CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Investigating Human-Computer Interaction with Motion-Capture Algorithms using a Microsoft Kinect

Bachelor's Thesis in Computer Science and Engineering

Jens Christensen

jens.christensen 'at' fripost.org

Oliver Carlsson

oliverc 'at' student.chalmers.com

Jonas Brandvik

jonasbrandvik 'at' hotmail.se

Department of Computer Science and Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY

Gothenburg, Sweden 2012

Abstract

This bachelor thesis investigates tracking of the human body in a 3D environment using known computer-visual algorithms. This is done by evaluating a variety of filter-, detection-, and tracking-algorithms, and assessing their strength and weaknesses in different environmental conditions that are found in most indoor situations. We will also discuss different combinations of algorithms that work together to optimize efficiency in a variety of conditions with limited resources.

The purpose of the developed application has been to evaluate how the algorithms perform and investigate the different possibilities to track using an ordinary laptop and a Microsoft Kinect. Finally, we discuss at what level of detail the tracking could be used and investigate if this kind of system could manage emotional capturing i.e. tracking facial expressions simultaneously with body movement.

Contents

1 Introduction

This section clarifies the benefits and reasons to develop a motion tracking system. Three key items are composed to concisely state the intention of this investigation. Because of the magnitude of this research area a few limitations had to be set in order to focus on the purpose. The limitations explains why 3D modulation and machine learning were left out.

1.1 Background

To communicate with computers through human interaction by using gestures and movements is fascinating. The data acquired from a motion capturing system can be used in the field of robotics for creating virtual interfaces and to make characters in computer games move realistically. The motion capture suit used in the movie industry has multiple references attached to it, which aids the software to acquire correct data, and analyzes where each reference-point is positioned. Motion capturing suits are often used by the game and movie industry to create realistic movements for computer generated figures. Positioning and motion analyzing of other objects are often done by lasers. In the production industry, laser-positioning systems are together with industrial robots used for locating objects and determine its position and movement. Both of these existing technologies are expensive, and therefore the opportunity to create an alternative solution arises. Is it possible to make an open source competitive motion tracking system that is easy to use and is addressed to people and companies with limited resources?

Research in the field of computer vision is not a new area. It has existed for a long time (by computer standards), and progress is being made continuously. One of the most common computer-vision libraries is OpenCV, which was originally developed by Intel cooperation. It includes features like shape recognition, a big collection of different tracking-algorithms, and motion-analysis techniques. With this specific library, and a few others, all the necessary tools are provided to implement a robust and accurate motion tracking application. The only thing required is a camera with depth sensors.

Microsoft released the Kinect input-device in November 2010. It provides depth and color amongst other features. It did not take a long time before the Kinect was hacked and libraries to communicate with the device was released. The source community quickly embraced this new technology, and all sorts of hacks circulated on the Internet. A framework for motion capture of an arbitrary object could be built using the Kinect input device and various computer vision libraries.

1.2 Purpose

As the development of the technology has progressed during the recent years, it has been proved that it is possible to do motion-capture with acceptable accuracy without having expensive high-tech equipment. The progression has gone from million dollar movie-projects, all the way down to controlling the latest video-games with the Microsoft Kinect. The problem is to take the technology to the next step, integrating motion-capture in everyday life and incorporate it into commercial and non-commercial products. The project has focused on fulfilling three key purposes:

- To create and implement an open source tracking application that may be used for human-computer interaction.
- To analyze a selection of tracking algorithms, find their strengths, weaknesses, and find appropriate combinations of techniques to get robust tracking results in an every day environment.
- To enable *emotional capturing*, capture facial expressions as well as body movement with color- and/or depth-camera e.g, Kinect.

1.3 Problem

Motion capture is the technique used for computers to detect and identify objects and movement during a video-stream. This can be solved by using a variety of algorithms that either identifies the object as a whole, *object detection*, recognizes certain features that the object has, *feature extraction*, or identifying movement in the video-feed, *motion tracking*. To know what algorithm should be used when and where will be investigated in this project.

1.4 Limitations

In all sorts of projects, there need to be limitations so that developers can maintain focus of the goal. This section will declare boundaries of the project and explain why certain aspects are left out.

1.4.1 Machine Learning

Within the subject of computer science developers and scientists strive to increase computers reasoning, and intellectual capabilities. This branch in computer science is called artificial intelligence, and a large part of this research uses machine learning techniques. The concept of machine learning is based on computers and programs ability to predict logical outcomes through information acquired from data sets [17]. By having machine learning in this kind of application, patterns can be stored and accumulated in order to predict certain patterns and recognize objects in images. This is something that will not be consider during the project because of its complexity.

1.4.2 3D Representation

When tracking objects with a Kinect input device, the depth and RGB data retrieved can be mapped into a 3D environment. One of the core functions in the motion capture application is the ability to track arbitrary objects[1]. These objects can be virtually represented in a 3D environment. The main purpose of this project is to focus on motion tracking. Therefore, mapping real world objects into a 3D environment will not be handled.

1.5 Outline

The report discusses what tracking is, different methods for tracking, various filters, and object feature enhancement techniques. Each section starts off with an explanation of how that particular algorithm or technique works, followed by a discussion on how and when to use it. The algorithms are also implemented in an application called *Kinect motion capture* where they can be evaluated. Note that the example code uses the python wrapper from OpenCV.

The report is organized in the following order: *Section 2* explains which technologies are being used, the approach taken for evaluating tests, documentation, and licenses for the actual application. *Section 3* introduces the Kinect followed by the definition of a digital image, how a computer interprets the information contained in an image, and a closer look on a variety of different filters. *Section 4* is about how to detect specific objects with certain features in an image. *Section 5* is about various tracking algorithms. In this section, tracking techniques are explained and evaluated. The final two sections, *Section 6* and *Section 7*, discusses conclusions and speculations of what can be further analyzed.

2 Method

In parallel with the report, an application was developed[25]. To enable an efficient work pace, a few key logistic issues were discussed before starting with the essential parts of the project. These issues includes documentation, licensing, libraries, and choice of programming language.

2.1 Documentation

To document the process and findings, there were two specifications required.

- *Version control compatibility*: Easy to keep track of what has been written, when, and by whom in all documents.
- *Merging*: There must not be any splices and the style must be consistent throughout the document.

LaTeX[4] is entirely text-based, which allows any text-editor to edit the content. This simplifies changes and additions to the content, because the structure and style is set at the top. Our version controller is able to read and track changes inside the documents.

[1] Arbitrary Object - Refers to an object with a limited size, an unspecified shape, and an unspecified color, for example a coke can, a toy, etc.

2.2 Unified Modeling Language

A distinct model for the object orientated parts of the project was acquired by the unified modeling language. Modeling and abstracting each part of the software in a good way helped the development process and is valuable for future improvements of the source code. To generate class, sequence, and case diagrams, we used a tool called *PlantUML*[8]. Since it uses plain text with syntax, it is really easy to modify it with scripts and get all the advantages which version control provides.

2.3 Version Control

It is preferable to have version control to keep track of all the changes being made in the project over time. The advantages of version control is the ability to check what code has been written or modified by whom, as well as getting the advantage of history and progress for the project. Keeping track of history is essential when there is a need to revert to a stable state if anything unpredictable would happen. *Git*[51] was selected because a developer has his/her own local repository that gives a decentralized structure where users can creates branches for different parts of the source code. Git is open sourced under GPL v2.

2.4 License

Selecting the ideal license for a software developed project is important. There are many of them and they may be hard to interpret from a legal point of view. Having a license is recommended if the project should remain to be open source. A license enables the code to be copyrighted to the author, and if a developer wishes to use the code, the copyright holder has to be asked for permission. The following licenses were discussed for the project: GPL, BSD and MIT

The standard version of *General Public License* (GPL)[35] prevents organizations or people from making the code proprietary, and all additional changes to the source code must remain in the form of free software. To ensure all source code would remain free, the license itself clearly states all the requirements that needs to be fulfilled in order to modify and/or release the source code.

Berkeley Software Distribution (BSD)[2] license is non-comprehensive and easy to understand, with a legal disclaimer. The copyright holder will not be held responsible for any damage that his/her source-code may cause and there are small restrictions for redistribution of source-code. There are currently two different versions of the license, excluding the original one: BSD-3, and BSD-2. Only difference between them is that the BSD-3 clause license protects the original copyright holders and/or organization to be used later for promotion purposes. This concerns software released using the original source code.

The MIT License (Massachusetts Institute of Technology)[6] is similar to the BSD-2 clause license. A disclaimer protecting the copyright holder from legal interventions if the source-code causes any damage and thereby redistribution of the source-code by a third party, has very small limitations, both in commercial products and open-sourced software.

It is desirable to select a license which reflects the overall spirit, and openness of the framework we are developing, without restricting the user. The license choice for this project was given in favor for the BSD 3-clause license. The main reasons for picking the license were due to its simplicity.

The license is easy to understand, and the fact that the framework can be used both in open source, and proprietary products.

2.5 Libraries

Various libraries have been validated thoroughly. What features the library provides, how active the community is, and which license it is released under are the most important criteria when the choice of libraries was made. Below is some information about different open source libraries that were considered being used during this project.

OpenCV

OpenCV (Open Source Computer Vision Library)[17][1] is a library with its main focus on real-time applications. It is written in C and C++ developed by Intel during the later years of the 1990's. The library has over 500 functions in different areas of vision and image analysis including: gesture recognition, facial recognition, motion tracking, and stereopsis (3D vision). It also includes *Machine Learning Library*. This sub-library is developed for statistical pattern recognition and clustering but is general enough to be used for most machine learning problems.

PCL

PCL (Point Cloud Library)[46] handles data acquired from modern visual sensors such as: laser scanners, stereo cameras, and ToF-cameras[2]. The scene is created by points, and each point contains a position in space (x,y,z) and optional RGB color or gray-scale. Point clouds create a three dimensional picture, PCL's purpose is to merge point-clouds together, filter out uninteresting data points, identify key points in a scene, and sort data into tree-hierarchies such as KD-trees or Octrees.

NumPy

The numPy[7] is a library created for Python exclusively to handle multidimensional arrays in an efficient and fast way. This will come in handy when handling large point-cloud matrices, and manipulating raw data acquired from the Kinect device.

OpenNI

OpenNI (Open Natural Interaction) refers to human interaction, voice and gestures. This framework defines APIs to enable voice recognition and hand gestures as an interactive way to communicate with a device.

OpenKinect

OpenKinect[27] is collection of open source libraries for Microsoft Kinect that enables the device to work on Windows, Linux, and OS X. Their primary focus for the time being is the development on *libfreenect[5]*. This library is the one being used as drivers for the Kinect in the project.

[2]Time-of-Flight camera estimates depth in a scene with a single light pulse instead of laser beam point-by-point.

These are the different libraries considered for the project. All except PCL has been used and tested in the application. OpenNI and OpenCV has their own solutions for creating 3D-scenes without using point-clouds.

2.6 Programming Language

The OpenCV library has three different languages to choose between: C[49], C++[45], and Python. A Java wrapper is in development as well but is not finished. Python[52] is a high level programming language and is the language that was used in this project. The following reasons determined the choice of language. Note, Python is used for all the OpenCV code examples.

First of all, Python uses automatic garbage collection. This frees programmers from the worry of memory allocation, as it will be taken care of. Both in C and C++ it must be micromanaged. Lists are powerful tools that do not exist in either C nor C++. There are ways of creating objects that acts like lists, but in Python, lists exist as a data type. Lists lets the developer do fast calculations. In Python, block delimiters consist of indentations instead of curly brackets such as in C and C++. This makes the code cleaner, easier to read and understand, and help developers to get a good overview of one another's code, which in turn provides the ability to work more efficiently.

When programming together with other developers, it is important to easily understand each others codes. Therefore, the PEP8 naming convention[53] will be used to make the code more consistent and readable. In general, we chose Python as our developing language because it is an efficient language that only requires a few lines of code to achieve greater effect compared to C and C++.

2.7 Testing

It is vital to have standardized ways of testing features and modifications of the implemented source code. To check the quality of the product is of course a necessity. Some code can easily be tested automatically with *unit testing*[58]. Unit testing will not guarantee the absence of buggy code, but it is nevertheless a good practice writing these tests, especially when refactoring or merging code from other developers and incorporating the changes into the software.

When all the unit tests has been executed, and evaluated, it is time for *integration testing*[43]. In this stage, all different modules are put together and validated one by one. The developer is supposed to make sure that various parts of the program work together as intended. If unit testing is not applicable, *interactive testing* is the only approach applied and the current input and output data should be evaluated. Interactive tests are very tedious but since a highly interactive software were developed, where arbitrary objects in the image can be tracked, this testing approach were used to evaluate tracking algorithms and different filtering techniques.

The system was also tested in various light conditions, examining the ability to detect different kinds of objects and analyzing the effects light conditions has. The objects varied in size, shape, and color to really test the robustness of the applications and tracking abilities. Finally various velocities, acceleration, and different directions was used for every object being tracked. The results from previous mentioned properties, is discussed for each specific algorithm.

3 Data Processing

Before there is any chance of tracking an object, there must first be an object to track. This is self-evident but the challenge is then to extract that specific object from an image. To improve results it is advisable to change the appearance of the image to enhance specific features of the desired object. This is done by using a variety of filter-algorithms. By then using a mathematical approach, it is possible or at least more likely that the desired object is registered.

To get a better grip of how filters affect the image, it is recommended to get an understanding of the hardware that has produced the image and what data the image contains. The project utilized two different camera devices, an ordinary webcam i.e. an RGB-camera or the Kinect. All the implemented software works on both devices if nothing else is written.

3.1 Microsoft Kinect

The Kinect has both an IR depth-sensor and a RGB-camera, each feeding in 640x480 pixels at a frame rate of approximately 30Hz. In comparison with other modern motion-capture equipment the hardware in the Kinect is poor, which leads to a great deal of noise[27]. In *Figure 1* a Kinect device is displayed. To use untouched raw data provided by the Kinect for our implementation would be inefficient and probably ineffective. Instead the data gathered will be filtered before applying it to any kind of tracking algorithms. Because the Kinect uses two different cameras, the feeds need to be processed individually. *The Image processing* will take care of filtering, colors, and scaling. Starting of, by enhancing colors or changing the input to gray-scale prepares the image for different tracking algorithms that may be color or contrast dependent. Filtering removes or enhances the data from the feed. Removing data refers to, e.g., noise reduction by using Gaussian filtering. Enhancement refers to, e.g., increase of contrast to ease the process of finding contours for tracking objects. Scaling frames allows algorithms to focus on a certain area in a frame or decreasing the resolution for faster execution of computationally heavy algorithms.

Figure 1: Microsoft Kinect

The depth processing handles scaling on the z-axis and also makes the image three dimensional.

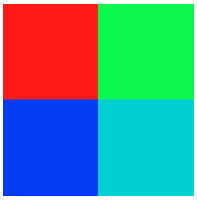
This is used to track objects or points of interest (*POI*) in a 3D environment. To be able to capture natural movement of objects and people this is essential due to the fact that most motions are affected in all three dimensions.
Both feeds will still be interpreted as images even though they will contain different kinds of data.

3.2 Image

The humans have an extraordinary ability to interpret images. To identify and track multiple objects and to distinguish them from each other in almost any kind of light-conditions are done unconsciously and is for the most part taken for granted. Unfortunately, to explain and define how objects should be separated and, which POI to track is not trivial.

To understand how to track an object with the help of images, it is probably appropriate to know how the data of images are represented. Because digital images consist of pixels, i.e. specified points with two-dimensional location coordinates and a set of values, which most commonly is a representation of color. The easiest way to manage the enormous amount of data that each image contains, is by representing the image as a two-dimensional matrix.

In *Table 1*, a four pixel image is represented, where each pixel has an unique color and its corresponding raw-data matrix. As seen, each element contains red, green, and blue value, which is the normal RGB coding. Each color is represented by 8-bits. This amounts to over 16.7 million color-combinations and contributes to the challenge of identifying objects in a scene. It is impossible for a person or computer to identify objects in an image by just traversing through one pixel at a time. But modern computers together with sophisticated tracking algorithms are rather good at identifying patterns and here is where filter-algorithms comes in handy.



$$= \begin{bmatrix} (255,0,0) & (0,255,0) \\ (0,0,255) & (38,207,213) \end{bmatrix}$$

Table 1: A 2x2 pixel image and its corresponding matrix

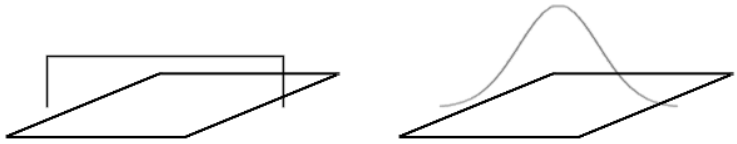
3.3 Filters

In order to make it easier for the computer to distinguish separate objects in an image filters are used to aid and enhance specific features. This makes each object easier to track. This section will be dedicated to explain how various filtering techniques are used and their different purposes.
Most of the filtering methods described in this section use small two dimensional matrices, called *kernels*, that are multiplied with the image, which in turn gives a new representation of the image, one that has the filter applied. This mathematical phenomenon is called *convolution*. A kernel's middle point is often referred to as the anchor, but the anchor's position can be changed within the kernel based on how each pixel value should be transformed.

3.3.1 Smoothing

One of the most common forms of noise reducing techniques is *smoothing*. This is achieved by convolving an image with a kernel, i.e. mask, such that a blurring effect that reduces noise is applied to the image. The simplest form of smoothing is by *mean filtering*, it changes the intensity of each pixel by calculating the average value from its surrounding pixels, *Figure 2a*. The size of the kernel depends on the amount of surrounding pixels that is taken into calculation and all the values in the kernel are the same, i.e. it is unweighted. However, this filtering method fails when it comes to preserving edges. Therefore, when smoothing an image for the purpose of reducing noise and still have patterns suitable for tracking, *Gaussian smoothing* is used.

Gaussian smoothing is applied in a very similar way as the mean filter, but instead it uses a weighted kernel. The kernel values are defined based on the bell shaped Gaussian distribution, *Figure 2a*, with the *anchor* as the maximum value. In other words, the kernel is weighted towards the middle with a circular shape. Even though Gaussian-filter is used, edges are still blurred but not in the same extent as with the mean filter. There is a trade-off between the amount of noise reduced and to which extent the image is blurred.



(a) Smoothing with mean filter (b) Smoothing with Gaussian filter

Figure 2: Smoothing filters

The following matrices are examples of kernels that are used for the respective smoothing technique. The major difference is how the kernels are weighted. A Gaussian kernel should usually be between 3x3 and 5x5 in order to achieve optimal result[28].

$$3x3\ mean\ kernel: \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} \qquad 5x5\ Gaussian\ kernel: \frac{1}{273}\begin{bmatrix}1 & 4 & 7 & 4 & 1\\ 4 & 16 & 26 & 16 & 4\\ 7 & 26 & 41 & 26 & 7\\ 4 & 16 & 26 & 16 & 4\\ 1 & 4 & 7 & 4 & 1\end{bmatrix}$$

Most of the tracking algorithms that are described in this report are very sensitive to noise. That is why smoothing has an essential role to play. Unless noise is reduced, features, such as edges and corners, will be harder to extract and object motions won't be calculated easily.

3.3.2 Edge Detection

To specifically distinguish among different objects in a 2D-image and be able to track them there exists techniques that are built upon using edge detection algorithms. There are loads of various edge detection methods when it comes to computer visualization. This project treats the most common ones which also are implemented in OpenCV: Canny, Sobel, and Laplace edge-detection.

The following section begins with an explanation of each technique followed by a comparison that decided which one that were used for this project.

Sobel Edge Detection

Sobel operation is a way of computing the gradient in an image by derivative calculations. By measuring the absolute gradient magnitude in the picture after the Sobel operation has been applied, edges can be found. To be able to calculate the first derivative along both the x and y-axis it uses two kernels, one for each axis. Convolving the kernels with the image returns a measurement of intensity changes in both directions which can be interpreted as edges, and with this information its gradient direction can also be calculated. The gradient direction tells how edges are oriented. The following two kernels are most common for Sobel operation[54][18].

$$3x3\ Sobel\ kernel\ for\ derivative\ of\ x: \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$3x3\ Sobel\ kernel\ for\ derivative\ of\ y: \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Laplace Edge Detection

This technique is similar to Sobel, which means it calculates the derivative with the help of convolution, but instead it calculates the second derivation and finds edges by searching for zeros in the derivative curve. Before adding the Laplace-filter a Gaussian-smoothing filter is applied in order to remove rapid fluctuation in the intensity changes[54][18].

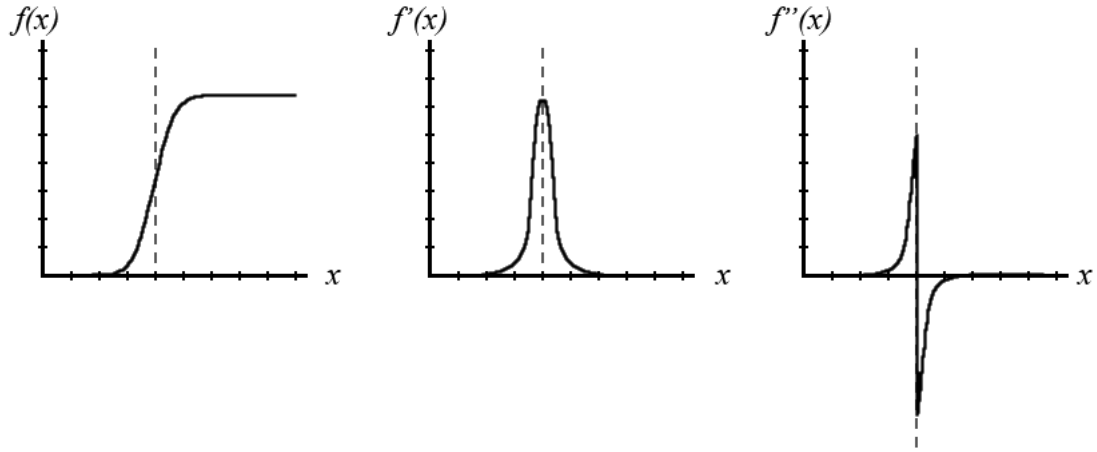


Figure 3: In the first graph to the left $f(x)$ represent a change in the intensity. In the second graph the first derivative, $f'(x)$, is calculated, this one is used by Sobel. Finally there is the second derivate, $f"(x)$, where the zero crossings gives the edge.

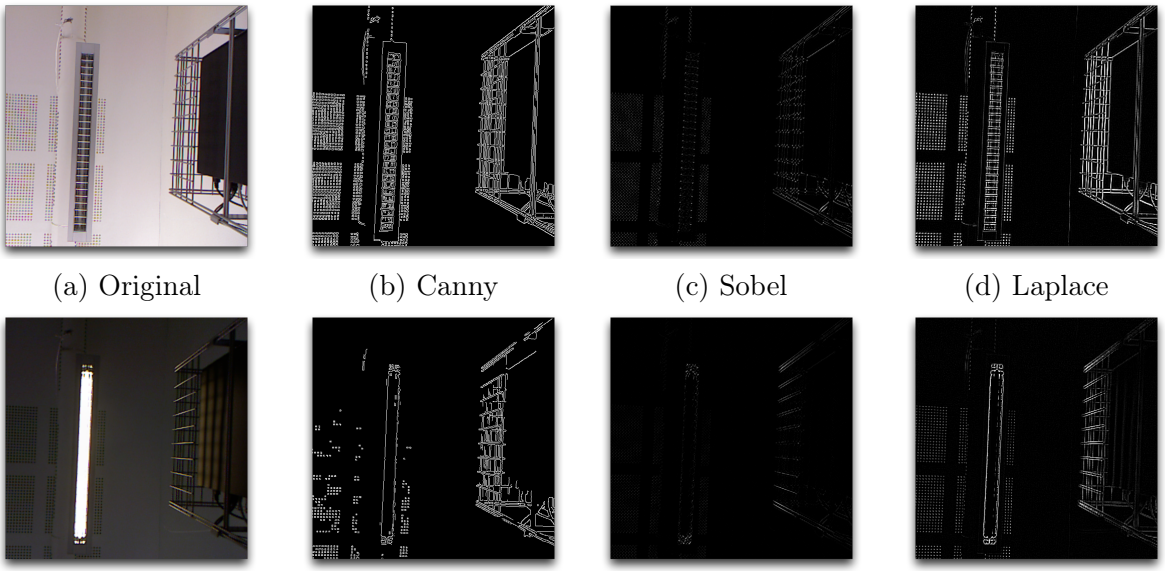
Canny

The Canny algorithm uses a higher and a lower threshold (a hysteresis), to determine edges. Each pixel's gradient is compared against the hysteresis. When a pixel has gradient that is below the lower threshold it is discarded. All pixels that are above the higher threshold will be marked as edge pixels along with all the pixels that are inside the hysteresis and connected to a pixel that is above the high threshold. This is one of the most common and used edge detection algorithm[23][19].

Edge Detection Results

There where some qualifications that needed to be satisfied during the selection process of the algorithms: noise-resistance, edge-clearness, and backlight-resistance. Empirical studies where done in order to research each quality. The Canny algorithm gave the clearest edges, this is because of the binary representation of the returned image where each pixel is displayed as either black or white. Sobel gave the most dull edges, but when it came to noise resistance, Sobel and Laplace were dominating. When Canny was applied to a series of images the video output were flickering due to lack of noise resistance. The algorithm that provided the best result against backlight was Laplace, while Canny handled it poorly.

For this project the Laplace were chosen due to best average results. In *Table 2* the various edge detection algorithms are compared with each other in backlight. In *Table 3* we have composed the comparison between the different methods like a scoreboard. Each number represents a grade, where three is the highest and one is the lowest.



(a) Original (b) Canny (c) Sobel (d) Laplace

(e) Original: backlight (f) Canny: backlight (g) Sobel: backlight (h) Laplace: backlight

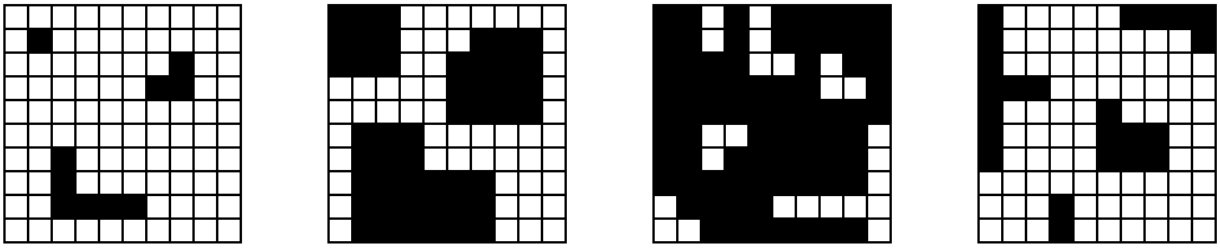
Table 2: Images of the different edge detection algorithms with and without backlight.

	Canny	Sobel	Laplace
Noise Resistance	1	3	2
Edge Clearness	3	1	2
Backlight Resistance	1	2	3
Total	**5**	**6**	**7**

Table 3: Comparison

3.3.3 Dilation and Erosion

The purpose of various filtering operations is to emphasize certain features by manipulating image contrast. *Erosion* and *dilation* are the most common morphological operations and has that fundamental property. Roughly speaking, applying erosion-filter on an image will expand the darker parts and shrink the lighter parts. The opposite will happen when dilation is applied. This is done by having a small window, normally with a size of 3x3 pixels, that moves over the image. For each pixel the algorithm will look for the largest pixel value, within the boundaries of the window, and replace the middle pixel with it. When erosion is applied the window searches for the smallest pixel value instead. In the *Figure* 4 a dilation and an erosion filter is applied on two binary images where dark parts mark low intensity. The reason for including erosion and dilation in this report comes down to its ability to reduce noise, remove openings in an object so that it is not recognized as several objects and separate different objects from each other. For example, when the Kinect displays the depth map of the recorded image it can give a porous impression in the way that the image contains a lot of holes. This can be adjusted by applying the dilation filter, such that the holes shrink. In this project we use OpenCV's implementation of these filters; cvErode() and cvDilate()[20].



(a) A binary picture with high intensity

(b) The erosion filter has been applied to the picture in a)

(c) A binary picture with low intensity

(d) In this image the dilation filter has been applied so that the dark parts has shrunk

Figure 4: Four images where the application of erosion and dilation has been done.

3.3.4 Histogram

As discussed in the image section, the data representation of an image consists basically of three different values: red, green, and blue. By combining these three values on each pixel with a value that goes from 0 to 255, each pixel can get any scale of colors.

Histogram is a way of collecting data in a simple and lucid model. Storing information from an image as a histogram by extract color data from each pixel makes it easier to identify and compare different images with each other. The way that the information is converted into a histogram is by changing the two dimensional representation into one dimension. This is done by first, defining how many bins the information should be divided among. Think of the bins of a partition of the x-axis. If there are just as many bins as there are x coordinates, each bin will hold a value that corresponds to the amount of color for all y-values at that x-coordinate. A histograms depth is determined by how many different colors that are measured. That means that each depth value corresponds to a certain color: red, green, or blue[21]. *Figure 5* gives a graphical representation of how histograms stores information.

During this project histograms was mainly used in combination with back-projection, a technique that with the help of histograms can filter background and foreground. This will be described more thoroughly in the following section.

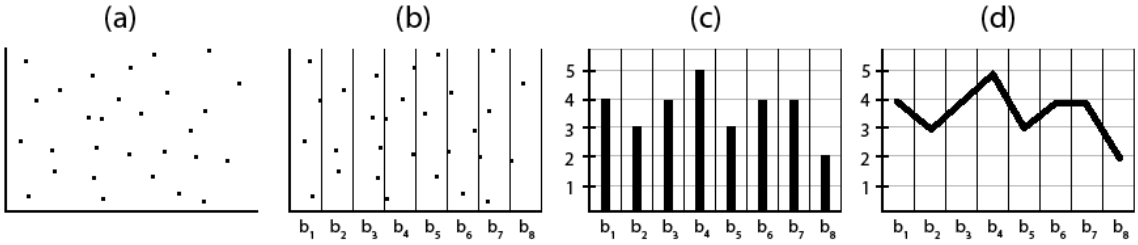


Figure 5: (a) First, there is the raw data represented in two dimensions. (b) This data is then distributed between 8 bins. (c) The representation is then changed such that the data is put as a value in each bin. (d) Applying splines to the new data representation gives the trend of data changes.

3.3.5 Histogram Back-Projection

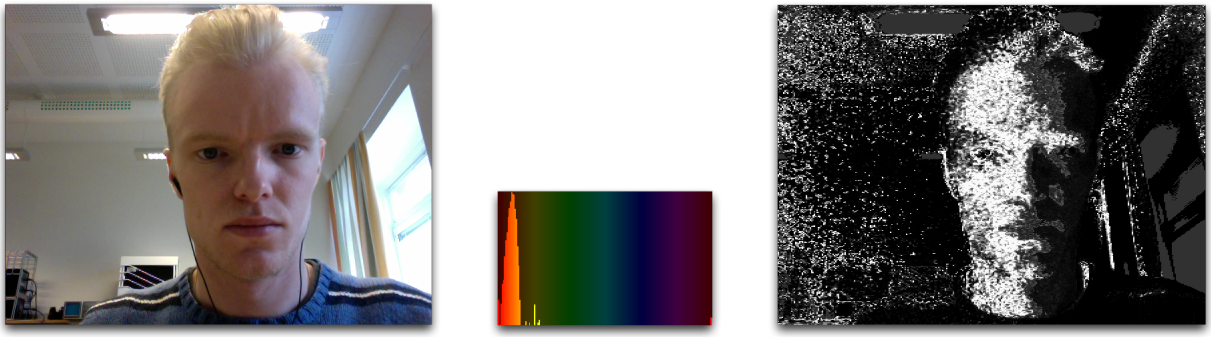
Tracking objects by color is very effective in sets where the object by itself distinguishes from the rest of the scene. There are a number of filters to prepare for colors tracking, the histogram back-projection is a primitive way of converting the standard image to an image where each pixel has been rated depending on the resemblance to the histogram.

Histogram back-projection is used to enhance a specific color combination in an image by using its histogram. It is a primitive algorithm that traverse through every pixel and gives a value from zero to L, where L represents the highest depth-value in the image. A high value implies a close match between the pixel and the histogram. This creates a gray-scale probability image. When all the pixels has been evaluated, the image gets rescaled. The formula below scales the probability-image according to a factor of the bit-depth divided by the highest-value. This basically means that the algorithm extracts the best and the worst match and then rates the image accordingly.

$$\left\{ \hat{p}_u = min\left(\frac{L}{max(\hat{q})} \hat{q}_u, L \right)_{u=1...m} \right\},$$

where $\hat{p}_u$ is a specific pixel in the image. The pixel's value $\hat{q}_u$ is recalculated by being multiplied with the factor of $\frac{L}{max(\hat{q})}$ where L is the maximum bit-depth and $max(\hat{q})$ is the pixel with the best match in the image[9].

It is worth noting that this primitive operation is very sensitive, this is due to the fact that the algorithm is all color dependent and it makes the result varying. There are a few reasons for this, the most obvious is light dependencies. The light setting in a scene will naturally affect the reflective light of the objects. An object with a solid color will in most cases reflect its natural color blended with the overall background light.



(a) Original (b) Hue-histogram (c) Histogram Back-Projection

Figure 6: Histogram back projection when light comes from side.

As seen in *Figure* 6, the Sample area for the histogram is taken from the left side of the face. Even though the skin is the same color, the light from the window is enough for the back-projection to prove that the right side of the face does not reflect the same color as the left. There are ways to manipulate the sensitivity of the Back-Projection, but the consequences of turning the sensitivity down is an increase of the background noise.

There are a few intuitive ways to improve the result. By selecting a well-lit scene with a single color background that differs as much as possible from the color being emphasized. As seen in *Figure 7*, There is minimum noise and the helicopter-body is very clear. The black-spots on the body are blinking led lights in various colors.

The standard function CalcArrBackProject() in OpenCV, is specialized in extracting skin color from the rest of the scene but it can be used for most colors. There is some work to be done before actually being able to use the CalcArrBackProject function.

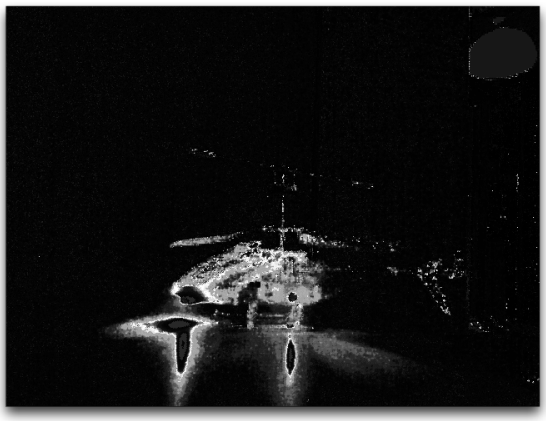


Figure 7: Histogram Back-Projection, helicopter

To begin with, the standard RGB-image must be converted into HSV, this is done by using the standard function CvtColor() with three parameters, where: *img* is the original RGB-image, *img_hsv* is the output image, and *CV_BGR2HSV* is an Integer that converts each element in the image matrix, from Red, Green, and Blue to Hue, Saturation, and Value (HSV) that can be translated to color, gray-tone, and brightness.

Hue is the only interesting value in this case Split() is recommended to extract hue from HSV. The function that does the magic is called CalcArrHist() with the parameters: [sel], hist, and nil. There is no official documentation of this function but it takes: *[sel]*, which is the part of the image wished to be converted to a histogram, *hist* is the output histogram, and *nil* is the minimum-value wished to be used, in most cases it is zero. The last step is to use CalcArrBackProject() with the parameters: hue, backproject, and hist where: *backproject* is the output image that is constructed by using the hue image and the histogram.

Histogram Back-projection Results

The Histogram Back-Projection filter is a fast, powerful algorithm, which is a great aid for color-based tracking algorithms, e.g., the mean shift algorithm. However, if there are no areas in the image that corresponds to the local hue-histogram the algorithm will still find the "best" result, which makes the back-projection image noisy or look like a chess-board because the image will still contain a relative scale.

3.3.6 Average Filter

The average filter is a method to enhance image quality and reduce noise. It does so by sampling an arbitrary sequence of images, and take the average RGB-value of every pixel. This operation requires a lot of performance, and for the time being is not recommended for a live video stream.

However, it could help to implement the code to execute on the graphics-hardware, in that case it would be possible to pipeline the process which, would reduce the stress on the CPU making it possible to run live. The filter's purpose is, for the time being, to produce screenshots for promotion purposes only.

4 Feature Extraction

It is not suitable to look at all the data within an image when trying to detect a certain object or property in the picture for performance reasons. This section will look at a few common algorithms to extract specific features or patterns from an image. The first subsections discuss ways to extract points suitable for tracking followed by, two algorithms for detecting a specific object in a picture.

4.1 Corner Detection

To track an object in a video-feed, each frame is compared against the previous frame in order to distinguish how the object has moved, this is done by observing the differences of the two frames. But in order to find differences in the two images there has to be points of interest that is related to the object and analyzed when tracked. These points are often corners and can be found by using corner-detection algorithms. For this project we have narrowed down the corner-algorithms discussed to Harris-corner detection, refined position and good features to track, because they are all supported by the OpenCV library and gives a picture of how corner algorithms works in general.

4.1.1 Harris Corner Detection

The main concept behind Harris corner detection is to use a small window to move over the image and compare the intensity after each shift at every pixel. In other words, the algorithm looks at how the intensity is changing in each direction from every point. Usually a threshold is set such that only intensity changes over a certain value is taken into consideration. There are three different cases of intensity changes:

- no change in either directions ⇒ no corners or edges are found.
- intensity change in one direction ⇒ an edge is found
- intensity change in both directions ⇒ a corner is found

This algorithm uses an auto-correlation function to find the changes:

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u,y+v) - I(x,y)]^2,$$

where $E(u,v)$ is the change after shifting, $w(x,y)$ is the window function over all x and y, $I(x+u,y+v)$ is the shifted intensity and $I(x,y)$ is the current intensity. By finding the Taylor series on the shifted intensity and extract the partial derivatives from I we get the following formula for $I(x+u,y+v)$:

$$I(x+u,y+v) \approx I(x,y) + \begin{bmatrix} I_x(x,y) & I_y(x,y) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix},$$

when applying this approximation into the original formula we get:

$$E(u,v) = \sum_{x,y} w(x,y) \left(\begin{bmatrix} I_x(x,y) & I_y(x,y) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \right)^2,$$

this in turn can be substituted down to:

$$E(u,v) = \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix},$$

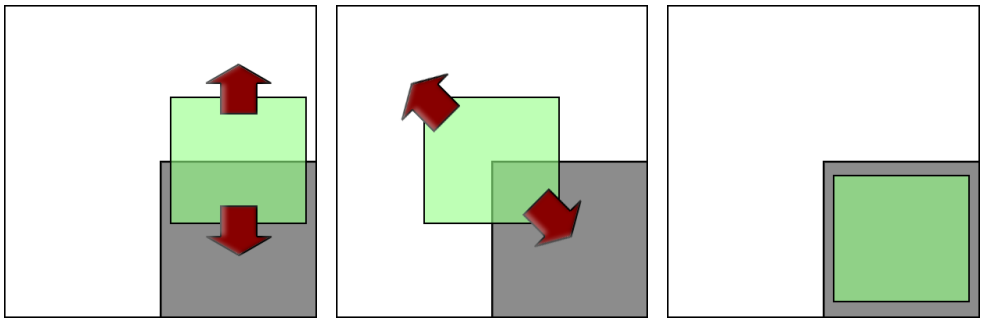
where the matrix M is defined as:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}.$$

By using these equations the intensity change is visible when analyzing the eigenvalues[10] of M, λ_1 and λ_2. Each of them represents a direction of where to find an edge. In order to determine the quality of the edge/corners that is found, the following formula will give a response value based on the eigenvalues:

$$R = Det(M) - k * Trace(M)^2 -> R = \lambda_1 * \lambda_2 - k(\lambda_1 + \lambda_2)^2,$$

where k is a constant that enhances the result and is found through empirical research. When $R < 0$ there is an edge present, $R > 0$ there is a corner, and when $|R|$ is small there is neither an edge nor a corner[34]. See *Figure 8* for illustration of R.



(a) The window detects an edge, R < 0

(b) The window detects a corner, R > 0

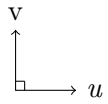
(c) The window does not find any major intensity changes, |R| is small

Figure 8: Three different cases when the search window is looking for corners/edges

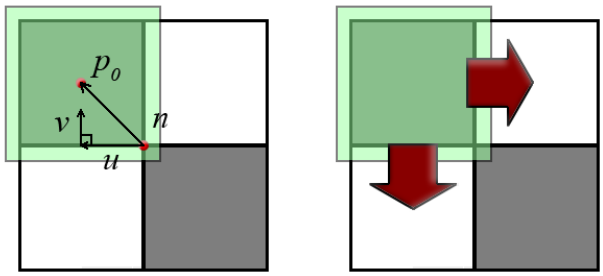
In OpenCV the Harris corner detection is implemented as a method, *CornerHarris()*, with the arguments: *img_ src, harris_ dst, blockSize, apertureSize=3, and k=0.04*. Given an image, *img_ src*, the method will return a new representation of the image, *harris_ dst*, where each pixel is represented by an R value that was described in the previous formula. The *blockSize* parameter decides how many neighborhood pixels[3] that should be taken into the calculation. When *CornerHarris()* is called in OpenCV it also uses *Sobel*[4] to find gradients. *ApertureSize=3* sets a default size of 3x3 on the Sobel kernel. The final parameter, k, is the variable that was mentioned in the previous paragraph which is meant to fine tune the calculations. Its default value is set to 0.04. The Harris corner detection algorithm was the source of inspiration when the good features to track algorithm was developed, which will be explained in *Section 4.3*.

4.2 Refined Accuracy

Often when trying to detect the position of an object, pixel level accuracy may be enough. However, this should not prevent a user to explore a higher accuracy for calibration of the camera or detailed tracking purposes, in these cases positions needs to be calculated on a *sub pixel* accuracy. To do this, $u \cdot v = 0 \Leftrightarrow v$ is *orthogonal* to u can be used. Note that $u \cdot v$ is the *dot product* of u and v.



The actual calculation of the refined position of the point is done by an iterative algorithm, which will loop until the accuracy criteria has been fulfilled. To do this calculation, it is not necessary or efficient for that matter, to incorporate all the pixels in the image. A smaller search window around the point of interest will suffice.



(a) Search window (b) Gradient movement

Figure 9: Execution of refined position

[3]The number of neighborhood pixels are blockSize x blockSize.
[4]The Sobel filter is described in the edge detection section.

The green rectangle in *Figure 9 (a)* is the search window, which will move in the direction illustrated by *Figure 9 (b)*. The position of the POI will be refined to the point p_0. For each iteration the search window will move in the *gradient direction* until the required accuracy is obtained. The formula used for this calculation looks as follows:

$$\sum_i (G_{p_i} \cdot G_{p_i}^T) n,$$

where G is the image gradient matrix, and p_i is one point $\in n$. The center of all neighboring pixels is the n variable.

The refined positions is executed by calling the FindCornerSubPix() method in OpenCV. The arguments for the method is: *image, corners, win, zero_zone, criteria.*
The image argument specifies in which frame the refinement should occur, for the list of interest points, the *corners argument*. The search window size, the *win* argument, is in the form of a tuple (width, height). *The zero_zone* argument which is also a tuple, (width, height), determines the area in the search window where the pixels should not be taken into account for the refinement calculations. *The criteria argument* is an OpenCV class called *CvTermCriteria*. The accuracy of the calculations is determined by the *epsilon*-variable in the class. If epsilon is assigned with the value of 0.2 the sub-pixel accuracy will be $\frac{1}{5}$th of a pixel. The last variable to assign in the criteria is the *max_iter* argument, which determines the number of iterations the algorithm should execute. The benefits of refined location of points are explained in various tracking algorithms in later sections.

4.3 Good Features to Track

The development of capable tracking algorithms is highly dependent on efficient use of the computers resources. Many calculations put a heavy load on both the CPU, RAM, and fast ways to find good features to track must be developed. A 'good feature' can vary from time to time depending on the information that is wished to be extracted and analyzed. However, a good feature will always have stable properties, which does not distort or diminish even though the conditions in the image vary. The different conditions could for example be: illumination, back lightning, and noise. One way to get values which fulfills previous mentioned criteria is to calculate *eigenvalues* in the picture. More precisely, finding eigenvalues which are aligned on a corner, of the object in the image[48].

Using corner Harris, or dividing the image into smaller squares, calculate the derivatives of all the squares, and extract the smallest eigenvalues will give you points suitable for tracking. This is the principle used in GoodFeaturesToTrack() method in OpenCV. The method requires six arguments: *image*, *eigImage*, *tempImage*, *cornerCount*, *qualityLevel*, *minDistance*. There exist four more arguments which are optional: *mask*, *blockSize*, *useHarris*, *k*. The first image argument is the picture to execute GoodFeaturesToTrack(), henceforth known as GFTT, method on. The two following images are temporary images to be used for the eigenvalues calculation. The properties of the points is determined by, the succeeding three arguments; cornerCount determines how many corners to be detected, the threshold for a corner to be returned is stated by qualityLevel, and the next argument determines the minimum distance between the found corners. The mask argument determines in what part of the original image the GFTT points should be extracted.

If features should be extracted using corner Harris, the useHarris argument is set to an integer bigger then zero. The blockSize and k argument is used in the corner Harris formula, See *Section 4.1.1*. The GFTT algorithm will return a list of points which are good features to track in the image.

Figure 10: Points returned after executing GoodFeaturesToTrack()

Good Features to Track Results

Even though an object is stationary in the images recovered from a camera device, the list of points returned from GFTT will not always be equal to the list returned from the previous frame. This is due to noise in the image, which will distort the edges of each object. Therefore, to be able to track an object using the data retrieved from the GFTT algorithm, an approximation calculation of the object center has to be made. Calculating the center of the set of returned points can be done by BoundingRect(). This method takes a set of points and returns a CvRect, which is a rectangle bounding every point in the set.

To track an arbitrary object in a specific depth, the GFTT algorithm could be used for satisfying results. The first thing to do is to set *clipping*[5] on the depth image. Clipping on the depth image can be achieved by, the logical operation *and*, on the depth array. The depth array is the raw depth data acquired from the Microsoft Kinect device, and the type of the array is a *numpy ndarray*, which is a multidimensional array. The next step is to apply the GFTT algorithm on the depth image with clipping. Finally, calculate the center of the rectangle using the CvRect returned from the BoundingRect() method, and this fairly simple tracking algorithm is complete. This approach works well if you want to track the entire object present on a specific depth, but it will not work for multiple objects or if only certain segments of the object is of interest when tracking.

[5]Clipping is when a max and min values of the field of view is set, i.e. remove objects in the foreground and/or background.

By adding some minor tweaks to the above algorithm there will be support for multiple objects tracking. The first problem to solve is to distinguish the separate objects in an optimal way in order to be able to track the objects with GFTT. One approach would be to: first display the first object, set the region of interest(ROI) of that object, and start tracking it. This procedure will be applied for all the objects to be tracked. Another solution would be to split the screen in multiple ROI:s, and start tracking the object in the various ROI:s. After the decision of which objects to be tracked and their corresponding ROI:s are set, save the center coordinate of each object. This coordinate is going to be used when updating all the tracked objects ROI:s for every frame the objects are being tracked. For each ROI, make a subtraction of the previous center coordinate with the current center coordinate of the object; let δ be the difference. Add δ to the (x, y) coordinate for every tracked objects ROI. The updating step will be done for every image where tracking should occur. Note that both these suggestions are using the depth image with added clipping.

The two proposed algorithms are fast and easy to understand, but they have large limitations. They will work really well if the objects in question will not be too close to each other, and are visible at all times. To cope with the possibility of blocked objects, template matching or preferable SURF should be used for updating the tracked objects current positions.

Another observation of GFTT is that it is especially good to use when an edge detection algorithm and the region of interest around the object is applied. However, it will give unreliable results if the image contains too much noise. The noise will distort and reduce the edge clarity when applying edge detection algorithms.

4.4 Cascade Classification

The tracking system can not identify similarities between two objects that are equal, if there are minor changes in: size, camera perspective, and illumination; even though a variety of filtering and enhancement technologies are used. Think of a human face in profile for instance. Even if different heads in an image would almost have the same shape, size, and skin color it would still be mathematically hard to recognize the patterns, and determine that it is indeed a human face. To be able to recognize a certain feature or property in an image, *cascade classification* algorithms can be used. The cascade classification algorithms, which will be discussed in this section, uses different machine learning algorithms. Since this report is not about machine learning, the focus of the discussion will be directed to how certain properties and data retrieved from machine learning algorithms can be used for object detection and tracking purposes. The focus is not on how that particular machine learning algorithm actually work.

In OpenCV, there are two algorithms to recognize objects in an image by using cascade classification. Viola01[55] is the first and Lienhart02[39] is the second algorithm used. The Viola01 algorithm can be divided into three major stages:

Integral image

This is where the pixels in the image are processed to a summation of the probability that a *Haar-like* feature is currently in a particular sub rectangle in the image.

Learning algorithm

The image is traversed, building a *decision tree* using the AdaBoost[31] algorithm. Every sub-tree is a *classification*, which matches Haar-like feature patterns in every sub-rectangle.

Cascade

The final stage of the Viola01 algorithm. This is the traversing step of the decision tree created in the learning phase. Each sub tree is matched with the Haar-like feature criteria and determined if it fulfills all the properties.

Integral image is created by traversing the original image in gray scale, and divide the image to smaller rectangular parts. Each rectangle is then summarized recursively with the following function:

$$g(x,y) = \sum_{x' \leq x, y' \leq y} h(x', y'),$$

where (x,y) is the pixel coordinate $\forall pixels \in Image$. Note that, the summation is done recursively $\forall rectangles \in Image$, but just for the sub-rectangles left of the (x,y) coordinate and all the way back to the first sub rectangle. The recurrence is then applied using the original image, which is the $h(x,y)$ function. The two recurrences are:

$$c(x,y) = c(x,y-1) + h(x,y),$$

$$g(x,y) = g(x-1,y) + c(x,y),$$

where $c(x,y)$ is the probability that the sum $c(x,-1) = 0$, and $g(-1,y) = 0$ for each row.
Let $subrectangles(s)$ and $image(i)$; the recurrence is executed $\forall rows \in s, \forall s \in i$.

The learning algorithm used by Viola01 is a machine-learning algorithm called *AdaBoost*. The boost algorithm is used for constructing a *decision tree* using a large set of images, containing the object which will be classified. Doing this procedure on a large amount of data will train the system to return more accurate results, and at the same time keep a generalized knowledge of the patterns[6]. The data the Viola01 algorithm is actually looking for when building the decision tree is, *Haar-like* features. The features are represented as two, three, and four rectangle features, *Figure 11*.

[6]Generalized knowledge means that the decision tree will just know 'enough' to determine if a specific feature fulfills the criteria. This is necessary since a well-trained system will just have a general idea of the features, which it is required to find. This means that it will, with high accuracy, find the correct patterns in the image, with the ability to learn even more for each iteration.

(a) Two-feature (b) Three-feature (c) Four-feature

Figure 11: Haar-like features

The borders are a representation of a smaller rectangle in the image, where the classification of an object will occur. The black boxes inside the borders are the sum of all the pixels in that specific rectangle. The summation is executed on all the pixels present in the black box and subtracted by the corresponding sum of every pixel in the white boxes. As shown above, the Viola01 algorithms proposes three different kinds of Haar-like features. The two-feature *Figure 11a* is the sum of pixels constructing a diagonal or vertical edge in the, currently searched, rectangular shaped region. If one box should fit, where two corresponding white boxes where needed to fill the whole rectangle; you have a three-feature *Figure 11b*. The last picture is the four-feature *Figure 11c* which contains boxes found on the diagonal in a rectangle fitting four boxes. Note that there are also permutations of all the cases of Haar-like features, e.g., the two-feature could switch positions of its boxes, making another Haar-like feature.

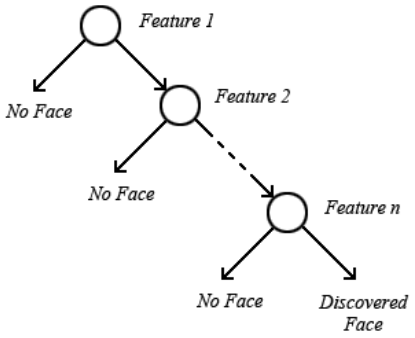


Figure 12: Decision tree constructed of Haar-like features

The decision tree, seen in *Figure 12*, is constructed when the original image is traversed. While traversing, the cascade algorithm will also check if all the requirements are meet in the sub-decision tree. Checking the requirements will be done for every sub-rectangle in the image. If some sub-rectangle should not meet the Haar-like feature requirements, it will be discarded, and the algorithm will cancel the construction of that particular decision sub-tree and move to the next sub-rectangle.

Canceling the search for Haar-like features of uninteresting parts of the original image, will increase the speed for building and traversing the decision-tree tremendously; this is the *cascading* part of the algorithm. The decision-tree is both constructed, and traversed when looking for a strong match of an object in the picture.

The method for actually using cascade classification, when detecting objects in an image, is called HaarDetectObjects() in OpenCV. The cascading data is a pre-constructed training data, in the form of a decision-tree, contained in a *xml*[3] file. The HaarDetectObjects() will return a list of detected objects. If no object is found, previous mentioned method returns an empty list. Each object found is in the form of a tuple, (x, y, width, height). A tuple of this form is called a CvRect in OpenCV, where the (x, y) variables is the left upper most corner of the rectangle of size (width, height). In other words, each rectangle in the list returned by the HaarDetectObjects() method will be a rectangle which completely surrounds the detected object.

Cascade Classification Results

This method works particular well on images with good light conditions, equalized histogram values, and if the image do not have too much noise. However, the OpenCV implementation of the Viola01 algorithm is computationally heavy. Running this algorithm on a live feed from a camera on every frame retrieved will put a lot of stress on the CPU. Therefore, it is not optimal to run Haar cascading as a standalone tracking algorithm. There are better alternative solutions which we will discuss later in the report. Using Haar cascading for detection of objects followed by performing the actual tracking with another tracking algorithm, is a better solution for faster tracking. The next part of this section will give an example of an algorithm for tracking a head.

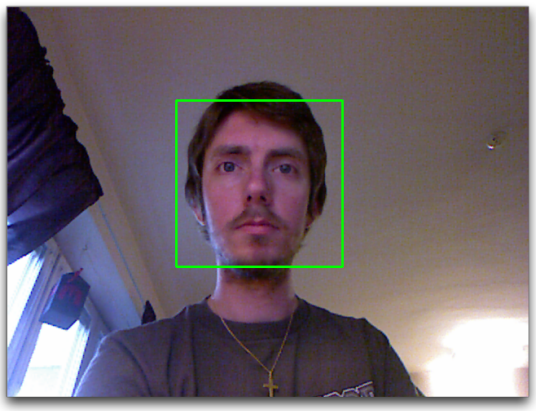


Figure 13: Found head using Haar cascading

To be able to track a human head, we propose an algorithm which first scans the image, looking for a head using Haar cascading. When a particular head is found, using HaarDetectObjects(), followed by generating an arbitrary amount of points to be tracked, using GoodFeaturesToTrack() along with FindCornerSubPix(). The algorithm has now located the head, and determined a set of points suitable for further tracking.

The final stage is to actually start tracking the set of points. Using *Lucas-Kanade*, a very robust optical flow tracking algorithm, which will give very accurate tracking results. Note that even though Lucas-Kanade is really good for solving this particular tracking problem, the tracking will fail miserably if the object is to be blocked or moving too fast. Previously mentioned limitations will have to be taken into account when developing reliable tracking algorithms.

This section was dedicated as an introduction for good features extraction, suitable for recognizing specific objects. There are other methods for accomplishing this task. The next section will be about how to extract features, comparing the key points, and finding similarities with methods other than Haar-like features.

4.5 SURF

In the field of computer-visualization there has been extensive research for finding specific robust features in images suitable for recognizing a certain object in a picture. A number of algorithms has been proposed, but the two most widely used are: Speed up robust feature (SURF)[12], and Scale-invariant feature transform (SIFT)[40]. They are both used for describing, detecting, and tracking a specific object.

A high-level description of the steps taken by SURF is to first extract feature points from the original image. The extracted features are described, and can later be matched with points extracted from another picture; where search for a potential match between the points will be made, and determine which point corresponds to which. The two most important features of the SURF algorithm, are the following:

Interest points extraction
: The *detector* is the entity for extracting certain interest points, *key points*. This detector extracts data from the *hessian-matrix*[38], using second order *Gaussian derivatives*. The most important property for accurate, and robust feature-extraction is repeatability. The points of interest should withstand noise in the image, changes in illumination, as well as coping with perspective changes between the images to be compared.

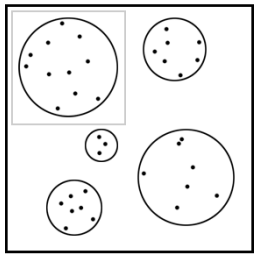
Description
: The image will be divided into smaller quadratic-shaped regions. The set of interests points within each *search region* will then be described. The descriptor should withstand the properties mentioned when extracting features as well as, having a very small amount of false positive detections, when matching the key points.

A smaller part of the image is represented as a two dimensional hessian-matrix. The matrix is a definition of the curve, derived from a second-order derivate, of that particular image region. The *hessian-detector* for the key point extraction will be derived from the hessian matrix:

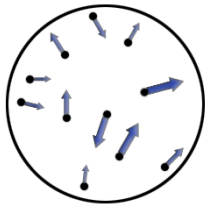
$$H(x,\sigma) = \begin{bmatrix} D_{xx}(x,\sigma) & D_{xy}(x,\sigma) \\ D_{xy}(x,\sigma) & D_{yy}(x,\sigma) \end{bmatrix},$$

where H is the hessian matrix for the variable x, which is a (x, y) coordinate in the image where the key point extraction will be executed. The scaling σ, and the location of the key point is calculated using *Gaussian derivatives* of the second order, D. The original image is converted to an integral image, and the calculations are executed over this image. Integral images are used for speed optimizations when executing the key point extraction phase. Gaussian derivatives can also be used for noise reduction, and constructing *scale spaces*, which can be used for restoration of images[50].

The calculation of the SURF descriptor is done in two phases. First, the direction for all the sets of interest points is calculated. All the sets are then divided into smaller squares. Secondly, the x, y direction for each point within a square is calculated and summed; the value from the summation is the descriptor. Below are illustrations and explanation of both phases:



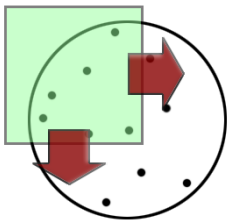
(a) Key points within the image, divided into circular shaped sets



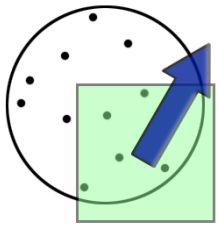
(b) Circle containing key points

Figure 14: Illustration of phase one of the descriptor calculation

All the extracted key points from the image are divided into sets of circles, *Figure 14a*. The orientation for each interest point incorporated within the circle is going to be calculated, using *Haar-wavelets*[24], as illustrated by *Figure 14b*.



(a) Search window traversing a set of interest points



(b) Direction vector for the set of interest points

Figure 15: Illustration of phase one of the descriptor calculations continued

The result from this calculation is a vector of a certain direction, for that particular key point. A quadratic orientation window, *Figure 15a* , moves across the circle, adding all the directions from each point, and calculates the overall orientation of the set; illustrated in *Figure 15b*. This vector is later used for satisfying orientation invariance of all the extracted key points. The second, and final stage of the SURF algorithm is to *describe* the found interest points.

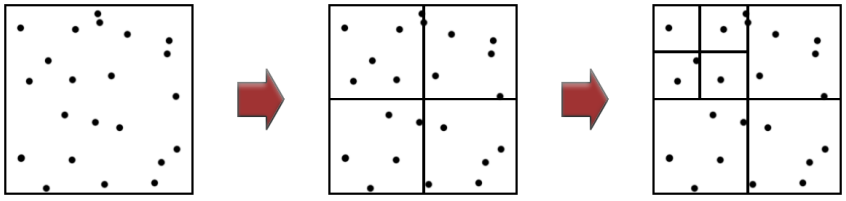


Figure 16: Illustration of phase two of the descriptor calculation

All the sets of key points are divided into smaller squares, and each square is in itself divided into even smaller squares as shown in *Figure 16*. The x, y direction is calculated for each point within a square for a particular interest point, using Haar-wavelets a second time. The formula for the calculation of the descriptor is:

$$descriptor = (\sum dx, \sum dy, \sum |dx|, \sum |dy|),$$

where d is the Haar-wavelet value for the x- or y- axis.
The descriptor can later be used for matching key points extracted from a template, with interest points selected from another image.

Extracting key points in OpenCV is really straightforward, using the method ExtractSURF(). The method takes four arguments: *image*, *mask*, *storage*, *params*. The features are extracted using the image, and the mask argument. The image argument declares where to extract key points, and the mask is an optional image which determines in what area of the original picture or what pixels you are interested in. Memory is a *cvMemStorage* class, and is used for temporary storage of the calculations when executing the SURF algorithm. Finally, the params argument is a *cvSURFParams* class in the form of a tuple containing four elements: *extended*, *hessianThreshold*, *nOctaves*, and *nOctaveLayers*. The extended variable determines the size of the descriptor; zero implies that the size is 64 elements, and one implies a size of 128 elements. The threshold is used for determining which extracted key points should be included or discarded. The last two arguments affect the underlying image traversing for the SURF algorithm. The nOctaveLayers variable determines how many pyramid layers each octave of the original image will be constructed upon, where the number of octaves is the size of the filter for each layer. If the latter variable is set to one, the scaling is unchanged. A value of two would declare that each layer would be twice as big, and the value three would declare another duplication. This method will return a tuple containing a list of key points, and another list of descriptors, extracted from the provided image.

SURF Results

SURF offers reliable results when trying to find interest points in a template image, search for similar points within another image, and finally match them. The algorithm is really robust against: noise, variations of brightness in the image, rotation, and different incidents of the tracked object. These mentioned qualities make the SURF algorithm a very reliable, and vigorous object detection technique. However, when traversing a live stream of images from a Kinect device or web-camera, executing the SURF algorithm on every frame is not sustainable. The CPU-load will be to great and the output image feed will lag. Instead use the algorithm when the task tries to detect or rediscovery a known object.

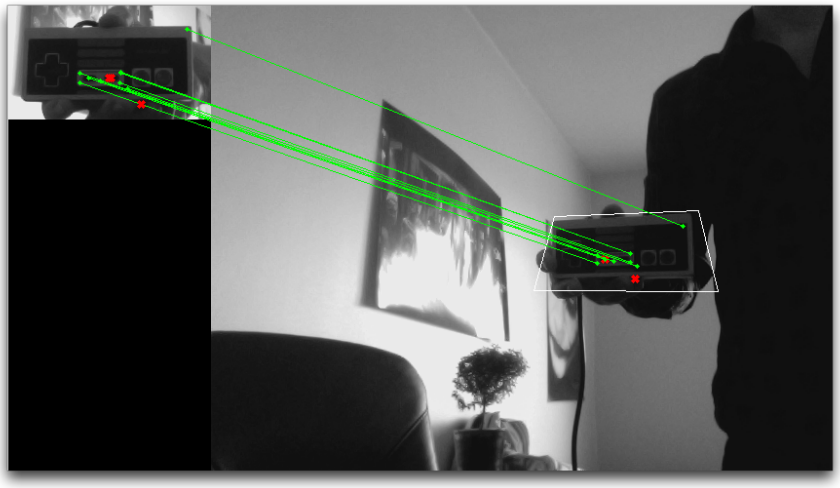


Figure 17: Key point matching using SURF

In *Figure 17* key points using SURF have been extracted from the template, the right upper most picture, and matched with key points in a live feed image. The green lines is an indication of a match between the interests points. There are also extracted key points where the algorithm could not find a match, which are represented by the red crosses.

5 Tracking

Tracking an object is a very vague and arbitrary abstraction of the actual work being performed of collecting data from an object in motion. A big challenge when tracking objects is for the application to determine somehow that the correct object is being tracked at all times. Another obstacle is to find unambiguity in the detected patterns. Conditions vary from scene to scene, and finding good patterns is challenging. The final stage for a tracking algorithm is to analyze and draw conclusions of the set of collected data. Analyzing images in various ways is a resource-demanding task. Efficiency when developing accurate tracking algorithms is a high concern. Depending on how and for what purpose the tracking algorithm is developed, a compromise between tracking accuracy and speed has to be made.

The application has to be able to determine a shape with high precision and unambiguity if a computer should be able to distinguish one object from another in a picture. To do this, certain properties specific to the object have to be highlighted and interpreted. There are many possible ways to accomplish this: using various filters, edge detection algorithms, optical flow, and segmentation techniques. There is a difference between the ability to detect, and track an object's movement. For the purpose of solving the tracking (an arbitrary object problem), two sub-categories of tracking was proposed by us; object detection and motion tracking. The following sections will first be about object detection, followed by motion tracking, and finally Optical flow.

Object Detection

Object detection is the process of looking at patterns and distinct features of an object in an effort to recognize which object[7] it is and what object to be tracked in the scene. Different techniques for distinguishing objects from one and other was discussed in the data processing section. Using one or more of those methods, the rediscovery of the object of interest is the actual object detection. By doing the rediscovery continuously for every frame, the system will track the object. A major problem is to make certain that the correct object is tracked by the system at all times. That is why, it is of utter importance to filter out the things you do not want the system to detect, and enhance features for the object you are interested of, for easier discovery of the tracked object. The actual analyze of the tracked objects movement will be the second sub category of tracking, *motion tracking.*

5.1 Template Matching

To find an object in an image and track it by looking at the similarities of a smaller part of the image is the basic principle in *template matching.* A template is a *region of interest* (ROI), in *Figure 18* it is marked with the green box. There are different proposed approaches to implement template matching[13][36], and this section will address the approach implemented in OpenCV. The algorithm traverses the image to find the object in question, and looks for matches that has similar properties with respect to the template, the small picture to the right in *Figure 18.* If the object is present in the picture, the algorithm returns a match.

The different methods for determining a template match in OpenCV are: squared difference, correlation matching, and correlation coefficient matching. The definition of the squared equation difference is:

$$G(x,y) = \sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2,$$

followed by correlation matching:

[7] Refers to shape, depth, color, and other properties in this context, not the actual awareness of what type of object it is in the real world.

$$G(x,y) = \sum_{x',y'} (T(x',y') \cdot I(x+x', y+y')),$$

where T is the template image, and I is the original image where you want to find matches using T. Let $r \in Region(R)$, $x' = 0 \ldots w-1$ and $y' = 0 \ldots h-1$, note that the summation applies $\forall (x,y) \in r$, where w is the width and h is the height.

Invoking template matching is done by calling the method MatchTemplate(). The first argument of the method is the image where template matching should be applied followed by a template image. The next argument is the result image, where the programmer wants to put the resulting matches in. Which template matching method to be used is determined by the last required argument. The matching regions can be extracted using the method MinMaxLoc(), which will return the global extreme values found in the image. The min/max value returned will be the coordinate for the left upper most corner where the object in the template matched with the original image.

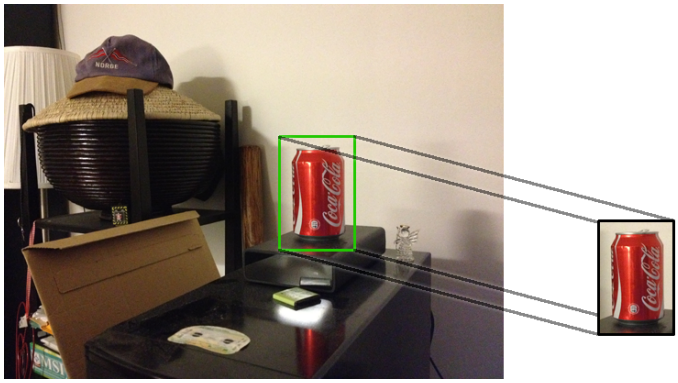


Figure 18: Template extracted from the original image

Template Matching Results

Tracking an object using template matching will result in rather precise localizations of that object. However, to only use this method when tracking arbitrary objects is really inefficient since template matching is too computationally heavy. The algorithm should be used for updating the location, or searching for an already known object and not used as a standalone tracking algorithm. The update or search could be performed if the tracked object can not be found at the moment. The object may be blocked in the image, temporarily exited the image, or there exists a template of the interesting object to be found. The algorithm begins by traversing the image, making matches for specific regions in the image, and finally compare the results, which puts a lot of stress on the CPU. However, there are a few ways to reduce the number of calculations needed and still get strong matches.

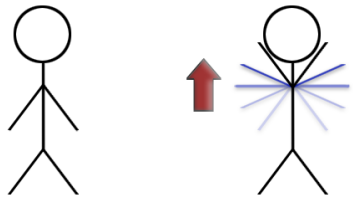
There is always noise in the image and when motion tracking is being performed, data[8] from the image can be discarded without sacrificing accuracy. To accomplish this, *down sampling*[9] is applied. Down sampling both the image being searched and the template, may decrease the number of calculations drastically when applying template matching or any other tracking algorithm.

Another possible speed improvement is to store the region where the strongest match was last found and start searching in that area. Even if the tracked object is moving, the difference in position from frame to frame tends to vary little. Therefore, when looking for the object in a new frame, it is preferable to start looking in the region where the object was located in the previous frame. There is a great chance that the object has not moved, or moved very little, since the last frame and a match is found right away. By also constructing the tracking algorithm to sample a few frames continuously and not traverse every frame will increase the speed of the template matching even further. This approach especially a good practice when tracking objects with low velocity.

Using the template matching method provided by the OpenCV library, and displaying just the RGB-data acquired from the Microsoft Kinect, the CPU[10] load on both the cores was around 50%. Doing the same test as previously mentioned, but also using PyrDown(), which is a down sampling function in OpenCV, the load on both cores was dropped to roughly 25% each.

5.2 Motion Templates

An interesting aspect of motion capturing is the possibility to recognize patterns in movements, and gestures. These patterns can later be used to enhance human computer interaction by triggering various events within an application when a particular gesture is recognized. To be able to accomplish this feat, the movement in the video stream has to be *segmented*. There has been a variety of suggestions for *segmenting motions*[56][44][14], and this section will look at one in particular, the Bradski00[16] algorithm. This algorithm relies on silhouettes of the object in motion that, for example, may be extracted using back propagation or dynamically learn what is the background/foreground in the video[37].



(a) Static person (b) Arms moving upwards

Figure 19: Motion history

[8]Data in this context is the sum of all pixels in the image.
[9]In this context down sampling is the process of reducing the amount of data in the image.
[10]Intel(R) Core(TM)2 Duo CPU P8600 @2.40GHz, on a Debian GNU/Linux unstable (sid) system.

Bradski00 starts of by analyzing if there is movement in the video feed by looking at the difference between the previous, and the current frame. If there is a movement, the current silhouette brightness intensity is maximized, and all the previous silhouettes will be slightly faded, *Figure 19b*. Noise elimination is performed by applying *dilation 3.3.3* on all the retrieved image silhouettes to make them as distinct as possible.

The retrieval of image silhouettes will proceed for a given time interval; by which a *motion history image* (MHI) is constructed. This image contains all the image silhouettes, of the object of interest from time t to t_i, and will later be used for describing the motion during that time interval.

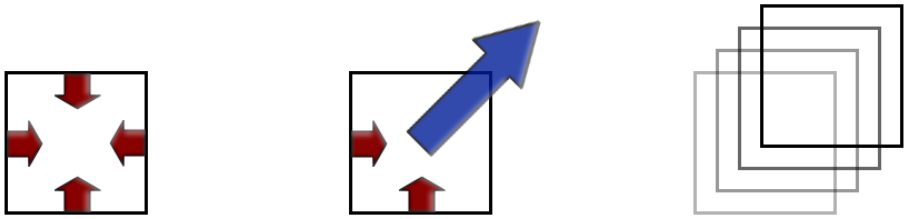


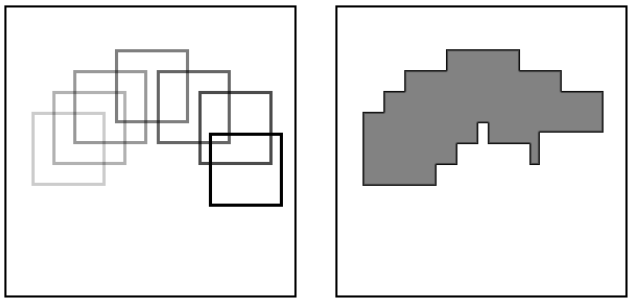
Figure 20: Global gradient orientation

To be able to tell the difference between motions which are very similar but moves in different directions, local, and global orientations are calculated. The left most picture in *Figure 20*, is a smaller part of the acquired MHI, with no movement at time t. Looking at the next frame fetched from the camera at time $t + 1$, the tracked object has moved diagonally. To be able to calculate the direction of the movement, *motion gradients* are used. Bradski00 proposes the use of a Sobel-filter 3.3.2 in order to calculate the orientation on the x- and y-axis for each pixel in every smaller square of the MHI:

$$G(x, y) = arctan\frac{S_y(x, y)}{S_x(x, y)},$$

where G is the orientation of the global gradient for x, y, during a given time interval t_i. S is the Sobel-filter applied on the x, y axis $\forall pixels$ within the silhouette.

Recognizing a specific pattern is done by constructing a motion mask while traversing the MHI; starting from the current silhouette the algorithm iterates back to the first silhouette within a given time interval. All the found silhouettes will be merged, *Figure 21b*, creating a motion mask which will be used when trying to recognize the motion. The silhouette movement in *Figure 21*, is an arc-shaped motion. When a movement in the video feed is discovered, and a positive match of the newly created motion mask with the already known arc-shaped motion mask is found a recognized motion is implied.



(a) Smaller part of a MHI (b) Motion mask retrieved from the MHI

Figure 21: Motion mask for gesture recognition

To construct the MHI the OpenCV method UpdateMotionHistory() is used. This method takes four arguments: *silhouette*, *mhi*, *timestamp*, and *duration*. The first argument is a picture containing the silhouette of the object of interest. Examples of how to extract silhouettes was given in the introduction of this section. The next argument, *mhi*, is an image used for the construction of the motion history image. The interval for the actual MHI data construction is determined by the last two arguments, where the *timestamp* is the initial time of the MHI construction, and the *duration* is for how long the construction should take place.

The orientation of the motion is calculated using CalcMotionGradient(), which takes six arguments: *mhi*, *mask*, *orientation*, *delta1*, *delta2*, and *apertureSize*. The first argument is the MHI retrieved from the UpdateMotionHistory() calculations. The *mask* is an image or matrix declaring in which part of the MHI the method should calculate the motion gradient. The gradient data is stored in the image argument *orientation*. The duration of the calculated motion is determined by the *delta1*, and *delta2* arguments: where the first one declares the maximum duration, and the latter the minimum duration. The last argument determines which size of the aperture window will be used when traversing the motion history image. What the aperture window is, and the problems with it will be brought up to discussion in *Section 5.8*.

The overall movement of the MHI is retrieved by the CalcGlobalOrientation() method, which takes five arguments: *orientation*, *mask*, *mhi*, *timestamp*, and *duration*. The *orientation*, and the *mask* argument is retrieved from the execution of the CalcMotionGradient() method. The *mhi* is calculated by UpdateMotionHistory(), where the *timestamp*, and *duration* determines for which time interval the global orientation calculations should be executed.

Motion Templates Results

Motion templates is not only suitable for recognizing motions, it can also be used very effectively when tracking a human. To accomplish this feat a tracking system extracts silhouettes, creates a motion history image, and calculates the orientation of the human within the MHI. As mentioned earlier the CalcGlobalOrientation() method returns the overall movement of the silhouettes.

The returned orientation could in this case be used to track the overall movement of the tracked human, *Figure 22*. The combination of the orientation along with the depth data retrieved from the Kinect will provide useful data where the tracked person is located together with its movement. Segmenting and tracking specific parts of the human may be accomplished by calculating motion gradients for that particular region where the interesting part is located. The location of the hand for example can be found using template matching 5.1 or SURF 4.5. One major downside by tracking an object of interest in a MHI is if the tracked object should enter the boundaries of another moving silhouette, the tracking will fail even if the object of interest is in front of the moving silhouette. This is due to the fact that local orientations can not be calculated inside another silhouette.

An implementation of idling in a motion tracking system can be accomplished by periodically look for changes in the image feed with the help of motion history images. If a movement should be detected, the system can generate tracking points within the object edges. The points could be generated using GoodFeaturesToTrack() 4.3, and all those points should be tracked using Lucas-Kanade 5.8.

Figure 22: Demonstration of a MHI of a moving human

5.3 Moments

After various filtering and shape detection techniques are applied, the ability to track by looking at the shape of an object is useful. Using *moments* is one possible way to accomplish this. In physics moment is often the force an object has when rotating around an axis[29]. However, in this section moments is a way to calculate the *gravitational pull* of an arbitrary object, *Figure 23*. If the object in question would have a circular shape, the central gravitational pull would be right in the middle of the circle. The gravitational pull on the x-axis would be the edge farthest to the left and the gravitational pull on the y-axis would be the upper most edge. Note that the circle is a set of points, and is limited to a finite area.

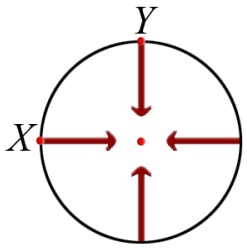


Figure 23: Moments on circle

The mathematical formula for calculating the center of gravitation is:

$$\mu_k = E[(X - \mu)^k],$$

where $E[X]$ or μ is the expected central gravitational pull, X is a continuous random variable of the kth order[47]. In the OpenCV, another way of calculating the gravitational pull towards the center of an object is proposed:

$$M_{x\_order, y\_order} = \sum_{x,y} (I(x,y) \cdot (x - x_c))^{x\_order} \cdot (y - y_c)^{y\_order},$$

where, I is the original image, (x, y) the pixel coordinates. x_c and y_c is defined as follows:

$$x_c = \frac{S_{first\_order\_x}}{S_{zeroth\_order}}, y_c = \frac{S_{first\_order\_y}}{S_{zeroth\_order}}.$$

The moments' algorithm needs to find all the extreme values[11] in the shape. This is performed by first looking at *spatial moments* (S) of the first order on the x-axis; which is the x_c value. The value returned of this calculation is the left most extreme value, i.e. the edge farthest to the left of the object. Doing the exact same thing by looking at spatial moments of the first order on the y-axis instead of the x-axis, will bring the minimum location of the extreme values in the shape, with respect to the y-axis; the y_c argument in the above formula. To be able to calculate the center of gravity in the shape, divide spatial moment of the first order with the spatial moment of the zeroth moment, for both the x-axis and the y-axis.

Moments Results

Here is another example[12] of tracking an arbitrary object using the depth data, *Figure* 24a, from the Microsoft Kinect. Clipping is applied to the depth as shown in *Figure* 24b. Since moments works with respect to shape, adding an edge detection filter algorithm, like Laplace, will provide the data of the outer contours of the object. Getting the spatial moment for x- and y-axis is done by calling GetSpatialMoment() from the OpenCV library.

[11] Extreme values in this context is the strongest gravitational pull using moments on an object with a specific shape.

[12] The previous example were described in *Section 4.3*

This method takes three arguments: *moments*, *x_ order*, *y_ order*. The computation of the moments argument is executed by the method Moments(), which takes the image as an argument and has an optional binary argument which determines how the pixels should behave.
Assigning the x_order = 1 and y_order = 0, will give you the gravitational pull for the x-axis of the object. The same goes for the y-axis, with the exception that, x_order = 0 and y_order = 1 in this case.

(a) No clipping on the hand (b) Hand with clipping

Figure 24: Depth images

To get the actual coordinates in the picture, divide moment axis and the central gravitational pull. The formula to retrieve the central coordinate is defined as:

$$coordinate = (\frac{m_x}{c}, \frac{m_y}{c}),$$

where m_i is the moment of a certain axis and c is the central moment of the shape. Getting the central moment is achieved by calling GetCentralMoment(); which takes the same arguments as GetSpatialMoment(). The values returned are the extreme values for the x-, y- axis, and central gravitational pull of the shape. The returned coordinate, from the above formula, is the center coordinate in the image where the gravitational pull is the strongest. For the interested reader, an improved version of calculating moments in images was proposed by Jan Flusser and Tomás Suk[30].

Tracking multiple objects can be done by using a list of all the regions of interest and previous center coordinates for each object. The list of object regions needs to be updated for every frame by subtracting the difference between current center coordinate and the previous one, followed by updating the positions of the regions for every object in the list of regions. This approach makes the assumption that the multiple objects being tracked never collide or get too close to each other. If two tracked objects are to close, and the gravitational pull is stronger for one object than the other, the moment tracking algorithm will fail. The object with the strongest pull will attract the other tracked center coordinate which will result in the loss of tracking capability of the other object.

There are various ways to solve this, and the by far easiest one, would be to use template matching continuously for updating the tracked objects positions.

5.4 Motion Tracking

Since tracking is just a form of rediscovering the correct object for every frame, and distinguish the object from other objects. The question is how should the motion be analyzed? The actual data acquired from the object-detection step is: the current-, and previous-position. Analyzing and drawing conclusions of that retrieved data is called motion-tracking. There are several motion-tracking algorithms. This section will essentially discuss the different *optical flow* techniques. The main advantage of using these algorithms for tracking purposes is first of all speed, as well as calculations of velocity, and the direction of object in motion. The programmer can use the newly acquired data when developing tracking algorithms which will try to accurately approximate the position of the tracked object even though, for example, the object is blocked. Discussion of the approach will be brought up in the optical flow section.

5.5 Mean-Shift

Mean-shift is a tracking algorithm that utilizes weight distribution in an image. It does so by analyzing clusters of weighted pixels and finds the most probable area in the image where the desired object currently exist. This algorithm requires a weighted input image, and is very efficient to track objects that stands out compared to the scene. For the project, the histogram back-projection, see 3.3.5, has been used. It creates a back-projection according to the objects hue-histogram.

The RGB-camera by itself does not provide with the weighted pixel matrix. The transformation from RGB to a weighted image can be done in different ways for example histogram back projection. In this case it is done by extracting a local histogram of the desired object. By using the histogram back-projection filter to generate a weighted image, the mean-shift algorithm continues were the histogram back projection algorithm finishes. By traversing through the new image and finds the object, based on the mass-center.

The algorithm used to measure the mass-center is called *Parzen window*[26], and it uses a specified size window and searches every frame for the new mass-center within the window and moves the window to get the mass-center in the center of the window, then it does the search again until the the maximum number of iteration is fulfilled or the shift finds the same mass-center again.

In OpenCV the mean shift algorithm is already implemented and is called by *MeanShift()*, it has three inputs, WImage, Window, Criteria. Were *WImage* is the weighted image, *Window* is the size and initial position, and *Criteria* has already been explained in *refined accuracy 4.2*.

The mean-shift algorithm tracks fluently and is relatively fast to compute, it can be used with confidence to track in situations where the objects is not expected to move in the depth-axis. This could for example be used in scenarios where there is a top-view camera tracking objects and analyzing movement through a course. Though in most cases and during this project tracking in three dimensions is the preferred approach. Fortunately there is a further development of the Mean-shift algorithm that is called Cam-shift algorithm.

5.5.1 Cam-Shift

The cam-shift algorithm uses both the histogram back-projection and the mean-shift algorithm to track an object through color. CAM stands for *continuous adaptive mean* and it has one more step that is activated after the mean-shift algorithm discovers the mass-center of the weight-distributed image. The cam-shift resizes the search window according to the total weight of the local area[9][15]. This allows full 3D tracking of an object in an efficient way. In *Figure 25b* the cam-shift algorithm is utilized by first extracting the local histogram from the helicopter's blue led-light. When in flight the cam-shift has no problem tracking the blue light even when the helicopter is changing course quickly.

(a) Back-projection (b) Cam-shift

Figure 25: Cam-shift in action

Cam-Shift Results

Though it is important to make sure the hue-histogram and histogram back-projection has a high quality for the tracking to work. We propose to use this algorithm as a secondary check-up that compares the tracking result with the first order tracking every ones in a while to make sure the tracking stays in check. A problem that can occur with other tracking algorithms is that the tracking loses focus and starts to follow a path on the object e.g., tracking the hand can result in tracking the wrist, to the elbow and so on. The cam-shift algorithm can make sure that the tracking stays on the hand if the operator wears a long sleeve shirt. But all in all it is a good algorithm that is very cost-efficient and is sensitive to dynamic changes, especially different light sources.

5.6 Optical Flow

It is a fair chance that some properties of an object is distorted in some manner when that object is tracked: the lighting conditions can change, the tracked object can be blocked of another object, and the object may temporally exit the image at a certain time. All these scenarios have to be taken into account when developing tracking algorithms. The need of a good way to approximate motion, and relative position for the tracked object arises; one solution is to use *optical flow*. This technique calculates the velocity, and direction of an object in motion. Using the acquired information from optical flow algorithms it is possible to make rather accurate guesses of the objects position 'some time ahead', even if the object is not present in the actual frame given.

Various optical flow algorithms will be brought up to discussion in this section. They will be explained, analyzed, and conclusions of when to use them will be proposed. Some of the algorithms are very good at detecting a motion, and start tracking the object. While others are exceptionally fast, and accurate when tracking a large set of points simultaneously.

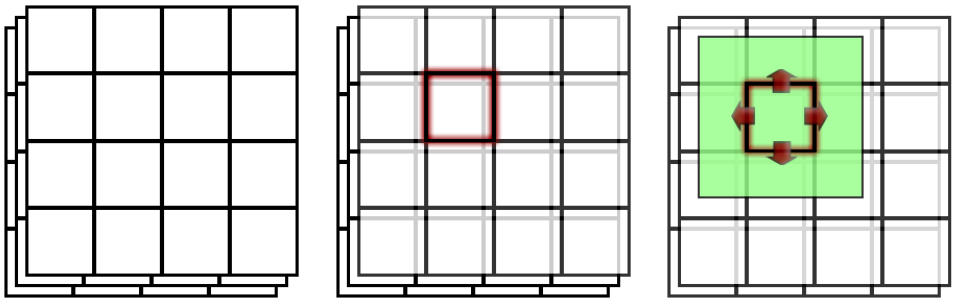
5.7 Block Matching

This subsection present yet another tracking algorithm that, among other things, also is used for compressing video formats such as MPEG and H.236. The Block Matching algorithm is a very straight forward method for tracking objects in a motion picture. It starts of by dividing each frame into disjoint blocks with equal size, *Figure* 26a. For each block in the current frame the algorithm searches within a predefined search area around each corresponding block in the previous frame in order to find a candidate[13] that has the best match with the block in the current frame, *Figure* 26b *and* 26c. So in other words, each block in the current frame is 'shifted' within the corresponding search area of the previous frame in order to discover how the block has moved. Since the comparison is done between pixel values this algorithm is sensitive to the aperture problem[14]. However, this can be solved by making the blocks larger.

The most common function for computing the cost of each pixel is *Mean Absolute Difference* (MAD)[33] which is expressed as follows:

$$MAD = \frac{1}{W * H} \sum_{i=1}^{W} \sum_{j=1}^{H} |C_{ij} - R_{ij}|,$$

where W is the block width and H is the block height, C_{ij} and R_{ij} represent the compared pixels of the block in the current frame and a candidate block in previous frame.



(a) This is a series of frames that has been divided into blocks

(b) Each block is matched against the previous frame in order to find its candidate block

(c) The block is 'shifted' across a searching area to find the best match

Figure 26: Illustration of Block Matching

[13] A candidate block is a block that potentially matches the block in the current frame.
[14] The aperture problem will be explained in Lucas-Kanade section.

Full Search

This is the most computationally demanding search algorithm, but at the same time the most accurate. It is a very straight forward brute force algorithm in the sense that it traverses the entire search window, compares the block against all candidate blocks, and picks the one that has the least divergence. But having large video feeds that has small changes between each frame makes this algorithm obsolete.

Three Step Search

Instead of traversing the entire search area like the full search method does, the search for a block match is divided into three steps, thus the name. First off, the size of the search area is set so that it holds up to 15x15 candidate blocks. Then nine search points[15] are placed out. One in the middle with eight surrounding points such that it forms a square with an area of 9x9 blocks (the red points in the *Figure* 27). Then the algorithm executes the first step of searching; beginning with the point in the middle and continues until all nine search points are investigated. If the middle point had the best match the algorithm ends and the motion of the block is found. Otherwise, if any other point has a better match, that one will be set as the new center point, where the next search will start, and eight new search points will be placed out around it, but this time with an area of 5x5 block (the green points in the *Figure* 27). The second searching step is commenced and it operates in the same way as the first one. Again, when the best match is found, that one will be the new center and eight new points will appear, but this time adjacent to it (the blue points in *Figure* 27). The final search step starts and the point with the best match shows where to the block has been moved and a new center point will not be declared. This method is not as accurate as the full search algorithm but the worst case scenario for this algorithm is 25 block searches instead of 225 with a search area of 15x15 candidate blocks. In *Figure* 27 the three different colors illustrates each step and the arrows point out where each step found the best match[11][33].

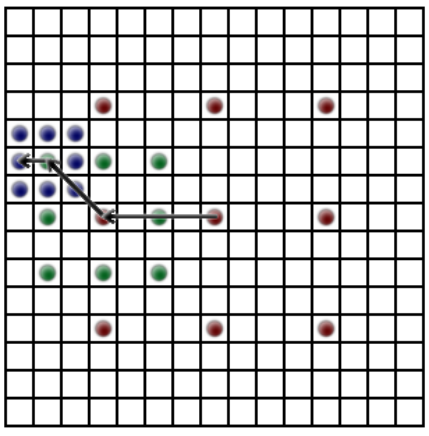


Figure 27: The arrow shows how the algorithm finds the best match and continues to the next step, this image illustrates three steps until a match is found

[15]Candidate blocks that are to be investigated.

5.8 Lucas-Kanade

In the tracking field it is often very convenient to be able to track arbitrary amount of points, independently of each other. Tracking in this way will increase the possibilities to track objects tremendously. A system providing this feature could be used for generating tracking points in a human face and capture facial expressions. The *Lucas-Kanade*(LK)[41] algorithm has the ability to provide with previous mentioned qualities. The algorithm is a fast optical flow algorithm proposed as early as 1981. The algorithm has the ability to track large set of points in an image, without losing its ability to track each point independently of each other accurately. Image transformations like: scaling, rotation, and shearing has little effect on the overall performance of the algorithm. However, Lucas-Kanade will fail to track a specific point in the image if the object being tracked is: moving too fast, or the lighting conditions in the image changes drastically. Why this is the case will later be explained when explaining the algorithm more thoroughly.
To explain how the Lucas-Kanade algorithm works three prerequisites has to be stated, which LK is heavily dependent on[22]:

Brightness
: The lightning conditions, from the previous frame to the current frame, should preferable not change at all. If the brightness should change, it has to be continuously smaller changes between the images.

Sustained movement
: For each point being tracked, their overall velocity should be low. Note that this assumption is not just applied to the point itself, but also for the neighboring points contained in the *window*.

Velocity coherence
: The overall velocity, and direction of the movement is coherent for the point being tracked and its corresponding neighbors.

With the previous mentioned properties, of the image in mind, LK starts of by dividing the image into smaller *windows*. Each separate window is a rectangular region in the initial image, where the actual optical flow calculations are being executed. Dividing the image in this fashion will speed up the calculations necessary for accurate results. However, if the window size is too small, this approach leads to another problem, the *aperture problem*.

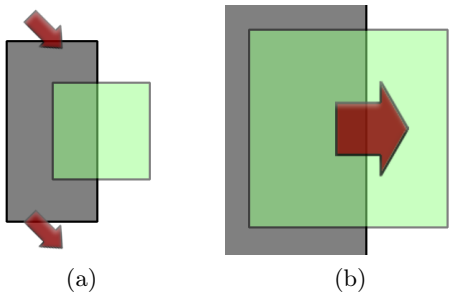


Figure 28: Illustration of the aperture problem

When the window is to small it is very hard, or even impossible to determine the flow of all the pixels within the edges of a specific window. *Figure 28* is an example of the aperture problem taking affect. The gray box in *Figure 28a* is a smaller part of the image being traversed, where the pixels has the same intensity. The green box in *Figure 28a* is the search window traversing the original image trying to determine which direction the gray-box object is heading. Looking from the search window perspective, the gray-box is moving from left to right. But this can not be claimed with absolute certainty. It may be very well, as in this case, that the object is moving in a diagonal motion from left to right. The search window is to small and it needs more information of the pattern, which movement it is currently trying to determine. This phenomenon makes it unreliable to apply too small window sizes, while trying to calculate accurate optical flow results, since the algorithm simply has not enough data.

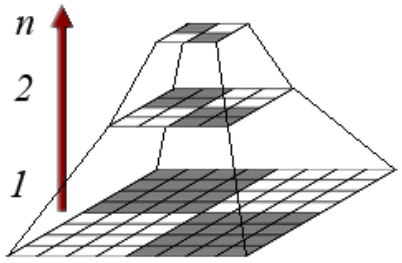


Figure 29: Image pyramid

One way to evade the aperture problem is to use image *pyramids* 29. Traversing each level of the pyramid while executing the LK algorithm will decrease the incorrectness from the aperture problem to nearly zero, without sacrificing accuracy when tracking an arbitrary point in the picture. Pyramids is often used as a method for *down sampling*[32]the original image for optimization purposes; it can also be used for recognizing patters[42] in the image.

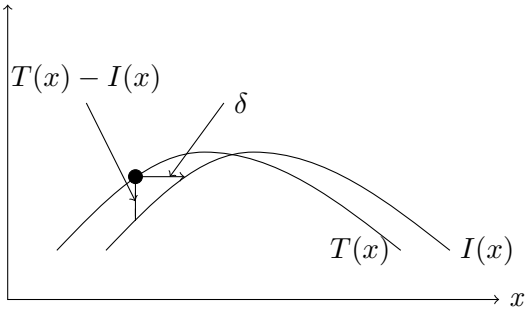


Figure 30: Calculate position of the black-dot

The next step is to traverse each window in the current image and check how well the current position of the tracked object matches with the previous position. The optical flow is calculated by matching the current position, of the point being tracked, with its previous position. The matching is done by looking at the derivative for both image and compare the dislocation; in doing so solving the *registration problem*. The problem is locating a point in space with sufficient accuracy over time. The black dot, in the *Figure 30*, is the point being tracked. The point moves in space from the $T(x)$ to $I(x)$ curve in one dimension. Both curves will be calculated with the following functions:

$$I'(x) \approx \frac{I(x+\delta) - I(x)}{\delta},$$
$$= \frac{T(x) - I(x)}{\delta}$$

when,

$$\delta \approx \frac{T(x) - I(x)}{I'(x)},$$

where $I(x)$ is the derivative curve for the current position of the *image*, and $T(x)$ is temporary image, that is, the previous image with respect to I. $I'(x)$ is the optical flow calculated in 1D, $\forall x$ *positions*. Note that the calculations for each derivative comparison is done over time, using the difference in space δ. The registration problem addresses the task of calculating the position of a point over time.

CalcOpticalFlowLK(), and CalcOpticalFlowPyrLK(), are two methods implementing the Lucas-Kanade algorithm. However, the second method is the preferable choice to use, since it traverses the LK algorithm over each pyramid level. The reasons why to use pyramids were discussed earlier in this section. The actual pyramid calculations is done by *Bouguet00*[57].

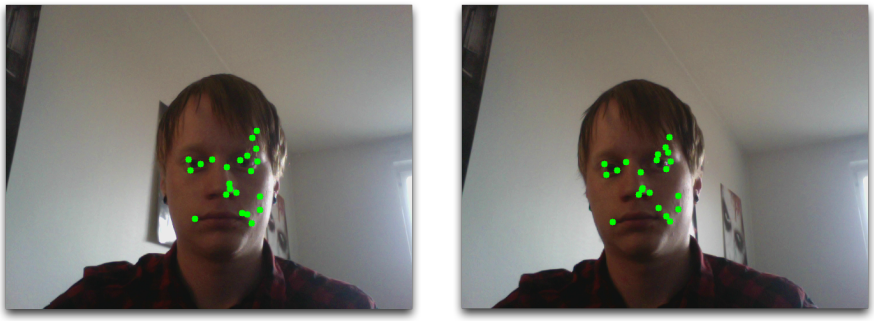
The method CalcOpticalFlowPyrLK() takes ten arguments: *prev*, *curr*, *prevPyr*, *currPyr*, *prevFeatures*, *winSize*, *level*, *criteria*, *flags*, and *guesses*. The first argument is the previous captured image, and the *curr* argument is the current retrieved picture. The next two arguments are temporary variables used for calculating the pyramid images. The *prevPyr* is the previous image pyramid and the *currPyr* is used for calculations of the current image pyramid. The *prevFeatures* argument, is a list of all the points being tracked. Traversing the pyramid is declared by the *winSize*, and the *level* arguments. Where the size of the search window is determined by the first one, and the number of pyramid levels is assigned by the latter. *Flags* determines if pre-calculation of the image pyramids should be used or not. The argument *criteria* specifies the maximum number of iterations, and the accuracy of the LK algorithm. Finally, guessing the coordinates for all tracked points can be done by assigning the last argument with a two dimensional array of coordinates.

The result, after executing the CalcOpticalFlowPyrLK() method, is a tuple of the form (*currFeatures*, *status*, *track_error*). Updated coordinates from the optical flow calculations for each tracking point is stored in the list *currFeatures*. Each point returned has a *status* variable which it is associated with. The status variable is represented as a binary integer of either the value one or zero. If the value is one, the optical flow calculations for that particular point was completed successfully. The status value will be zero if either the criteria could not be met or the optical flow calculations failed for some other reason.

This indicates that filtering out points where the LK algorithm fails to determine the movement of a specific point can be made by the status variable. The final variable, *track_ error*, is the location difference for each point between the current picture, and the previous one.

Lucas-Kanade Results

Tracking arbitrary amount of points simultaneously works really well using CalcOpticalFlowPyrLK(). How the set of points are being generated depends on your specific domain. Two alternatives for generation of points could be: using the mouse by clicking on specific parts in the image, or the usage of the GoodFeaturesToTrack() method. While the GFTT method will return points suitable for tracking, certain positions of the points may be inadequate to be used with LK. This is due to the fact that higher position accuracy is needed. The possibility to refine the position of the points is made by FindCornerSubPix().



(a) Initial position of the head (b) Moved the head slightly to the left

Figure 31: Real time tracking of a set of points using Lucas-Kanade

Even though the set of points have all the requirements necessary for solving the task of tracking, the programmer also needs to take into consideration of what to do if the point is being blocked by another object or the point is moving in such a speed that LK loses the ability to track that point. There are two reasons for the loss of tracking capabilities of a point which has a to high velocity. The first one is due to the aperture problem when a search window is too small. The second problem is that the tracked point can end up outside the current pyramid level when the LK algorithm is traversing it, resulting in a discarded tracking point.

6 Result

This section summarize the results and general observations encountered during the project. Spanning from multi-library issues to the primary difference between detection and tracking algorithms. Many of the more specific results discovered has no fundamental proof and has instead been added in discussion where it has been thoroughly motivated.

It was decided to develop the application on the three platforms: Windows 7, OSX, and Gnu/Linux. Even though there were no problems to install the drivers and libraries on Gnu/Linux there were some issues on Window 7 and OSX. The first problem encountered, at the start of the project, was to get Python and the libraries: OpenCV, Libfreenect, and numPy, to work together on two of the platforms OSX, and Windows 7. This problem demanded its resources and the issues were not completely solved until the first quarter of the project had passed. The reason for this was complicated operations with inconclusive documentation. Most of the problems came with Windows 7, where there were issues to get the Kinect drivers to work. Even though we created a new document[25] that describes each step of the process there are probably still scenarios uncovered.
On OSX there were a few problems to connect the libraries with the right version of Python, but in the end it was not an issue after all. Unfortunately, these problems delayed the progress of the project with approximately two to three weeks.

However, when the libraries were up and running, OpenCV proved to be invaluable to the progress of the project. It's overwhelming collection of image processing algorithms allowed us to analyze and compare a variety of different filter-, detection-, and tracking algorithms. All the algorithms provided were in the end implemented in the application, and a continuous trend was discovered. Tracking and detection is first and foremost, noise-, and scene-dependent, which means that all the algorithms tested have prerequisite conditions for optimal performance. Less noise is better, unfortunately with the equipment used, noise was inevitable. Different filter-algorithms can cover up the noise in different ways, but in the process data will be lost. Therefore, the critical part is to choose the right filter for the right tracking- or detection-algorithm. In *Section 7*, different combinations of filter, detection-, and tracking-algorithms are presented as suggestions for different scenarios. These suggestions are based on observations only, and there is no scientific proof to back them up them.

It is important to know that there is a difference between detecting objects and motion tracking. All the detection-algorithms i.e., corner detection, GFTT, SURF, and template matching, can track objects, however they are inefficient. That is why there are optical flow algorithms, which anticipates where the objects should appear in the current frame, and searches the area until it finds the objects in motion. Whereas detection algorithms searches through the whole frame in the stream regardless of the position of the objects.
During the project it was discovered that a combination of algorithms could be used for emotional capturing, with the use of a laptop and a webcam or Kinect. Unfortunately this was not achieved completely, but an algorithm has been proposed A, which uses a combination of algorithms that has been analyzed during the project. The proposed algorithm should be able to generate an arbitrary number of points on a person's face in order to capture expressions. The algorithm is hypothetical and will be further discussed in *Section 7*. It has not yet been fully implemented and tested.

7 Discussion

Early in the project we realized that motion-tracking can be divided into three steps: filter, detection, and tracking. We decided to investigate a collection of algorithms in each category with different characteristics to identify preferred algorithms individually, and together in a daily environment, that may contain multiple objects in different colors and shapes. Together with variation in light quality with single and multiple light-sources; and investigate the effects on the algorithms when objects move with different velocities.

When the research began we quickly realized that there are far too many algorithms constructed for filtering and motion capture. The strategy was to identify the most common algorithms that are used in other tracking projects. The average filter, *Section 3.3.6*, was implemented and its purpose was to remove noise, the same as smoothing, *Section 3.3.1*. After the first prototype, it was discovered that in order to get it to work in real-time, the process needed to be pipelined, which we realized that we did not have the time to do. The goal was to implement and compare various filter-, detection-, and tracking-algorithms. These reasons resulted in the convenient decision to only use algorithms that has already been implemented in a library. Because it is a fast way to implement a prototype, which has been used for analysis and evaluation of the algorithms capabilities.

The primary consequence of working with low-fidelity camera equipment is a larger amount of noise. The problem with noise is that the data is not fully accurate, and there are no algorithms that can identify the difference between noise and correct data. Though, there are noise-reducing filters that deals with the problem in different ways, we have selected three algorithms: smooth, dilate, and erode. Smooth-filters calculate a new pixel value by considering its surrounding area. this creates a blur-effect on the entire image that makes it noise free. Erode and dilate creates a small window that traverses through the image and replaces the center value with the highest or lowest value. This enhances the dark areas or the light areas in the image and at the same time discards the noisy pixels in the image. We found that these three noise-removal algorithms has been able to aid the tracking algorithms well in our project.

There is an uncertainty of the effects when noise-filters are used compared to e.g., edge-enhance filter. No matter what happens to the rest of the image, the user can at least expect the edges within the image to be enhanced. Noise reduction results in a blurred up image and the question is what is preferred, noisy or blurry? It is worth noting that noise-filters are favorable as long as the right one is picked depending on what tracking method is going to be used. In our case most of the tracking algorithms are based on using contrast and edges. Of course it is a balance but there are few scenarios were there are so much noise that it has been impossible to do the tracking in a sensible way. Though, it should be mentioned that when the tracking is based on colors, to our experience it does not matter if the image is blurred a little bit.

Edge detection filters are probably one of the best ways to aid the segmentation process. For tracking, one of the most important part is to be able to distinguish the objects in the image e.g., different parts of the body, especially hands. However, if the background is clean of edges, this technique is good to get clear lines and get the exact shape of the whole body. Note, motion capture focuses on capturing joints and rarely on shapes per se, but it is beneficial to start by excluding everything that is of human shape.

Opposite to what we believed when we started this project, depth is a surprisingly small part of the detection and tracking technology. There are very few techniques that utilize depth to its potential that we know of. The reason for this is probably because the technology is very new and there is still limited algorithms to detect specific three dimensional shapes. As mentioned in the previous part, edge detection can be used to extract high precision shapes. It is worth noting that depth extraction probably has an even higher precision as long as the object being detected is the only object at the particular depth. Combining depth and edge detection would probably complement each other. Because even if the scene contains a lot of objects, and the background is messy. Clipping on a specific depth, with the help of an edge-detection algorithm, allows the detection algorithm to ignore most of the uninteresting objects in the scene. We believe that using the depth data has potential, perhaps even enough to detect objects by itself. To combine depth together with modern color- or detection-algorithm is probably the next step for precision detection, segmentation, and tracking of a specific object.

A major concern for tracking- and detection-algorithms is when objects can not be identified, it ends up outside the frame, or is blocked behind another object. As soon as the objects disappear one of the detection algorithms is activated e.g., template matching or SURF. We suggest a time-dependent sequence of operations to rediscover the missing object. A point of probability can be extracted by estimating the object acceleration, velocity, orientation, and direction before it disappeared. The algorithm starts the search after it has created a local search-window. If the object is not found through the iteration, the search-window expands. It expands until the whole frame is covered in the local search-window. If the point of probability is estimated outside the image-frame, the local search-area should be moved to the closest edge. This search method is used because of its superior attribute in speed and performance optimization compared to searching the whole frame from the start. This method should not be applied to the SURF-algorithm, because it uses very different techniques to discover objects.

The focus when tracking people is often not to track the whole body and get the complete shape. Instead, as mentioned above, the idea is to extract the position of the joints. By doing so, there are no problem to get the interesting data, that is movement and position of the person, without restricting tracking capabilities to a certain individual. This can be achieved by the application OSCeleton, which utilizes OpenNI. The application is straight-forward and we recommend it for experimenting. But to implement this yourself is rather complex. We suggests to use one of the feature extraction techniques that has been reviewed in this report, preferably *good features to track*. As explained, it is not that simple to identify where a knee is in an image with no reference. But by placing multiple 'good feature' points close to the knee, a relative position to the joint can be extracted out of the group of points and by using LK, tracking can be achieved. This method works as long as the joint is not too far away from the camera, because the pixel area can become too small.

OpenNI is a library that has great possibilities. However, OpenNI can not be connected to a camera device or Kinect if OpenCV already is in use. This problem can be solved by retrieving the image-data via OpenNI and convert the data to an image format supported by OpenCV.

We wanted to combine these two libraries but did not because there is no simple way to convert the raw-data from OpenNI to Images in OpenCV. We used OpenNI in a stand-alone application called OSCeleton, to extract the joints in the human body making it easy to use for motion-capture. In the application a 3D-scene together with objects and skeletons for every user was implemented. The user controls the skeleton by body motion, and can interact with the virtual objects.

One of our goals was to investigate emotional capture, unfortunately, due to other priorities we did not get the chance to implement such an algorithm in our application, but we did come up with a proposed algorithm. Even though this algorithm is not tested in its complete form, every part of the proposed algorithm consist of algorithms tested and evaluated in this report. Note, the algorithm and the explanation is presented in *Appendix A*.

For further development the proposed algorithm should be implemented, investigated properly, and integrated with skeleton tracking. Also hand tracking is also an important part of motion-capture. There are various solutions to track fingers independently, but to get emotional-capture, motion-capture, and tracking fingers to work together would be something to investigate in the future.

8 Conclusion

The task was to analyze different motion tracking techniques using laptop systems. This report starts by investigating different filters suitable for tracking algorithms. The investigated filters are: Gaussian, mean, average, Sobel, Laplace, Canny, dilation, erosion, and histogram back-projection. Various feature extraction techniques were analyzed: Harris corner detection, good features to track, refined accuracy, cascade classification, and SURF. Finally, the report discusses the following tracking algorithms: template matching, motion templates, moments, mean-shift, cam-shift, block-matching, and Lucas-Kanade.

It has been concluded that in order to track an object, specific properties for the tracked object should be enhanced. Image noise from the camera feed prevents optimal tracking performance. Different conditions needs to be taken into consideration when choosing computer-visualization algorithms, such as changes in illumination and disappearance of objects.

References

[1] *OpenCV Reference Manual*, v2.2 edition, December 2010.

[2] Bsd 3 clause lisence. `http://www.opensource.org/licenses/BSD-3-Clause`, June 2012. Director of the Office of Technology Licensing of the University of California.

[3] Extensible markup language (xml) 1.0 (fifth edition). `http://www.w3.org/TR/REC-xml`, June 2012.

[4] Latex wikibook. `http://upload.wikimedia.org/wikipedia/commons/2/2d/LaTeX.pdf`, June 2012.

[5] Libfreenect. `https://github.com/OpenKinect/libfreenect`, June 2012.

[6] The mit license (mit). `http://www.opensource.org/licenses/MIT`, June 2012. Massachusetts Institute of Technology.

[7] Numpy. `http://numpy.scipy.org/`, June 2012.

[8] Plantuml. `http://plantuml.sourceforge.net/`, June 2012.

[9] John G. Allen, Richard Y. D. Xu, and Jesse S. Jin. Object tracking using camshift algorithm and multiple quantized feature spaces. In *Proceedings of the Pan-Sydney area workshop on Visual information processing*, VIP '05, pages 3–7, Darlinghurst, Australia, 2004. Australian Computer Society, Inc.

[10] Lennart Andersson, Anders Grennberg, Torbjörn Hedberg, Reinhold Näslund, Lars-Erik Persson, Björn von Sydow, and Inge Söderkvist. *Linjär algebra med gometri*, chapter 7. Studentlitteratur, Lund, 1999.

[11] Aroh Barjatya. Block matching algorithms for motion estimation. Technical report, Blekinge Institute of Technology, IEEE, 2004.

[12] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *European Confeiefrence on Computer Vision*, pages 404–417, 2006.

[13] Alexander C. Berg and Jitendra Malik. Geometric blur for template matching. Technical report, Univeristy of California, Berkeley, 2001.

[14] Aaron F. Bobick, James W. Davis, and IEEE Computer Society. The recognition of human movement using temporal templates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23:257–267, 2001.

[15] Gary Bradski. Real time face and object tracking as a component of a perceptual user interface. In *Applications of Computer Vision*, volume 4th, pages 214 –219. IEEE Workshop, October 1998.

[16] Gary Bradski and James Davis. Motion segmentation and pose recognition with motion history gradients. In *Machine Vision and Applications*, pages 238–244, 2000.

[17] Gary Bradski and Adrian Kaehler. *Learning OpenCV*, chapter 13. O'Reilly Media, Inc., Sebastopol, CA, 2008.

[18] Gary Bradski and Adrian Kaehler. *Learning OpenCV*, pages 148–151. O'Reilly Media, Inc., Sebastopol, CA, 2008.

[19] Gary Bradski and Adrian Kaehler. *Learning OpenCV*, pages 151–153. O'Reilly Media, Inc., Sebastopol, CA, 2008.

[20] Gary Bradski and Adrian Kaehler. *Learning OpenCV*, pages 115–118. O'Reilly Media, Inc., Sebastopol, CA, 2008.

[21] Gary Bradski and Adrian Kaehler. *Learning OpenCV*, chapter 7. O'Reilly Media, Inc., Sebastopol, CA, 2008.

[22] Gary Bradski and Adrian Kaehler. *Learning OpenCV*, pages 323–334. O'Reilly Media, Inc., Sebastopol, CA, 2008.

[23] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679 –698, nov. 1986.

[24] Phang Chang and Phang Piau. Simple procedure for the designation of haar wavelet matrices for differential equations. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, volume 2, 2008.

[25] Jens Christensen, Jonas Brandvik, and Oliver Carlsson. Kinect motion capture. `https://github.com/prebz/kinect-motion-capture`, June 2012.

[26] D. Comaniciu and P. Meer. Mean shift: a robust approach toward feature space analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(5):603 –619, may 2002.

[27] Openkinect community. `http://openkinect.org`, June 2012.

[28] E. Roy Davies. *Machine Vision*, chapter 3. Elsevier Inc, 3rd edition, 2005.

[29] Lennart Ekbom, Stig Larsson, Lars Bergström, Uno Jönsson Alf Ölme, Sigvard Lillieborg, and Thomas Krigsman. *Tabeller och formler för NV- och TE-programmen*, page 100. Författarna och Liber AB, 2003.

[30] Jan Flusser and Tomás Suk. On the calculation of image moments. Technical report, Institute of Information Theory and Automation Academy of Sciences of the Czech Republic, 1999.

[31] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. Technical report, AT&T Labs, 1997.

[32] H. Greenspan, S. Belongie, R. Goodman, P. Perona, S. Rakshit, and C. H. Anderson. Overcomplete steerable pyramid filters and rotation invariance. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 222–228, 1994.

[33] Aglika Gyaourova, Chandrika Kamath, and Sen ching Cheung. Block matching for object tracking. Technical report, Department of Computer Science, University of Nevada, Reno; Center for Applied and Scientific Computing Lawrence Livermore National Laboratory, 2003.

[34] Chris Harris and Mike Stephens. A combined corner and edge detector. Technical report, Plessey Research Roke Manor, United Kingdom, 1988.

[35] Free Software Foundation Inc. Gnu general public license. `http://www.gnu.org/copyleft/gpl.html`, June 2012.

[36] Freâ Deâ Ric Jurie and Michel Dhome. Hyperplane approximation for template matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 996–1000, 2002.

[37] Robert Laganière. *OpenCV 2 Computer Vision Application Programming Cookbook*, pages 272–277. Packt Publishing, Birmingham and Mumbai, 2011.

[38] Ruan Lakemond, Clinton Fookes, and Sridha Sridharan. Affine adaptation of local image features using the hessian matrix. In *IEEE International Conference On Advanced Video and Signal Based Surveillance*, Geneoa, Italy, 2009.

[39] Rainer Lienhart and Jochen Maydt. An extended set of haar-like features for rapid object detection. Technical report, Intel Labs, Intel Corporation, Santa Clara, USA, 2002.

[40] David G. Lowe. Distinctive image features from scale-invariant keypoints. Technical report, Computer Science Department University of British Columbia, Vancouver, B.C., Canada, 2003.

[41] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of Imaging Understanding Workshop*, pages 121–130, 1981.

[42] James MacLean and John Tsotsos. Fast pattern recognition using gradient-descent search in an image pyramid. In *Proceedings of the 15 International Conference on Pattern Recognition*, pages 877–881, 2000.

[43] Steven C. McConnell. *Code Complete*, chapter 22. Microsoft Press, 2nd edition, 2004.

[44] Claudia Nieuwenhuis, Benjamin Berkels, Martin Rumpf, and Daniel Cremers. Interactive motion segmentation. Technical report, Technical University of Munich, Germany and University of Bonn, Germany, 2010.

[45] Stephen Prata. *C++ Primer Plus*. SAMS, fifth edition, 2005.

[46] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, 9 may 2011.

[47] Lennart Råde and Bertil Westergern. *Mathematics Handbook for Science and Engineering*, page 429. Studentlitteratur, Lund, 1998 and 2004.

[48] Jianbo Shi and Carlo Tomasi. Good features to track. Technical report, Computer Science Department Cornell and Stanford University, 1994.

[49] Jan Skansholm and Ulf Bilting. *Vägen till C*. Studentlitteratur, 3rd edition, 2000.

[50] Jon Sporring and Joachim Weickert. Information measures in scale-spaces. *IEEE Trans. Information Theory*, 45:1051–1058, 1999.

[51] Linus Torvalds and Junio C Hamano. Git. `http://git-scm.com/`, June 2012.

[52] Guido van Rossum. Python programming language – official website. `http://python.org/`, June 2012.

[53] Guido van Rossum and Barry Warsaw. Style guide for python code. `http://www.python.org/dev/peps/pep-0008/`, June 2012.

[54] Rebecca Vincent and Dr. Olusegun Folorunso. A descriptive algorithm for sobel image edge detection. In *Proceedings of Informing Science & IT Education Conference (InSITE)*, 2009.

[55] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. Technical report, Mitsubishi Electric Research Labs and Compaq Cambridge Research Lab, 2001.

[56] King Yuen Wong and Minas E. Spetsakis. Motion segmentation and tracking. Technical report, Intel Corporation, Microcomputer Research Labs and MIT Media Lab, 2002.

[57] Jean yves Bouguet. Pyramidal implementation of the lucas kanade feature tracker. *Intel Corporation, Microprocessor Research Labs*, 2000.

[58] Andreas Zeller. *Why programs fail*, chapter 3. Morgan Kaufmann, 2nd edition, 2009.

A Appendix A

```
let threshold_points: threshold forall nr points
    threshold_intensity: the intensity difference
                         between the tracked head images
    threshold_velocity: thereshold for movement
                        of the tracked points

head_object = extract head from image                          (1)
Save the extracted head as an image

tracking = True
list_points = []

while tracking:

    if length of list_points <= threshold_points:
        clear list_points
        list_points = generate points in head_object           (2)

        for pt in list_points:
            refine position of pt                              (3)

    calculate optical flow                                     (4)

    for pt in list_points:
        if get intensity pt <= threshold_intensity:            (5)
            pt current_pos = pt prev_pos

        calculate the velocity for pt                          (6)

        if get velocity pt >= threshold_velocity:
            remove pt from list_points


def get_intensity(pt):                                         (7)
    check diff in intensity
    for pt.previous_position and pt.current_position
```

1. Select and save the face using *Haar cascading 4.4*.

2. The point's auto and/or manually generated by using *good features to track 4.3*, the points should be scattered all over the face and head.

3. *LK* needs a higher accuracy than pixel accuracy can provide, and therefore *refined accuracy 4.2* is used.

4. The optical flow is calculated on each point by *LK 5.8*.

5. If the point is lost or does not meet the intensity requirements, the point is moved to its last known position. This happens when the face is blocked.

6. LK might fail to update the points position if they are moving too fast, to prevent the tracking from failing completely, a point that moves too quick is discarded.

7. Compare the pixel-intensity with previous frame.

If too many points have been discarded, the object can be relocated with SURF and new points will be generated. The operation is looped from step two, but as mentioned above this algorithm is untested.