UNIVERSITÄT BASEL

A framework for tracking ball movements in sports using fixed HD cameras

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Databases and Information Systems Group
dbis.cs.unibas.ch

Examiner: Prof. Dr. Heiko Schuldt
Supervisor: Ihab Al Kabary MSc.

Luca Rossetto
luca.rossetto@stud.unibas.ch

4. July 2012

Acknowledgments

I would like to thank Prof. Dr. Schuldt for giving me the opportunity to work on this project and the liberty to design this framework from scratch. I also want to thank Ihab Al Kabary for supervising this work and supporting me during the whole time of this project. Special thanks goes to the entire DBIS group and everyone else who spent some time generating the video data which was essential for building and testing this framework.

Zusammenfassung

Ein Framework zur Erfassung von Ballbewegungen mittels fixierten HD Kameras

Die wachsende Multimedialität heutiger Daten erschwert die Suche nach bestimmten Informationen. Bestand die Mehrheit der Daten vor nicht allzu langer Zeit noch aus Text, so werden Videos immer häufiger. Um die Suche von bestimmten Inhalten in einer Vielzahl von Videos zu ermöglichen, ist es notwendig, zuerst bestimmte Informationen aus den Videos zu extrahieren. Im Fall von Sportaufzeichnungen sind diese relevanten Informationen meist eng mit der Bewegung des Balles oder eines äquivalenten Spielobjekts verbunden. Durch Extraktion der Bewegung dieses Objektes kann die zur Suche notwendige Information gewonnen werden. Diese Arbeit beschreibt den Prozess, welchen das Java Motion Tracking Framework (JMTF) verwendet, um solche Bewegungsinformation zu extrahieren. Das JMTF, welches das primäre Resultat dieser Arbeit darstellt, besteht aus einem Set von Modulen zur Bildvorverarbeitung, Objektdetektion und Verfolgung sowie zur Nachbearbeitung der gewonnenen Positionsinformationen. Die modulare Architektur erlaubt dem Benutzer, Detektionssysteme für eine Vielzahl von Sportarten zu erstellen.

Abstract

A framework for tracking ball movements in sports using fixed HD cameras

Because of the increasing multimediality of data, search gets increasingly difficult. While data used to consist primarily of text, video gets more common nowadays. To be able to find certain information in a plurality of videos, some descriptive information has to be extracted beforehand. In the case of sports the interesting information is mostly connected to the movement of the ball or an equivalent game play object. By extracting the movement of the ball from a video, the necessary meta information can be created. This work describes the process which the Java Motion Tracking Framework (JMTF) uses to extract such motion data. The JMTF, which is the primary result of this work, consists of a set of modules for image pre-processing, object tracking and tracking data post-processing. Its modular architecture enables the user to build tracking pipelines for a wide range of sports.

Table of Contents

1 Introduction

1.1 The bigger picture

Most of the data generated today is not in an easily searchable format like text but rather in a more multi medial form like images or video sequences. Especially video sequences which often contain tens of thousands of frames require a lot of space to store and are difficult to index directly. To be able to make these kinds of media usable in a Content-Based Image Retrieval system, which is capable to find a image or video subsequence based not on the images themselves but on the content they show, more abstract data has to be extracted. In the case of sports like tennis, soccer, ping-pong or the like, this data consists mainly of the movement of players and game objects, mainly the ball. The extracted data can then be used by spatiotemporal database systems to query for certain movement patterns which can be used to reference the image data of their origin. These queries can be produced by sketching or by giving an example either out of an existing database or directly as a video sequence. This data extraction process which transforms video data into a format with which it is possible to perform this kind of querying is the motivation of this work. The goal of this work is to provide a framework to perform extraction of such motion data. A special focus lies in ball movements.

1.2 The challenges

The goal of this work was to design and implement a system, that could be used to extract motion data from image sequences for a wide range of sports. It had to be adaptable so it could be configured to generate reliable results in different scenarios of application. It also had to be fully implemented in Java since the environment it is going to be used in is written in Java as well. Additionally, it had to use as few external dependencies as possible, to be able to profit from the platform independence of Java. Lastly, the resulting framework had to be easily expandable to be able to add functionality in case the one present does not suffice in a future application.

1.3 Design of the framework

The Java Motion Tracking Framework (JMTF) is the central result of this work. It consists of multiple parts for image acquisition, image filtering, object tracking, tracking post-processing and data export. The way image data which is passed through an JMTF pipeline is inspired by the concept of node based compositing which is often used in image and video post-processing, visual effects and computer generated content like 3D animation. Every part of a JMTF pipeline consists of freely interchangeable modules, similar to the nodes in a compositor, through which the data is passed and manipulated on its way. There is therefore no fixed structure which has to be used within the different parts. Pipelines can be created to fit the requirements of a certain scenario. The five parts described above are however (more or less) fixed within that particular order. The framework is designed for offline data processing and is not capable of tracking objects in real time.

Figure 1.1: Basic structure of a JMTF pipeline

1.4 Applications of the framework

The Java Motion Tracking Framework is designed to be useful in a wide range of applications. Its main focus lies however on sports applications, indoor and outdoor. It was built with the intention to be able analyse the movement of balls and other sports game play items. In its current state, the framework can be used to track movements of balls in for example soccer, tennis, handball, ping-pong and many more. It is also capable of tracking the players in scenarios like ping-pong or tennis, where there is a small number of players and the regions in which these players move do not overlap. It is however not capable of distinguishing lots of different but similar objects like players in a soccer or handball game and can therefore not be used for tracking their movements.

1.5 Related work

The JMTF is not the first framework which covers the topic of motion tracking in Java. It is however more dynamic and flexible than the other ones described in this section.

1.5.1 OpenCV

OpenCV [1] is the probably most powerful library for computer vision applications. This library was originally developed by Intel and is written in C/C++. It is very popular in the field of computer vision because of its wide range of most highly optimised filters. To be able to use this library in Java, a wrapper called JavaCV can be used. This native library

provides all functionality of OpenCV at the expense of the platform independence. Since all the functionality is implemented in C/C++, all the image data has to be managed outside the Java Virtual Machine but referenced inside the Java program which controls the flow of the data. When the data has to be manipulated inside the JVM, it has to be moved into the corresponding memory and moved back afterwards. It is therefore difficult to keep the data references consistent while not loosing the flexibility of interchangeable image processing modules.

1.5.2 JMyron

JMyron [2] is a real-time image manipulation library for Processing. It augments the image manipulation capabilities of Processing with things like background subtraction, edge detection or motion detection. It is quite limited in its functionality and more useful for artistic purposes than for general data processing. For video input it relies completely on the video functionalities of Processing which again depend on native functions which are not available for all operating systems. Due to the integration in the Processing context, JMyron is not particularly flexible and difficult to use in a different environment.

1.5.3 Java Motion Tracking API

The Java Motion Tracking API [3] is a fixed pipeline for detecting all kinds of motion in a video, either live or pre-recorded. It uses the Processing video libraries for live video input and shares therefore all platform dependency problems of JMyron. Due to its fixed pipeline architecture it offers almost no flexibility for changing the way it operates.

2

Image filtering and preprocessing

The JMTF consists of a range of different types of image filters. These filters can be applied to isolate the parts of the image, which are interesting for the subsequent tracking process.

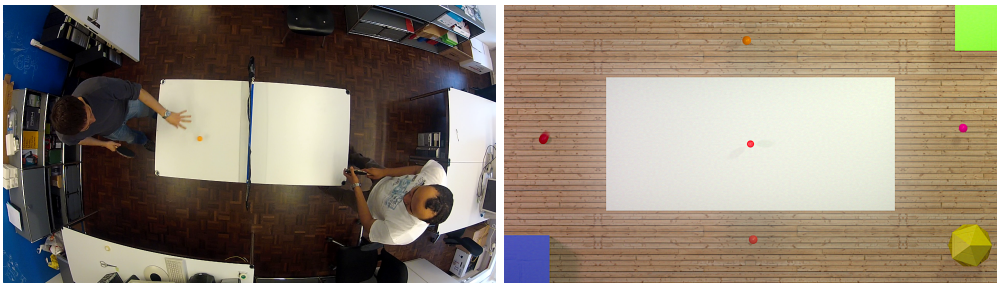

Figure 2.1: The test images used to demonstrate the filters described in this chapter (left: test setup recorded at 1080p, right: synthetic test scene rendered at 1080p)

2.1 Pixel based filters

Most of the filters contained in JMTF are pixel based filters. In a pixel based filter, a function is applied to every pixel individually without considering any of the other pixels in the image.

2.1.1 Color cut filters

The color cut filters are the simplest filters in JMTF. A color cut filter simply evaluates a given binary function for each pixel. If the function returns true, the value pixel does not get changed, otherwise, it is set to solid black.

2.1.1.1 Hue cut-out filter

The hue [4] cut-out filter is initialized with a range (H_{min} to H_{max}) of allowed hue values. It checks for every pixel if its value lies within this given range. If so, the value gets preserved.

$$H_{min} \leqslant H_p \leqslant H_{max}$$

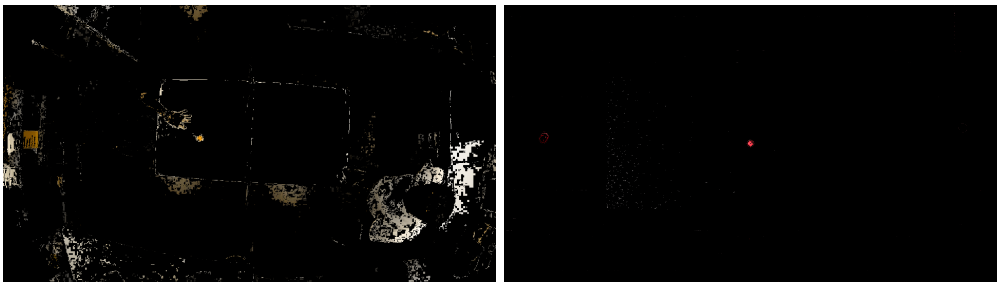

Figure 2.2: Hue cut-out filter applied to test images

2.1.1.2 Gray cut-out filter

The gray cut-out filter or intensity cut-out filter takes one threshold value (T) for initialization. It then calculates the brightness of the pixel (b) and evaluates the distance between b and the color values of the pixel (R_p, G_p, B_p). If all three distances are greater than the specified threshold, the value is kept. (If the threshold value is negative, the absolute value is taken for comparison and the result is inverted.)

$$b = \frac{R_p + G_p + B_p}{3} \qquad (|R_p - b| > T) \wedge (|G_p - b| > T) \wedge (|B_p - b| > T)$$

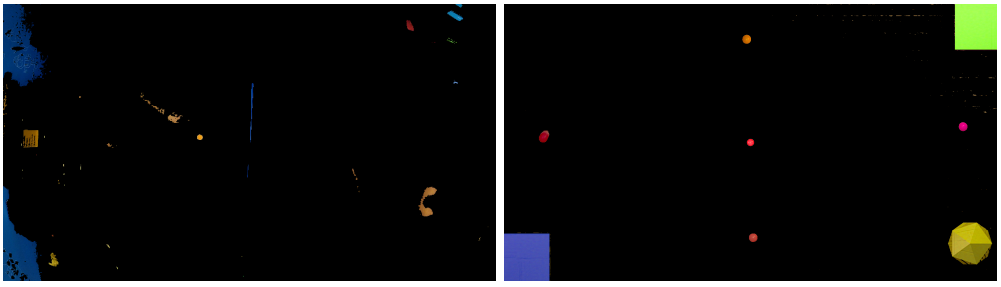

Figure 2.3: Gray cut-out filter applied to test images

2.1.1.3 Color distance cut-out filter

The color distance cut-out filter simply calculates the euclidean distance (D) between the color of a pixel and a given one.

$$D = (R_1 - R_2)^2 + (G_1 - G_2)^2 + (B_1 - B_2)^2$$

If the distance is greater than a specified threshold, the pixel is set to black.

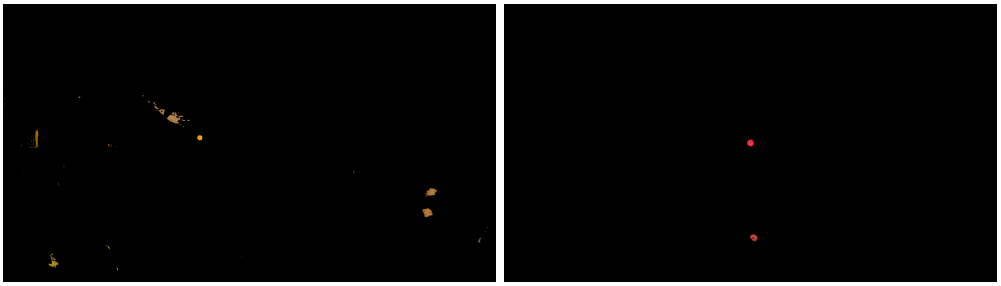

Figure 2.4: Color distance cut-out filter applied to test images

2.1.2 Color space transformation

Color space transformation filters in general map an image from one color space into another. This can be used to isolate certain features of the object to track which may be more easily to distinguish from the rest of the image when using another representation of the color. Therefore it does not matter if the target color space is a standard one like HSV, CIEXYZ [5] or grayscale or a custom one.

2.1.2.1 Color mapping filters

A color mapping projects a set of colors into another mostly smaller set. It does this every color of the input set with that one from the output set which has the minimal distance (D) to the one from the input. This filter can be used for reducing in the colorspace of an image in a way that the color density in regions of low interest while keeping a higher density in an area in which regions of high interest are found. Therefore, a special reduced color set has to be constructed to fit the requirements of the respective scene.

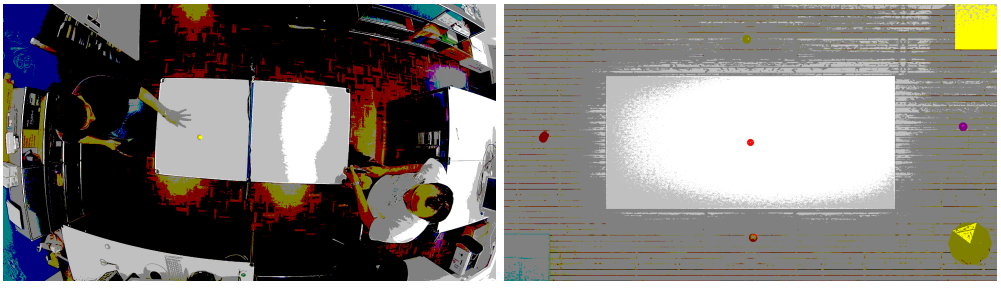

Figure 2.5: Color mapping filter applied to test images to reduce color count to 16

2.1.2.2 Image to grayscale

The grayscale filter calculates the average of all three color channels of an image.

$$(R'_p, G'_p, B'_p) = (v, v, v) \qquad v = \frac{R_p + G_p + B_p}{3}$$

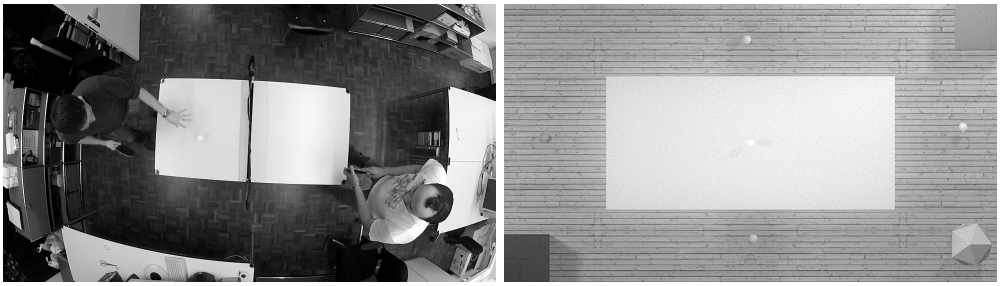

Figure 2.6: Test images converted to grayscale

2.1.2.3 Image to hue

The image to hue filter converts the image from RGB to HSV. It then takes the Hue value from it, and assigns it to every channel of the image. The result of this filter is therefore a grayscale containing the hue instead of the intensity of the color.

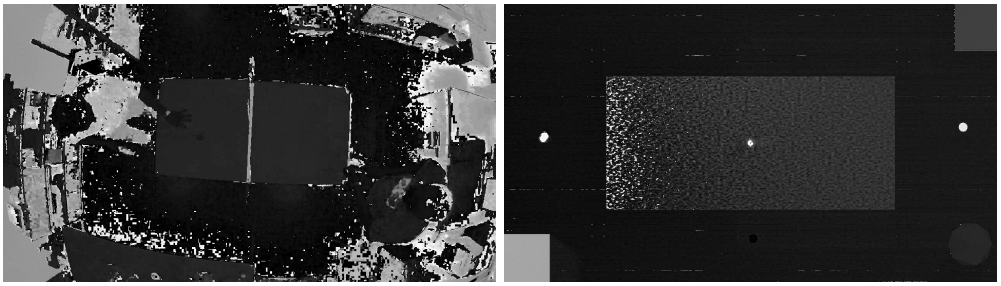

Figure 2.7: Image to hue filter applied to test images

2.1.2.4 Image to hue * saturation

The image to hue * saturation filter is quite similar to the image to hue filter but with one advantage. The image to hue filter generates quite a lot of noise in areas with a saturation close to zero since the hue is not properly defined in those areas. This problem can be avoided by multiplying the hue value of a pixel ($H_p \in [0, 1)$) with the saturation value ($S_p \in [0, 1)$). To avoid loosing values with a high saturation, H_p is offset by 1 so that only pixels with low saturation get a value close to. To get a value between 0 and 255, the resulting product has to be multiplied by 128 and rounded down.

$$HtS_p = \lfloor (H_p + 1) * S_p * 128 \rfloor$$

The value HtS_p is then assigned to all three color channels of the pixel so that the resulting image is grayscale.

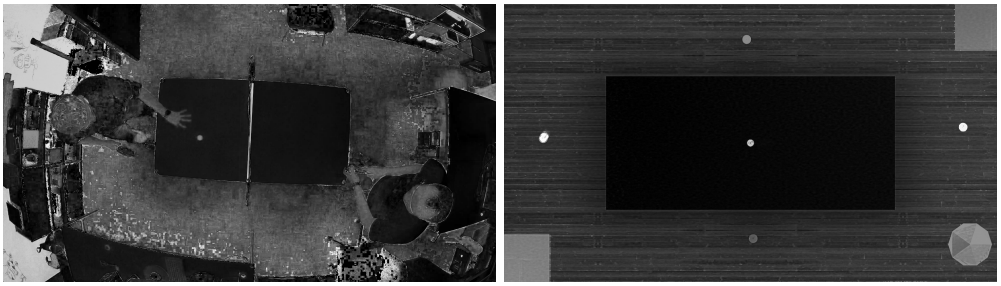

Figure 2.8: Image to hue * saturation filter applied to test images

2.1.2.5 Linear colorspace transformation

The linear color space transformation filter performs a linear transformation to map the color of a pixel in the input color space (R, G, B) to a new one (R', G', B'). This linear transformation is defined by a 3x3 matrix A.

$$(R', G', B') = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} (R, G, B)^\intercal$$

This filter can be used to transform to a common color model like XYZ or a custom one for example for color balance or isolation of special regions.

2.1.2.6 Color space transformation and PCA

Using a Principle Component Analysis [6], a custom color space can be created which isolates the color region of the object of interest. To do this, a special image has to be created which contains cutouts of the object of interest from different frames and ideally with different backgrounds and in different lighting conditions. Then, the PCA has to be applied to this image using the provided MATLAB script. This script will return a 3x3 Matrix which then can be used to perform a linear colorspace transformation as described above. This transformation will highlight the color regions which were most prominent in the input image while suppressing everything else. This method will only work with single colored objects.

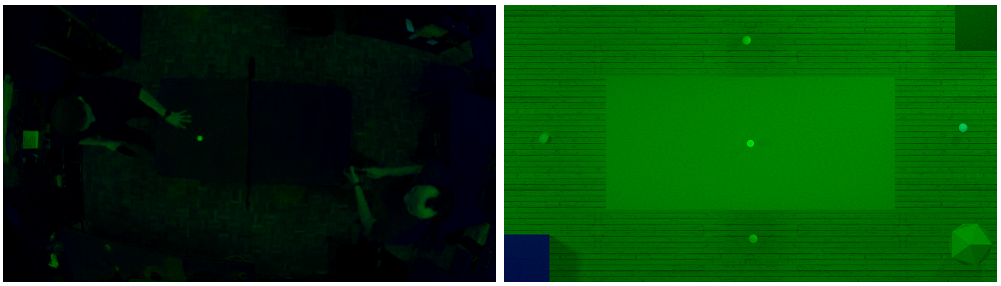

Figure 2.9: Color space transformation applied to test images

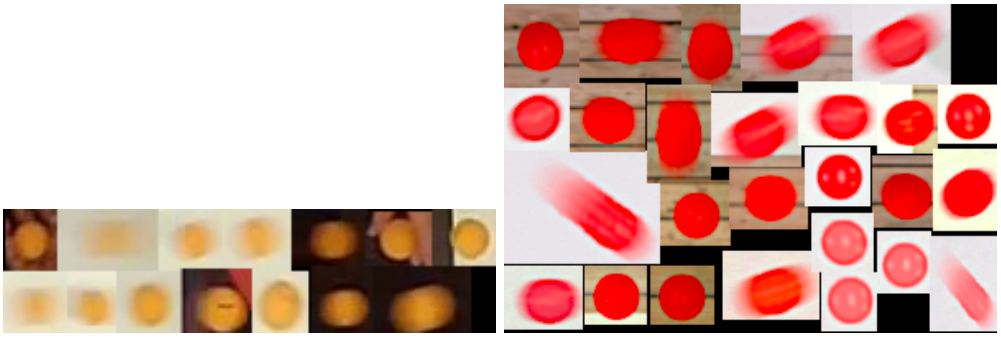

Figure 2.10: Input images for the image PCA

2.1.3 Color intensity mapping filters

The color mapping filters all apply a mapping function (f) to every color channel intensity of every pixel (I_p) to get a new intensity value (I'_p).

$$I'_p = f(I_p)$$

With the corresponding map, this filter can be used for example as gamma correction or color curve filter.

2.2 Image based filters

Other than the pixel based filters, the image based filters take multiple pixels as well as their spatial relationship as an input. The input can therefore be a cutout of the image or the image as a whole.

2.2.1 Histogram filters

Histogram filters are essentially color intensity mapping filters with a dynamic mapping function. The mapping function is newly generated for every image based on information extracted from its histogram. This allows to perform automatic image modification without the need of estimating parameters manually.

2.2.1.1 Histogram stretching

The histogram stretching filter is the simplest possible filter to normalize a histogram of an image. It works by stretching a histogram that only covers a part of its possible range to the full range. To do that, the filter finds the minimum (H_{min}) and maximum (H_{max}) value of the histogram which is not zero and builds the following mapping function.

$$f(x) = \frac{255(x - H_{min})}{H_{max} - H_{min}}$$

Since camera features like automatic aperture control and white balance generate images where the histogram already fills the maximal range, this filter has a very limited range of application.

2.2.1.2 Histogram equalization

The histogram equalization filter [7] is an automatic image quality improvement filter without any additional parameters. It maximises the color range of an image by transforming its histograms so they become as close to a uniform distribution as possible. To do so, the histogram is normalised so it can be treated as a probability density function. Afterwards, the cumulative distribution function is calculated. The new histogram is obtained by simply multiplying the CDF which has a range of $(0, 1)$ by 255 to stretch it to its full range of $(0, 255)$. This is done for the three color channels individually. This simple algorithm is used often for automatic image color correction, since it is able to compensate (to a certain degree) for changes in the image brightness as well as the white balance. It is however not able to compensate for local changes in image illumination since it always operates on the image as a whole.

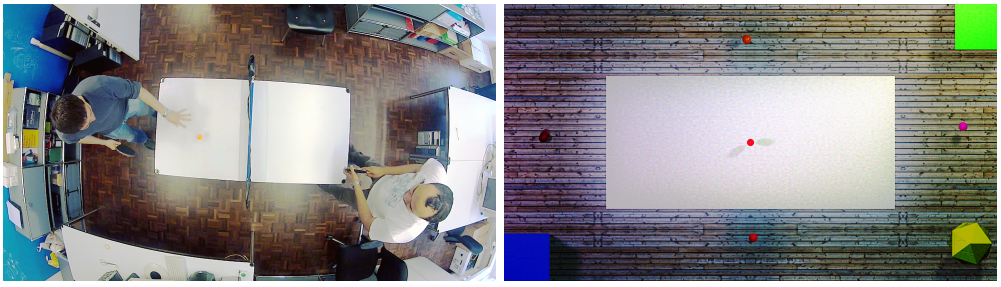

Figure 2.11: Histogram equalization applied to test images

2.2.1.3 Lloyd-Max quantisation filter

The Lloyd-Max quantisation filter [8] reduces the number of intensity levels ($q_0 to q_{K-1}$) in a color channel of an image to a specified number (K). Rather than equally spacing the new quantisation levels, it positions them in a way that the mean-square quantisation error (E), which indicates the loss of information due to the lower quantisation, is minimized. To do so, the histogram needs to be normalized so it can be regarded as a probability density function ($p(z)$).

$$E = \sum_{k=1}^{K} \int_{z_{k-1}}^{z_k} (z - q_k)^2 p(z) dz$$

In a grayscale image, this filter is able to significantly reduce the number of intensity levels without decreasing the visual appearance. In a RGB image with three similar histograms it tends to drastically reduce the saturation of the image due to the fact that the newly estimated brightness levels for the three color channels are relatively close to each other. To attenuate this effect, it is recommended to choose different numbers of intensity levels for the three color channels, which should ideally be coprime to each other. It is also advised to use more intensity levels in those channels in which the object of interest is easiest to distinguish from the background to insure that the object can still be separated from the background after applying the filter.

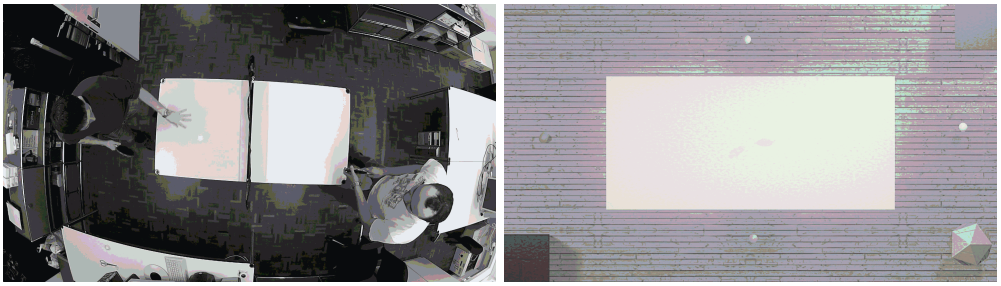

Figure 2.12: Lloyd-Max quantisation filter applied to test images with 8 remaining colors per channel

2.2.2 Convolution filter

The convolution filter of the JMTF uses a vector as a kernel for separable convolution which is applied in both directions of the image. This particular filter uses the same kernel for the X and Y direction of the convolution.

2.2.2.1 Convolution blur

The convolution blur filter uses an approximated gaussian bell curve as a convolution kernel. To approximate the bell curve, a normalized row from the sierpinski triangle is used. To normalize the row, it is divided by 2^l. It is not advised to use this filter with a big kernel size, since it may decrease the tracking results.

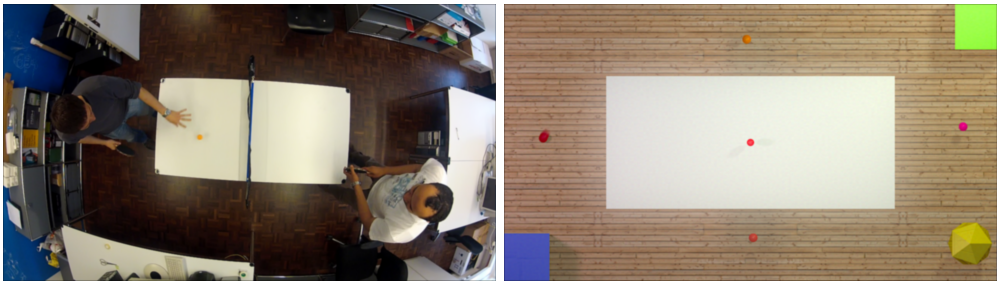

Figure 2.13: Convolution blur applied to test images with a radius of 20 pixels

2.2.3 Noise reduction

The noise reduction filter only works for grayscale or binary images with a black background. It analyses for every non black pixel the box of its surrounding pixels. If less then a specified number of pixels within that box which has a diameter of $2r + 1$ is different from black, the center pixel will be regarded as noise and set to black. Since one iteration of this filter may leave too much noise in the image, it can iterate over the image multiple times. The settings of this filter have to be balanced carefully, since a too small tolerance value will leave to much noise in the image, a value too big will result in removal of non-noise objects. The same is true for the number of iterations. Too many iterations may remove to much, where as to few iterations may leave to much. Unfortunately, there is currently no heuristic to automatically estimate appropriate values for this filter.

2.2.4 Median filter

The median filter is a mostly edge preserving color and detail reduction filter that also acts as a noise remover. It therefore calculates for every pixel the median color of a box surrounding it. The bigger this box is chosen, the less detail will remain after the filtering. To calculate the median color of this box, the median of the three color channels have to be calculated separately, since the median is only defined for one-dimensional data. To calculate this median efficiently, a color histogram is used. To calculate the median using the histogram, it is successively summed up from both sides until the two sums are equal. The position obtained in this way is the median of the data out of which the histogram was generated. Since combining three independently calculated values into one color may produce a color which was not present in the input set, the color from the input set which has the smallest distance to the calculated color is returned.

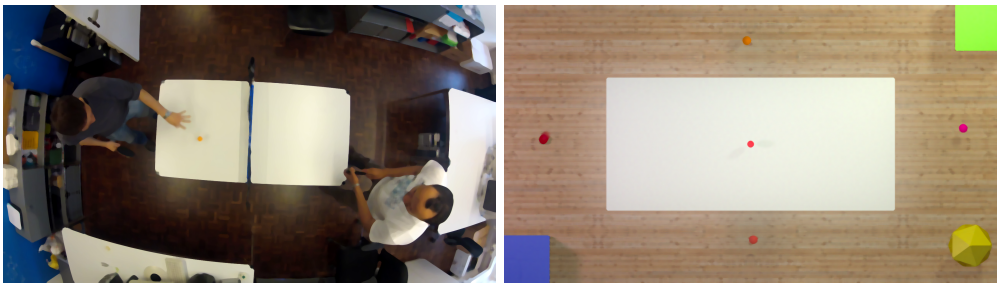

Figure 2.14: Median filter applied to test images with a radius of 7 pixels

2.3 Dynamic filters

A dynamic filter is a filter that does not operate on only one image but on multiple ones. They are therefore able to accumulate more general information about the images and improve their performance over time.

2.3.1 Background estimator

In contrast to most other commonly used background subtraction filters, this filter does not simply subtract an image from its predecessor to estimate the foreground. The background estimator uses a simple learning algorithm to continuously estimate the background. Everything in the current image that differs to much from the estimated background is regarded as foreground. To estimate the background, the filter has two models for every pixel, one of which is initially completely black, the other one is completely white. Additionally, there is a certainty value for every pixel and model which is initialized with zero. In every iteration, the distance between the input pixel and the two models is calculated. If the distance between the input and the better model (which is the one with the higher certainty) is below a certain threshold, the certainty value for this model is increased linearly, otherwise it is decreased exponentially. If the distance for the worse model is below the same threshold, its certainty gets increased linearly by the same amount. If the distance is however bigger than the threshold, the color is set to the input color and the certainty is reset to one. By changing the threshold, the time the filter takes to adjust to mayor changes in the image can be

modified. For estimating the foreground, the distance between the more certain model and the input is calculated. If this distance is within a certain threshold, the pixel is regarded as background, otherwise it is regarded as foreground.

2.4 Image combination filters

An image combination filter takes two images as an input and combines them into one. These filters are useful for overlaying information over an image or mask out certain areas. Since color channels only contain values between 0 and 255, resulting values outside of this range need to be cropped.

2.4.1 Image adder

The image adder is the simplest image combination filter. Aside from two input images (I_1 and I_2), it takes a factor (a) as an input. This factor is multiplied to the first image before adding it to the second one. The equation for this filter is therefore:

$$I' = a * I_1 + I_2$$

2.4.2 Image mixer

The image mixer is very similar to the image adder. The only difference is that the image mixer multiplies the second image by $1 - a$. So if a is between 0 and 1 it determines the percentage of the first image in the resulting image.

$$I' = a * I_1 + (1 - a) * I_2$$

2.4.3 Image multiplier

The image multiplier simply multiplies the two given images together with a factor. To be able to do that, it needs to normalise the images.

$$I' = 255a(\frac{I_1}{255} * \frac{I_2}{255})$$

3

Object tracking

3.1 The blob

A blob is an amorphous area of connected pixels of equal or at least similar intensity which differs from its surrounding area. A blob by itself does not contain any other information but its position and its size. Therefore a blob by it self is meaningless and only defined by the process which lead to its creation. Blob detection is never the less preferred over object detection because isolating the objects of interest using global methods and then detecting blobs is more efficient and generalisable than using local methods to detect the object directly.

3.2 The tracker

The blob tracker or blob detector is the module which uses a grayscale image as an input and returns all blobs found in this image in a descriptive way. The process of finding the blobs in an image is called blob detection. Blob tracking is the process of applying blob detection to multiple sequential images. The images which are fed into the blob detector are pre-processed in such a way that only the objects of interest are remaining and everything else is set to black. Therefore, every blob in the image results from such an object. By tracking the blobs in these processed images, the locations of the objects of interest can be determined. This is the essential abstraction step in JMTF which transforms image data into position data. There are many ways to do blob detection. The method used in JMTF is called growing region blob detection.

3.2.1 Growing region blob detector

The growing region blob detection algorithm consists of two major parts. Line blob detection and line blob merging.

3.2.1.1 Line blob detection

A line blob is a one dimensional blob which is only characterised by its line, a start and an end point. These line blobs are easy to detect by scanning the image line by line. As soon as a pixel with a value greater than a certain threshold is found, it is regarded as the starting point of a new line blob. The scan is continued until a pixel with a lower value is found or the end of the line is reached. After a line blob is found, there is checked if a overlapping line blob exists in the line above. In that case, the new line blob gets the same id as the one above, otherwise, a new id is assigned. In case there are multiple overlapping line blobs in the line above with different ids, the line blobs with the higher ids get reassigned. All line blobs are stored in a list of lists with their id as a key. After the image is completely scanned and all line blobs are detected, the second part of the algorithm begins.

3.2.1.2 Line blob merging

All the line blobs with the same id need to be represented using a simple blob representation. In JMTF, a so called rect-blob is used, which is basically just the bounding box of the blob. To get this bounding box, the extremal values in X and Y for each list of line blobs is determined. After the blob is described by its box, it is stored in a struct containing the source image and a Hashtable which is returned at the end of the process.

4

Tracking data post-processing

4.1 Post-processing

The post-processing step is needed to clean the tracking data, filtering out the false positives and interpolating where the tracker was not able to locate the object of interest. Other correction processes can also happen in this step like translation, scaling or undistortion of the coordinates of the tracked objects. All these post-processing steps are done by modules called tracking data transformers.

4.1.1 Data filters

The data filters remove some blobs from a tracking data set which do not meet certain criteria. The goal is to be left with only one blob which (most likely) represents the object of interest.

4.1.1.1 Extremal area size filter

The extremal area size filter removes all blobs from a tracking data set except for either the one with biggest or the smallest size depending on the setting of the filter. This filter is useful for removing false positives resulting from remaining noise in the image after the pre-processing step.

4.1.1.2 Closest distance filter

The closest distance filter takes a set of coordinates for initialisation and leaves only the one blob in the tracking data set which is closest. The coordinates of this blob are then used for

the next tracking data set and so on.

4.1.2 Translation and scaling

JMTF contains multiple transformers to translate and scale the blobs in a tracking data set. This is necessary to be able compensate for camera movement between different recordings or for centering the origin point which is needed for undistortion.

4.1.3 Undistortion

Most object tracking systems perform undistortion on the image at the beginning of the process. When there is no interest on the exact shape of an object, it is however much less computationally expensive to just undistort the coordinates of the found objects after tracking. The JMTF contains a tracking data transformer for radial undistortion [9]. This transformer uses a list of previously estimated parameters (k) which represent the properties of the optical which was used to create the image to map the distorted input coordinates (x_d, y_d) to the undistorted coordinates (x_u, y_u).

$$r = \sqrt{x_d^2 + y_d^2} \qquad D = k_1 r^2 + k_2 r^4 + ...$$

$$x_u = x_d + Dx_d \qquad y_u = y_d + Dy_d$$

4.1.4 Interpolation

The interpolation step can compensate for small gaps where the tracker was not able to detect the object of interest. The JMTF offers a linear interpolation module to bridge these gaps. It works by buffering the results from a precedent tracking data source until a blob is found again. Is the number of frames between two blobs smaller than a specified threshold, the gap is bridged. At the current state, this interpolation only works for one blob per tracking data set since the togetherness of multiple blobs can not be reliably determined. It is therefore advised to use a filter before interpolating, since the interpolation module takes only the first blob in a set into account which may not be the one of interest.

4.1.5 Outlier detection and removal

The outlier detection and removal step eliminates false positives by the tracker. It finds single blobs which are far away from their surrounding blobs in the previous and next frame. It does that by comparing the distance between the blobs across three frames against a specified threshold. This filter is only capable of removing a single outlier and is unable to detect cluster of outliers.

4.2 Data export

Data export is the final step in a tracking pipeline. JMTF provides a simple automated data exporter for comma separated values. The CSVExporter automatically requests new tracking data sets and writes the position of all contained blobs either to an output stream or directly to a file until the tracking data source is exhausted.

5 Implementation

5.1 Concept and structure

The Java Motion Tracking Framework (JMTF) consists of freely interchangeable modules through which an image, represented through the JMTFImage class, is passed and manipulated on its way. These modules can be categorised into four different groups: image sources, image filters, trackers and tracking data manipulators. Within these categories, modules can be combined and interchanged freely, the order of the categories themselves is however fixed. The JMTF is written in pure Java and does not depend on any other libraries other than the Java default ones. It is therefore completely platform independent.

5.1.1 JMTFImage

The JMTFImage is the primary image class in the JMTF. It stores a 24bit color image using an integer array and provides all basic tools for manipulating pixels. A custom class for image storage was chosen over one of the existing Java image classes for two reasons. The most commonly used image class in Java is the BufferedImage which is optimised for getting displayed. Since the goal of JMTF is to manipulate images for tracking and not displaying them, using BufferedImages would have resulted in a decrease in performance. The other and more important reason for using a custom image class is the flexibility. The custom class can be extended freely to add missing information or functionality when needed. The JMTFImage does for example have a flag for grayscale images which is needed for certain filters and the trackers. To store even this little additional information in a BufferedImage would require much more effort.

5.1.2 Image sources

An image source is the entry point for every JMTF pipeline. It provides a sequence of JMTFImages which are either generated or read from the file system. When an image source is exhausted, null is returned. To avoid checking for null values, every image source provides the capability to check, whether or not the next request for an image can be satisfied. Since

an image source does generally not know the exact state of its underlying data source, it is not possible to determine how many images are left before exhaustion of the source.

5.1.2.1 Color image source

The color image source produces an image of specified dimensions completely filled with a specified color. It is an inexhaustible image source since it returns in every iteration a new copy of the same image. By itself, this image source is rather useless, but in combination with an image combination filter it can for example be used for global color manipulation.

5.1.2.2 Single image source

Like the color image source, the single image source is an inexhaustible image source. It takes a JMTFImage for initialisation and returns then in every iteration a copy of this image. This source can for example be used to provide image masks or constant overlays which can be combined with another image.

5.1.2.3 Folder image source

The folder image source uses java ImageIO to read all images within a specified folder and returns them one by one. By default it only loads JPG, PNG and BMP images, but it accepts different FileFilters on initialisation. The folder image source scans the specified input folder only once in the beginning and therefore assumes that the content of this folder won't change during the runtime. In contrast to the two image sources described above, this is an exhaustible image source, so before requesting a next image there has to be checked if there are images left to read.

5.1.2.4 Listening image source

The listening image source bridges the gap between the active image requesting and the passive image notification by buffering the last image update it received via notification and providing it on request. See 5.2.1.2 for further information.

5.1.3 Image filters

All image filters share the property that they take at least one image source as an input and provide an image as an output. To be able to chain multiple image filters, image filters are also image sources. For detailed information about all the different filters of JMTF see chapter 2.

5.1.4 Trackers

A tracker is a module that takes an image as an input and returns a list of objects found in the image. It therefore terminates a chain of image filters and starts a new chain of tracking data sources. The object returned by a JMTF TrackingDataSource a struct called TrackingDataSet which contains the source image and a Hashtable with Integer as key and Blob as value. For more information about trackers, see chapter 3.

5.1.5 Tracking data transformers

What image filters are for JMTFImages, are tracking data transformer for tracking data sets. They are made for post-processing the tracking data to, for example, remove false tracks or translate the coordinates to fit within the expected range. Since every tracking data transformer is also a TrackingDataSource, they can be chained like image filters. For more information see chapter 4.

5.2 Architecture of the code

The architecture of JMTF is based on the decorator pattern [10]. This means that almost every module that takes some kind of data source (except the ones that make a transition form one data type to another) is itself such a source. To be able to provide the flexibility which such a framework needs to be usable, some base classes and interfaces are needed.

5.2.1 Interfaces

To describe the fundamental functionality of the framework, some interfaces are used. To keep the structural overhead as small as possible, these interfaces contain as few methods as possible.

5.2.1.1 ImageSource

The ImageSource interface describes the way in which images are passed actively through modules. It defines methods for requesting the current or the next image of this source as well as a method to check whether or not there are more images.

5.2.1.2 ImageUpdater

There are two ways in which images are passed between modules, an active way by request and a passive way by notification. The ImageUpdater interface describes one side of the passive way. It only describes two methods, one to add and one to remove an ImageUpdateListener from this particular ImageUpdater.

5.2.1.3 ImageUpdateListener

The ImageUpdateListener can be attached to an ImageUpdater to receive new images. This interface describes only one method, the update method which is called by the ImageUpdater whenever there is a new image.

5.2.1.4 ImageManipulator

The ImageManipulator interface is an extension of the ImageSource interface. It additionally describes methods to change the ImageSource of the ImageManipulator as well as the manipulate method which when implemented performs the actual image manipulation.

5.2.1.5 TrackingDataSource

The TrackingDataSource describes, similar to the ImageSource, two methods to get the current and the next tracking data set.

5.2.1.6 TrackingDataUpdater

The TrackingDataUpdater interface is the equivalent to the ImageUpdater for tracking data. When implemented, it notifies all TrackingDataUpdateListeners when a new tracking data set is available. This interface is implemented by the AbstractTracker as well as the AbstractTrackingDataTransformer.

5.2.1.7 TrackingDataUpdateListener

The TrackingDataUpdateListener interface is the equivalent to the ImageUpdateListener. It describes an update function which is called by the ThrackingDataUpdater.

5.2.2 Abstract classes

Some abstract classes are needed to implement the basic functionality of the framework. These classes form the structural base upon which all classes that perform the actual tasks are built.

5.2.2.1 AbstractImageSource

The AbstractImageSource partially implements the methods described in the ImageSource interface and fully implements the ImageUpdater interface. It handles all the ImageUpdateListeners and makes sure they get updated when needed. It also introduces a buffer for the current image and returns it on request.

5.2.2.2 AbstractImageManipulator

The AbstractImageManipulator inherits from the AbstractImageSource. It also partially implements the ImageManipulator interface and fully implements the ImageUpdateListener interface. The only abstract method in this class is the actual manipulate method. All image manipulations are performed in-place.

5.2.2.3 AbstractImageCombiner

Similar to the AbstractImageManipulator, the AbstractImageCombiner is an extension of the AbstractImageSource. It does however not implement any additional interfaces but defines the only abstract method needed, the combine method, itself.

5.2.2.4 AbstractTrackingDataTransformer

The AbstractTrackingDataTransformer implements the basic functionality of the TrackingDataSource by fully implementing the interface. It additionally defines the abstract method transform which is equivalent to the manipulate method in the ImageManipulator.

5.3 Helper classes

JMTF contains some classes which don't generate or manipulate data in any way. These classes are here to assist the user in setting up a pipeline using the framework.

5.3.1 Image display

The image display is an ImageUpdateListener which does nothing more than displaying the received image on a monitor. Since the images are manipulated in-place by the image manipulators, the image display makes a copy for displaying whenever there is a new image.

5.3.2 Image to file exporter

Similar to the image display, the image to file exporter implements the ImageUpdateListener interface. The image to file exporter uses Java ImageIO to write all updated images to a file. As file name it uses a specified prefix together with the frame number of the image. The format in which to save the images can be specified as well.

5.3.3 Tracking data display

The tracking data display is a TrackingDataUpdateListener. It behaves like an image display but additionally marks all the found blobs in the image.

6

Evaluation of camera data

6.1 The camera

The camera used to generate the test video sequences is a HERO 2 from GoPro [11]. This camera is originally designed to be used in outdoor sport applications, but it has become very popular in research applications because of its small form factor, wide range of resolutions and frame rates and its relative low cost. It uses a fixed focus 170° wide angle lens which allows to capture big area from a relatively short distance. The biggest disadvantage of this camera is its automatic white balance which has no manual mode. Automatic white balance can lead to color shifts when the conditions in which the recording is made change. This is problematic when the used tracking pipeline relies on color to identify the object of interest. Since there is no information of the settings used for white balance and their changes over time, there is no general way to compensate for it.

6.2 The setup

Due to space restrictions, the test environment for generating the evaluation video data was quite small. It consisted of an office table which was equipped with a ping-pong net. Above the table, a specially designed mount was holding the camera which was placed on top of two ceiling lights. Since the camera does not support any remote configuration without any specialised additional hardware, the camera had to be configured and the recording had to be started before placing the camera into the holder.

Figure 6.1: The setup to produce the test videos

6.3 The procedure

For evaluating the performance of the framework and to be able to compare different pipelines against each other, two types of video were used. The first set of videos consists of three synthetic test videos generated using a 3d suite [12]. Each of these three videos shows the same ball movement, but the number of other objects, lighting and color changes as well as artificial noise variates. Since the position of the ball is known exactly for every frame, it is possible to directly compare the input coordinates used for rendering with the coordinates generated by the different pipelines. These videos all have a resolution of 1920 by 1080 pixels and are 1200 frames long. The second set of test videos consists of seven videos generated by the setup described above. All of them are 1000 frames long but differ in resolution and frame rate at which they were recorded. See the list below for details. From each of these sets, a random subset of 20 frames in which the ball was visible has been taken and manually tracked. By comparing the results of the tracker with the manually generated true position, the performance of the tracker can be estimated.

6.3.1 Video formats

To be able to evaluate the importance of resolution versus frame rate, a video for every setting which is supported by the camera was recorded.

- 1920 by 1080 pixels @ 30 fps
- 1280 by 960 pixels @ 30 fps
- 1280 by 960 pixels @ 48 fps
- 1280 by 720 pixels @ 30 fps
- 1280 by 720 pixels @ 60 fps
- 848 by 480 pixels @ 60 fps
- 848 by 480 pixels @ 120 fps

6.3.2 The pipeline

The pipeline used in this evaluation is a very simple one. It was designed to contain as few manual settings as possible. The pre-processing stage consists of a folder image source which provides the images, a colorspace transformation node with a transformation matrix resulting from an image PCA which isolates the ball, a background estimation node and a color channel isolation node which isolates the channel with the interesting information. The post-processing step consists of a brightest blob filter, a linear single outlier detector and a single linear interpolator limited to 10 frames. The data is exported using a CSV exporter.

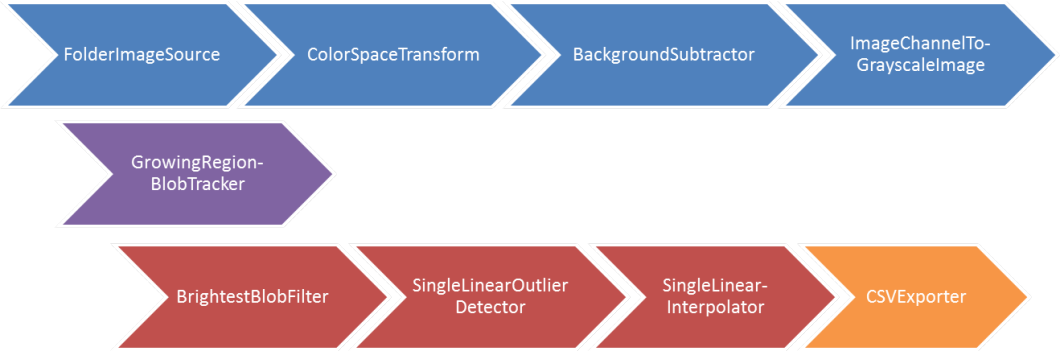

Figure 6.2: The pipeline used

This pipeline is a good general all-round pipeline for detecting single colored objects. It can be used everywhere where the object of interest is primarily described by one color like in this table-tennis scenario.

6.4 The results

6.4.1 Recorded videos

The randomly chosen frames can be divided into two groups, one containing all the frames where the ball was within the area on or closely around the table and moving in a way in which it was clearly recognisable as a ball, meaning its shape was more or less round. The other, much smaller group is composed of all the frames in which the ball was way outside the area which could be considered as playing field or was moving so fast that it appeared more as a line than as a round object. All random samples from the first group, which in a real world scenario would be the important ones, were tracked with a 100% accuracy, independent from framerate and resolution. The results of the second group contained false positives and negatives due to the fact that the ball in these frames did not actually look like the ball the system was looking for. Even tough the tracker itself was not able to detect the ball in some images, the following interpolation step was able to compensate for some of these cases. Due to the small data set, it is difficult do make quantitative statements regarding the overall performance of the pipeline depending on the resolution and framerate of the input footage. The tracking accuracy of the random subsets was never below 85% and in most cases at 100%. These numbers are however not very significant since the distribution of the frames from the second group was not the same in all the test sets. The ones with the highest portion of these frames where also the ones with the worst overall results. These results could potentially increased even more by a better post-processing stage which is able to better compensate for such frames. Since a higher framerate leads to fewer frames with severe motion blur, it can be beneficial to have a higher framerate as long as its increase does not decrease resolution to much.

6.4.2 Synthetic videos

The results for the three synthetic test scenes are as expected. The tracks first and second test scene were perfect or nearly perfect. The track for the third scene contained some false positives due to the fact that it contained moving objects which look very similar to the object of interest. These objects are hard to distinguish from each other and in certain situations, especially where the actual ball was moving very fast or was occluded, the tracker mistook one of the other balls in the scene for the right one. These kinds of problem could be solved by using a more sophisticated post-processing pipeline. The results of these three test scenes were compiled into one video for further evaluation [13].

7 Conclusions and Outlook

7.1 Conclusions

7.1.1 Successful approaches

During the time of production of this work, many approaches have been evaluated. This section highlights some of them which worked surprisingly well.

7.1.1.1 Linear color space transformation using PCA

The maybe most useful color transformation filter in the JMTF is the linear color space transformation with a transformation matrix resulting from a PCA. It produces very reliable results, is fast and easily configurable. It is in fact the filter with the highest potential for automatic configuration. Since the matrix used is calculated from image samples, it is sufficient to provide a small set of these samples to configure this node. When this node is used in a pipeline which primarily relies on it for image preprocessing, no further configuration of the entire pipeline might be necessary.

7.1.1.2 Background estimation

The background estimation used in JMTF differs from commonly used background subtracters because it does not use a fixed window of previous frames to remove the background in the current frame but uses all the previous frames to estimate how the background looks. This makes this method more reliable and less vulnerable to image glitches within a sequence.

7.1.2 Unsuccessful approaches

This section covers the approaches which did not turn out to be particularly helpful or did not even make it into the current version of the framework.

7.1.2.1 JavaCV

The first intention was to build this framework using JavaCV since it had already provided some of the functionalities which had to be implemented otherwise. After some experimentation, it turned out not to be an optimal solution. Not only was JavaCV difficult to set up on some operating systems, some parts of it, especially the memory management was either not documented at all or the documentation was written for a much older version. The experiments also showed no noteworthy performance difference but since this evaluation was not their purpose, this might be just coincidental. Lastly, it would have resulted in much less clear code structure when external libraries had been used for some essential parts of the functionality.

7.1.2.2 Blur

One of the first filters which were implemented was the convolution blur filter. The expectation was that the blur filter would improve the tracking results by removing some of the noise in the image. This turned out not to be the case. When the blurring had any effect at all, it decreased the tracking performance. Because there might be some situations where blurring the image is actually beneficial, the filter is kept in the framework.

7.1.2.3 Dynamic quantisation

An earlier design of the framework included the possibility to specify the quantisation on a JMTFImage. Since some evaluations showed that this would have increased the complexity of the code and decreased its performance while offering only minor functional benefits, the quantisation was fixed to 24bit per pixel. This is also the most common format cameras will produce images so there is no general information loss in doing so.

7.1.2.4 Direct video input

Multiple ways of reading frames directly from a video file have been evaluated. All of them turned out to be either outdated, not available for all platforms or not yet stable. So all the videos had to be broken up into image sequences before they could have been loaded by JMTF. The frame isolation was done using ffmpeg. The video from the camera had to be converted first to a different container format since ffmpeg had trouble reading directly from the mp4 container.

7.2 Future work

While at its current state offering a basic set of functionality for motion tracking, it is far from a state that would qualify as finished. As well as adding additional functionality, introducing processes to speed up a pipeline would be desirable.

7.2.1 Expansion

There are countless of potentially useful functionalities which could be added to this framework. The following ones are however the probably most important currently missing features.

7.2.1.1 Direct video input

In its current state, the JMTF is not capable of directly reading from video files. This is due to the absence of reliable, pure Java video libraries. The general video input capabilities of Java are not well supported. There are multiple solutions currently in development but none of them is stable enough for production use at the moment. Possible candidates for future consideration are xuggler and jcodec.

7.2.1.2 3D reconstruction

To be able to describe motion accurately, it is not sufficient to only consider two dimensions. It would therefore be desirable to use multiple cameras to combine two ore more two dimensional tracks into a three dimensional one. To be able to do this, the resulting coordinates from multiple pipelines could be combined together with the position of the cameras and their individual optical properties and used to calculate the three dimensional spacial position of the tracked object.

7.2.1.3 Interpolation

The interpolation functionalities offered by JMTF are rather basic. It is only possible to interpolate linearly. To represent motion more accurately, an interpolation polynomial of at least second order is needed. Quadratic or even cubic splines could be used to achieve a much more plausible trajectory.

7.2.1.4 Outlier detection

The outlier detection capabilities of JMTF are currently limited to a single point per frame and time. There is no way of detecting clusters of outliers. The detection also does only take one blob into account and does not work optimal for a tracking data set containing multiple blobs. The capability of detecting clusters of outliers and handle multiple blobs per tracking data set would greatly improve the reliability of the tracking results.

7.2.1.5 Correlation

With the JMTF in its current state, it is difficult to track objects which are not easily describable by their color. A description using a pattern is not possible at this stage. It would be beneficial to have a module to perform image correlation which would be able to provide such pattern matching capabilities.

7.2.1.6 Registration

JMTF is designed to operate using image data from fixed cameras. In certain situations, it might not be possible to get the camera fixed and simultaneously observe the whole area in which the object of interest is moving. An image registration step could solve this problem to map an image from a moving camera into a precomputed image which represents the whole area of interest.

7.2.1.7 Auto-configuration

Currently, setting up a JMTF pipeline requires a lot of manual adjustment of the configuration of the individual filters. There are multiple possibilities to automate certain of there processes. Ideally, one would manually label the object of interest in a small subset of frames and the system would learn from the required configuration to isolate this object. This could be done by automatically generate the input image for the image PCA.

7.2.2 Optimisation

Because the goal of this work was to build a framework for offline processes, the runtime performance was a lower priority during the implementation. The framework was however designed with the possibility to add some performance improvements later on.

7.2.2.1 Parallelism

All modules of the JMTF are completely single-threaded. This is due to the absence of multi-processor utilities like OpenMP. The internal frame structure would generate way to much overhead to parallelize every filter individually even tough most of them would allow it. The runtime performance could be improved however by parallelizing the entire pipeline or certain sections of it, since only very few filters depend on a continuous image stream.

7.2.2.2 ROI selection

Currently, all filters always process the entire image disregarding the position of the object of interest in the previous image. Therefore, a lot of time is spent on manipulating image regions which do not contain any information. By defining a region of interest (ROI) based on the results of the previous image before starting the pipeline for the next one, efficiency could be improved. The easiest way to do this would be to just cut out the region of interest and only process this partial image. Before evaluating the coordinates extracted from this image, an adjustment for the new point of origin has to be made.

Bibliography

[1] Official OpenCV Wiki. URL http://opencv.willowgarage.com/wiki/.

[2] Zananiri, E. JMyron Processing Library. URL http://www.silentlycrashing.net/p5/libs/jmyron/.

[3] Becker, J. Java Motion Tracking API on sourceforge. URL http://sourceforge.net/projects/javamotiontrack/.

[4] Wikipedia: HSL and HSV. URL http://en.wikipedia.org/wiki/HSL_and_HSV.

[5] Wikipedia: CIE 1931 color space. URL http://en.wikipedia.org/wiki/CIE_1931_color_space.

[6] Wikipedia: Principal component analysis. URL http://en.wikipedia.org/wiki/Principal_component_analysis.

[7] Roth, V. CS252 Biomedical image analysis: Histogram equilization (2012). URL http://informatik.unibas.ch/lehre/fs12/cs252/slides/bmia12_intensity.pdf.

[8] Cattin, P. CS252 Biomedical image analysis: Lloyd-Max Quantisation (2012). URL http://miac.unibas.ch/BIA/03-Sampling.html#(40).

[9] Cattin, P. Image Enhancement: Radial Undistortion (2012). URL http://miac.unibas.ch/SIP/pdf/SIP-05-Enhancement.pdf.

[10] Wikipedia: Decorator pattern. URL http://en.wikipedia.org/wiki/Decorator_pattern.

[11] GoPro Hero 2. URL http://gopro.com/hd-hero2-cameras/.

[12] Blender 3D content creation suite. URL http://www.blender.org/.

[13] Synthetic test scene evaluation. URL http://www.youtube.com/watch?v=wt99pxEST0c.

Appendix A: Examples of filtered images

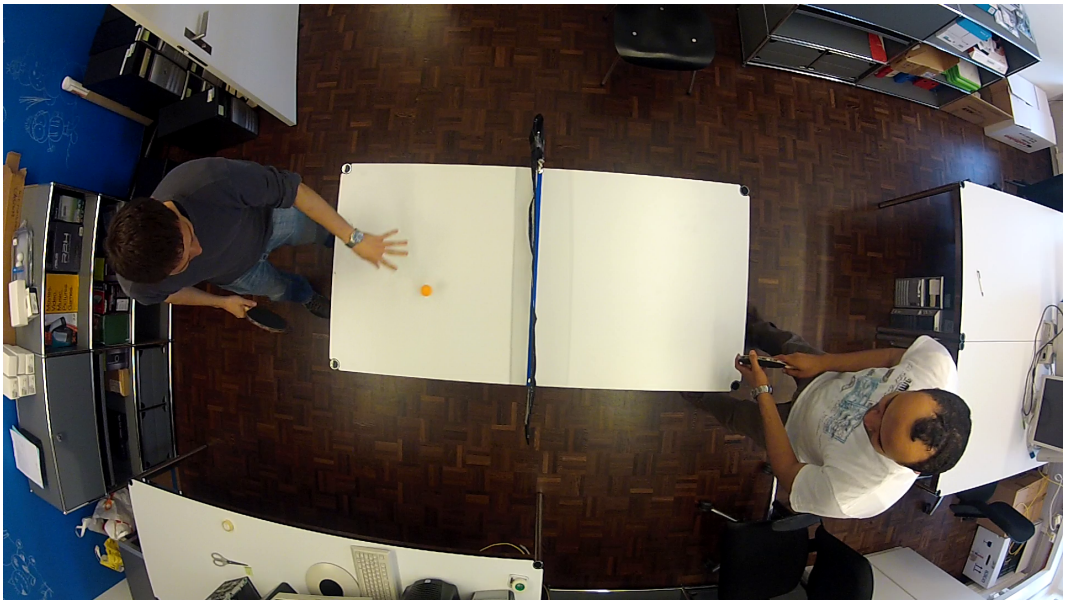

Figure 1: The first test image used to demonstrate the filters described in chapter 2

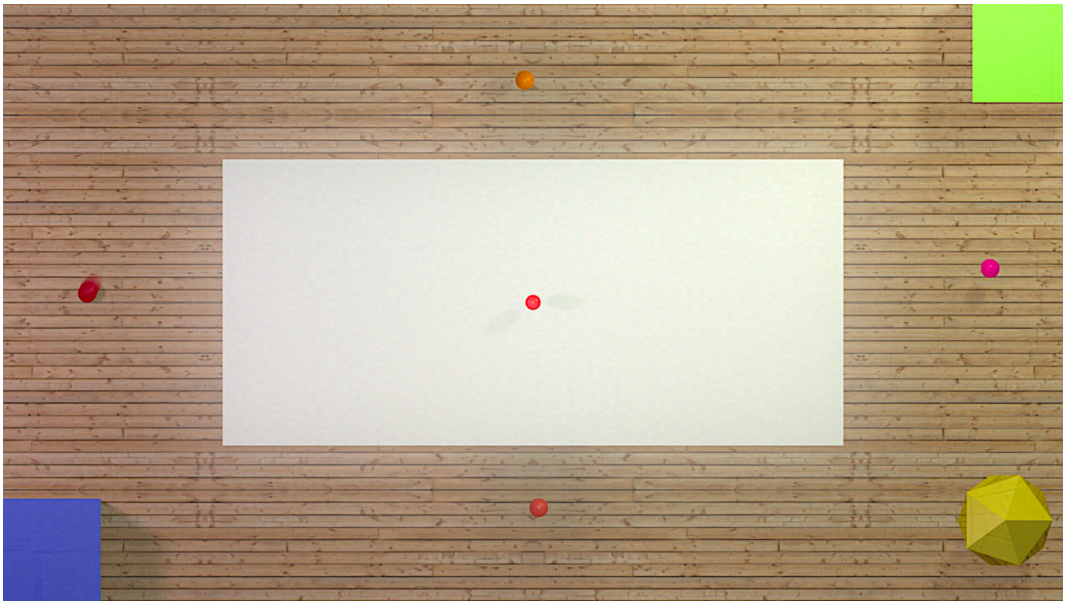

Figure 2: The second test image used to demonstrate the filters described in chapter 2

Figure 3: Hue cut-out filter applied to test image 1

Figure 4: Hue cut-out filter applied to test image 2

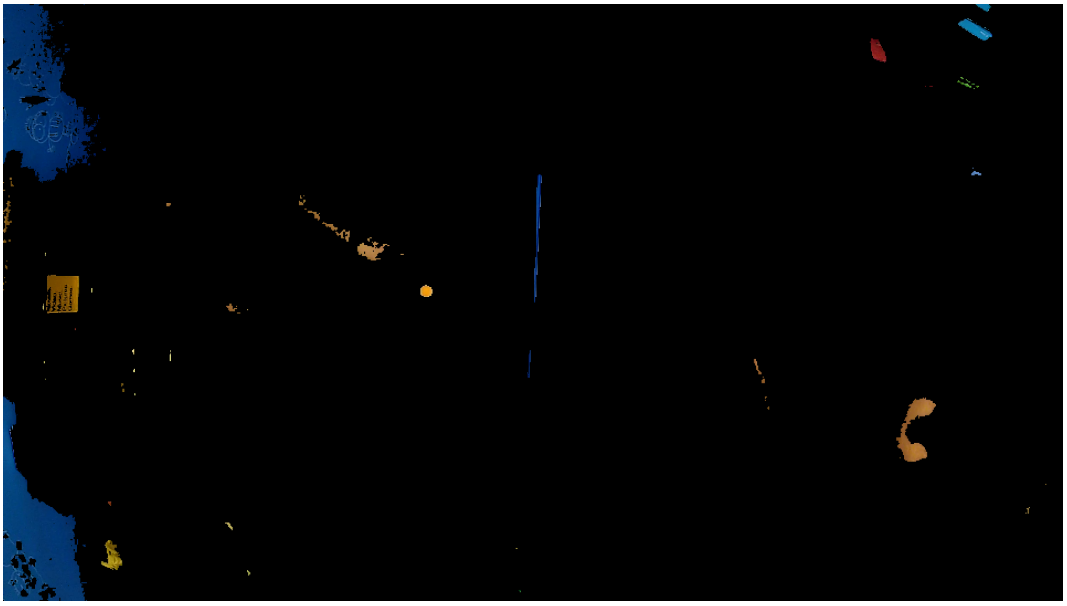

Figure 5: Gray cut-out filter applied to test image 1

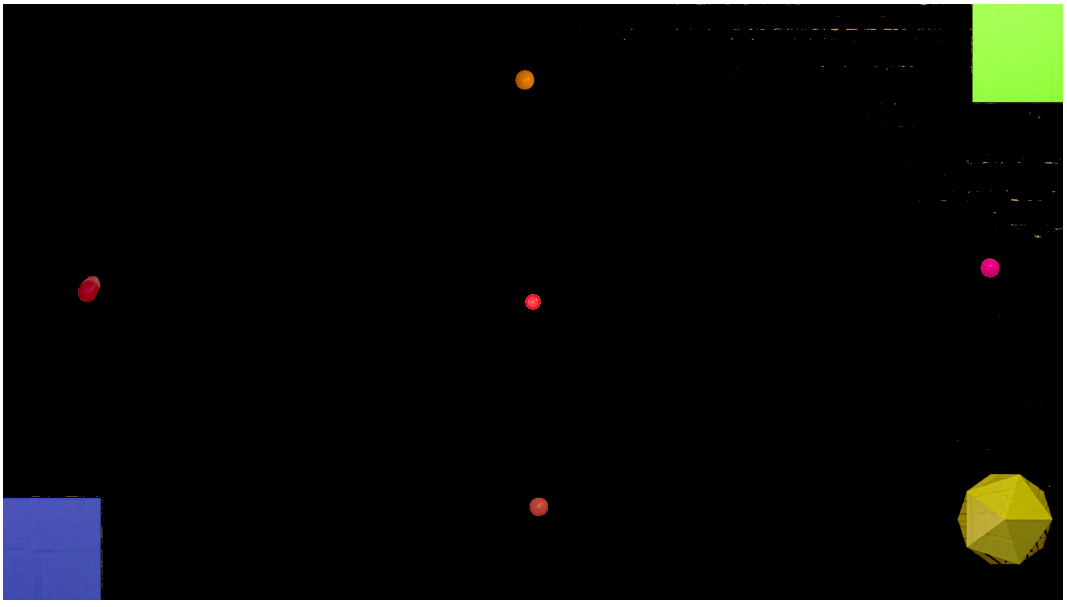

Figure 6: Gray cut-out filter applied to test image 2

Figure 7: Color distance cut-out filter applied to test image 1

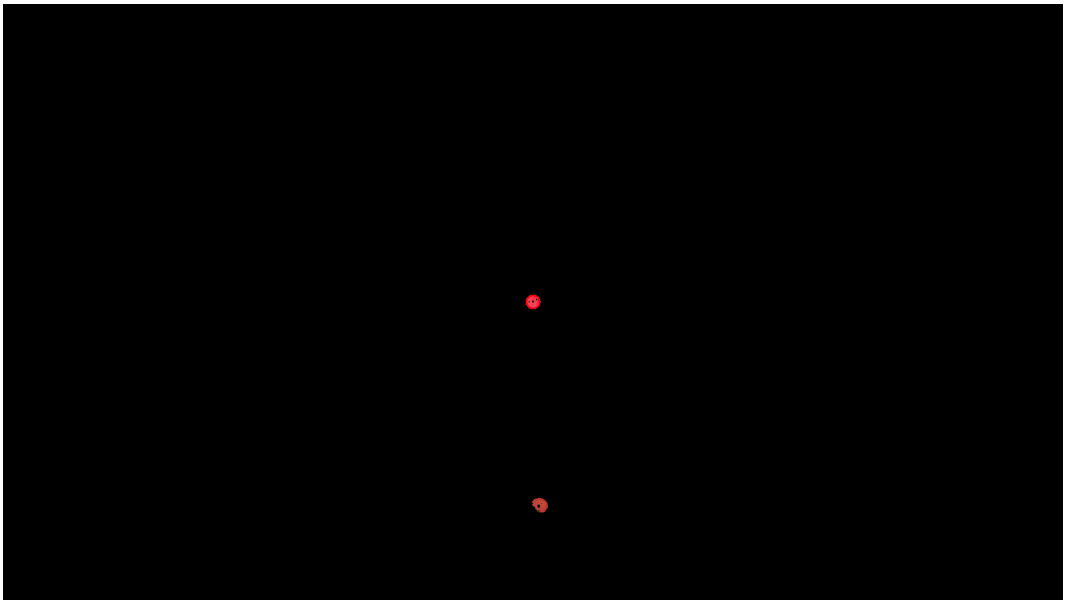

Figure 8: Color distance cut-out filter applied to test image 2

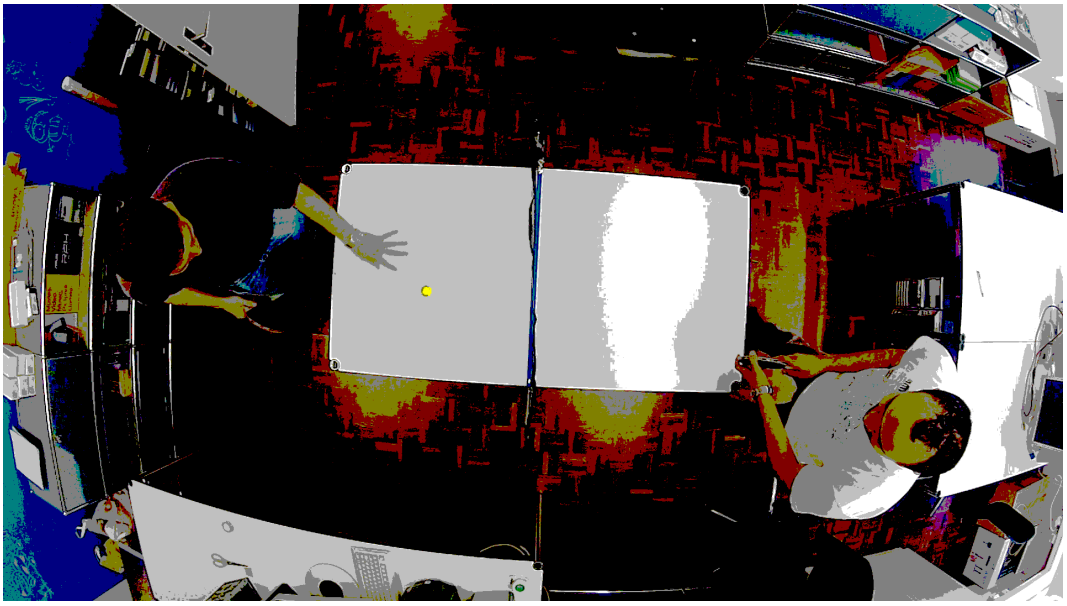

Figure 9: Color mapping filter applied to test image 1 to reduce color count to 16

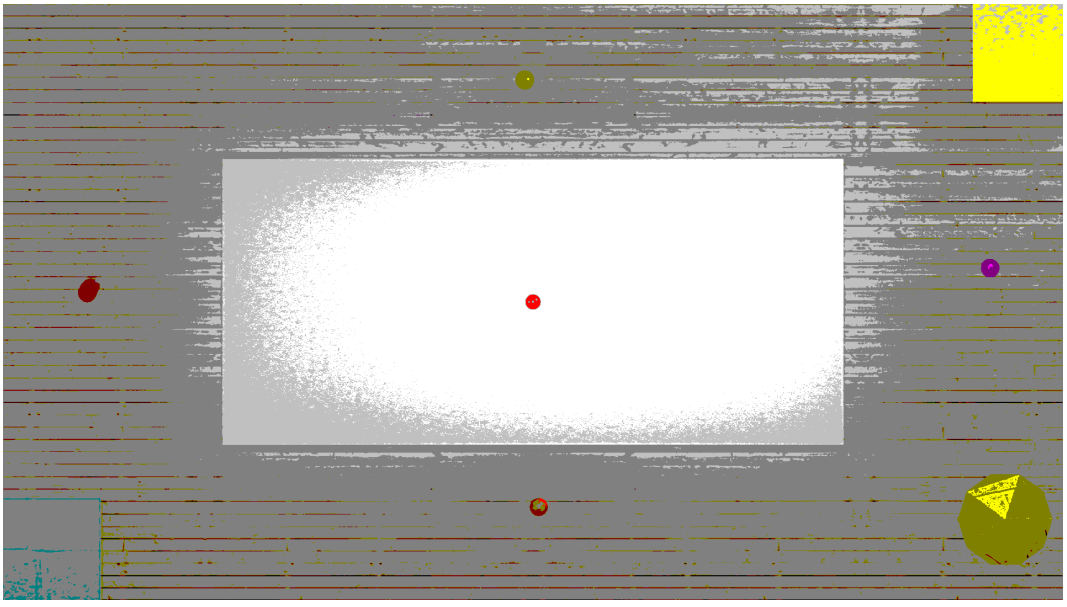

Figure 10: Color mapping filter applied to test image 2 to reduce color count to 16

Figure 11: Test image 1 converted to grayscale

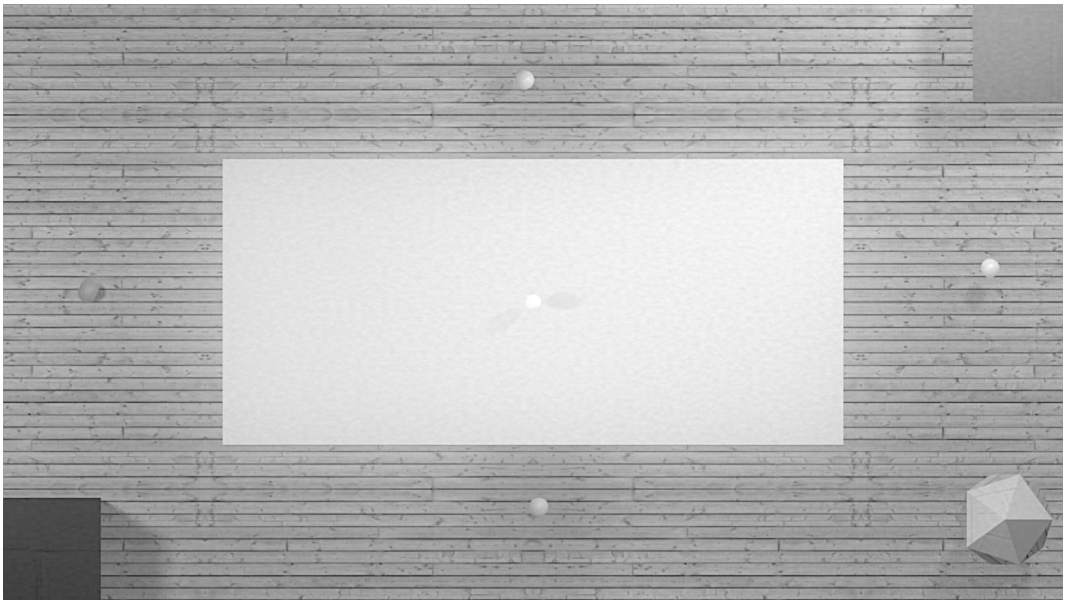

Figure 12: Test image 2 converted to grayscale

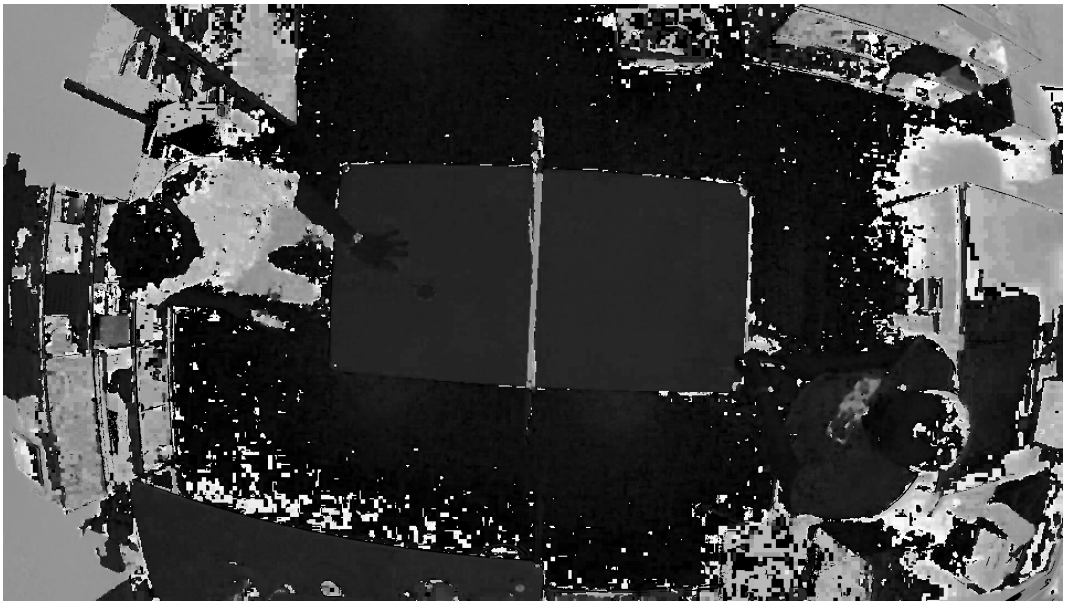

Figure 13: Image to hue filter applied to test image 1

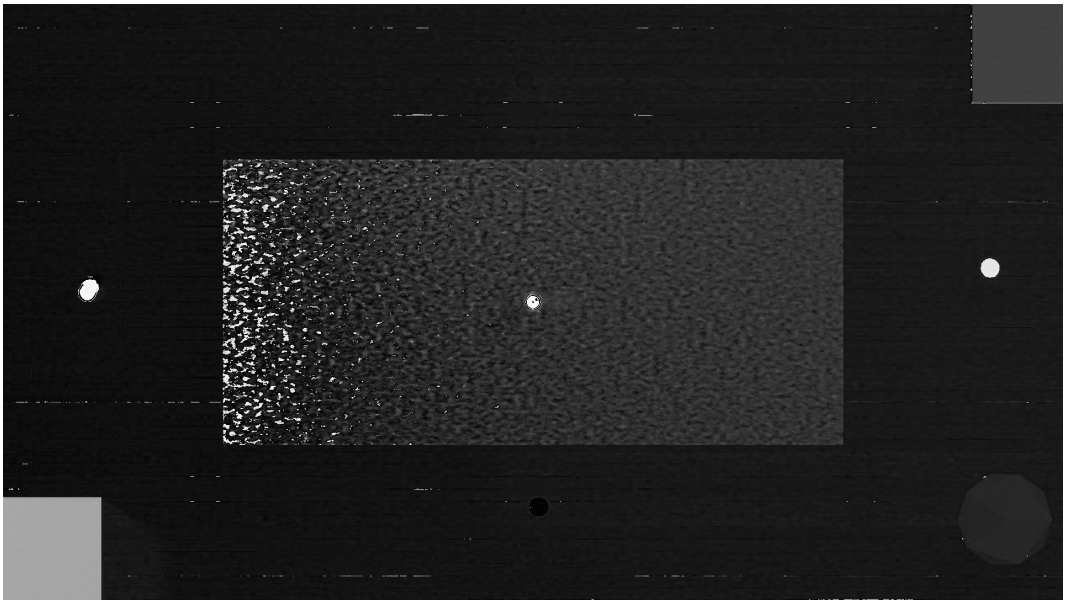

Figure 14: Image to hue filter applied to test image 2

Figure 15: Image to hue * saturation filter applied to test image 1

Figure 16: Image to hue * saturation filter applied to test image 2

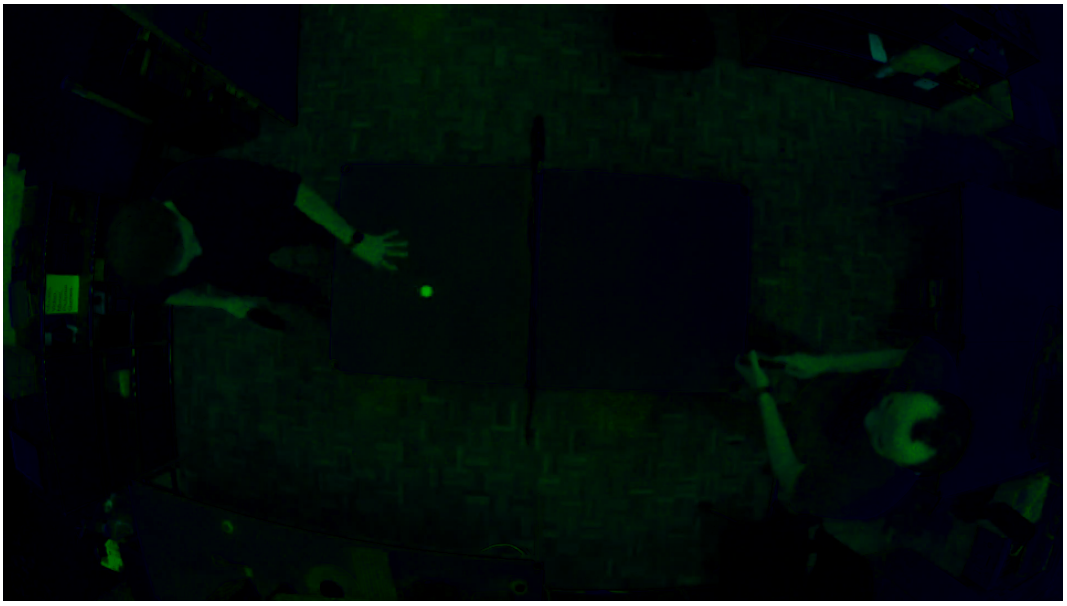

Figure 17: Color space transformation applied to test image 1

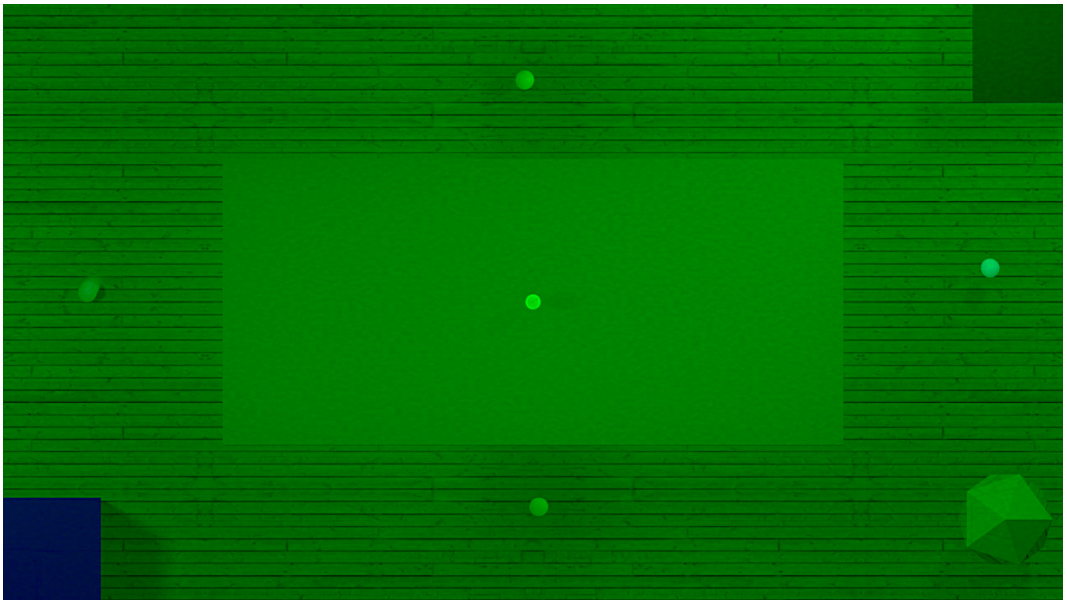

Figure 18: Color space transformation applied to test image 2

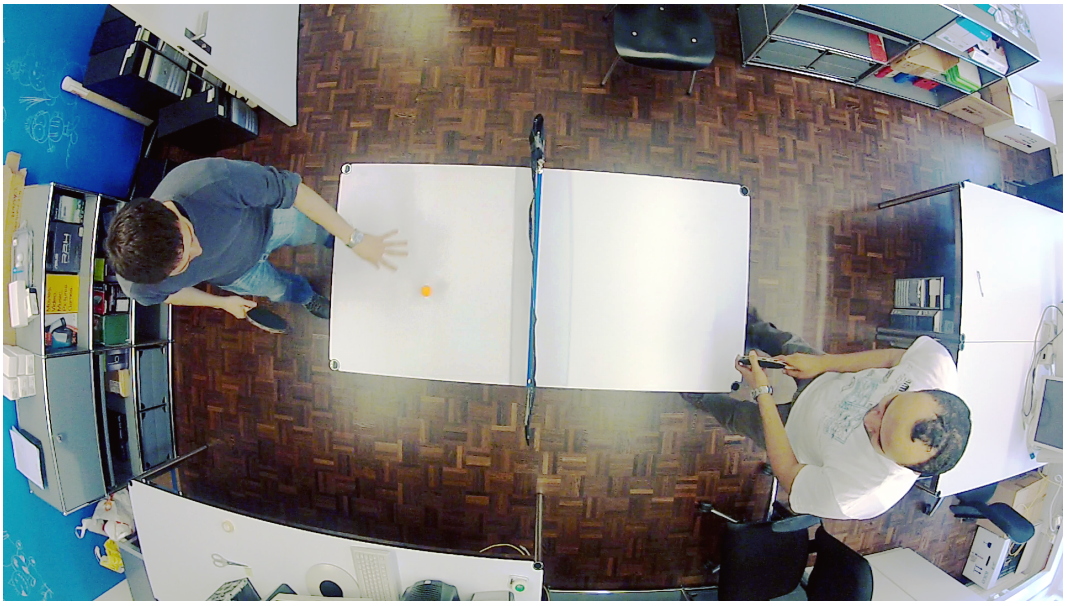

Figure 19: Histogram equalization applied to test image 1

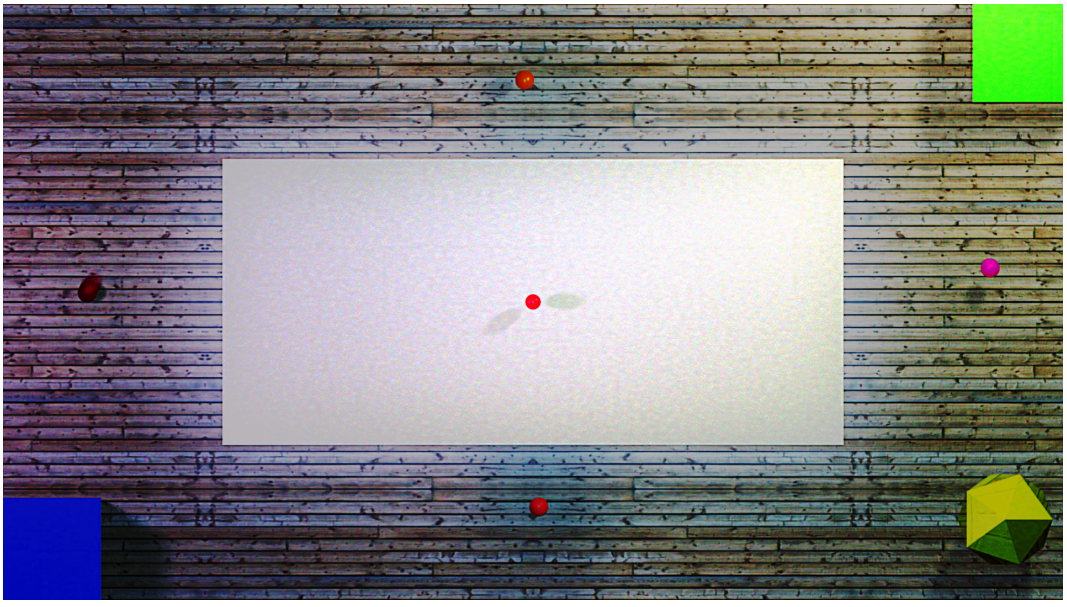

Figure 20: Histogram equalization applied to test image 2

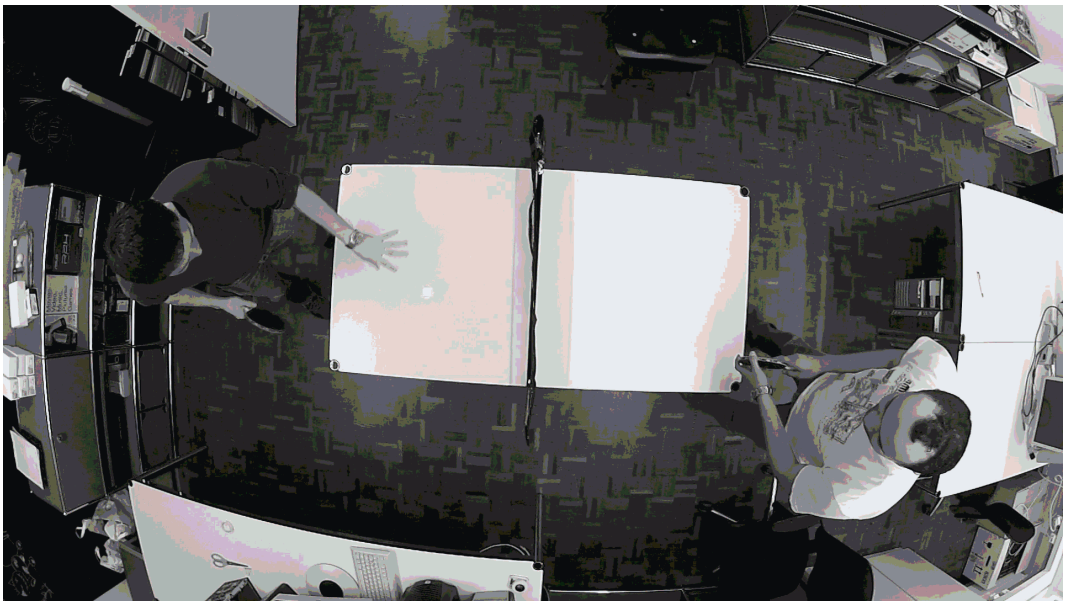

Figure 21: Lloyd-Max quantisation filter applied to test image 1 with 8 remaining colors per channel

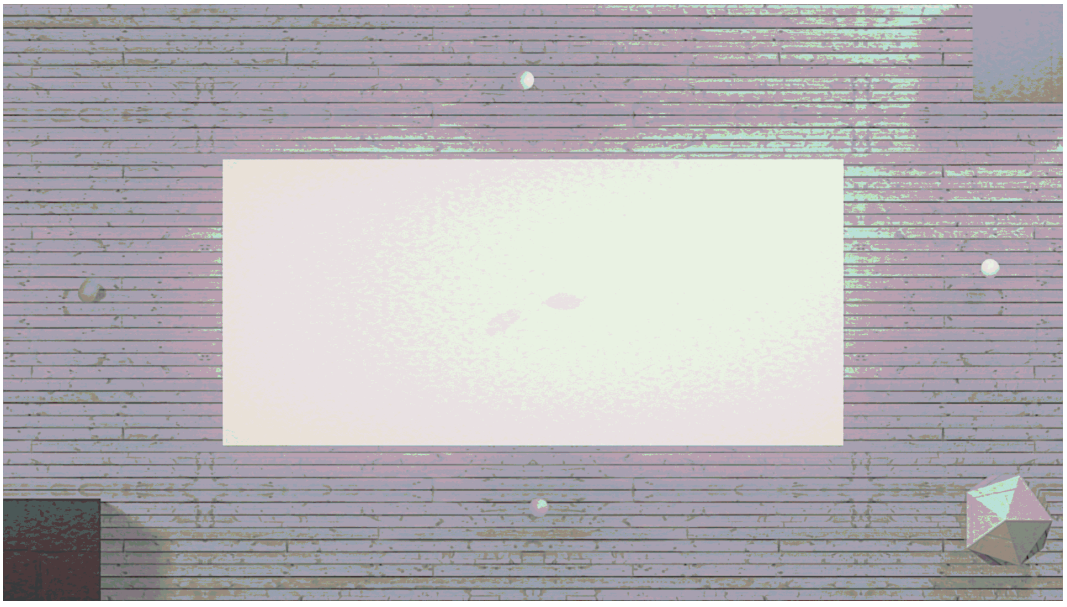

Figure 22: Lloyd-Max quantisation filter applied to test image 2 with 8 remaining colors per channel

Figure 23: Convolution blur applied to test image 1 with a radius of 20 pixels

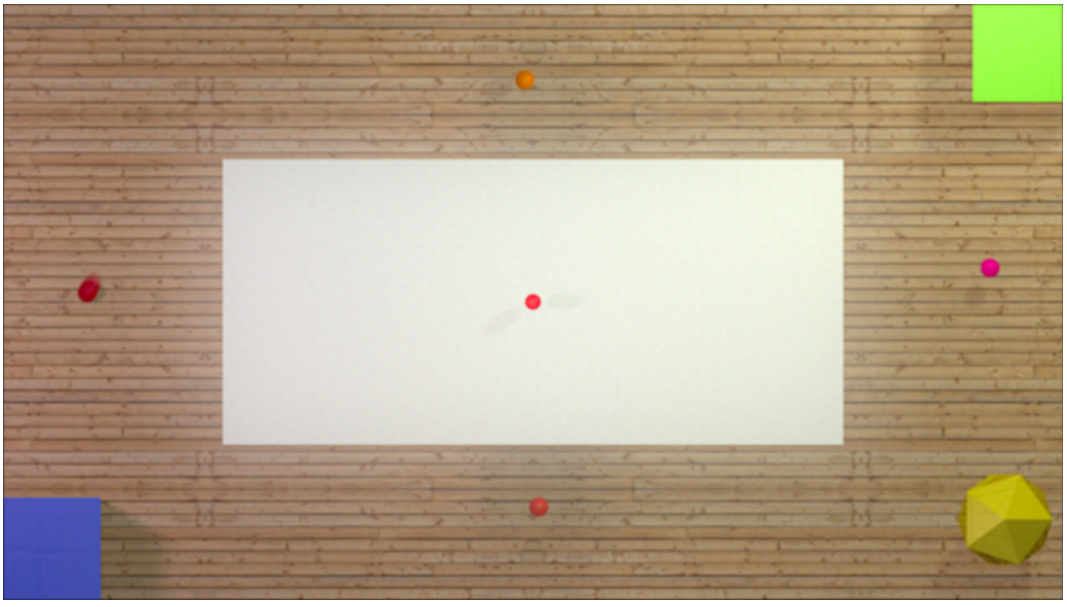

Figure 24: Convolution blur applied to test image 2 with a radius of 20 pixels

Figure 25: Median filter applied to test image 1 with a radius of 7 pixels

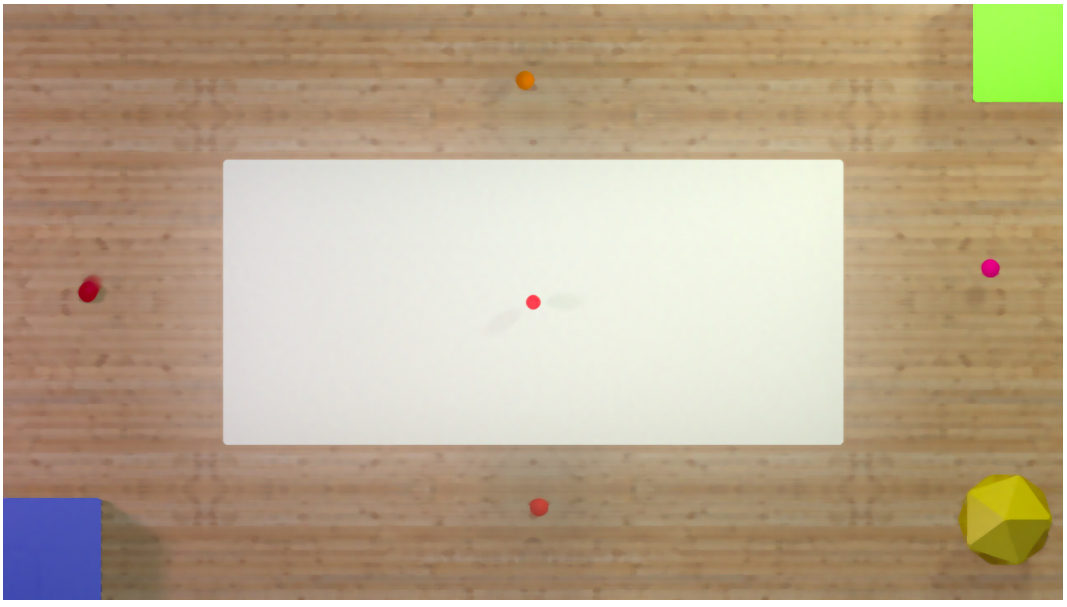

Figure 26: Median filter applied to test image 2 with a radius of 7 pixels

Appendix B: Examples of intermediate steps of the used pipeline

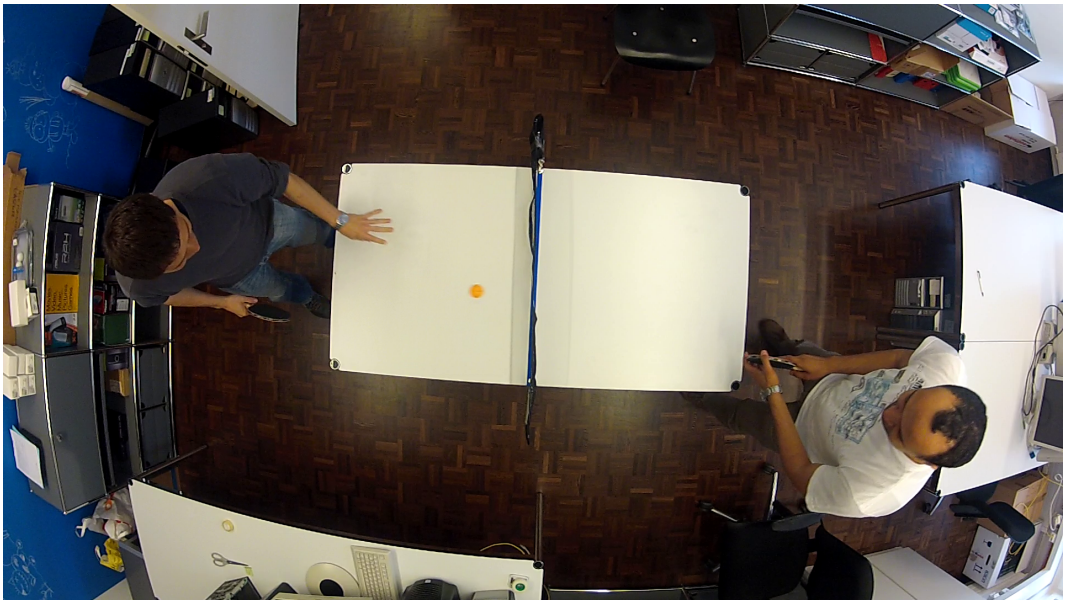

Figure 27: The input image from the recorded video

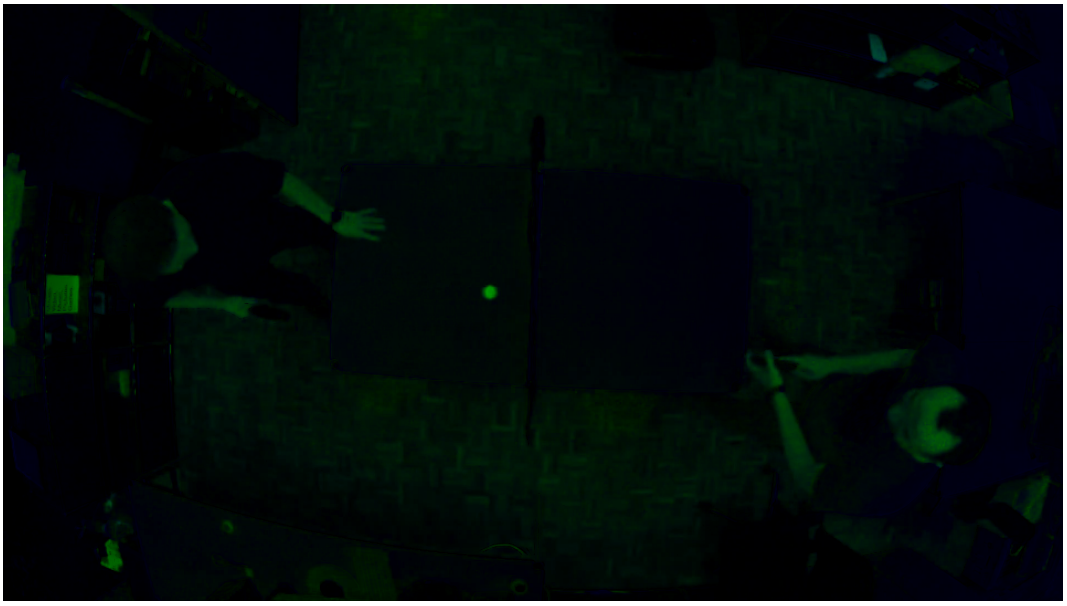

Figure 28: Color space transformation applied

Figure 29: The estimated background

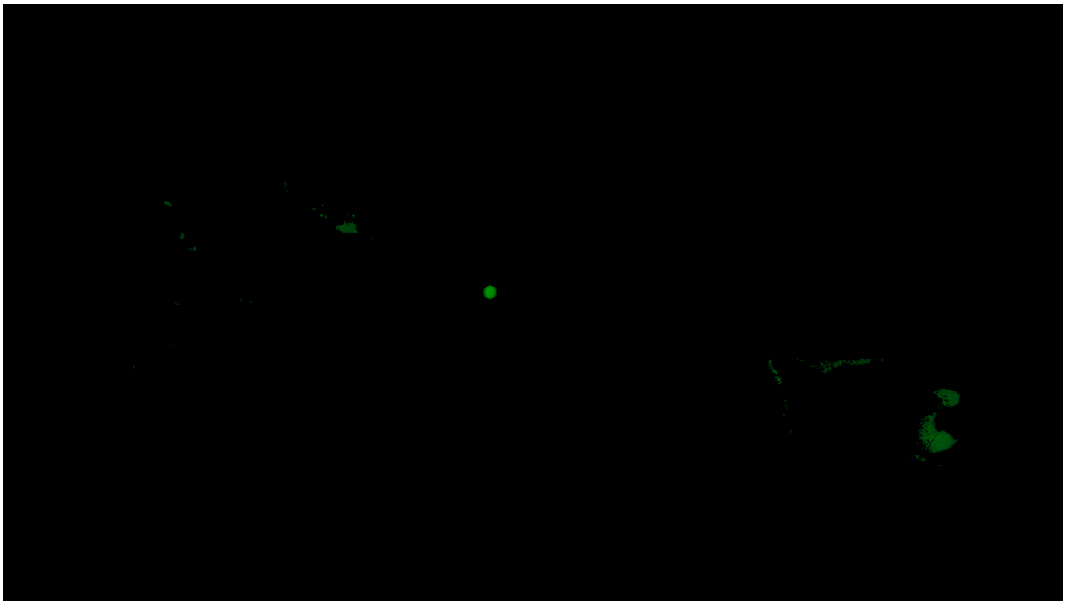

Figure 30: The estimated foreground

Figure 31: Extracted green color channel

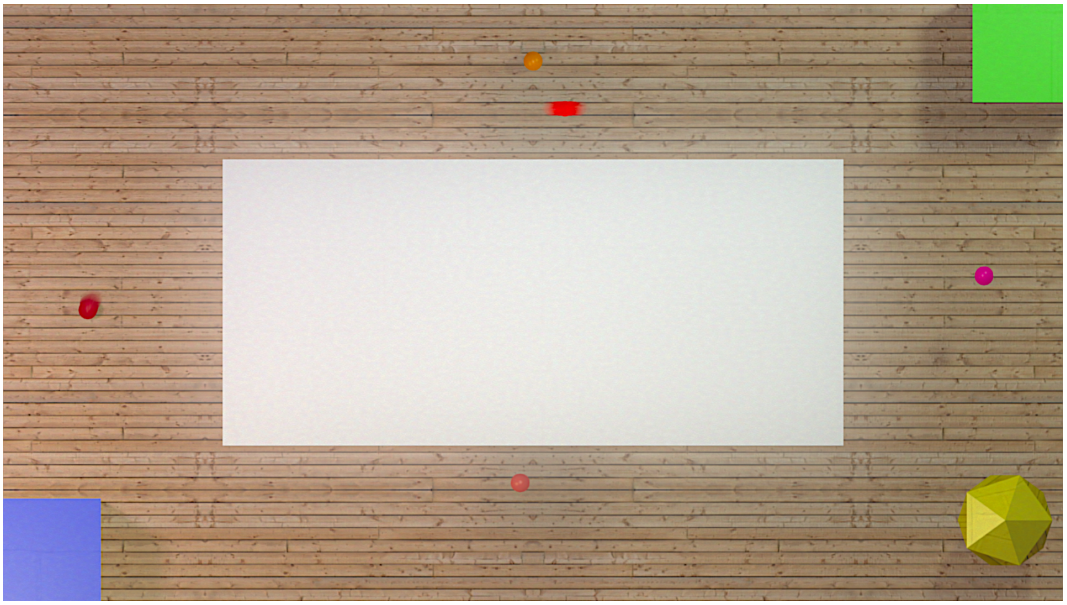

Figure 32: The input image from the synthetic video

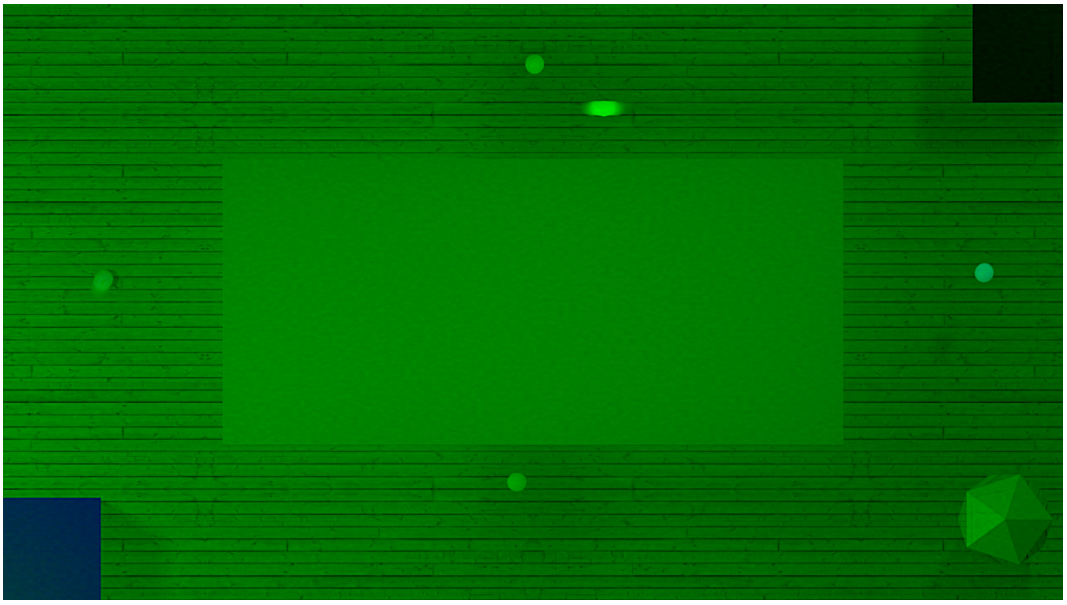

Figure 33: Color space transformation applied

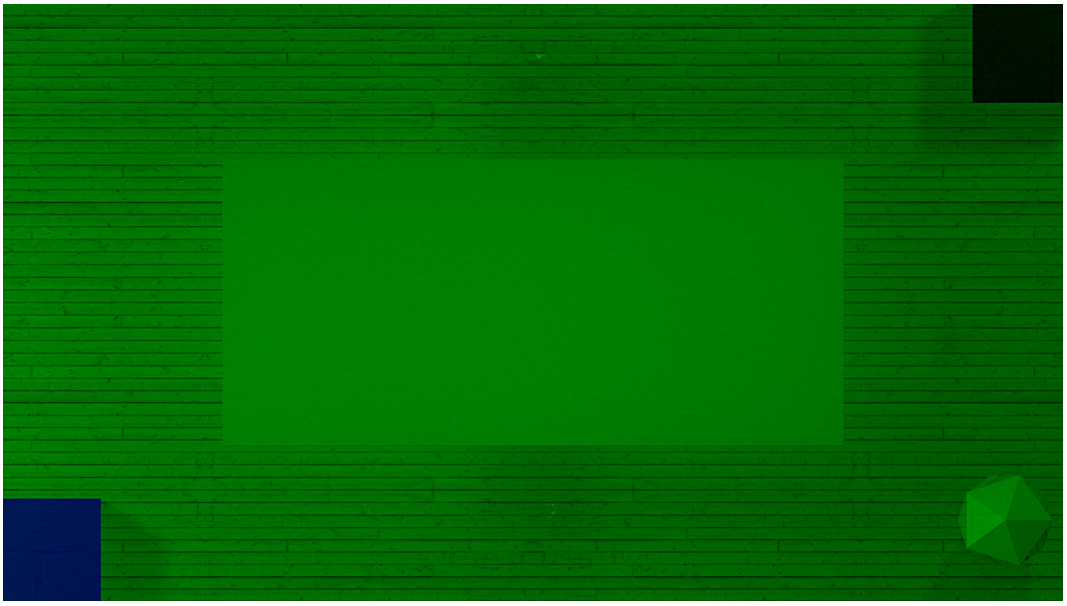

Figure 34: The estimated background

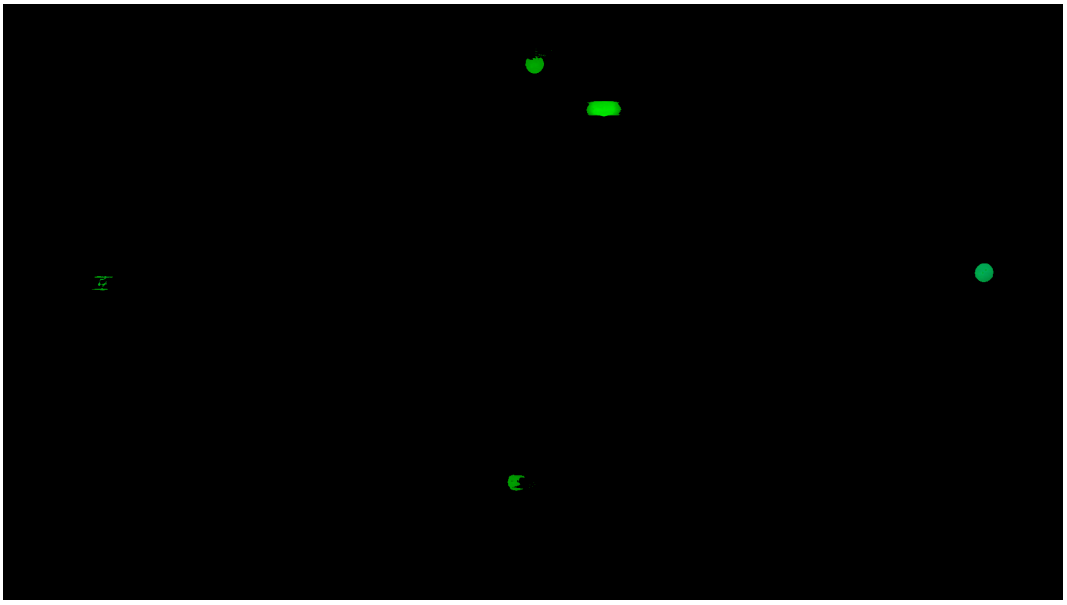

Figure 35: The estimated foreground

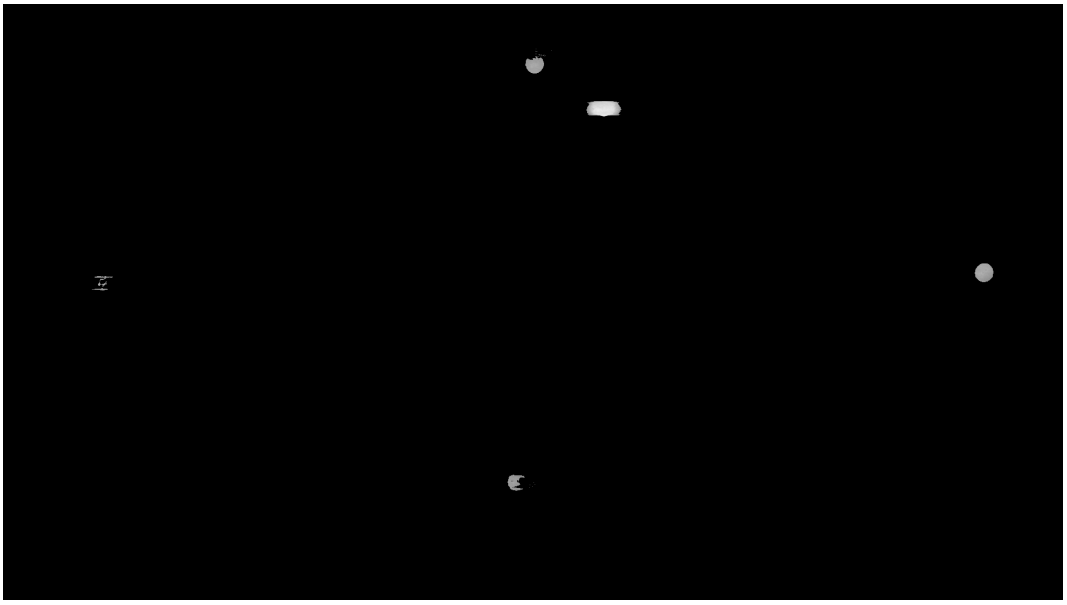

Figure 36: Extracted green color channel

Declaration of Authorship

I hereby declare that this thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the bibliography and specified in the text.
This thesis is not substantially the same as any that I have submitted or will be submitting for a degree or diploma or other qualification at this or any other University.

Basel, 04.07.2012

Luca Rossetto