heightOfROI]-200].posX;
31        found = true;
32        break;
33      }
34    }
35    return found;
36  }
```

If this function also returns false, it means that there must have been a mistake with the person and that it has disappeared out of the picture (8).

6.7.5 The move vector adaption

When configuring and setting the threshold values for the program, it was often hard to find the optimal values at which the program would operate best. This was especially the case when having to say how fast the person initially moved into the frame. That is why it was decided to let the program measure how fast people are moving from the first frame to the second.

Here we have to look at the rest of the loop that goes through the persons that are not explained yet. The idea is that when the program is running, it would take the average of the initial move vector of all the persons that have walked into the picture, and write that to a file. This file could be read the next time the program started, getting a more accurate result.

The way the average of the initial move vectors is made is that every time the person

counter is increased, the measured actual initial move vector is added to the product of all the move vectors. For this the boolean variable **notAddedToTeInitialMoveVector-Product**, which was set to true in the **lookForNewPersons()** function, is used. When finding a person or multiple persons again, this boolean is checked (9) and when found true, the move vector gets added (10). In most cases this works; the problem is just when a person is found and disappears in the next frame. To not confuse the person counter system in the program, it was chosen to add the initial move vector that the program started with again (11).

6.8 Using Unity to handle the graphical output

After doing the initial prototype, the group realized that it would be difficult to use OpenCV to both extract the data from the camera and display it in some kind of graphical way. Since the program has to loop through a lot of data continuously, there was little resources left for it to actually display the results in an interesting way without being too slow. First the group thought about using multiple threads running in the program, but even then it would be hard to display more sophisticated graphics. OpenCV is not meant as a tool for displaying graphics, but more for the analyzing/extracting part.

Then the group came up with the idea of using the Unity game engine to display the graphics based on the data from OpenCV. This had the added bonus of being able to use elements such as particle effects, physics and sound. Since some group members already had experience with Unity, it seemed a good choice. The only concern was how to send the data from OpenCV to Unity. OpenCV uses C++, while Unity uses `C#`, JavaScript or Boo as scripting languages.

6.8.1 Getting a link between OpenCV and Unity

It was first considered to make C++ write to a text file that could then be read by Unity. This approach was not used in the end, since it would need a lot of reads and writes to the hard drive - doing this repeatedly could destroy it. Also, there is a chance that there would be a mismatch with Unity trying to read the file while C++ is writing to it (the text file has to be opened, then written to, and finally closed). It was also found that this method was quite slow. Therefore it was decided to write to the Clipboard in Windows.

The Clipboard acts as a short-term buffer to store temporary data in the RAM (Random Access Memory). Using copy and paste functionality, it can work like a carrier for transferring information from one program to another. [Janssen]. This avoids writes/reads from a physical hard drive and will not wear it out, as well as being faster than saving to a text file on the computer.

Copying data from C++

In the C++ program a text line is saved as a string. This is done using the function **SetClipboardData()** where the data type has a format called CF_TEXT that stores the data as text [Microsoft, b] [Microsoft, a]. This string describes the information of where a person is currently located.

First the C++ program assigns a unique ID name, and then a floating point value between 0 (left) and 1 (right). If the float is -1, it means that the person is currently not there, i.e. there is no person in front of the camera. The reason for this is to make it easier for Unity to move the characters around on the screen. If a position is different from -1, it means that it should be shown on the screen. Else if the position data is -1, the character will be moved outside of the screen area and not shown.

Below is an example of the string that is copied to the Clipboard from the C++ program:

q iObject_1p0.1 iObject_2p0.2 iObject_3p0.3 iObject_4p0.4 iObject_5p0.5 iObject_6p0.6 iObject_7p0.7 iObject_8p0.8 iObject_9p0.9 iObject_10p-1 q

Even though the program in theory could track more people, a finite number of 10 was chosen to reduce the information used. Also, it seems unrealistic that there would be more than a total of 10 persons in front of the camera at the same time. Having this limit made it easier to handle the objects in Unity: instead of spawning a new character every time a person steps in, it just reuses the same 10 characters and move them around appropriately.

Reading data in Unity

In Unity, using `C#`, the text is read from the Clipboard via the static function **GetSystemCopyBufferProperty()**. The first thing that happens is that whitespaces are removed. Then the string is split multiple times using different delimit characters such as the q's, i's and p's.

'q' is simply used to start and end the string, so nothing before or after is used. This was to ensure that incorrect data would never be read in Unity, i.e. if there was any leftovers from the last time data was copied to the Clipboard. It made debugging easier, since it would produce an exception error if the data in the Clipboard did not start and end with 'q'.

The character 'i' is used to identify each person and assign it a name. In this case the name consists of a number that increases for each person that has been in front of the camera. Doing this makes it easier to identify a specific character in Unity and check if it is moving as intended. The floating value after the 'p' is the actual position of the character, which is then translated into a coordinate system in Unity where e.g. 0.1 is to the left side and 0.9 is to the right side. As mentioned before, a -1 is an invalid value, meaning that the person is not present and will therefore be moved outside of the visible screen.

In the end, all the position data are stored in an array with the position data called **Position[]**. In this case Position[0] would be the character named "Object_1" and have the position 0.1. Position[9], on the other hand, has a -1 and would therefore not be shown on the screen. These values are then mapped to a coordinate system in Unity.

It should be noted that these values are only for the X position. It was chosen not to use the Y position, and since there is no depth, the Z axis is totally excluded (the program is kept in two dimensions). However, the characters move somewhat randomly up and down in the Y direction every time they are moving around on the X direction - depending on what type of character it is. For instance, the angel character should float in the air, while the snowman should be on the ground. Doing this helps avoiding characters occluding

each other, since more of the screen is being used (see figure 6.6 and 6.7).

Figure 6.6. Without random Y position.

Figure 6.7. With random Y position.

Having this array of position data makes it easy to loop through a list of Unity game objects and move them around accordingly.

Additional functionality was implemented to make the characters move more smoothly than simply jumping from point to point in a one-to-one relationship with the data from the Clipboard. First of all a threshold value is used so a character will only move if the person moves enough for the system to trigger it. This means that if a person is standing still in front of the camera, the character on screen will not jump around due to the small changes in the decimal values (e.g. from 0.542 to 0.523 would not make the character move).

To make the characters move even more smoothly, it was chosen to de-synchronize the position data gathered from the camera and the character's actual position; a target system was implemented for this purpose. This works by at intervals finding a new target to move to, and then a character will gradually move towards that target. Then after a small period of time, the target will be updated. Like the threshold, it makes the characters move in a more natural line.

6.8.2 Using Unity to display the characters

The Unity game engine is typically used to make 3D games, but it is still possible to make programs in 2D, using 2D sprites/textures and an orthographic camera that doesn't display depth.

The scene used is quite simple. It consists of a background image showing mountains. To achieve a more dynamic look, a couple of elements are added to make the whole scene spring to life.

First there are are a collection of Christmas trees with some Christmas balls and bells hanging on top. Using built-in physics in Unity, these bounce back and forth, simulating the wind blowing. Along with this is the falling snow, which is made as a particle system in Unity. This ensures that even when there are nobody in front of the camera, the program still feels alive.

6.8.3 Time manager and Santa Claus

As mentioned in chapter 2.1, the library is build on the theme of serendipity, i.e. to surprise and be new. When making the program, the group had an idea about having it change depending on the time of the day. Using the clock in Windows, the program could do something different at different times.

In the program, the sun will gradually rise as time passes, as well as the light will become brighter/dimmer, just as it is happening outside. Figures 6.8, 6.9 and 6.10 illustrate this. One thing we needed to have in mind was the light setting. On a computer screen the image can be really bright and really dark, but on a canvas and a projector it is not easy to see a dark image. Therefore the brightness level was set a little higher than normal, so it would still be possible to see it properly on the canvas.

Figure 6.8. Morning.

Figure 6.9. Noon.

Figure 6.10. Evening.

Another aspect is Santa Claus who will come at a randomly-chosen time (see figure 6.11. When he arrives, jingle bell sounds play, as well as his iconic "ho ho ho" laughter. Santa will then drop packages that can be interacted with. This happens by moving into the objects, pushing them with the physics system in Unity. It should be noted that Santa does not come too often, since it would disturb other visitors at the library. In average, Santa Claus arrives once an hour.

6.8.4 Snow ball

To get a little more interaction, a snow ball is placed in the scene (see figure 6.12). This can be pushed around by the characters, and it will gradually grow bigger and bigger until it at some point explodes and disappears for a short period of time.

As mentioned in 5.0.3, sheets of sprites were used to create a sense of motion. This was done by looping over an image containing multiple frames. Depending on which way the character is facing (left, right or idle), the animation will change. Without these animations the whole program would feel rigid and stiff.

6.9 Making an automatic batch file

One of the goals in the problem statement (2.2.1) was to have the programs work as easily as possible. We could not expect the library staff to use a lot of time to turn everything on each day. Therefore we wanted everything to start automatically. This was done by creating a batch file that starts when Windows boots up. The batch file then opens the OpenCV program that initializes itself. After 15 seconds Unity opens and starts receiving

Figure 6.11. Santa Claus comes at random times and drop presents. These can be pushed by the characters. They disappear after some time.

Figure 6.12. The characters can push the snowball to make it grow. When it becomes big enough, it explodes into colorful fireworks.

data from the Clipboard. In the end, the only thing needed is just to turn on (and off) the computer; no additional input is required for the programs to start.

6.10 Changes and limitations

During the setup at the library multiple things were learned. Since it was not possible to be at Hjørring Library every day, especially not in the beginning of the project, it was difficult to test the proper camera settings, etc. Therefore we needed to re-adjust some elements when we visited the library.

6.10.1 Use of LED strips

The original plan was to be able to track people in front of and behind the bookshelf (dubbed "den røde tråd). Therefore two sets of LED strips were put up: one behind the canvas (see figure 6.13) and one beneath the red bookshelf (see figure 6.14).

What the group did not have in mind was the perspective that the camera would capture (figure 6.15). Since the program had already been written without this in mind, it was difficult to make it work with both LED strips. Due to lack of time it was decided to not use the LED strip in the background, but instead solely focus on the LED strip placed underneath the red bookshelf.

One thing that could be interesting to work on in the future would to be the make the background LED strip recognize hands. When trying out the program, it seemed intuitive to either jump or raise one's arms in the air. The LED strip could be used to track this, so the characters in the program could react accordingly. Sadly, there was no time to implement this idea.

Figure 6.13. Behind bookshelf.

Figure 6.14. Beneah bookshelf

6.10.2 Santa Claus and his "ho ho ho" sound

As mentioned previously, it was important not to disturb visitors with too much sound. After having the program running for a few hours, one of the library staff members contacted us and asked if it was our program that kept making weird noises. It turned out that she was talking about the Santa Claus "ho ho ho" laughter, which, in her opinion, sounded like a deep and growling voice. Apparently the speakers that we were using made

Figure 6.15. Camera point of view.

the "ho ho ho" sound different than on our laptop computers. The staff member said that it was a little disturbing and that she thought kids would be afraid, since it sounded a little creepy. On top of that, it sounded like she was a little annoyed of the sound, since she was working close-by. In the end the group decided to just get rid of the sound and only use jingle bells for Santa's appearance.

6.10.3 Shadows on canvas

An unexpected problem was the fact that when somebody was standing in front of the camera and projector, it draws a rather big shadow on the canvas. When the group first went to Hjørring to research and look for the location, it didn't seem like it would be a problem.

Figure 6.16. If a visitor is standing too close to the red bookshelf, a shadow will cover the canvas.

Figure 6.17. The projector needed to be placed higher to minimize shadows.

When the final setup was built, it turned out that in the worst-case scenario the shadow

would hide a lot of the graphics drawn on the canvas (figure 6.16). To reduce the shadows, the projector was placed on top of a couple of boxes (see figure 6.17). However, to completely remove the shadows one needed to use a lot of boxes, or, alternatively, hang the projector in the ceiling. Due to time constraints this was not possible. Also, the library staff was afraid that the projector would be too unstable.

6.10.4 Needing sound to draw attention

One thing that the group quickly learned while setting the equipment up was the fact that a lot of people didn't even realize the fact that they were interacting with the canvas when they walked past. Many people looked straight forward when they were walking, instead of looking to either sides.

It was suggested to use sounds to draw attention to the canvas. An idea was to make a "magical fairy dust sound" play every time a person stepped into the scene. We tried this, but it didn't give a good result. One reasons was the fact that the speakers are placed inside the projector which is placed on the opposite side of the canvas. Figure 6.18 illustrates the problem.

Figure 6.18. Since the speakers are inside the projector, audio will draw attention to the wrong side.

The solution would be to place some external speakers near the canvas, but due to time constraints this was not implemented. Also, the sounds turned out to be a little disturbing, so in the end it was chosen to just drop them.

6.10.5 People walking too fast

Another reason why the canvas did not draw proper attention when people walked past was the fact that it was quite slow to display the characters. In the beginning there was no real connection between a person walking and the character on-screen. This was due to multiple things: the C++ program having to process the data and send it to Unity, and the speed of the characters. It took some time for the characters to "catch up" to the actual person, so the character would be lagging behind. The solution was to teleport the character if the distance became too big.

6.10.6 Camera not steady

Initially there were some problems with the webcam moving just slightly. This would make the whole program malfunction, since the region of interest would change. To avoid this the webcam needed to be steadied with some strong tape. Also, the OpenCV program was modified to re-synchronize itself every hour. This would mean that if somebody accidentally touches either the camera or the LED strips (which happened surprisingly often), the program would re-adjust after some time.

The group needed to make a decision: how often should it re-configure itself? If it did it too often, the chances of people standing in front of the camera while it was trying to re-configure was bigger. On the other hand, if it does it less often, the time where it won't work properly would be longer. In the end an hour was chosen to be a good compromise.

6.10.7 Programs crashing during the day

Before testing the program at the library, they hadn't run for more than a few minutes continuously. Since the goal was to let them run for a whole day (10:00-18:00), it was important that the programs would not crash. Initially, there were problems with both C++ and Unity.

Even though the Unity program is quite simple, it turned out to hog a lot of memory over time. It took several days for the group to realize why. After multiple debugging sessions they found that apparently Unity does not deleted old textures and materials. As mentioned earlier, animations are done using sprite sheets. Doing this changes the material on a Unity object (a material describes how a texture should be displayed using a shader). Every time a new material is applied, i.e. when the animation needed to change, the old material was kept in the background. Over time, these materials stacked up. This meant that the RAM usage constantly increased until the program finally crashed.

To resolve this problem proper deletion of the materials were implemented. It turned out that the C# garbage collector did not do this automatically, so we manually had to delete old instances of materials. This lead to more stable memory usage. Just to be sure that Unity does not use too much memory during a whole day, it was made so that it reloads itself every third hour. This only takes a fraction of a second, so it shouldn't disturb the interaction.

Testing the installation 7

7.1 Introduction to testing

Testing is an important part of the project. It is a great opportunity to get an impression of how the visitors of the library experience the installation.

The testing of the installation took place at Hjørring Library during its opening hours in December. The goal was to see how people interacted with the installation and whether they grasped the concept. The focus for the project is to entertain people of all ages, so it is interesting to get an overview of how this goal is achieved. It was also of great importance to observe if the installation actually drew visitor's attention, or if people just walked past it without noticing it. It was of interest to see if any patterns or tendencies emerged.

7.1.1 Observations

The test was focused on the visitors at the library. During the test, one group member focused on observing people's habits, gender and ages, while the other group member interviewed the visitors.

For the purpose of the test, two types of visitors were defined: active users and passive users. An active user is a visitor that by own initiative chooses to engage with the installation. A passive user is defined as a visitor that casually walks past the installation without deciding to engage with it further.

Two group members were present at the installation to observe people walking by. It was important that the group members did not position themselves too close to the installation, so that they would have an influence on the way people reacted. One of the group members focused on people using the installation actively and the other group member focused on people using the installation on a passive level.

7.1.2 Interviews

The testers were positioned on either side of the installation, to ask people who walked by the installation to participate in answering some simple questions. People were kindly asked if they were interested in giving some feedback to improve the user experience.

The idea of placing two members on either side of the installation was in order to gather information after the visitors had the chance to engage with the installation. The

questionnaire prepared for the interviews is illustration on figure (7.1) and it contained the following questions:

Age?
Gender?
Why did you visit the library today?
Did you try the installation?
If yes:
What made you use the installation?
Do you think the installation fits the library and the Christmas theme?
What could be done to improve the experience?
If no:
Why did you not se the installation?
Do you think the installation fits the library and the Christmas theme?
What can be done to make you use the installation?

Figure 7.1. Figure showing the questionnaire used for the interview.

7.2 Results

After going to Hjørring Library to run the different tests, the analyzing part was to be made. The testing ran for an entire day, to see if there were any changes in tendencies during the day. The testing will be evaluated and analyzed in the following subsections.
All test participants were offered some Christmas treats in return for the help. In addition the majority of visitors were positive and friendly-minded. Unfortunately, December is a busy month, so many people were occupied and did not have time to participate in the testing. On the day of testing there was a low amount of visitors at the library. According to one of the staff members at Hjørring Library, December is not the most visited month of the year due to Christmas.

7.2.1 Results of the observations

The following is based on observations during a Wednesday at the library. Even though this might not be representative of all of the library's opening hours, some general observations were made.

Testing the common tendencies was done by observing how people engaged in the installation. The group members positioned themselves away from the installation to avoid interference and to sustain consistency.
After about an hour it was clear that there was some common tendencies, some more desirable than others. The one tendency that drew the most attention is the fact that people did not notice the installation when they walked past it. Many visitors went straight from one location to another without really looking around. This proved a problem for the installation and it was considered critical.

Another tendency was that children were more likely to notice and interact with the installation compared to adults. It was assumed that the cartoon art style appeals more to young people.
It seemed that the children visiting Hjørring Library were more likely to engage in the Christmas exhibitions and decorations around the library.

7.2.2 Results from the interviews

The interviews were made after the visitors had had the chance to engage in the installation, as described in the above. It was decided to interview both passive and active users of the installation in order to compare the answers from the questions for both active and passive users. Secondly, it was important to generate some specific questions that applied to either group, in order to receive valuable feedback. A total amount of 10 persons (five males and five females) were interviewed. The test participant's ages ranged from 14 to 67.
Out of the 10 testers, four engaged actively in the installation, while six engaged on a passive level.
The main reason why the persons were visiting the library varied widely but can mainly be divided into two groups. The first group consists of persons who came to the library to have fun. The other group of people who ventured to the library, did so to borrow books. The fact that people visiting the library are so different gives a great approach for the testing, as it provides different perspectives.
First people who engaged passively in the installation will be evaluated and later the answers submitted by the active users will be evaluated.

Passive interviewees

The passive group were asked why they did not use the installation. Five of the six passive users did not use the installation because they did not notice it. The only passive visitor who actually noticed the canvas but did not interact with it, was a 19-years-old male. The subject's reasoning was that the installation was too childish.

Next question was how the installation fitted the library and the Christmas theme. The majority, 5 out of 6 participants, thought that the installation fitted the Christmas theme. Furthermore a woman expressed that the installation fitted Hjørring library with excellence, as its interactivity complimented the general interactive element of Hjørring library well.

The last question for the passive users regarded which changes that had to be made to make them interact with the installation. Most of them mentioned that there had to be drawn more attention towards the use of the installation.

Active interviewees

The first question the active users were asked to answers was concerning the use of the installation. It was of interest to know what made them interact with the installation, however this gave very different answers. One tester said that that primary reason for

using the installation was due to his daughter who's attention was drawn immediately. However the most common answer was that the installation drew attention as it differed from the rest of the exhibitions at the library.
Like the passive interviewees, the active users were asked to elaborate on how the installation fitted the library and the overall Christmas theme. All active users said that the Christmas theme fitted the library very well. One tester even said that he hoped that the library would do something similar the upcoming years.
The last question asked, was how to make the installation even better. Two tester said that further development of the interaction would improve the quality of the installation. In continuation of improved interaction, a proposal was to make the user able to touch the screen.

Conclusion 8

In this chapter we will conclude whether or not the project was a success. After having reached a conclusion in the project, we will reflect upon the experiences and knowledge gained during this semester project.

In the beginning of the report a series of problem definitions were listed. Below these are included and we will describe to which degree each of the goals were met.

Problem statement:
The group decided that in order to succeed, the program should meet the following sets of requirements:

- Low maintenance: The program should be easy to use for the library staff.
- The installation should fit the overall theme of Hjørring Library during December.
- It should be entertaining for both active and passive users.
- Multiple people should be able to use it simultaneously.
- All of the image processing functionality should be written by the group.

The parts from the problem definition will be listed throughout this chapter and commented upon.

Low maintenance: The program should be easy to use for the library staff

From the beginning of the project, it was a goal to make the program easy to maintain for the library staff. This was accomplished by limiting the actions required to start the program. The only thing that is needed to turn on the installation is power on three electronic systems. This was explained to the staff in the form of these simple instructions:

1. Press button 1 - on/off button to turn on the LEDs.
2. Press button 2 - on/off button on the projector.
3. Press button 3 - on/off button on the computer.

Ultimately this should prevent errors and make it possible for the staff and the library to initialize the program without assistance.

The installation should fit the overall theme of Hjørring Library during December

To make the installation fit the theme of Hjørring Library during December month, Christmas was brought into the project. All the graphical elements support the Christmas theme. As a part of the final testing, visitors of the library were asked whether the

installation fit the library. Almost every participant answered that it did indeed fit into the library and the Christmas theme.

It should be entertaining for both active and passive users

Visitors at the library are able to walk past the canvas, spot a character moving, and then decide to engage further or not. The fact that there are multiple characters selected randomly, gives the installation an interesting touch.

Multiple people should be able to use it simultaneously

Implementing a way to track multiple persons from frame to frame, makes the program able to represent up to 10 characters at a time.

All of the image processing functionality should be written by the group

All image processing functionality was written by the group, the OpenCV library is only used to access pixel values.

Concluding experiences

During our time at the library, both our observations and the information learned from interviews correlated to highlight that the installation was often overlooked. Despite this the project received positive feedback both from the visitors and the library staff, as seen in the attached e-mail.

> E-mail December 13, 2012
> Tak tak, det skal vi nok klare!
> Og tak for jeres indsats!
> Måske føler I at det drukner lidt i alt julehalløjet her hos os, men for os er det en værdifuld erfaring. Som I kan se så er de andre juletiltag og udstillinger lidt mere traditionelle og mere til at kigge på. Og for os, og brugerne, er det vigtigt at vi er så alsidige som muligt, og hele tiden afprøver nyt.
> Med venlig hilsen Tone Lunden

Perspective 9

In the program's current state, it would be relatively simple to change the graphics, making the installation fit other themes than Christmas. Also, more awareness about it could be gained by placing information banners close to the canvas, describing what the project is about and how to interact with it.

During the development of the Magic Canvas, several ideas for improvements arose. When performing the tests is was clear that people wanted more ways to interact with the installation, such as using their arms and legs. Multiple or more sophisticated cameras could be used to gain more possibilities for this purpose.

There are also multiple ways that the concept could be expanded upon. As mentioned in 1.1 an early idea was to scan books. Now that the project is finished, this still seems like an idea that could be worked on in the future. One could image a more game-like program where people were encouraged to go out and explore their local libraries. This could be coupled with learning material from the library to create a setting for playful learning.

Appendices 10

10.1 Appendix A: OpenCV code

The complete code can be found on GitHub: `https://github.com/Wikzo/MagicCanvas_MedialogyP3_2012`

10.2 Appendix B: AV Production

The AV Production for Magic Canvas can be found on YouTube: `http://youtu.be/HlmA0AzRx0o`

10.3 Appendix C: Instructions for the library

Look at 10.3 The following e-mail was sent to the library, describing how they should start the programs. A hand-written note was also given to the library.

December 10, 2010
Hej Tone og Martin.

Vi er nu færdige med vores "nissehue-spejl", så det skulle være klart for jer at sætte op.

Jeg har givet to siders papir med instruktioner til en bibliotekar, som skulle give det til dig, Tone, i morgen.

Vi har forsøgt at gøre opsætningen så simpel som muligt:

1. Tænd for strømadapteren som er placeret i reolen under lærredet.
2. Tænd for projekteren som står på toppen af bogreolen. Man skal holde knappen i bund i nogle sekunder.
3. Tænd for computeren. Den skulle gerne starte op med programmerne selv. Alting burde køre nu. Mus og tastatur er ikke nødvendig.

Når I skal slukke, skal I blot trykke på de samme kontakter. Dog skal man trykke to gange for at slukke projekteren, og knappen skal holdes i bund i nogle sekunder på computeren (I behøver ikke at slukke ved at trykke på "Start" og "Luk Computer").

Vi håber, alting fungerer. Hvis ikke kan I kontakte os på telefon.

Mange julehilsener fra Medialogi 1223
http://medialogyp3.wordpress.com

10.4 Appendix D: Library visit November 19

Tools we need

- Record equipment (for timelapse)
- Both IR and normal webcam
- Canvas with reference points

General agenda

1. Look at the setting (lights, projector, etc.)
2. Where to setup the camera and do we have a region of interest?
3. Observation on raw people - record timelapse.
4. Initial test setup - see how people react, and if we need anything for the system to work better

Talk with people

How do they experience the library? Do they like the concept of "an experience place"? Do they want to stay there and hang out? Casual?

Are they in a hurry, going from A to B quickly, or do they want to walk around and browse?

Did you notice the camera? - How did being recorded make you feel?

Measure what their "Christmas mood" is

Bibliography

Cambridgeincolour.com. Cambridgeincolour.com. *Understanding Digital Camera Histograms: Tones and Contrast*. URL http://www.cambridgeincolour.com/tutorials/histograms1.htm.

Janssen. Cory Janssen. *Techopedia*. URL http://www.techopedia.com/definition/4646/clipboard.

LPI. LPI. *IR spectrum*. URL http://www.lpi.usra.edu/education/fieldtrips/2005/activities/ir_spectrum/images/emspectrum.jpg.

Microsoft, **a**. Microsoft. *Using the Clipboard (Windows)*. URL http://msdn.microsoft.com/en-us/library/windows/desktop/ms649016(v=vs.85).aspx.

Microsoft, **b**. Microsoft. *How the Clipboard Works, Part 1 - Ntdebugging Blog*. URL http://blogs.msdn.com/b/ntdebugging/archive/2012/03/16/how-the-clipboard-works-part-1.aspx.

Moeslund, **2012**. Thomas B. Moeslund. *Introduction to Video and Image Processing - Building real systems and applications*. ISBN: 978-1447125020, Handbook. Springer, 2012.

Mukherjee. Manasij Mukherjee. *A Gentle Introduction to C++ IO Streams*. URL http://www.cprogramming.com/tutorial/c++-iostreams.html.

SIEMENS, **2004a**. *SIEMENS*, 2004. Datasheet: SFH487P - GaAIAs INFRARED EMITTER.

SIEMENS, **2004b**. *SIEMENS*, 2004. Datasheet: SFH484/485 - GaAIAs INFRARED EMITTER.

Suenson et al., **2010**. V. Suenson, H. Harder, N. Tradisauskas, A. K. Simonsen, M. Knudstrup, Institut for Arkitektur og Design, Natur-og Sundhedsvidenskabelige Fakulteter De Ingeniør og Aalborg Universitet. *Walking the Library*. Aalborg Universitet, 2010.

Tsesmelis, **2012**. Theodore Tsesmelis. *Image Processing - From Theory to Practice*, 2012.

Visualizeus.com. Visualizeus.com. *Color pencils*. URL http://cdnimg.visualizeus.com/thumbs/fb/72/color,colorful,heart,love,photoart,photography-fb725aeec58606d28e4b3375569be31e_h.jpg.

Yilmaz et al., **2006**. Alper Yilmaz, Omar Javed og Mubarak Shah. *Object Tracking: A Survey*. 2006.