TU UB
Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at
TU WIEN UB Universitätsbibliothek
The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

DIPLOMA THESIS

Object Detection and Flightpath Prediction

A Parallelized Approach Using a Graphics Processing Unit

Submitted at the
Faculty of Electrical Engineering and Information Technology,
Vienna University of Technology
in partial fulfillment of the requirements for the degree of
Diplom-Ingenieur (equals Master of Sciences)

under the supervision of

Em.O.Univ.Prof. Dipl.-Ing. Dr.techn. Dietmar Dietrich
Institute number: 384
Institute of Computer Technology

and

Univ.Ass. Dipl.-Ing. Martin Pongratz
Institute number: 384
Institute of Computer Technology

by

Maximilian Götzinger BSc
0826279
Krottenbachstraße 110
A-1190, Wien

May 30, 2015

Kurzfassung

Das vermehrte Verlangen nach individuellen Produkten steigert den Bedarf an flexibleren Produktionslinien. Ein zukunftsweisendes System hierfür könnte der „Wurftransport-Ansatz“ sein, bei dem sich Roboterarme die zu transportierenden Güter gegenseitig zuwerfen. Obwohl auf diesem Gebiet schon viel geforscht wurde, ergab sich bis dato noch keine völlig zufriedenstellende Lösung für dieses Transportsystem. Ein neuer, biologisch inspirierter Ansatz könnte die Antwort auf dieses Problem darstellen. Wenngleich dieses System bereits hinsichtlich seiner Genauigkeit untersucht wurde, so ist die Erforschung seiner Echtzeitfähigkeit noch ausständig. Diese Arbeit zeigt, dass die Detektion des Balls und die Vorhersage seiner Flugbahn schnell genug durchführbar sind, um das Kamerasystem bei einer Bildwiederholungsrate von 130 FPS arbeiten lassen zu können. Mit Hilfe einer NVidia GTX 560 Ti GPU ist es möglich gewesen, alle nötigen Berechnungen hierfür, in durchschnittlich, unter 7,7 ms durchzuführen. Für Bildwiederholungsraten von über 85 FPS wird jedoch ein Puffer benötigt, der selten auftretende Rechenzeiten von bis zu 11,7 ms kompensiert. Darüber hinaus zeigen die Resultate ebenso ein um das 3,46- bis 7,17-fach schnellere Ausführen des implementierten Programmes, wenn anstelle einer CPU eine GPU, für die nötigen Berechnungen, verwendet wird. Basierend auf diesen Resultaten können nun weitere Forschungen angestellt werden, um die Zuverlässigkeit und mögliche Einschränkungen des Systems zu untersuchen. Etwaige zukünftige Programmänderungen, im Zuge weiterer Forschungen, könnten zu längeren Ausführungszeiten führen. Jedoch ist es möglich, diese unter Verwendung einer aktuelleren GPU oder mit Hilfe einer Rechenschrittaufteilung auf verschiedene GPUs zu kompensieren.

Abstract

Advanced personalized customer needs and requirements lead to the demand for more flexible types of production lines. One trendsetting system apt to replace the old and static conveyor belt could be Transport-by-Throwing, which consists of robotic arms throwing objects to each other. Much research has been carried out in the field of robotic catching, but more needs to be done to meet the challenges involved. Despite many novel approaches, no fully satisfactory solution to catching a ball has been developed so far. A new approach that deals with this problem in a biologically-inspired way could be the answer. While it has already been proven that such a solution can lead to accurate results, its real-time constraints have not been examined. This thesis shows that computing ball detection and flightpath prediction can be done fast enough to capture the scene with a frame rate of 130 FPS. With the help of a NVidia GTX 560 Ti graphics processing unit, it was possible to execute all necessary calculations for the predictions in less than 7.7 ms on average. Because of maximum times of up to 11.7 ms, a small buffer is required for frame rates over 85 FPS. The results here demonstrate that the use of a GPU greatly accelerates the entire procedure and can lead to executions 3.5 to 7.2 times faster than on a CPU. Based on these results, further research can be carried out to examine the prediction system's reliability and limitations. Possible changes in the algorithm that lead to additional demand for computational power can be made when using a newer GPU or distributing the tasks on different GPUs.

Many thanks to myself ;)

Contents

1. Introduction 1
   1.1 Motivation 1
   1.2 Problem statement 3

2. Related Work and State of the Art 5
   2.1 Transport-by-Throwing 5
   2.2 Prediction of the object‘s trajectory 9
   2.3 Object detection and localization 17
   2.4 GPU programming 28

3. Setup, Procedure, and Concept 36
   3.1 Environment 36
   3.2 Tasks 37
   3.3 Preliminaries 39
   3.4 Subtracting the scenes background 41
   3.5 Canny Edge Detector 42
   3.6 Hough Circle Transformation 47
   3.7 RANSAC algorithm 55
   3.8 Obtaining the object’s 3D position 60
   3.9 Prediction 63

4. Results and Discussion 70
   4.1 Comparing different approaches implemented 70
   4.2 Testing worst-case execution times with artificially generated data 82
   4.3 Results of the entire procedure 84

5. Conclusion and Future Work 87
   5.1 Conclusion 87
   5.2 Future work 88

Literature 90

Internet references 94

Abbreviations

2D	Two-Dimensional
3D	Three-Dimensional
AFD	Adjacent Frame Difference
ALU	Arithmetic Logic Unit
BMP	Bitmap
CSV	Comma-Separated Values
CUDA	Compute Unified Device Architecture
FPGA	Field Programmable Gate Array
FPS	Frames per Second
GPU	Graphics Processing Unit
GPGPU	General-Purpose Computing on Graphics Processing Units
k-NN	k-Nearest Neighbors
MMX	Multi Media Extension
OpenCL	Open Computing Language
OpenCV	Open Source Computer Vision
RANSAC	Random Sample Consensus
SIMD	Single Instruction, Multiple Data
SIMT	Single-Instruction, Multiple-Thread
SFU	Special Function Unit
STDIO	Standard Input/Output

1. Introduction

In 1886, Carl Friedrich Benz presented the first automobile to the public [1]. At that time, the far-reaching impact of this invention was beyond imagination. Not only have the pace of human society and the interactions among people become faster and more flexible through eased mobility, the entire industry has been radically transformed by the advent of cars. In the early days of the automobile, each unit was handmade, one at a time. This would be downright unimaginable in today's automotive market. This pivotal change to the automotive industry came in 1913 when Henry Ford revolutionized his factory through assembly line work [2]. For the first time, it was possible that workers could work simultaneously on several cars. With his idea, Ford transformed the entire concept of manufacturing, making it possible to raise manufacturing efficiency. The invention of the assembly line led to the conveyor belt, which eventually became fully automated. However, the idea of using a continuous transport system is not new; it was first conceived by Oliver Evans who lived in the early 19th century [3]. Nowadays, most factories are equipped with these fully automated transport systems to speed up production. With this fast transportation and robot-assisted assembly, it is possible to produce annually about 83 millions cars [4]. Due to the transportation of passengers and goods as well as ease of communications today, people's life is becoming faster and more flexible. As a result of these changes, customer needs and requirements are becoming increasingly personalized in almost all product fields. Thus, manufacturers have to produce a lot of products that are similar, but not completely the same. The use of automatized fabrication processes leads to cheaper production costs and more precisely manufactured products. The challenge consists in finding a solution that allows enhancing the variabilty of products without losing the price advantage of mass production [Pon09, 1]. This could be summarized with the term *mass customization* [5] and describes the conflict between two requirements in manufacturing: automation and flexibility [GH03, 2]. Hence, a demand for other more flexible types of product lines already exists. As the economy becomes increasingly fast-paced and more flexible, there is a need for exchangeability in technical and manufacturing environments, such as a flexible and easily reconfigurable product line.

1.1 Motivation

Since not every personalized product passes through the same production stages, the currently used static conveyor belt no longer matches the application profile. There are

already approaches to solutions for a replaceable production line, including appliances such as transport carts and trollies that can carry goods autonomously from A to B. As a result, it has become possible to eliminate the necessity of navigating through the same production stages each time. The transport route is adaptable and the goods can be carried to the stations as needed. Drawbacks of such solutions include the slower velocity of locomotion as well as the lack of facilities for easy reconfiguration and rerouting. Such a modification would cause long production breaks, which prohibit a high-load output. [Pon09, 1]

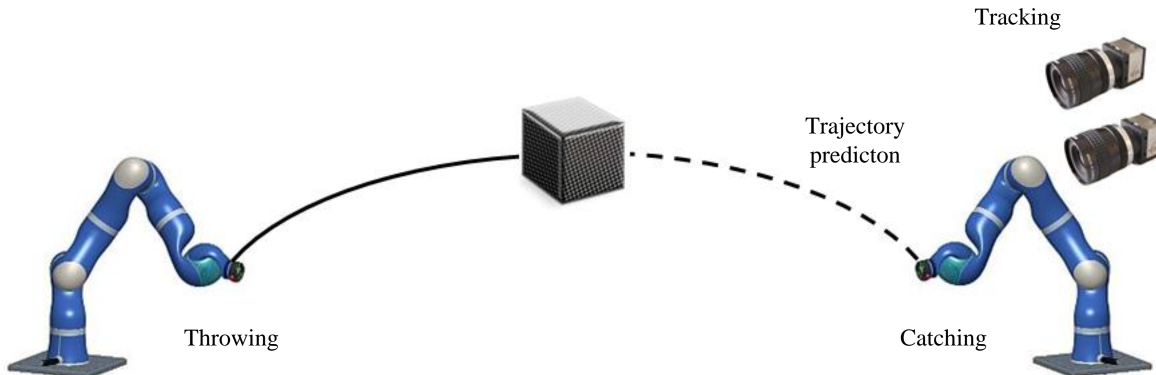

Figure 1: The Transport-by-Throwing approach deals with robot arms that throw and catch goods. The picture shows the transportation of an object from the left station to the right station. [PP12, 2]

Another solution for this rising demand for individual products is represented by the *Transport-by-Throwing* approach, which facilitates changing routes or even the whole production line. Through the frequent use of robots [6], it seems obvious to develop a system that would be based on robot arms that pass their payload: one throws it and another one catches it (Figure 1). This passing is being repeated until the transport good reaches its final destination in the production process, similar to communication networks with hop-by-hop routing [PP12, 2]. A system like this would be a lot sleeker than a conveyor belt that runs through the whole industrial hall. Furthermore, it would be extremely flexible because different objects could be conveyed to various stations. It would be much easier to adapt to different circumstances; which means that a reconfiguration of the transport route would not stop the whole production as is the case with transport trollies and conveyor belts [Pon09, 1]. Avoiding idle periods not only saves time and money, the transport system would be the fastest way to transport goods from A to B [BFP09]. The only requirements for the setup of the robot arms would be the communication between them. Hence, the robots would only have to exchange information about their location and the production steps done at their places.

In addition to the flexibility and lack of fixed transport routes, another big benefit of this solution would be the increased availability of the whole transport system. This is again similar to communication networks in that a drop-out of one station can be masked through bypassing the goods with the help of other devices [BFP09]. As shown in Figure 2 above, the drop-out of station B leads to a transport route A-D-E-C instead of A-B-C.

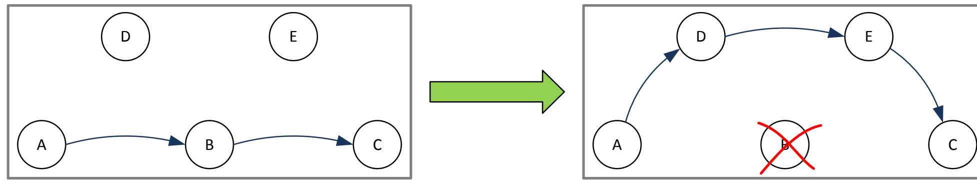

Figure 2: A drop-out of station B could be masked through bypassing it with the help of stations D and E.

It would be mistaken to assume that this solution is applicable only in manufacturing processes; indeed, it also could be used in supermarkets to stock goods anywhere in the store and place them on the shelves in the appropriate location. Another application could be the sorting of letters and parcels in a post office. The Transport-by-Throwing approach could be applied in almost every field where humans can throw things to each other to get them to the right place. [Pon09, 69]

1.2 Problem statement

Much research in this field has been carried out previously, including computer vision, object detection, and Transport-by-Throwing in general. Interest in such an approach to transportation should not be confined to universities wishing to demonstrate robot technology for academic purposes, but is of interest to industry as well [PP12, 1]. Despite many novel approaches, a fully satisfactory solution has not yet been developed. The approach that is in focus of this thesis shall solve the problem of catching in a biological way. For this purpose, predictions of trajectories shall not be forecasted by physical laws and calculations as is done in many other studies [FBH01][HS91][HS95][YLJ10]. The goal is to prognosticate the flightpath by benefitting from already gathered practical experiences [PKH10]. Think about a child who is learning to catch a ball. It does not contemplate about physics when trying to catch. A child only learns from its experiences: after numerous failed attempts, it will be able to catch an object in the right way. By analogy, many flightpaths from thrown objects were recorded and saved in a trajectory database. If an item flies to a catching robotic arm, the flight parameters such as distance, velocity, and flight altitude, shall be measured and compared with the already known trajectories in the database to enable a prediction about this actual throw.

In addition, it is very important to catch the object in a soft manner. That means that the robot arm follows the object's trajectory for a short while after grabbing it. That way of proceeding, the bulk of kinetic energy should be dissipated to catch the object safely and softly. On the other hand, hard catching would involve a robot arm waiting at the right interception position followed by grabbing its target without following its trajectory. The forces originating from this sudden deceleration can harm the object or destroy it altogether. [PP12, 3][PMB13]

Pongratz wrote in his diploma thesis [Pon09, 65] that it is possible to develop such a transport system, however, the real-time constraints have not been examined until now [PKH10]. Not only is accuracy important, the system also has to be fast enough to catch the

ball [Pon09, 52]. Image processing as well as comparison of the determined data with all the numerous database entries require a huge number of calculation steps. If calculating takes too much time, the object will pass the interception point before the robot arm moves in its direction. In this scenario, the thrown object would make impact with the ground and could be harmed by incidental forces. To manage the massive amount of computation in a short period of time, usage of a GPU ("Graphics Processing Unit") is an obvious option. Such hardware has a massively parallel architecture and can perform many computational steps very fast. For a GPU it is normal to have 4 to 120 *multiprocessors*, which have again between 16 and 48 cores (Figure 3). [CUD14, 191][OCL09, 51]

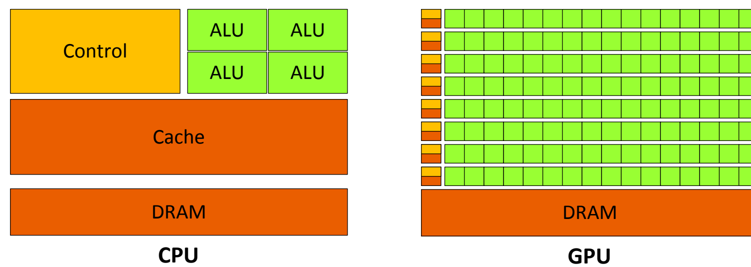

Figure 3: A CPU with four ALUs (Arithmetic Logic Units) and a GPU with eight multiprocessors are to be seen. Each of these multiprocessors has 16 cores. A program which runs on a GPU could be accelerated a lot through the massive parallelism of this device. Modified from [CUD14, 3]

That brings up some questions that will be answered in theframework of this diploma thesis:

- Does the use of a GPU reduce the time needed to compute a flight prediction?
- Is it possible to achieve a frame rate of 110 fps?
- Which developement platform should be used to implement the program?

Besides performance check and comparison between similar flight forecasting programs on a GPU as well as on a CPU, this thesis will deal with different implementations of different program parts.

2. Related Work and State of the Art

For almost 25 years, scientists have been trying to develop robots that can catch objects. Throughout this entire period, catching a ball was the challenge or benchmark for developing robots and testing other key technologies [BSW11]. Such a system can be assessed by its catching rate, which is the proportion of caught to thrown objects or conversely, by quoting the percentage of dropped objects, the so-called dropping rate. [Pon09, 14]

There has been a lot of research in the various fields of this project: apart from the subject of Transport-by-Throwing, also the use of a GPU for General Purpose Computation, object detection, and object recognition has been studied. For this thesis, some studies are more important than others, but in the following subchapters these themes will be introduced and briefly discussed. Furthermore, results of other research will be shown to provide an idea about the various strategies, changes, and trends in the last three decades.

2.1 Transport-by-Throwing

While there have been a large number of publications on Transport-by-Throwing [BWH10] [HS91][HS95][NIN97][FBH01][RA02][NI03][MHM04][SC07], it has only recently become a focus of scientific research. Proposed by Frank [PKH10] in 2010, this topic has gained momentum at the Vienna University of Technology.

In summary, a robot has to know how the object is moving in space and time to enable the arm to catch it. To meet this challenge, it is necessary to know how the object will move in order to correctly position the robot arm. The concept of this type of transport includes four different activities that have to be performed more or less consecutively: *Throwing*, *Tracking*, *Prediction*, and *Catching*. [PP12, 3]

2.1.1 Related work

Flightpath forecasting has enjoyed much attention long time before Transport-by-Throwing came into the focus of reseach. As aforementioned, the Transport-by-Throwing approach would be interesting for industrial purposes, but at this point in time, research is only done in academic fields [PP12][PMB13][PKH10][MPD14]. At the turn of the millennium, the situation was similar regarding the studies of Namiki and his colleagues [NNI99][NI03][NII03][INH04] from the University of Tokyo. These applications have not the aim to transport goods by

throwing and catching, they only should demonstrate the skills of robots [PP12, 4]. The system was equipped with a low-resolution black-and-white vision system, which worked at 1000 frames per second. Namiki's team constructed a three-finger hand to grab objects, and they implemented a special algorithm for tracking these items and holding them in the center of the image. This algorithm was tailored for this application and, therefore, is not comparable to other algorithms that are made for tracking objects [INI96]. While [NII03] enabled the robot hand to catch foam balls and foam cylinders, [SNI05] was about dribbling a ball on a plane ground. Although not all of these papers directly pertain to the Transport-by-Throwing approach, they demonstrate the skills of robots and connect the field of computer vision with that of mechanized interacting. Therefore, they constitute the basis for further researches in this sector like those summarized below.

In 1995, Hong and Slotine from the Massachusetts Institute of Technology published a paper [HS95] about the topic of robotic catching. The goal of this research was the implementation and improvement of the *Hand-Eye Coordination*. In this connection, a robotic arm should catch a ball, which was thrown over a distance of about 1.5 to 2.5 meters. The system they created was based on a simple parabolic function, into which the ball's trajectory was fitted. Therefore, the estimation unit needed two points from the trajectory of the actual throw to be able to evaluate the movement of the object and to predict the further pathway. With less than two of these points, it would not be possible to calculate the vectors of velocity and direction [HS91, 383]. The prediction unit received this important information about the trajectory from a two-camera system, which consisted of CCD (Charge-Coupled Device) video cameras with a baseline distance of 0.8 m between them. The researchers did not use a throwing device for tossing the specially painted ball, but a person threw it with an under-hand toss. The ball flew approximately for half a second in the air and was localized through color BLOB detection. For this purpose a simple BLOB detector vision board [Wri93, 3] was used, which compared the input frames from the two color cameras with a color histogram to display the location of the specially painted ball. The success rate of catching the balls ranged between 70 and 80 percent, and the longest run was 14 consecutive successful catches. In addition, this was the first instance of robotic catching in a soft way. Catching the ball was only possible if the prediction resulted in a deviation from the inception point not larger than 1.5 cm and a variance of timing less than 5 ms. Their aim was not to develop a perfectly working complete system, but to establish a basis for further research. [HS95, 1]

Additionally, they provided with their research another important insight and proposed that the unsuccessful or abortive attempts at catching were caused by noisy data. Insufficient compensation of time delays as well as bad exposure leads to such noise in the data. In this context, they wrote about the problem of alternating lighting conditions during the flight of a ball: When the ball gets closer to the light source, the reflections of light on the ball are considerably larger than when it is further away. [HS95, 8]

In 2001, Frese, Bäuml, and other colleagues of the Institute of Robotics and Mechatronics at the German Aerospace Center published a paper [FBH01] on their solution of a catching robot arm. The project's goal was to develop a well working catching system made out of off-the-shelf hardware as all of the previous studies [And98][HS91][HS95] had made use of special

hardware, which is usually very expensive. In order to create a low-cost system, they used two standard PAL video cameras in a stereo vision configuration. They were vertically positioned with a baseline distance of 1 meter and enabled a localization of the thrown ball with a precision of about 3 cm. The captured images were sent to two standard frame-grabber cards, which digitalized and forwarded them to a computer with a 300 MHz Pentium II processor. The captured frame was compared with the background of the scene to detect the ball. If the difference between a pixel of the recorded frame and that of the background is larger than a previously set threshold, then it can be identified as belonging to the ball. A Kalman filter was used to track the ball and predict the further flightpath taking into account the air drag. The work was presented at a fair in Hannover and various guests were invited to throw balls to the catching device. Of approximately 100 throws, two thirds of these balls were caught. The main reason every third ball was dropped was due to the coverage of the cameras as throws that partially went beyond the captured scene led to a late prediction and caused many faults. Three problems had to be dealt with using off-the-shelf hardware. Firstly, they had to handle interlaced frames from the PAL cameras. Secondly, the implemented program had to use the MMX (Multi Media Extension) instruction set to make the processor more efficient by working on more pixels simultaneously. Thirdly, the limited memory bandwidth of the CPU led to huge performance losses. To circumvent this drawback, comparison was only done with a small subframe, called Region of Interest, which constituted 2 and 40 percent of the entire image. [FBH01, 1623ff]

Almost one decade later, in 2010, Bäuml presented another paper [BWH10] with a similar setup as his previous study [FBH01]. The differences were the higher computational power provided by a 32 CPU big cluster as well as the use of a robotic arm coupled with a hand of four fingers. The main challenge was to grab the flying object in the right moment with the claw and preserve the DLR-Hand-II [BGL01] from getting harmed by the impact forces of the throw. In this context, there were three different catching modes: soft, latest, and cool. With the reuse of the Kalman filter, it was possible to achieve a success rate of over 80 percent for the soft and latest mode. [BWH10, 2592ff]

The last point in this subchapter is the research [PP12] carried out by the KOROS Initiative at the Vienna University of Technology and constitutes the basis of this work. A KUKA LWR 4 robot arm is available for testing different approaches. For studies in this field, it is of particular interest to use state-of-the-art equipment to ensure a cheap solution that is affordable for industry. Furthermore, a solution involving a soft catching strategy is at the center of attention to minimize the forces occurring when catching the object [PMB13]. Again, it is a tennis ball that will be fired by a coil-based throwing device that stands about 2.5 meters away from the catching robotic arm. A stereo vision-based detection shall create a prediction as accurate as possible for the ball traveling at approximately 5 m/s. The bio-inspired approach, which implies a comparison with recorded reference trajectories of thrown objects, enables such a manner of transportation and considers these constraints. This solution is the base of this master's thesis and will be explained more detailed in Subchapter 2.2.7. [PP12, 2, 5]

2.1.2 Throwing, catching, working area, and interception point

Obviously, before an object can be caught, it has to be thrown. The catching device must, therefore, know the trajectory of the thrown object and the best interception point. To create a forecast of the flightpath, it is important to recognize and locate the object. Different techniques to obtain a prediction of the trajectory will be shown in the following Section 2.2.

While throwing will not be part of this thesis, it will still be briefly introduced to help understand the topic as a whole. During its launch, the object will be accelerated until it has gathered enough speed to negotiate the way from the sender to the receiver [PP12, 3]. In addition to the speed with its underlying acceleration, the throwing direction, too, must meet certain requirements. When looking at the xy-plane (left side of Figure 4), the operation area of such a robot arm looks like a circle with the catching device placed in the middle. When looking at the yz-plane (right side of Figure 4), the area where the robot arm can interact looks like some sort of circle. Considering the 3D view, this leads to a sphere-like working zone where the catching device can grab the object. Furthermore, it is important to clarify that this zone does not necessarily have to look exactly like that; it solely depends on the robots construction and movement. Not every robot has to have a spherical leeway of movement! The shape of the area as shown in Figure 4 shall only serve for the better understanding of the following explanations.

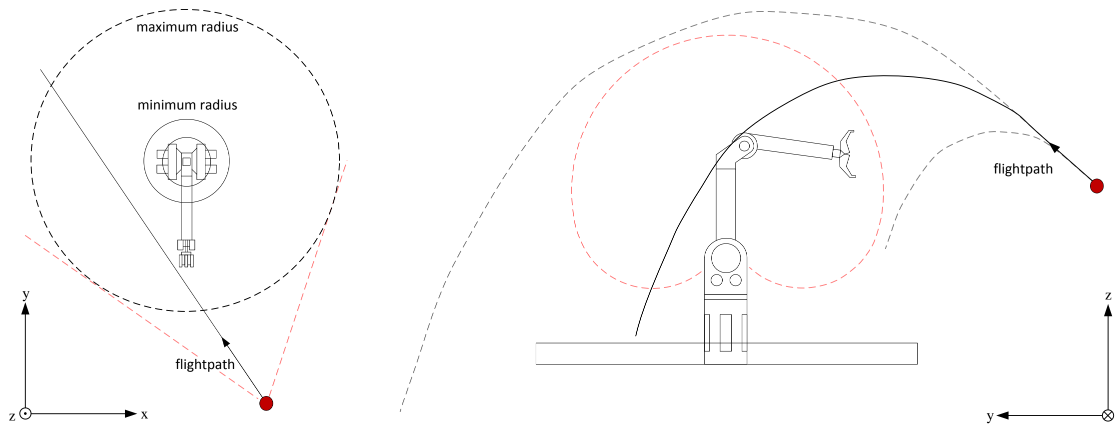

Figure 4: On the left side is the operation area of a robot arm viewed on the xy-plane, on the right side on the yz-plane. The ball can only be caught when it flies through the robot's working area. Modified from [HS95, 5]

The inception point where the thrown object touches the catching device must be in the working area of the robot. If it is beyond the reachability of the catching unit, the robot will not be able to catch it. That raises the question of the correct direction, in which the object should be thrown. Evidently, the flightpath must pass through the section between the two red lines on the xy-plane (Figure 4, left). Additionally, there is one fact to consider regarding the soft catching strategy. If the flightpath runs close to the borders of the robot's working area, the object's trajectory is likely to be traced less accurately. Additional forces that will come with such redirection will impact on the object and have to be taken into account as well [PP12, 3].

Flight altitude and distance are also very important for transportation and direction (Figure 4, right). As previously mentioned, the initial acceleration has to be set to a value high enough to provide a velocity, needed to traverse the desired range considering the set throwing angle. To catch the ball is theoretically possible when the flightpath runs through the operating zone of the catching device [HS91, 382]. On the other hand, acceleration and velocity must not be so high that the thrown object overflies the catching zone or possibly even damages the catching robot or the object itself. They might also get damaged when the thrown object weighs too much to be grabbed safely by the robot.

2.2 Prediction of the object‘ s trajectory

Predicting the future was always a desire and a great challenge for humans [Pon09, 11], be it the weather forecast [7], which interests the general public, be it rocket ballistics, which is of military interest. A prominent example of the second case is the Iron Dome [8], which is used in the Israeli military defense system. If a missile flies in Israel’s direction, this guard system will track the flightpath of the launched rocket to predict its further flight and destroy it with own rockets. If the prediction of the enemy’s missile projectile is accurate, it will be destroyed by the Iron Dome, hereby eliminating danger for the country. If not, the rocket will plunge to the ground and hit something or someone.

To successfully catch a thrown object, it is necessary to have a system that estimates the object’s trajectory. This prediction unit has to “know” the movement of this object, which means that its trajectory must be observed [BFP09]. To understand the ball’s movement in space and time, a 3D view of the surrounding area is indispensable, therefore, most of the projects handle this challenge with video-based systems. There are solutions with single-camera- [BFK08], two-camera- [PKH10][INI96][NI05][INH04], and multiple-camera-systems [BSW11].

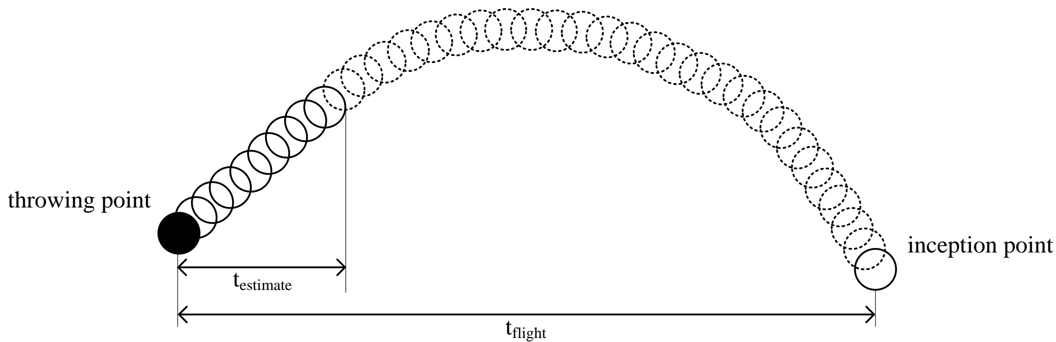

Figure 5: The ball is flying from left to right and after a few captured positions of it (done after $t_{estimate}$), the further flightpath can be estimated.

In the majority of researches, the basis for obtaining a prediction of the ball’s trajectory is the same. The ball will be localized through a stereo vision system to determine its movement in the 3D space [SPV05][BWH10, 1]. After the time $t_{estimate}$ all necessary information of the ball’s movement is known (Figure 5) and a first forecast for its further flightpath will be

made [HS91, 383]. The duration $t_{estimate}$ depends on the method of predicting and the whole setup of the vision system. Accuracy of the prediction, too, depends on these two facts and determines whether the catching will be successful.

For reasons of simplification, all works so far have only dealt with highly symmetrical (point-symmetrical or axial-symmetrical) objects [HS91][HS95][FBM08][BFK08][FMS09][BFP09][PKH10]. Subchapter 2.3 describes different kinds of object detection and shows that it is much easier to detect symmetrical objects such as a tennis ball or a cylinder. When complexity rises, requirements for the object detection unit will also increase [PP12, 4] and necessitate a higher development effort as well as a longer computation period [PKH10]. Therefore, this work deals with the forecasting of the trajectory of a thrown tennis ball to avoid additional difficulties.

2.2.1 The importance of timing

The term "catching" describes the process of controlled decelerating of a thrown object by the catching device, no matter the manner of catching: soft or hard. Whether the object is fixed after touching, lands in a basket, or is only held by friction is irrelevant here and depends on the construction of the robot. [PP12, 3]

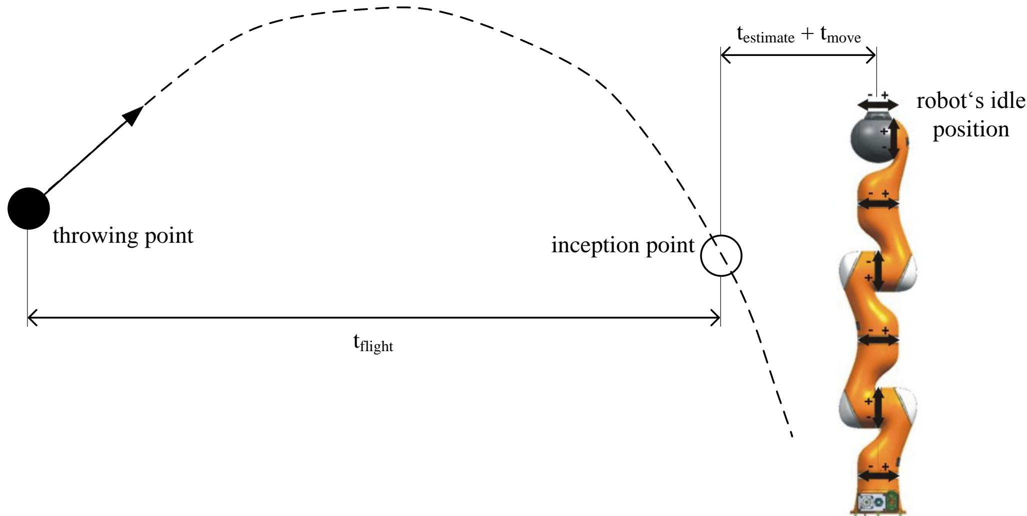

Figure 6: The ball takes the time t_{flight} in order to traverse the distance from the throwing device to the inception point. To move the catching device of the robot arm to the same point, two things are necessary: estimating the location of the inception point and moving the robot to this position, which need $t_{estimate}$ respectively t_{move}.

The working area of the catching device depends also on the swiftness of the arm's movement [HS91, 381] and its starting point as well as on the moment when the prediction of the flight appears. Catching will fail when the flight estimation arrives too late or the distance of the robot's idle position to the interception point is too far for it to reach in time. In a nutshell, there are three periods of time that affect the outcome of a successful transport: t_{flight}, $t_{estimate}$, and t_{move}. The thrown object will only be caught if the following condition (shown in Equation 2.2.1) is fulfilled. In other words, the time needed for estimation plus the

time taken to move the robot arm to the inception point have to be less or equal to the duration of the flight (Figure 6). If this condition is not fulfilled, the robot arm will arrive too late at the correct position, therefore, the ball will be missed.

$$t_{estimate} + t_{move} \leq t_{flight} \tag{2.2.1}$$

2.2.2 Demand for a reliable prediction

Catching would simply not work without a prediction. In other words, grabbing the thrown object is possible only when the forecasting is more precise than the coverage of the catching device [PKH10][FBW07]. Table 1 shows that the flightpath of such a ball will be influenced by several forces, which can be classified into two groups: mass forces and aerodynamic forces [9].

mass forces	aerodynamic forces
gravity	drag
coriolis force	lift
centrifugal force	magnus
	pitch damping
	transversal magnus

Table 1: These two groups of forces influence the flightpath of a thrown object [9].

Although the effect of some of these forces on the trajectory of the flying body varies in significance, they each have an effect. Consideration of these forces thoroughly depends on the chosen method of predicting. However, there are more than these ascendancies influencing the trajectory of the thrown ball: local air flow, different air density, and differences of flying properties of different objects [Pon09, 2]. Even small variations in the manufacturing of such a ball or in the launching speed of the throwing device can modify the trajectory significantly. [PKH10]

After the tennis ball is discovered and its movement known, different algorithms can calculate its further course. However, a proper model that well describes the monitored behavior has to be found to enable a reliable prediction. If the model describes the movement of the ball not good enough, it has to be revised or replaced by another. When the model accurately depicts the behavior, the further flightpath can be predicted in an acceptable way. [Pon09, 11]

2.2.3 Physical-based predictions

The most logical choice of an approach for predicting the physical behavior of a flying object would be a physics-based model. The simplest of these would only involve the impact of gravity, which forces the ball to fall towards the ground. The velocity vector of the thrown object consists of three particular vectors in different directions (Figure 7). The advantage of this model is its simplicity since it calculates all velocity vectors separately.

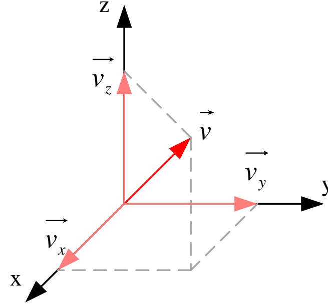

Figure 7: The object's velocity can be described by its three components.

Equation 2.2.2, Equation 2.2.3, and Equation 2.2.4 show the velocities in the different directions while the ball is in the air. Gravity only affects the object's movement in direction z. Therefore, velocities in direction x and y absolutely correspond to the initial velocity, which came from the throwing device in direction x and y. For calculating the position of the object in the x, y, and z direction for a given time, Equation 2.2.5, Equation 2.2.6, and Equation 2.2.7 are necessary. But the simplicity of this model is not only an advantage, it is a drawback as well. On the one hand, the calculations are so easy that the model does not need much computational power, but on the other hand, it neglects far too many influencing variables for predicting the trajectory in an accurate manner. [Pon09, 11]

$$v_x(t) = v_{x0} \tag{2.2.2}$$

$$v_y(t) = v_{y0} \tag{2.2.3}$$

$$v_z(t) = v_{z0} + g \cdot t \tag{2.2.4}$$

$$x(t) = x_0 + v_{x0} \cdot t \tag{2.2.5}$$

$$y(t) = y_0 + v_{y0} \cdot t \tag{2.2.6}$$

$$z(t) = z_0 + v_{z0} \cdot t + g \cdot t^2 \tag{2.2.7}$$

Aerodynamic forces are not linear and, therefore, it is not possible to separately calculate the three components without further simplifications [PKH10]. These simpler equations (Equations 2.2.8, 2.2.9, and 2.2.10) make it possible to calculate the three directions separately [BFK08]. However, they result in minor mistakes, which depend on the proportion of the velocity components (vx, vy, and vz). [Pon09, 13] [PKH10]

$$\dot{v}_x = -k \cdot \frac{v_x}{|v_x|} \cdot v_x^{\,2} \tag{2.2.8}$$

$$\dot{v}_y = -k \cdot \frac{v_y}{|v_y|} \cdot v_y^{\,2} + g \tag{2.2.9}$$

$$\dot{v}_z = -k \cdot \frac{v_z}{|v_z|} \cdot v_z^{\ 2} \qquad (2.2.10)$$

2.2.4 Trajectory Fitting Model / Polynomial Model

If the parameters of the object launch such as its velocity, direction, etc. are not known, it will only be possible to evaluate the flight regarding its movement in space and time. Since the trajectory depends on these parameters, they are implicit in this progress of movement. Therefore the first step is to record the various object's positions during its flight (Figure 5). Afterwards, a polynomial function (Equation 2.2.11) can be fitted into the measured data in the best possible way. Indeed, the polynomial model is the simplest approach of all.

$$p_i = p_0 + p_1 \cdot t + p_2 \cdot t^2 + \ldots + p_n \cdot t^n \qquad (2.2.11)$$

An example for a solution to finding such a polynomial function like the one described in Equation 2.2.11 is shown in Equation 2.2.12. This error function should be as small as possible, which means that the smaller E(p) is the better fitting of the chosen polynomial function in the captured trajectory [Pon09, 13]. This approach to finding an appropriate function is called Method of Least Squares and was formulated by Joseph-Louis Lagrange [10].

$$E(p) = \sum_{i=0}^{N} [p(x_i) - f_i]^2 \qquad (2.2.12)$$

In theory, order n of the polynomial function determines how closely it fits to the trajectory of the flying object [PKH10]. In testing this approach in the field, it was recognized that a function of third-, fourth-, or higher order barely provides a better result. Indeed, functions of higher orders tend to be unstable at higher frame rates of the captured scene [PKH10]. According to paper [PKH10], a second order function is a good compromise between sensitivity and a stable behavior.

2.2.5 Kalman filter

Another approach is the dynamic model, which uses a Kalman filter for predicting the next steps. Two steps are alternately carried out when using the Kalman filter: a predicting step and an updating step (Figure 8) [11].

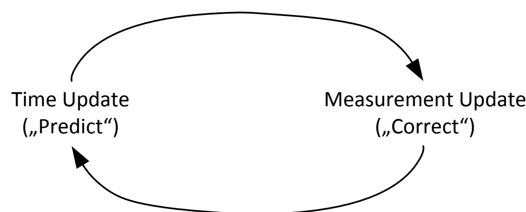

Figure 8: The Kalman filter consists of two steps that are processed alternately. Modified from [11]

Since the "normal" Kalman filter does not involve non-linear influencing variables, the estimation of the flightpath is not sufficient for our purposes. One remedy of this drawback is the use of the extended Kalman filter [11][12], which is able to consider those non-linear effects. However, it contains Jacobian matrices that are sometimes hard to calculate. Therefore, the process is not always reliable. Furthermore, the algorithm is hard to implement and presents a very sophisticated task for computation [11][12]. These big disadvantages led to the development of another approach to predict through using the so-called Unscented Kalman filter. With its faster computation, this algorithm is the better choice for estimations of real-world problems [JU97, 2]. Nonetheless, this algorithm does not offer the perfect solution as well: it assumes a random Gaussian variable, which does not lead to an adequate calculation of all the different problems [VSH04, 1].

2.2.6 Comparison of the presented models

In the direct comparison between physical and polynomial models, two different assertions were made. Various works and papers [Pon09][PKH10] suggest that the polynomial model performed the worst. The physical approach performed better, but slightly worse than the separated physical model that consider the non-linear effects. However, they still do not provide the perfect solution. Most recent works that deal with catching flying objects have been based on solutions with a Kalman filter [FBH01][BWH10] or a physical model [FBH01][BFK08]. A direct comparison between these two methods of prediction was not found, but as previously stated in Section 2.1.1, no study achieved perfect results, regardless of the model used. The best results achieved a success rate of about 80 percent, which is high but not high enough to be considered a consistent catching device! Although some models consider more factors affecting the flightpath, none of the methods take into account all the leverages that play a role in modeling such a flightpath.

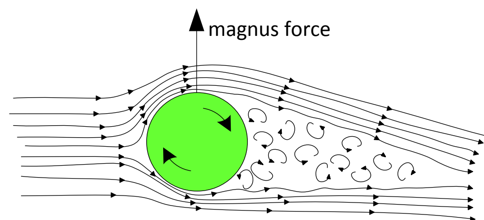

Figure 9: The spin of a ball can greatly affect its flightpath. Modified from [13]

For example, the spin of an object is not being considered in most models. The throwing device is responsible for the rotation around the thrown ball's own axis (Figure 9), which leads to the flight-path-changing Magnus Effect. A low speed spin means a slow rotation of the ball and can be neglected in most cases[1], whereas a high-speed spin has to be considered [ATW07, 327]. The spin of a tennis ball in some test scenarios was about 1000 min^{-1} and therefore it has to be considered for a successful capture [PKH10][ATW07, 326]. Figure 10 demonstrates this issue by means of a table tennis shot.

[1] This only applies for point-symmetrical objects like a ball.

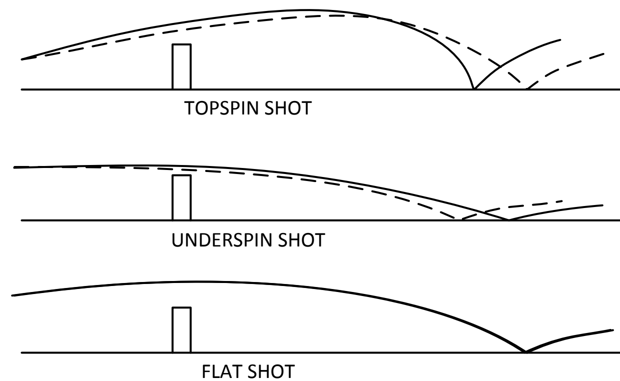

Figure 10: The spin of a ball greatly affects its flightpath. Top spin means the ball is spinning forward and leads to its rapid descent. If the ball is hit with an underspin, it behaves the other way round. Modified from [14]

For a stereovision system, measurement of such a spin is very difficult, sometimes almost impossible, and its influence on the flighpath varies because of production variances from tennis ball to tennis ball [PKH10]. Therefore, it is important to search for other solutions to consider this effect in the prediction progress to enable a reliable estimation system. To improve the throwing device or only use flat and hairless balls would not solve the problem, but it would lead to better results [Pon09, 68]. For example, while a table tennis ball is flat and hairless, its trajectory is still affected by the spin given by the bat when playing it (Figure 10).

In addition, a catching device should be in a position to catch all kinds of objects, regardless of the throwing device used. Hence, there is a demand for another model, which will be able to handle all these influences. The diploma thesis of Pongratz [Pon09, 68] must be mentioned in this context. Following a summary of the different models, a vision of a future model, whose approach will be to compare the captured flightpath with a small set of earlier recorder reference trajectories, was presented.

2.2.7 Bio-Inspired approach / k-Nearest Neighbors algorithm

This method represents one of the newer approaches in the field of Transport-by-Throwing research and is very similar to the way a human catches something in motion. Hence, this is the biological approach based on experiences such as a child improving his ball-catching skills. With each attempt at catching it, another experience with the ball's movement will be ingrained in the child's memory. Analogously, numerous flight trajectories have been recorded and stored in a database. When an object is flying towards the catching device, the parameters of its flight will be captured and compared with a set of stored reference throws to enable a prediction about the trajectory as well as the point where it can be caught [PP12, 6].

The wealth of experiences of the catching system significantly determines the quality of prediction and the results obtained. All the factors influencing a flightpath can be theoretically taken into account when the trajectory database is sufficiently comprehensive. Not only will gravity and air drag be considered, the Magnus Effect, which is caused by the

spinning of the ball, receives attention as well. To find a well-fitting trajectory by comparing the actual flight with the entire database, a k-NN (k-Nearest Neighbors) searching algorithm is used [MPD14]. This method was primarily used for the assignment of pattern classification in the field of computer vision [CH67, 1] and later for a time series forecasting as well [Yak87, 235].

$$y = \frac{1}{k}\sum_{i=1}^{k} x_i \tag{2.2.13}$$

To predict the further flightpath of the actually thrown ball, the average of the k best fitting trajectories from the database is calculated [AC13, 1]. This procedure is shown in Equation 2.2.13 where y is the output flightpath and x_i is one of the k best fitting trajectories from the database.

Besides this “normal” k-NN approach, there is also another method where the k chosen tracks are weighted by means of their resemblance to the actual flightpath (Equation 2.2.14).

$$y = \sum_{i=1}^{k} w_i x_i \tag{2.2.14}$$

Whereby both below-stated conditions (Equation 2.2.15) have to be fulfilled:

$$\sum_{i=1}^{k} w_i = 1 \qquad \& \qquad 0 < w_i \leq 1 \tag{2.2.15}$$

To get a better understanding of this method, a simplified example will be explained: Imagine that the flightpaths are only two-dimensional and the parameter k is equal to 2. With these simplifications, the trajectories would look similar to the ones in Figure 11. Furthermore, imagine that curve C is the trajectory of the actual flying object whereas tracks A and B are the most similar matches from the database. The fact that the new trajectory is between those two sheds light on its further movement and permits the calculation of the point where the ball will land [MPD14].

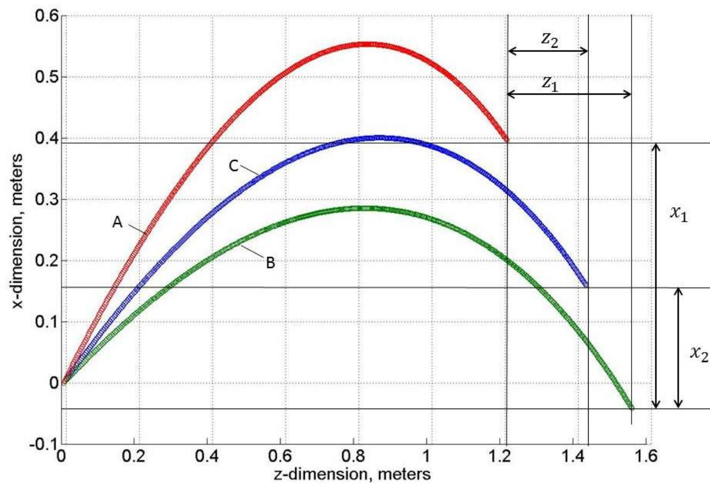

Figure 11: Flightpaths in a 2D aspect would look like these [MPD14].

A possible simplification of the design would be a consequence of neglecting some of the influencing factors when projecting the captured points on a 2D plane. It can be called *Plane of Flight* and is collinear to gravity and the speed the object gets at its launch [MPD14]. Figure 12 shows an optional object coordination system stretched between x* and z*, in which the Plane of Flight could lie. The results of this simplified approach depend on the foreign influences that could cause the flightpath to deviate. However, in general, the outcomes of this simplified approach (caused by neglecting some possible forces) will be slightly worse than the "normal" one without the Plane of Flight.

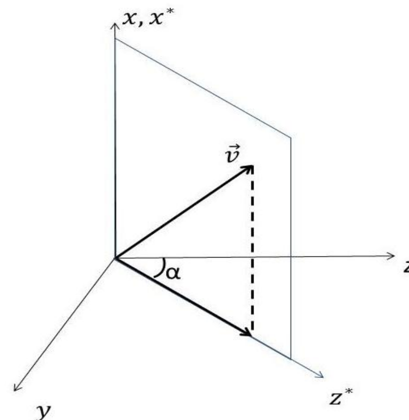

Figure 12: The plane of flight is a simplification in predicting [MPD14].

Mironov and Pongratz published a work [MPD14] about the implementation of a Matlab Code for predicting an object's trajectory with above-mentioned biological approach. Simulations showed that it produces better results than past approaches that were based on physical laws, the Kalman filter, or the fitting of a simple polynomial function. It was possible to estimate the correct inception point for soft catches in 90% of the simulated throws. [MPD14]

2.3 Object detection and localization

As previously explained, detection of the flying ball is absolutely essential to making a prediction about its future movement. It is impossible to catch a ball if one doesn't know where it is! This subchapter will provide an overview of the different techniques of object detection and localization.

Catching a ball is not the only application for detecting an object. Modern cars often have safety-relevant features like a pedestrian detector [DWS12, 743f] that warns the driver or breaks the car when a human is on the street. Other developments in the automotive field go one step further, like the autonomously-driven cars from Google or Audi [15], which are currently tested in the USA. A "simple" pedestrian detector would not be enough for this task because cars have to see everything around them: from pedestrians and other obstacles to street signs and road markings as well as other traffic participants. Additionally, there are other non-safety-relevant applications like the "Hawk Eye" [16], which are used in various

sports and tournaments to reproduce the ball's movement to support the referee in difficult decisions.

While much has been published on object detection, projective geometry, and triangulation [FLP01] [HZ03][MGV09][Sze10], in this framework we will only discuss the basics.

2.3.1 Triangulation, stereo vision, localization

Depth information is essential to knowing the ball's movement in space and time. Using two or more cameras for locating the ball is advisable because to do this with only one camera is a very ill-conditioned problem [FBH01, 1623][BFK08]. Triangulation, which can be done with a stereoscopic vision system, is a good and cheap option to obtain a 3D view [SPV05]. These two cameras are located side by side with a predefined distance between each other and should synchronously acquire their images [Pon09, 18]. The way to turn two 2D pictures into a 3D scene is similar to the biological way a human assembles the two images obtained from the left and the right eye to one three-dimensional image [17].

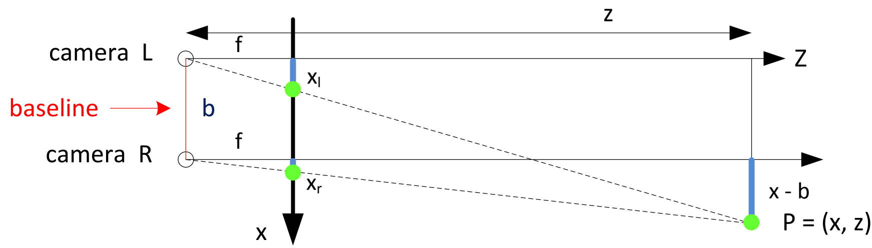

Figure 13: Triangulation enables to calculate the location of the object when some properties are known: baseline distance (b), focal length (f), and the measured disparity (d = xl – xr). Modified from [18]

However, knowing only the distance between the two cameras, called baseline, is not sufficient. Other important and relevant intrinsic parameters of the cameras are used: the focal length, the principal point coordinates, the skew coefficient, and the image distortions coefficients [18][SPV05]. Pinpointing the object of interest (the thrown ball) in an accurate way is only possible when all these specifications are known [19]. Finding the same point of the same object in both pictures is necessary to determine the location of this object and can be described with the term *Correspondence Problem* [SPV05]. In subject literature, this point is called *point of interest*, and there are several ways to find it. As shown in Figure 13, it is possible to measure the disparity of the two positions from the point of interest in the two captured frames. Equations 2.3.1 and 2.3.2 show that it is possible to calculate the distance z to the chosen point when the information about the base length b, the focal length f, and disparity d is known. [Sze10, 48ff]

$$d = x_l - x_r \tag{2.3.1}$$

$$z = \frac{f \cdot b}{d} = \frac{f \cdot b}{x_l - x_r} \tag{2.3.2}$$

The question of finding the point of interest is easy to answer. In the case of a tennis ball, one particular point of reference makes the most sense: the center point; in a 2D-view, a ball has the shape of a circle and, therefore, it makes sense to look for its center. The way the center of the ball can be identified in the left and the right image will be shown in Subchapter 2.3.3. The coordinates of the ball's center in both frames can be transformed to a coherent World Coordinate System, so that the catching device is able to catch the ball (Figure 14). [Pon09, 36]

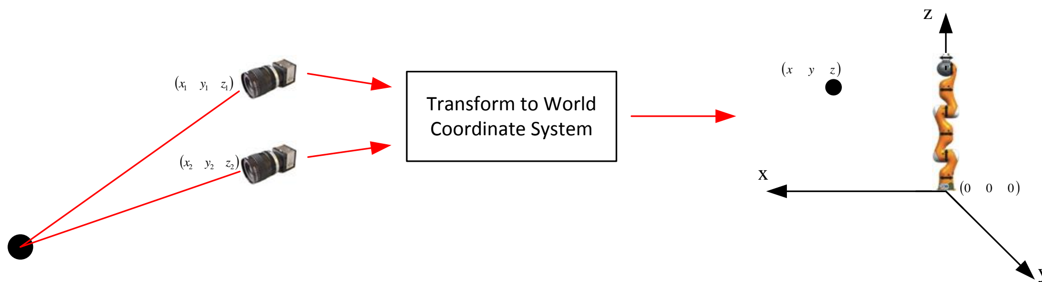

Figure 14: The location of the ball has to be transformed into coherent World Coordinates to enable the robot to catch it.

The precision of object localization depends not only on the resolution and other intrinsic parameters of the cameras, but also on their position and alignment. Figure 15 shows the three ways for setting up the camera alignment, which lead to different visual views: divergent, parallel, and convergent. [Pon09, 17]

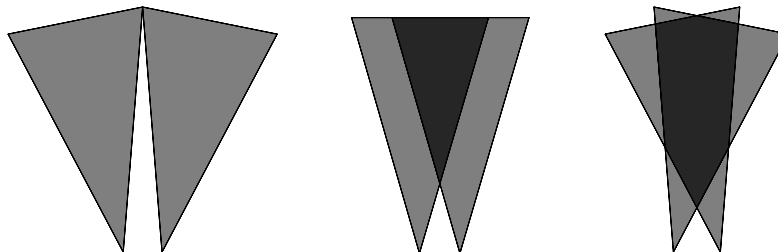

Figure 15: The three different alignments for a stereo vision system from left to right: divergent, parallel, and convergent. Modified from [PON09, 17]

The divergent variant is inadequate for the task of getting a 3D view through triangulation because the point of interest has to be present in both images [Pon09, 17][Sze10, 537]. The parallel alignment has the advantage of not needing a keystone correction, which is normally needed to remove the distortion of a picture caused by recording in an angle other than 90fl to the filmed plane of interest (Figure 16) [20].

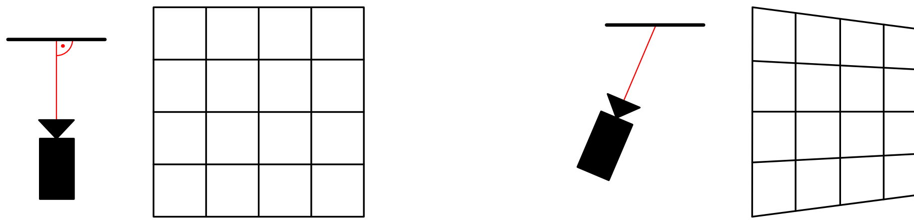

Figure 16: The left side shows a distortion-free image while at the right side a distorted picture is imaged.

Again, the advantage of using a ball as a transported good is in its shape. A ball does not have a plane, and it will not be distorted regardless from which angle filmed. It will always look like a circle! But distortion is not the only impact of filming from a slanted angle. A convergent stereo vision setup possibly leads to a smaller quantization error, which leads to the deviation distance Δp, than a parallel setup [SSL01, 1]. As shown in Figure 17, the smallest localization error Δp can be achieved at an angle of 90 degrees between the two cameras [Pon09, 18].

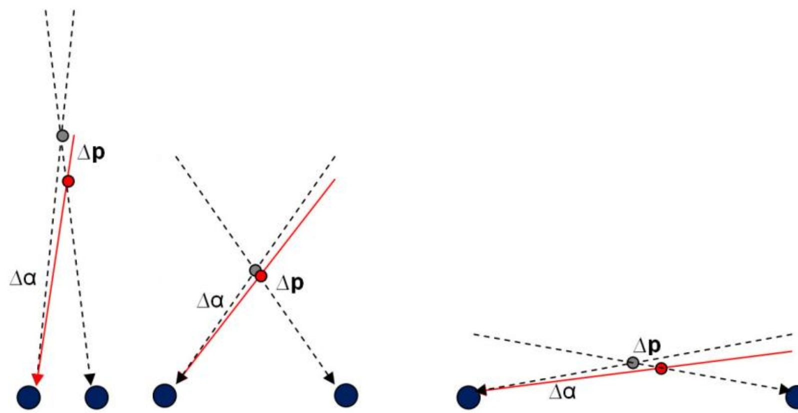

Figure 17: Various filming angles with the convergent camera setup lead to different quantization errors [Pon09, 18].

2.3.2 The position of the Stereo Vision System

The object can be localized more precisely when it is closer to the Stereo Vision System. For this reason, positioning the cameras behind the throwing device is advantageous for achieving better results. When filming from this position, the tracking and predicting of the ball's trajectory will already be as accurate as possible in the early flight phase [FBH01, 1629], which leads to a quick estimation of the inception point. As a result, the robot arm has more time to move to this point. The study [FBH01, 1629] showed that a position behind the throwing robot, with a baseline distance of 1 m between the cameras, leads to better results than trials from other positions. However, placing the cameras behind the throwing device will not work in every application because of the need for an information channel between the Vision System and the catching device.

2.3.3 Point of interest

Finding the point of interest is essential for the purpose of localizing the flying object through triangulation. Choosing the center of the ball as the point of interest appears obvious and logical. The following subchapters will present a few techniques to detect circles and their centers.

2.3.3.1 BLOB Detection based on Difference Image forming

A BLOB (Binary Large Object) is a connected region in an image that represents an object and is associated with a local minimum or maximum [Lin93, 33][FBH01, 1624].

BLOB Detection can be simply calculated through a difference image and can, therefore, be done very quickly.There exist two methods that will be briefly presented here. AFD (Adjacent Frame Difference) is the procedure of making a difference image from two consecutive frames [Pon09, 5]. The ball can be localized at the place where the changes of pixels have been bigger than a previously set threshold [GW07][TKB99, 3]. This approach is independent of changes that are slower than the frame rate of the Vision System, which in turn means that problems can occur when the movement of the ball is not fast enough for the chosen frame rate. In this case, the ball in the actual frame would overlap with the one in the previous image. This problem is called ghosting (Figure 18) and leads to an inaccurate localization of the ball's new position.

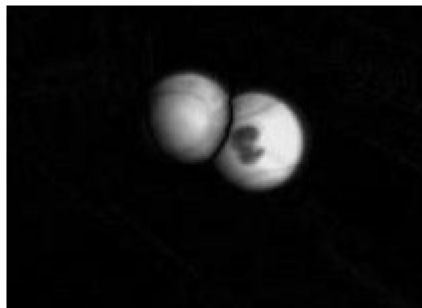

Figure 18: Ghosting describes the fact that object is overlapped in two adjacent frames. Left, the ball in the previous frame, right, the ball in the actual one. [Pon09, 5]

The other feasible solution makes use of a difference image made out of the captured frame and a picture of the scene‘s background. But this method is not without problems either because of the possibility of an alternating scene that differs from the recorded background. Such changes can be caused by outside influences such as varying lighting conditions or flickering lamps. Hardware features, too, can lead to problems such as automatic gain control of the cameras or frame-grabber cards. A way around this problem is the use of a slow, adaptive background image that accommodates changes of the scene [FBH01, 1625]. However, Pixel Jitter effects [21] or the interlacing of the cameras also lead to a difference of the background and can disturb the detection algorithm. These issues can be avoided when using a reference interval instead of fixed thresholds for determining pixel changes (Figure 19). If the intensity shift of a pixel is so large that it exceeds or falls below the reference band, this picture element will be counted as BLOB-related.

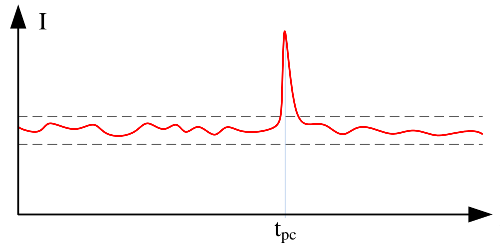

Figure 19: The continuous change of a pixel's intensity is caused by Pixel Jitter effects. A real change of a Pixel is indicated when its intensity that exceeds or falls below the reference band as it happens at the time t_{ps}. Modified from [FBH01, 1625]

The main drawback is in the possible inaccuracy of finding the ball's center. Basically, it is not very complicated to detect the center of such a created BLOB, but it will not be in the right position if pixels that are not part of the same object are counted to the BLOB and, vice versa, if pixels from the ball are not counted to the BLOB. This error can be enhanced by well-deliberated motion detection algorithms, but a total prevention of this behavior is not possible [Pon09, 7]. To make a prediction as accurate as possible, localization has to be extremely precise. Therefore, another approach is needed to detect the flying ball.

2.3.3.2 BLOB Detection based on Color Histogram comparison

Besides making a difference image, there is another possibility to create BLOBs. The captured frames can also be compared with a color histogram when the ball is specially painted for this job [FBH01, 1623][HS95, 2]. The use of a color that does not occur again in the rest of the scene is essential; otherwise this colored area would also count as a ball. One drawback of this method is again the inaccurate center detection of the discovered ball, as described in the previous Subchapter 2.3.3.1. But this is not the only problem: the ball could appear in different colors, which might be caused by the different lighting conditions in the room in which it is flying. This would make it difficult to detect it through color comparison [FBH01, 1623].

2.3.3.3 Edge Detection as preparatory work for accurate Object Detection

BLOB detection can be computed extremely fast, especially when only processing an Area of Interest that can be determined through the correlation of consecutive frames. The ball in the actual frame will have a similar color and will be around the same place as in the previous frame [SPV05]. However, the tendency to provide inaccurate results makes this approach useless and raises the demand for other solutions.

Figure 20: Transformation from Input Image to Edge Image through the Canny Edge Detector [22].

Due to widespread occurrence of circles and spheres in nature, Circle Detection is one of the most important and most frequently used applications in the field of Computer Vision [SPV05]. Various methods for detecting circles exist [DAC02], but two of them are very well-known and often used for object Detection: the *Hough Transformation* and the *RANSAC algorithm* (Random Sample Consensus). Both algorithms will be discussed in the following two subchapters, but they require a preparatory step [YR08, 2][Pon09, 33], which will be explained here. An analysis made with the Hough Transformation or the RANSAC algorithm requires an edge magnitude map. Therefore, a filter is needed that transforms the input frames (Figure 20, left) into images where only edges, or rather local changes in intensity (gradients), are drawn (Figure 20, right) [JWD13]. These filters are called Edge Detectors and can either be first derivate filters or second derivate filters. The advantage of first derivate filters is in their fast computation resulting from simple algorithms, but they are very sensitive to noise in the image [Pon09, 7]. In contrast, second derivate filters need more calculations steps, but provide better results. One of these last-mentioned filters is the Canny Edge Detector, which was published by John Canny in 1986 [CAN86, 1]. This filter consists of more than just a simple convolution with one matrix, it is carried out in four steps: Gaussian filtering, Sobel filtering, doing non-maximum suppression, hysteresis thresholding [OIN10]. After these four steps, an edge magnitude map like that on the right side of Figure 20 is achieved [GW07]. The Canny Edge Detector distinguishes itself from other edge filters through its global inspection of the entire image with the opportunity of detecting possible closed loops [SHB07]. As it tends to achieve good results, this filter is nowadays very popular among developers whenever an edge image is required [YR08, 1]. However, the iterative process of the Canny Edge Detector's hysteresis step could lead to longer execution times: the more pixels are present, the more time will be needed [YR08, 7].

2.3.3.4 Hough Circle Transformation

The Hough Transformation was developed by Paul Hough in 1959 [Hou62] and is a possible procedure to detect objects on the basis of their parameters [Pon09, 33] out of a created edge image [HP62]. Its original task was to detect lines in images [Kol02, 313]. Soon however, improvements and extensions were made such as another way of calculating with polar

coordinates [DH72] or an algorithm that involves the direction of the gradients [OC76, 544]. Furthermore, a method for the detection of circles and arcs [IK88, 93] was developed, which is of interest here for the task of finding the center of the thrown ball.

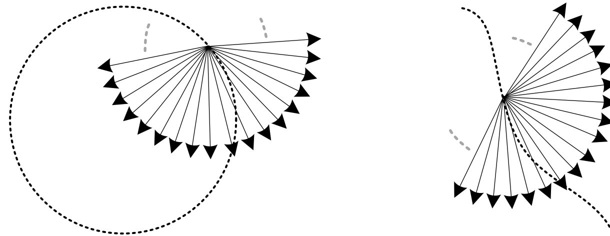

Figure 21: Every edge point "looks" around itself for a potential center point.

To perform the Hough Circle Transformation, each edge point will be considered part of a dedicated circle with the radius r. This will enable looking for potential center points (Figure 21). This is done by writing in an array called Hough Space, which is the storage for this voting process. At the beginning of this algorithm, every field of this array has the value zero. And every time one of these fields (pixels) is voted to be a potential center point, the value of this field will be incremented by 1. With an accurately defined radius, we will only need a 2D array, which represents the various pixels. After the voting process is completed, the local maximum can be determined. The field with the most votes represents the center of the circle. In other words: k edge points belonging to one circle with a radius r will lead to a k times increment of the field in the Hough Space, which is representing their center point (Equation 2.3.3).

$$A(x, y) = k \tag{2.3.3}$$

Figure 22 shows the Hough Space superimposed with the edge pixels of the corresponding circle. As can be seen, the most votes are in the center of the circle.

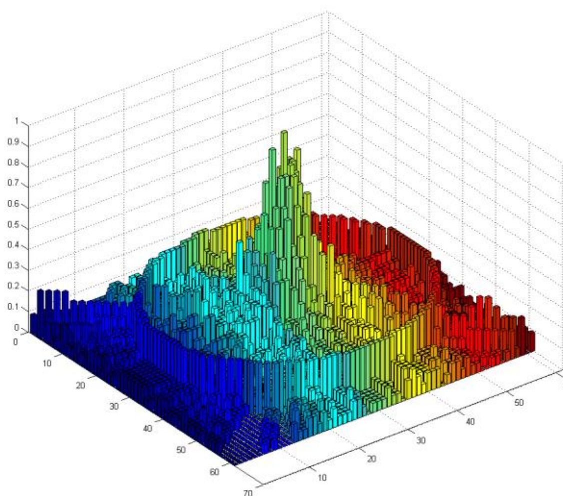

Figure 22: The Hough Space with its votes superimposed with the edge of the corresponding circle [PKH10].

Rarely will we know the exact radius r and, therefore, a range of radiuses to be voted for has to be defined. If one looks for circles with different radiuses, a 3D-Hough Space will be needed: one dimension for x, one for y, and one for the different radiuses. Figure 23 shows when every edge point is voting for potential center points for different radiuses. Ultimately, the result will be the same: the field with the most votes is the center of the circle. [SPV05][Pon09, 9][PKH10]

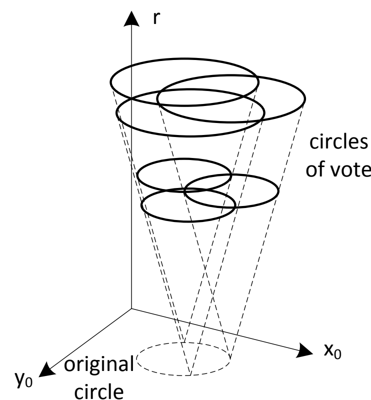

Figure 23: The 3D Hough Space for detecting Circles with a radius in the defined range of radiuses. Modified from [DH72]

On the one hand, the Hough Transformation achieves accurate results, on the other hand, it requires much computational power. Therefore, enhanced approaches are welcome to simplify the procedure. A growing Hough Space leads to an exponential increase in the process' complexity. As stated above, when detecting circles with an unknown radius, the voting process has to cover three instead of two parameters in the Hough Space [AME13, 216]. Therefore, another method was developed and used in various works [RFQ03][KBS75][MS81]. Instead of examining the image only on the basis of its single edge points, a segment of a circle, an arc, will be analyzed regarding its direction. As soon as the direction of the arc is identified, a line along the normal of the arc's tangent will be drawn (Figure 24). The pixel where most of these lines intersect is the center of the circle. Storing straight lines instead of circles in the Hough Space leads to a reduction from three parameters to two [SPV05].

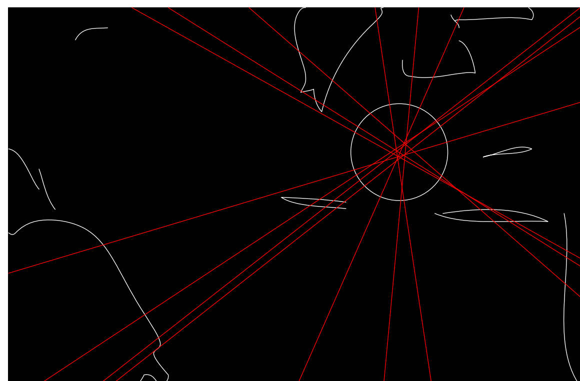

Figure 24: Another approach of the Hough Circle Transformation consists in drawing lines along the normal of the various arcs of the circle. Modified from [SPV05]

However, this approach does not offer the perfect solution because it might achieve inaccurate results caused by noise. When some of the circle's segments have more pixels than they should, the edge image will not be well-defined, and the drawn lines will not be exactly perpendicular to the tangents. Conditions of the edge images and the desired accuracy have to be considered to fulfill all requirements for the system. Furthermore, an algorithm optimally matching the hardware's capabilities will be needed.

To sum up: the Hough Transformation is noise tolerant, highly stable, and achieves accurate results [JWD13]. In addition, it can be adapted for objects of different shapes. All these features render it a viable option for this project's purposes. A possible drawback could be its costly computational requirements.

2.3.3.5 RANSAC algorithm

Another popular method for detecting objects, called RANSAC algorithm, is a non-deterministic technique [JWD13] and was published by Fischler and Bolles in 1981 [FB81, 1]. Lines, circles, planes, and other shapes are detectable with RANSAC just as with the Hough Transformation. Since this thesis deals with the catching of a ball, circle detection will be explained here (Figure 25, left). In the first step, three edge points are randomly chosen to draw a circle through them. Then, the distances between the image's edge points and the circle's points are examined. If the distance of one of these edge points to the circle is smaller than a predefined parameter, the pixel belongs to the circle and is called an *Inlier*. If the edge point is not close enough, it does not belong to the circle and is called an *Outlier*. A circle has been detected when the quantity of Inliers exceeds a predefined threshold. Subsequently, three further edge points will be randomly chosen and the procedure repeats. These steps will be repeated for a predefined period. An example for finding a circle is shown in Figure 25. Besides the parameter of the maximum distance for qualifying as an Inlier, other parameters have to be set as well: the number of iterations (how often new random pixels will be chosen), and the minimum distance these random points must have to each other.

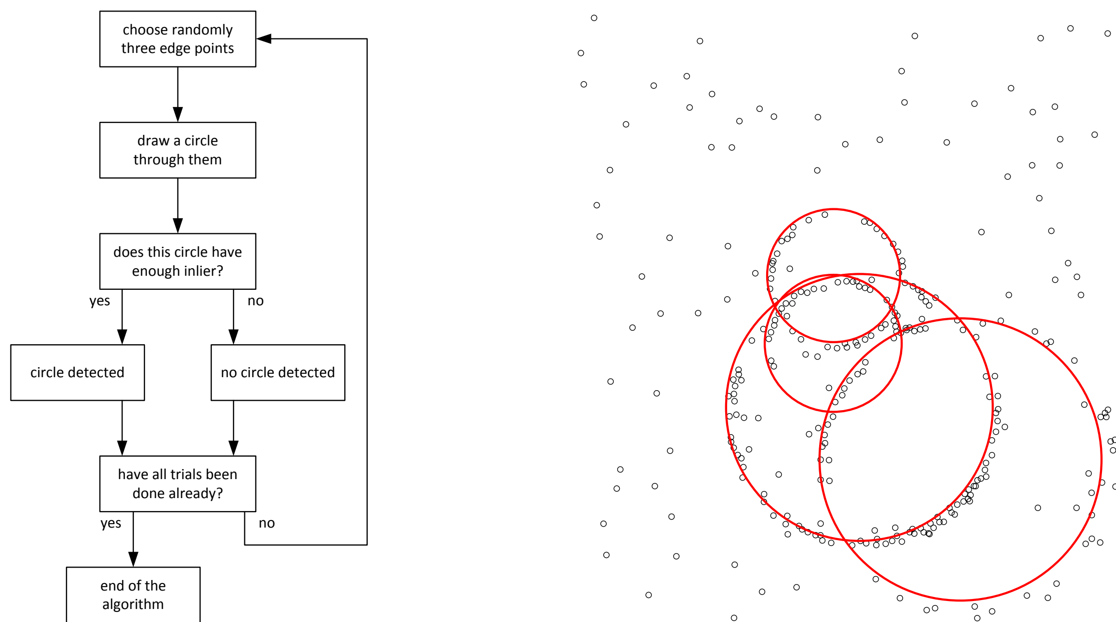

Figure 25: Left Left: steps of the RANSAC algorithm; right: possible results of the algorithm. Modified from [23]

The accuracy of the achieved results depends on the number of iterations and the amount of edge points that are not part of the circle. The more frequently random points for searching are chosen, the higher the likelihood of finding a circle. The more edge points not belonging to a circle are in the image, the less the likelihood of finding a circle.

2.3.3.6 Hough Circle Transformation vs. RANSAC algorithm

Both procedures allow detecting various shapes such as lines, quadrangles, circles, plains, etc. Additionally, edge magnitude maps, which were presented in Section 2.3.3.3, are essential for executing one of these algorithms.

One published work [JWD13] made a direct comparison between the Hough Transformation and the RANSAC algorithm. The experiments were about finding lines in data sets from radar, which were partly experimentally recorded and partly artificially generated. The first conclusion seems very logical and stated that both procedures do their jobs better when there is no noise in the images.

However, the Hough Transformation was highlighted as being more accurate when detecting lines in the presence of noise. This algorithm was still highly stable and able to detect in an accurate way [JWD13]. The reason for these good detecting qualities is the search algorithm, which examines each edge point as to whether it is part of the wanted shape. In case of circles, their center will be searched for in a predefined range of radiuses around each edge point. This leads to the assumption that detection will work properly if the image is not totally overloaded with noise and the correct radius is present in this range of radiuses. On the other hand, RANSAC is a stochastic algorithm and strongly depends on various characteristics of the image and some other predefined parameters. The quantity of edge

points present, the number of iterations, and the predefined minimum distance between randomly chosen points are crucial for the success of the procedure [JWD13].

The comparison of the two approaches showed that the Hough Transformation was better than the RANSAC algorithm at detecting lines in the experimental data. However, RANSAC achieved better results than Hough when examining the generated data [JWD13]. The issue of the algorithms' required calculation times is not covered here in depth, but another paper states that the Hough Circle Transformation is fast enough for real-time applications at a frame rate of 70 FPS [Wei08, 7]. The duration of this algorithm depends on the quantity of pixels in the edge image [WL12]: the more pixels are present, the more time will be needed because of the necessity to examine each of them. In comparison, the time required by the RANSAC algorithm mainly depends on the number of iterations and will probably be less than with the Hough Transformation. The voting process for various radiuses carried out by the Hough Transformation requires more computation steps than the RANSAC algorithm. Furthermore, an edge image will probably have more edge points than the number of iterations needed for finding a circle with the RANSAC algorithm. The smallest possible distance between the chosen points might be affecting the temporal behavior as well: When the chosen points are too close to each other, new points have to be chosen until all three of them fulfill the requirement of minimum distance. If this happens frequently, the entire process will take more time.

2.4 GPU programming

Programs or parts of them can be accelerated through the massively parallel architecture of a GPU (Graphics Processing Unit), which leads to shorter computation times [SHH07, 1][FM05]. Previously, using the computer's graphics card for general purpose computing was only possible with the help of a trick. It was necessary to wrap the necessary program parts in a graphics framework to fool the GPU into "believing" that it computes a normal monitor frame while its capacity was used for other purposes. Coding in this manner was a hard task, therefore, it could not be expected to be carried out by the general public, only by a selected few developers who were up to the challenge [YR08, 1]. Driven by the demand of the market to have a programmable graphics processing unit for general purpose computing, called GPGPU (General-Purpose Computing on Graphics Processing Units), NVidia eventually developed CUDA (Compute Unified Device Architecture) in 2006. Since then, it has been possible to use this programming platform to easily implement programs running on a GPU and speed them up with the huge number of parallel computing processors. [CUD14, 4][OLG07, 7]

2.4.1 The necessity of using a GPU

The real-time constraints of the bio-inspired Transport-by-Throwing approach are an open topic until now. The arm of the catching device has to move early and quickly to the interception point to be there on time [PKH10]. The following example shall help understand the challenge of timing and what it means to do the prediction in real-time. In case of the

robot used, the KUKA LBR 4+, the interception point must be known at least 33.5 ms before the object arrives there. The flight of the thrown object takes approximately 780 ms if the ball travels with a speed of 5.3 m/s over a distance of 2.5 m with a launching angle of 42 fl [PMB13]. Consequently, the result of the prediction of the inception point has to be ready roughly 746.5 ms after the object was thrown. This short period between the launching of the ball to the time when the trajectory estimation is needed provides an illustration of the required speed of computation, however, the major challenge are the high FPS rates (Frames per Seconds) of the recording cameras.

$$t = \frac{1}{FPS} \tag{2.4.1}$$

Table 2 shows various exposure times of frames when filming with various frame rates. These times can be calculated with Equation 2.4.1.

FPS	time interval
24	41,6 ms
25	40,0 ms
30	33,3 ms
48	20,8 ms
60	16,7 ms
90	11,1 ms
110	9,1 ms
120	8,3 ms

Table 2: Different frame rates lead to different exposure times of frames.

Pongratz and colleagues showed in their paper [PKH10] that accuracy of prediction not only depends on higher resolution, it also critically depends on the frame rate. In general, a prediction is better when working with a higher FPS rate. Indeed, in some cases it can even be useful to downscale the resolution to have more bandwidth on hand to operate on a higher frame rate. [PKH10]

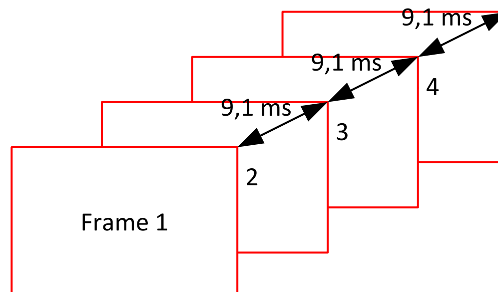

Figure 26: For a capture with 110 FPS, the time interval between the frames is 9.1 ms long.

For example, when capturing the scene with a frame rate of 110 FPS, the interval between two frames will be about 9.1 ms. All steps of image processing, object detection, and flightpath prediction have to occur in this short span. To accomplish this in real-time, all necessary calculations have to be completed in less than 9.1 ms (Figure 26). The deadline for the GPU or CPU to compute one frame is 9.1 ms after it has appeared in the computational device. To ensure that there will be sufficient free capacity for calculating the next frame when it appears, all procedures have to be completed before the deadline. However, if the estimation algorithm takes up too much of the processor's capacity, computation time will interfere with the camera's sample rates [HS95, 8]. In this case, treatment of some frames at the beginning of the flight would be completed only shortly after the deadline. Whether that is necessarily a serious problem, is unclear. It depends on the frequency of the occurrence and the length of the delay. To be on the safe side, however, every execution should be on time!

For example, the Hough Transformation needs huge computational power [SPV05]. How long it takes depends on its implementation and the quantity of pixels present in the edge image. As explained in Section 2.3.3.6, the more edge points there are in the input frame, the more time the Hough Transformation will take to execute a frame [WL12]. An adaptation of this method for detecting more complex shapes could even lead to a higher computational demand. Performing the Hough Transformation on a GPU was tested and documented in some studies [CJ11][AME13] [WL12], which came to the same conclusion that execution was much faster on a GPU than on a CPU. The obtained acceleration was measured with differing result: from 45.7x in [WL12] and 65.4x in [AME13, 220] to 400x in [CJ11].

The RANSAC algorithm and the k-NN search for predicting a flightpath will need much computational power [MPD14] as well. However, they are also partially suited for parallel computing. To shorten execution time, it makes sense to swap these program parts on a GPU.

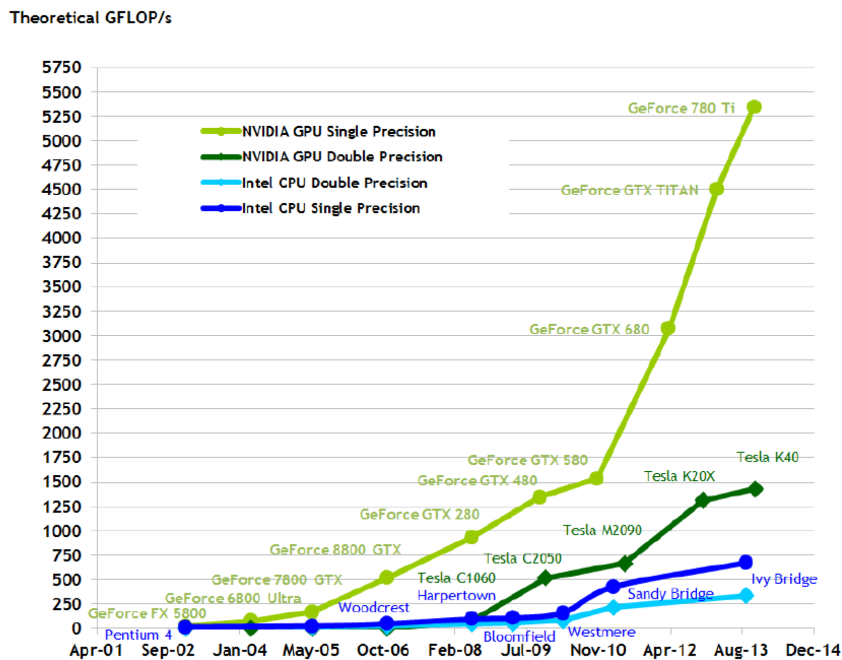

Figure 27: Floating-Point Operations per Second for the CPU and GPU [CUD14].

It is obvious that there are differences in computational power and execution time when using different hardware. That means that different GPUs will take different time to execute the program; the same is true for different CPUs. However, generally speaking, a GPU has considerably more theoretical computational power than a CPU (Figure 27). GPUs are based on the SIMT (single-instruction, multiple-thread) architecture, which is akin to SIMD (Single Instruction, Multiple Data). In other words, a single instruction processes a complete data set all at once; therefore, a GPU is the perfect hardware for processing images [OCL09, 14]. Such images consist of huge amounts of pixels that have to be processed by the same instructions. However, the implementation has to be optimized to process the data sets parallel and in an efficient way [FBH01, 1625].

It is no coincidence that a graphics card computes the frames for a computer monitor; after all, it is designed for such graphical tasks. Therefore, it seems natural to outsource program parts for image processing to the GPU to speed up the whole procedure. Another possible solution would be the use of a FPGA (Field Programmable Gate Array), which could also accelerate the execution of the algorithm [Pon09, 67], but this would probably lead to much higher development costs than an implementation for an off-the-shelf hardware such as a GPU.

2.4.2 The functionality of a GPU

There are a few fundamental issues to consider when implementing a program that will run on a GPU. Such a graphics card has its own memory and it is necessary to swap data to be processed by the GPU onto this memory first. Accordingly, a distinction is made between *host code* and *device code*. Like a regular program, the host code runs on the CPU and only processes the data sets on its main memory. On the other hand, the device code is executed on the graphics processor and edits only the data of the GPU's memory. The latter consists of so-called *kernels*, which are functions that run on one or more multiprocessors of a GPU. Such a Kernel creates a couple of light-weight processes, called *threads*, which run simultaneously on different processors (cores) of a multiprocessor to simultaneously process multiple data (Figure 28 and Figure 29). [CUD14, 9f] [OCL09, 11f]

To understand the connection between threads, multiprocessors and cores, it should be explained that threads are pooled together in *blocks* (Figure 28), which can contain between 1 and 1024 threads each. Blocks, which run on several multiprocessors of the GPU, are also pooled to a so-called *grid*. [CUD14, 11f][OCL09, 12ff]

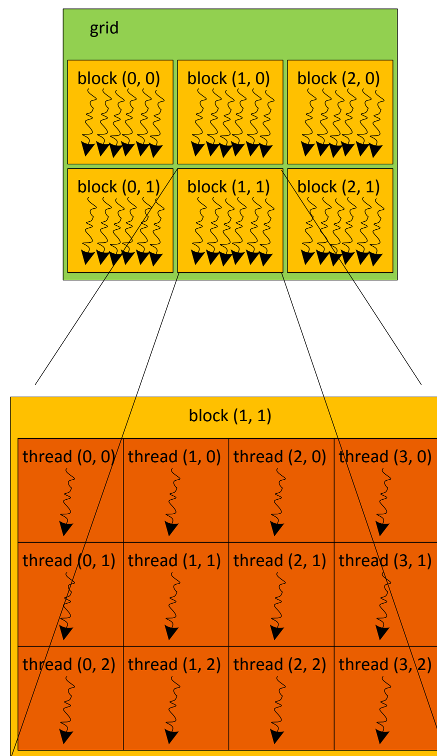

Figure 28: Up to 1024 threads are combined in a structure, which is called block. The sum of all various blocks is pooled in one grid. Modified from [CUD14, 11]

The program runs through the following steps: at first, all necessary data sets are moved to the memory of the GPU for further processing in the next step. After the GPU has processed this data, the results can be swapped back to the main memory of the host where they can be used or displayed [CUD14, 14, 225].

It should be noted that such a graphics chip contains, apart from this single device memory, also several other memories shown in *global memory*, has the advantage that all blocks can read from it and write on it. Despite two associated caches, which can considerably accelerate reading accesses on the global memory, it is the slowest of the GPU. Another storage, called *shared memory*, which is assigned to the various blocks, is much smaller than the global memory and can only be edited by its assigned block, yet, it is much faster than the global memory. [CUD14, 12, 187f][OCL09, 18]

However, this is not all: Each of a multiprocessor's several cores has its own registers, which can be used only by the assigned thread running on this core. Such registers have a much smaller memory than the other two types of storages, but they are by far the fastest on the GPU. [OCL09, 14f]

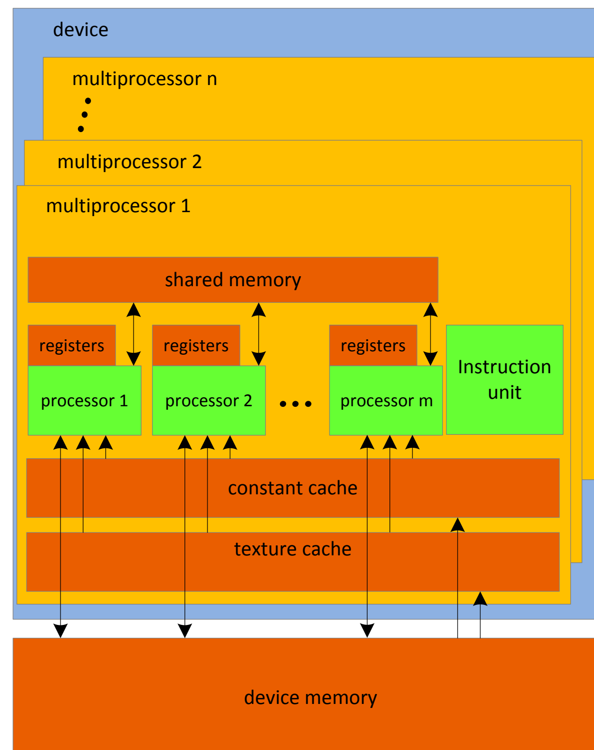

Figure 29: The schematic view of a GPU shows a device memory (supported by two caches) that can be read and written from all cores of the whole Grid. There is a faster storage, called shared memory and it is only accessible to the threads of the same assigned block. Additionally, each processor has its own registers, which are only useable for a thread assigned to it. However, these registers are the fastest storages on the GPU. Modified from [OCL09, 16]

When trying to improve the performance of the implemented algorithm, it makes more sense to optimize the program by using the shared memory instead of the two caches, which accelerate reading from the global memory. This course of action brings about a reduction in slow global storage accesses and, therefore, leads to a minimization of the program's execution time [LWT12, 1]. It is also important to mention that such optimization requires developers with broader knowledge and experience. The risk of a deadlock or other failures exists when the program is not correctly implemented. To avoid these problems, such optimization will not be carried out in the framework of this thesis.

2.4.3 CUDA vs. OpenCL

Besides CUDA, there exists another option for programming a graphics processor: OpenCL from the Khronos Group [24]. Subchapters 2.4.3.1 and 2.4.3.2 will show the existing research on the two platforms. The differences as well as the similarities between them will be worked out.

2.4.3.1 Formal differences between the two platforms

While OpenCL represents an open software standard to implement portable programs for GPUs and Multi-Core CPUs from various vendors, CUDA is proprietary software that works

only with NVidia graphics cards [KDH10, 1][FVS11, 9]. Nevertheless, these two different platforms show a lot of similarities: both of them are an extension for C (the programming language) and are used to program a host- as well as a device code. They have the same memory model and are both equipped with an equal segmentation of blocks and threads. The various interface-dependent expressions and their relations to each other are displayed in Table 3. This provides a simple way to translate programs from CUDA to OpenCL and vice versa [MGW11, 1].

CUDA	OpenCL
global memory	global memory
constant memory	constant memory
shared memory	local memory
local memory	private memory
thread	work-item
block	work-group

Table 3: Terminological differences between CUDA and OpenCL [FVS11, 2].

Besides these terminological differences, the two platforms are very different regarding the compilation of a program. The pure C-Code is not the only program part being translated into an executable file when compiling the CUDA program, the device- and the host code will be compiled at the same time. At runtime, both of them can be executed without any additional preparatory steps. When developing an OpenCL program, the software designer has to ensure that the device code will be compiled at the beginning of the execution time. As a consequence, the duration of the initialization of an OpenCL program needs more time than a CUDA program. On the other hand, compiling the device code at runtime can be of great advantage; it offers the possibility of optimizing the code for the hardware used for executing the program at that moment. [KDH10, 10]

2.4.3.2 Performance differences between the two platforms

Karimi, Dickson, and Hamze stated in their study [KDH10, 9f] that CUDA is preferable if the focus is on performance and shorter execution time. If the developer intends to implement the program platform on hardware from various vendors, the development environment of OpenCL should be used. Karimi's team wrote two nearly-identical programs for various tasks to compare the performance differences between the two platforms. The measured data and results read as follows: Execution times of the OpenCL kernels were about 13 to 63 percent longer than those of the CUDA kernels. There were also similar outcomes regarding the two programs' overall execution time: the OpenCL program was about 16 to 67 percent slower.

However, this is not the only paper comparing the performance of both platforms. Fang, Varbanescu, and Sips wrote [FVS11, 1, 9] that an unfair comparison used in other works accounted for about 30% of the performance gaps. Additionally, they implied that there occurred illegal storage accesses, which accelerated execution and were camouflaged as an access to a texture cache when using the CUDA interface. Moreover, the CUDA compiler

optimizes in a better way than its OpenCL counterpart. If the texture cache is turned off and the program manually optimized, the performances of both programs will be roughly equal.

As conflicting statements have been made in several papers, the most straightforward way for continuing this project seems to make a comparison of our own between these two platforms. To avoid redundancy by creating another program, the Canny Edge Detector, which is needed as a preparatory step for object detection (explained in Section 2.3.3.3), will be implemented for both programming interfaces.

3. Setup, Procedure, and Concept

At this point in time, all Transport-by-Throwing research is only at the level of academic research. Therefore, the experimental setup of this work is shaped accordingly and makes use of some simplifications. The transport system examined in this project consists of various parts that can be considered and treated separately: developing a throwing device, setting up the vision system, detecting the thrown object, creating a flightpath prediction, and developing a catching device. Since each of these parts require a huge amount of research and work, this diploma thesis will only deal with an accurate detection and prediction of the flying tennis ball. The advantage of using such a ball lies in its easier detection resulting from its point-symmetrical shape.

3.1 Environment

The ball was synchronously captured by two IDS uEye UI-3370CP cameras that were aligned convergent with a baseline distance of approximately 0.92 m. A coil-based throwing device accelerated the ball to overcome the distance of about 2.5 m. Four 500 W halogen floodlights were used for a sufficient illumination of the scene. A Matlab toolbox [25] was used to calibrate the stereo vision system to achieve accurate results from the triangulation of the two images. Furthermore, the toolbox provided all necessary intrinsic parameters of the vision system: distortion, skew coefficient, etc. [PKH10]. The two cameras support a resolution of 2048-by-2048 pixels at a frame rate of 80 FPS and 2048-by-800 pixels at 110 FPS.

Because of the need for highly accurate detection, processing these large images would need too much computational power and time to predict the flightpath in time. To maintain the advantages of the higher resolution, which leads to more accurate results (see Subchapter 2.4.1), a system is required that acquires only the relevant area of the full image. To retain any important information and be able to determine the ball's exact position, the subimage has to show the ball; in addition, information about the location of the subframe must be provided. Figure 30 shows a possible subframe that is delivered with the parameters x_{offset} and y_{offset}, which provide a back-calculation of the ball's location in the large image.

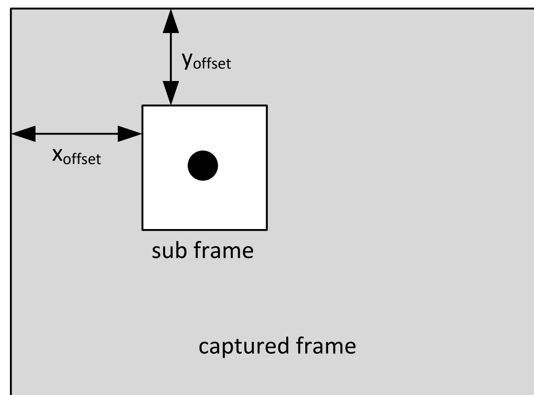

Figure 30: An already existing unit crops the big images to smaller subframes with the ball more or less in the middle. The offset parameters enable the calculation of the ball's center point in the big images.

The cropping system, which is not part of this thesis, already exists and provides output images with 300-by-300 pixels, where the ball is roughly in the center. A technique similar to BLOB detection, which calculates difference images based on a comparison with the scene's background, enables cropping large images to small regions of interest. It would be possible to make a flightpath prediction based solely on BLOB detection. However, this would probably lead to an inaccurate detection, which in turn would result in an insufficiently accurate prediction. In the framework of this thesis, a system that detects the ball in the cropped images and predicts its flightpath will be designed, implemented, and tested.

The Institute of Computer Technology at the Vienna University of Technology provides the hardware for these tasks: A computer with an i7-4770S CPU clocked at 3.10 GHz, 8 GB main memory clocked at 667 MHz and an NVidia GeForce GTX 560 Ti graphics card [26] with 384 cores assigned to 8 multiprocessors clocked at 822 MHz. The GPU features compute capability 2.1, 1024 MB global memory, and 48 KB shared memory per multiprocessor. A 64-Bit version of Windows 7 Enterprise was chosen as operating system and Microsoft Visual Studio 2012 as development environment. The CUDA platform is at Version 6.5 [27] and its OpenCL counterpart at 1.0 [28].

3.2 Tasks

The project described here deals with the following tasks: detection of the ball in both of the 300-by-300 pixels images, its localization in 3D space, and forecast of its flightpath (highlighted in Figure 31).

For reasons of convenience and reproducibility as well as to enable practical implementation, the small subframes of some flights were recorded to analyze them instead of always capturing real-time flights. Without these recordings, it would be necessary to throw the ball each time when testing the written software, which would add an undue amount of work for the developer. Implementing software in this manner has the benefit of easier troubleshooting

when an error occurs. It is easier to find a software failure when it is possible to precisely reproduce the malfunction.

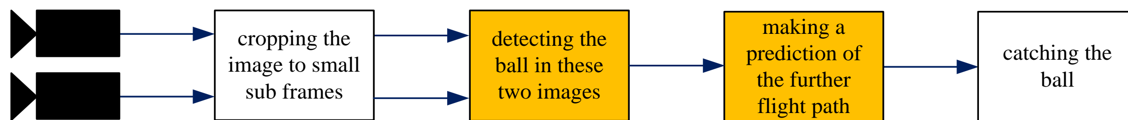

Figure 31: This diploma thesis deals with the detection and prediction of a flying ball. An already existing unit crops the big images to smaller 300-by-300 pixel subframes, in which the ball has to be detected. The prediction of the ball's flightpath will be sent to the robot arm to enable it to catch the ball.

The procedural steps, which are highlighted in Figure 31, can be broken down into multiple smaller tasks: load the trajectory database, load the subframes from the hard disk, convert them to uniform 8-Bit grayscale images, create edge magnitude maps, detect the ball's center on this maps, triangulate the two received center points, and make an estimation of the flightpath. Figure 32 shows the entire procedure that will be implemented in the framework of this thesis.

The following subchapters will show the procedure respectively its implementation step-by-step. A distinction must be made between preliminaries that have to be carried out specifically for the experimental setup and preliminaries that are generally necessary for object detection. The former consist of loading the images and convert them to data that can be processed in the appropriate manner. While these preliminary measures are necessary for this setup, they will probably not be needed for real world applications. In contrast, edge detection, for example, is a necessary preparatory step for accurate object detection and must be executed both in the experimental setup as well as in real world applications.

The focus is on detection, localization, and prediction of the ball, but preliminaries, like loading the picture and converting it will also be briefly explained. Following these fundamentals, the Canny Edge Detector, which was chosen to produce the edge images, will be explained.

Another preliminary measure can be taken to improve the quality of the edge images. In this step, the background of the scene is subtracted so that the resulting image will only display the ball. Removing all objects present except the one to be localized, minimizes the probability of detecting wrong edge points. For this reason, both cameras recorded the scene without the ball. These images were stored with the cameras' maximum resolution (2048-by-800 pixels) to have all the information about the background available. Again, the parameters x_{offset} and y_{offset} (see Section 3.1) are needed as additional information to use the correct 300-by-300 pixels of the big image. Only the ball's pixels should remain when making such a difference image.

Two different techniques will be tested to detect the object: the Hough Circle Transformation and the RANSAC algorithm. On the one hand, it has to be assumed that RANSAC will probably be faster than Hough, but on the other hand, the Hough Transformation will probably provide more accurate results. To validate these assumptions, both have to be implemented. Subsequently, implementation of the localization, which is done through

triangulation, will be explained. The prediction, which constitutes the final step, will be explained in the last subchapter.

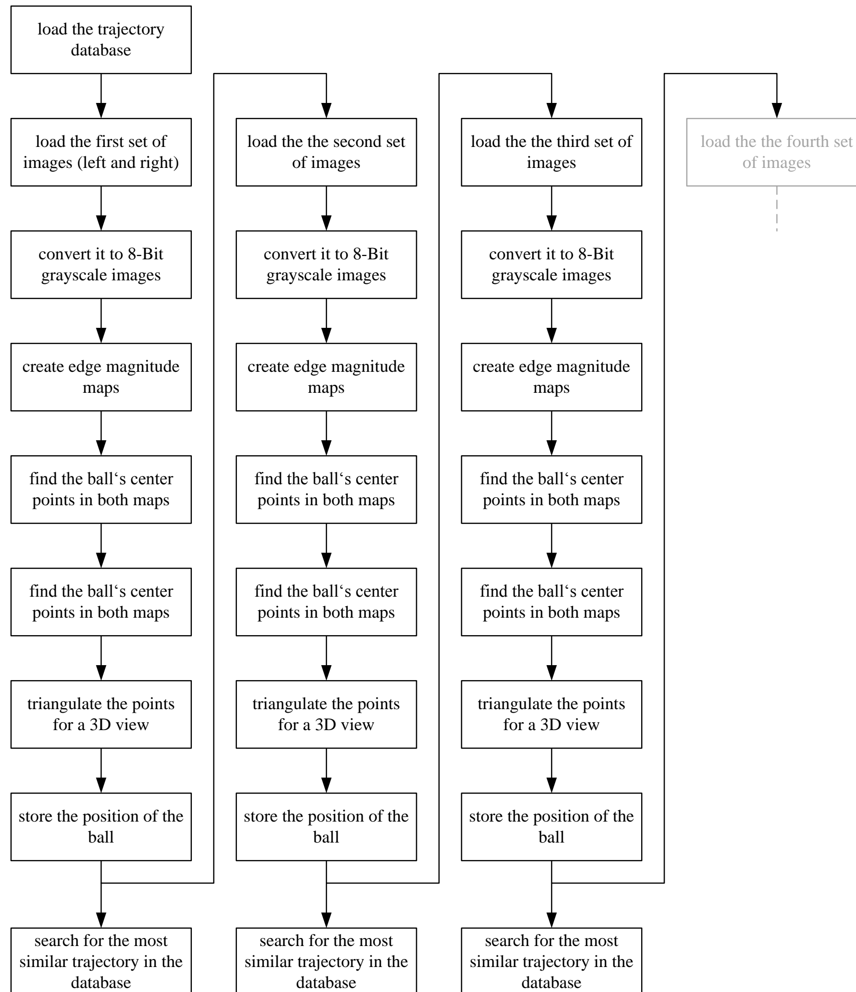

Figure 32: After the trajectory database has been loaded, the first set of stereo frames can be loaded and converted to grayscale images. Afterwards, edge magnitude maps will be created so that the ball's center can be detected. The next step is the localization of its position in the 3D space to examine its movement. The actual flight positions will be compared with the database to find the most similar trajectory.

3.3 Preliminaries

As explained in Chapter 3.2, some steps have to be made to enable localization and estimation of the ball. The following subchapter will give an insight into the implementation of these preliminaries. After all these steps have been carried out, the main tasks can begin.

3.3.1 Loading the trajectory database

The first step is about loading the trajectory database from the hard disk to the main memory of the CPU. The database is stored as a CSV-file (Comma-Separated Values), which makes it a simple task to load it. The only functions needed are in the *stdio*-library (standard input/output) [29], which comes with every normal C/C++ compiler. Subsequently, all trajectories, which are stored in multiple arrays filled with the 3D positions, will be copied from the CPU's main memory to the global memory on the GPU.

An optional procedure would be to load the trajectories when the positions of the flight are already known. However, the accruing latency times, which occur when loading the database from the hard disk to the main memory and then to the global memory, would most probably make an on-time forecast impossible.

3.3.2 Loading two of the flight' s subimages

As stated in Section 3.2, the 300-by-300 pixel subimages have been recorded to provide an easier implementation of the detection software. The images are stored as BMP-files (Bitmap) and can be opened with different libraries that can be downloaded from the Internet. OpenCV [30] is one of the most popular library packages for opening, editing and processing pictures as well as videos. Because of the easy-to-use libraries as well as the vast amount of explanations and tutorials, OpenCV will be used to load the images from the hard disk to the main memory of the CPU. This procedure will be done by an instruction called *imread* [31].

In the case of processing frames from a live scene, a video stream would be directly captured instead of loading stored images from the hard disk. This would probably lead to much shorter and more regular latency times. Therefore, the time needed for moving the required data sets of a frame to the main memory will not be treated in great detail here.

Copying frames from the host's main memory to the GPU's global memory is quite another thing than loading them from hard disk to main memory. This step is always necessary when processing images with a GPU. Therefore, the time needed to do this will be worked out here. This copying task is embedded in another step that will be explained in Subchapter 3.5.1.

3.3.3 Converting the frames to uniform grayscale images

It would also be possible to process the 32-Bit color images, sent by the cropping unit, on the GPU. However, further calculation steps only require images with intensities; colors are not needed. To reduce host-to-device workload and save memory on the GPU, the 32-Bit images will be converted to 8-Bit grayscale images. Another OpenCV function, called *cvtColor* [32], can be used for executing this conversion on the CPU. Subsequently, the 8-Bit grayscale images will be moved to the GPU to be processed in the following computation steps.

Why do we use a CPU to convert the images when a GPU would probably do the job faster? It would also be possible to obtain the images in this grayscale format directly from the cameras. This approach would render conversion unnecessary; therefore, the time needed for converting the frames to 8-Bit grayscale images will be addressed here only marginally.

3.4 Subtracting the scenes background

This step may be executed, but is not mandatory. The idea here is to improve the quality of the edge images, which means reducing the possibility of detecting wrong edge points. The following implementation of the Canny Edge Detector (see Section 3.5) will be the same, regardless whether the background subtraction will be performed or not. However, the input frame for the Canny Edge Detector will be an image in which, more or less, only the ball is present; therefore, the edge images will be of a higher quality.

$$P_S(x, y) = P_I(x, y) - P_{BG}(x + x_{offset}, y + y_{offset}) \quad (3.4.1)$$

Therefore, 2048-by-2048 pixel images, without the ball, have been recorded from the left and the right camera. The resolution of these two images is equal to the highest resolution of the cameras. In other words, having these images is equal to having complete information about the background of the entire scene. To subtract the background (Equation 3.4.1) means to make a difference image of the input image[2] as well as of the background image: the value of the resulting pixel's value P_S with the coordinates x and y is equal to the pixel's value P_I from the grayscale image (in place x and y) minus the pixel's value P_{BG} of the background image (in place $x + x_{offset}$ and $y + y_{offset}$). The offset parameters, which are delivered with the 300-by-300 pixel subimages, are important for this task. Without them, it would be impossible to use the right part of the big image for this subtraction (Figure 30).

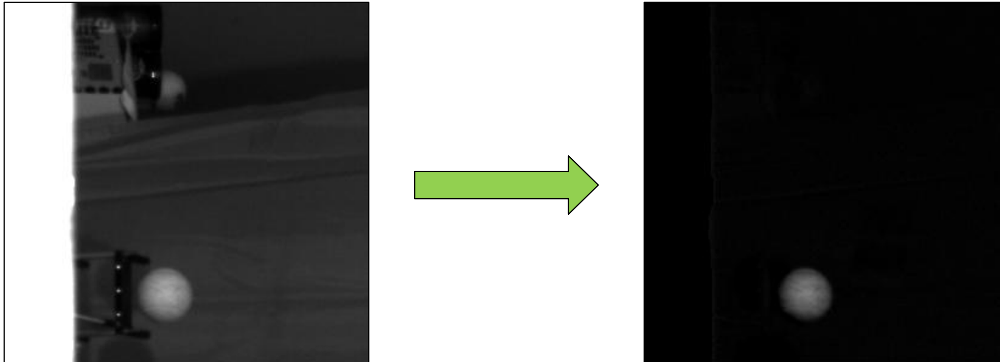

Figure 33: Background subtraction.

Because of changing lighting conditions, the difference image will probably not be perfect. The recorded background image can be slightly brighter or darker than the actual frame with the thrown ball. As shown in Figure 33, there are still contours of other objects in the resulting difference image. However, these remaining contours are very weak and, therefore, it will be no problem to remove them in the Hysteresis step of the Canny Edge Detector, which will be explained in Subchapter 3.5.4.

Additionally, it is important to note that the subtraction of a pixel must not result in a negative number. This would happen if the value of the background image was higher than its counterpart on the grayscale image. Because 2D arrays of Unsigned Char data type are

[2] In the case of performing a background subtraction, the subimages will be transferred from the main to the device memory in this step.

used for storing the subimages, a negative value would cause an overflow of the pixel value. In other words, the pixel in the difference image would not become negative or 0; rather, it would have a very high value if the subtraction's result would be negative. A pixel with such a high intensity would be wrongly interpreted as an edge point. Therefore, subtractions of pixels that would lead to a negative result will not be calculated with Equation 3.4.1, they will set to 0.

3.5 Canny Edge Detector

If no background subtraction is carried out, the Canny Edge Detector will be the first program step executed on the GPU[3]. A GPU can be programmed in two different ways: with CUDA or OpenCL. Based on statements from different papers about the two platforms' performances, the Canny Edge Detector will be implemented for both. The time needed to complete all four steps of this algorithm will be measured to obtain authentic information about possible performance differences. The procedure will be explained in detail in the following subchapters and the results (shown in Subchapter 4.1.1) will determine which interface will be used in the following steps of the prediction algorithm.

3.5.1 Gaussian filter to smooth the images

The Canny Edge Detector's centerpiece, which will be explained in the next subchapter, examines the gradients of all pixels. If the magnitude of a pixel's gradient is high, it will be an edge point. However, a camera frame does not contain only high gradient magnitudes that represent edges. Indeed, superimposed white Gaussian noise may lead to high gradient magnitudes as well. This would disturb accurate edge detection and result in detecting false edge points. Therefore, the Gaussian noise has to be removed as effectively as possible. Therefore, the first step of the Canny Edge Detector is a Gaussian filter whose implementation will be discussed now.

1	2	1
2	4	2
1	2	1

$\frac{1}{16}$

Figure 34: The center field of the Gaussian filter matrix has to lie on the pixel to be calculated. Its new value will be the accumulation of its own initial value multiplied by 4 and its neighbor's values multiplied by 2 respectively 1. The calculated value is then divided by the sum of all factors.

Besides its good performance, the Gaussian filter is also predestinated to run on hardware with parallel architecture. Because of its linear separability in x- and y-direction, it is possible to calculate each pixel value independently of others. In other words: All pixels can be calculated simultaneously and do not have to wait for other results of the Gaussian filter

[3] Therefore, the subimages will be transferred from the main to the device memory in this step.

operation [GSW03][LWT12, 2] [YR08, 3]. The value of a smoothed pixel is the result of its own initial value and the initial value of its direct neighbors. Figure 34 shows an example of a Gaussian filter matrix that will be convolved with the input image. A pixel's value can be calculated when the center field of the matrix lies directly on it. In this example, the new value of the center pixel would be an accumulation of its own initial value multiplied by 4 and the neighboring pixels multiplied by 2 respectively 1. Since the sum of all factors has to amount to 1, the calculated pixel value has to be divided by 16. For the new intensity I_{P2-2} pixel P_{2-2} with the coordinates x = 2 and y = 2, the intensity would be calculated with Equation 3.5.1. The results of all of these pixel calculations will be stored in a 2D array located in the global memory.

$$I_{P2|2} = (I_{P1|1} + 2 \cdot I_{P2|1} + I_{P3|1} + 2 \cdot I_{P1|2} + 4 \cdot I_{P2|2} + 2 \cdot I_{P2|3} + I_{P3|1} + 2 \cdot I_{P3|2} + I_{P3|3}) \cdot \frac{1}{16} \quad (3.5.1)$$

As mentioned in Section 2.4.2, the host program can invoke a kernel on the GPU that starts as many threads as needed. Because of the resolution of 300-by-300 pixels, 90,000 threads have to be created for calculating all pixels simultaneously; every thread is assigned to one pixel.

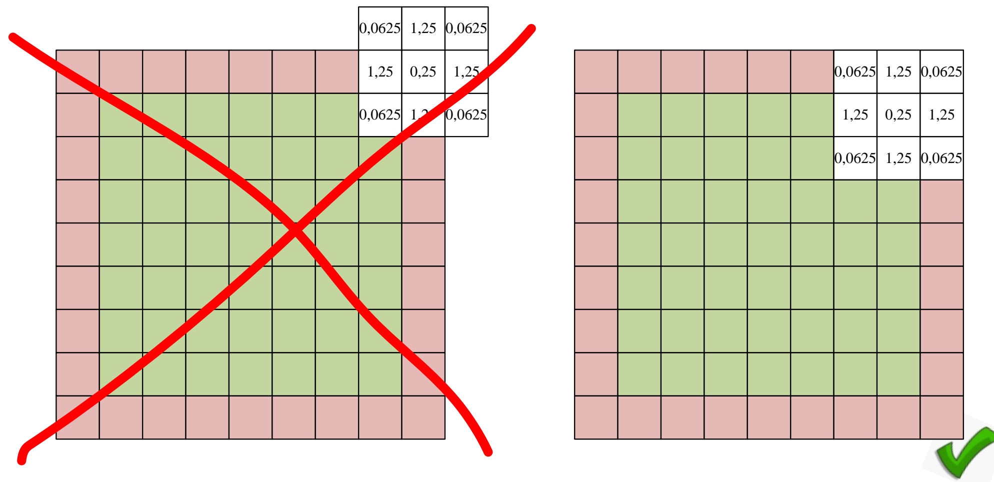

Figure 35: If the matrix' center field lies on one of the outermost pixels, some fields would read undefined memory. Therefore, it is not allowed to calculate the edge lines and columns in the same way.

However, it should be noted that calculating the pixels on the image border with such a 3x3 matrix is not allowed. If the matrix' center field lies on one of the outermost pixels, three of the fields would read undefined memory. When calculating the corner of the image, indeed, five of the fields would read undefined memory (Figure 35). Two options are available to obtain values for the pixels in the outermost lines and columns: inherit the pixel's initial value directly or calculate it with the help of a 4x4 matrix that is placed in a way all fields lie on the image. To consider the outermost lines and columns in the following calculations steps, a 4x4 matrix will be used to process them.

Additionally, to accelerate the procedure, a configuration of a part of the main memory, called page-locked memory, was used that is only supported at compute capability 2.0. A

special conversation between two defined pointers in the program can be made: one of the two points to an address in the host's memory and the other one to an address in the device's memory. This setup qualifies the GPU to access the main memory directly. This access is a slightly slower than the access to the device memory, but it obviates exclusive copying between the two memories.

3.5.2 Sobel operator to create edge magnitude maps

After the image has been smoothed, the Canny Edge Detector's centerpiece will process the image with the so-called Sobel operator, which consists of two 3x3 matrices. Because of its linear separability, it is possible to invoke multiple kernels for every pixel; just like in the previous step. The pixel that should be calculated has to lie in the matrices' center field as well. Therefore, it is not possible to calculate the gradients of these lines. The difference to the Gaussian filter is the use of two matrices instead of only one that produce two different parameters, which represent the derivations in x and y direction for every pixel: G_x and G_y.

G_x

-1		+1
-2		+2
-1		+1

G_y

+1	+2	+1
-1	-2	-1

Figure 36: The parameters G_x und G_y can be calculated by the two Sobel matrices.

As an example, the direction-dependent gradients $G_{x2—2}$ and $G_{y2—2}$ for the pixel $P_{2—2}$ with the coordinates x = 2 and y = 2 would be calculated with Equations 3.5.2 and 3.5.3.

$$G_{x2|2} = -I_{P1|1} + I_{P3|1} - 2 \cdot I_{P1|2} + 2 \cdot I_{P3|2} - I_{P1|3} + I_{P3|3} \tag{3.5.2}$$

$$G_{y2|2} = I_{P1|1} + 2 \cdot I_{P2|1} + I_{P3|1} - I_{P1|3} - 2 \cdot I_{P2|3} - I_{P3|3} \tag{3.5.3}$$

However, an edge magnitude map is required for the following step of the Canny Edge Detector. Therefore, further calculations have to be made to obtain the magnitude —G— of a pixel's gradient (Equation 3.5.4).

$$|G| = \sqrt{G_x^{\ 2} + G_y^{\ 2}} \tag{3.5.4}$$

The right side of Figure 37 shows the resulting edge magnitude map that has to be stored in the global memory for further processing in the next step.

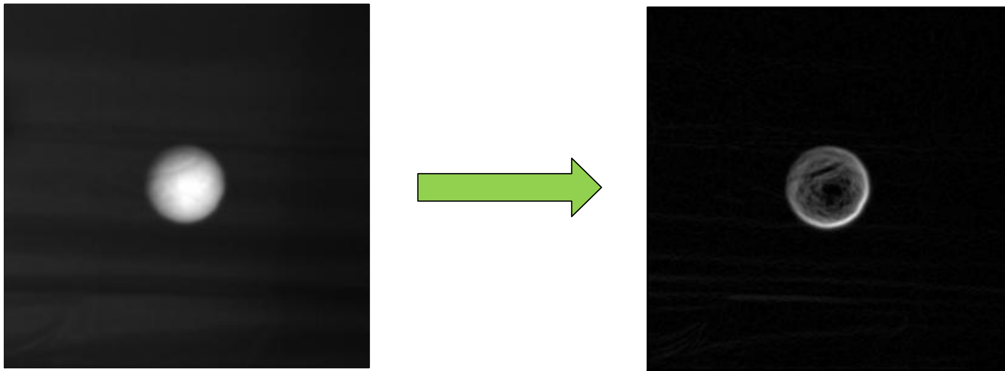

Figure 37: On the left, result of image smoothing, on the right, magnitude edge map.

3.5.3 Non-maximum suppression to thin the edges

As Figure 37 shows on the right side, the Sobel operator outputs edges that are too thick. This is the result of bad lighting, a rather vague focus, and, of course, the first step of the Canny Edge Detector, which smoothed the image. Since clear and thin edges are required for detecting the object as accurately as possible, non-maximum suppression has to be performed. The pixel's gradients have to be reconsidered to "thin" the edges. However, this time the direction, which is represented by the angle Θ, has to be known and can be calculated with Equation 3.5.5.

$$\Theta = \arctan\left(\frac{G_y}{G_x}\right) \tag{3.5.5}$$

When a pixel's gradient is fully known, the pixel's magnitude can be compared to the ones in positive and negative direction of the gradient to find a local maximum of intensities. A maximum found in this way will be preserved for further computation, the others will be suppressed. The result, which is again calculated simultaneously by as many threads as pixels are contained in the image, is shown in Figure 38.

Figure 38: The non-maximum suppression function leads to thinner and more precise edges.

3.5.4 Hysteresis to track edges

The hysteresis function is the final step of the Canny Edge Detector and will be explained next. Another look at Figure 38 shows that some pixels are already present although they should not count as edge points. On the other hand some pixels that should count as edge points have too little intensity. Therefore, the goal of this step is the perfection of less marked pixels and the removing of false edge points.

Two predefined thresholds will pass judgment on the given edge points. Is a point's intensity less than the lower threshold, it is not an edge point and will be removed. However, is the intensity bigger than the higher threshold, it is definitely an edge point and will be preserved. If the intensity is between the two thresholds, the pixel is a potential edge point and will be upgraded to a definite edge point if it is either directly or indirectly, that is, through other potential edge points, connected to a definite edge point. As stated in Subchapter 2.1.1, lighting conditions change during the flight: The ball reflects more light when it is closer to a light source and less when it is further away. Therefore, the threshold values will be changed in the course of a ball's flight.

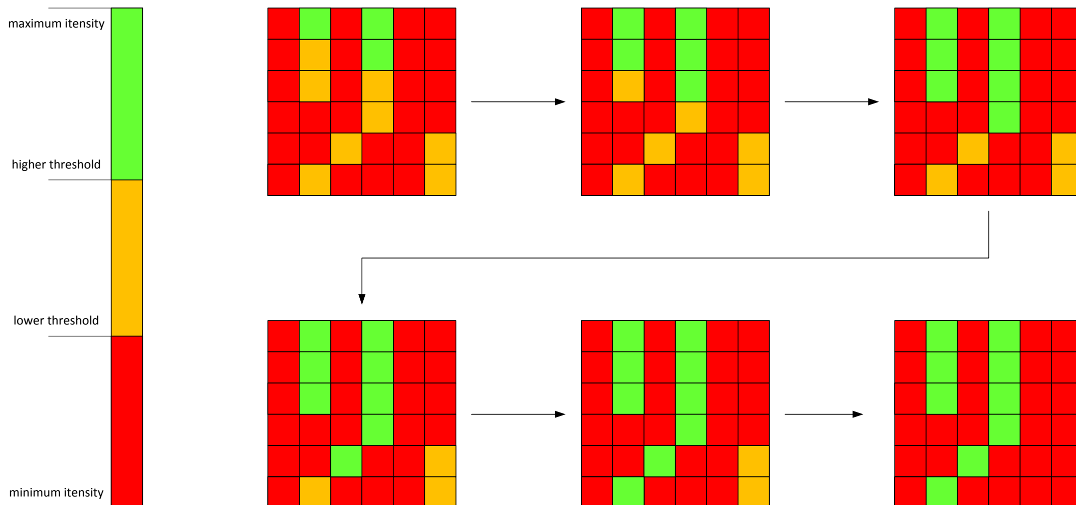

Figure 39: Hysteresis is an iterative procedure whereby all pixels are divided into three groups. If the intensity of a pixel is above the higher threshold, it is a definite edge point and will persist. If it is below the lower threshold, it is not an edge point and will be removed. If the intensity is between the two thresholds, the pixel is a potential edge point and will be upgraded to a definite edge point if it is either directly or indirectly, that is, through other potential edge points, connected to a definite edge point.

The hysteresis function (Figure 39) is an iterative procedure whereby all pixels' intensities will be checked. Here again, one thread will be used for each pixel. If the intensity of a pixel is above the higher threshold, it will enter a queue, in which all definite edge pixels will be registered[4]. Subsequently, the next kernel will be started and will invoke as many threads as there are definite pixels in the queue. Each of these pixels will be set to maximum intensity[5] and their neighbors will be checked. If the intensity of a neighboring pixel is bigger than the

[4]To improve performance and save time, the first queue will be filled in the non-maximum procedure.
[5] The maximum intensity value in an 8-Bit grayscale image is 255.

lower threshold, it will be upgraded to a definite pixel and registered in a new queue. If the intensity is below the lower threshold, its value will be set to 0 and will not be registered in the queue. Now a new iteration can start, and every neighbor of the new definite pixels will be checked. The procedure ends when no pixels are registered in a queue (the result is shown in Figure 40).

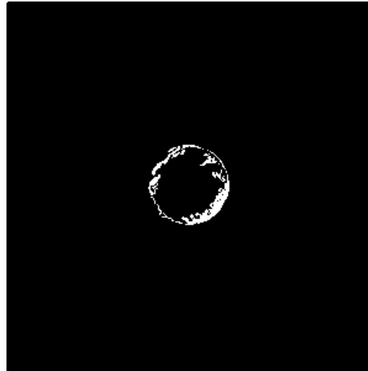

Figure 40: The image shows a result of hysteresis, which constitutes the Canny Edge Detector's last step.

Attention should be paid to the fact that neighbors of a definite pixel have to be ignored if they have already been checked. Their reexamination would lead to an endless loop.

3.6 Hough Circle Transformation

When an edge image is available, object detection can start. This subchapter deals with the implementation of one of two different detection algorithms: the Hough Circle Transformation. This technique usually achieves highly accurate results, but it can be computationally costly and time-consuming. A faster version of the Hough Circle Transformation, whereby the center is detected through lines perpendicular to the tangents, was suggested in Subchapter 2.3.3.4. However, because of the partially blurred recordings and, hence, moderate edge images, this algorithm will not be applied. Its use would probably lead to an inaccurate detection, as happened in [SPV05]. There, only 90 percent of wrong detections could be avoided. Furthermore, such implementation will not fit perfectly on a parallel architecture.

To enable detection as accurately as possible, the "normal" Hough Circle version was chosen. The studies [AME13][CJ11] describe two different approaches to this "normal" Hough Circle Transformation: the Straightforward Strategy (see Section 3.6.1) and the Inverse-checking Strategy (see Section 3.6.2). Both of them will be implemented and tested to identify the strategy that better meets the timing requirements for Transport-by-Throwing.

3.6.1 Straightforward Strategy of Hough Circle Transformation

At the beginning of this algorithm, a kernel invokes as many threads T as pixels make up the image multiplied by the range of radiuses (Equation 3.6.1), which is the same number as the

Hough Space has fields. To save memory and execution time, the radius range was set to 10. However, since the ball flies in the direction of the cameras, the minimum and maximum radiuses that have to be considered will change with the progress of the flight. This ensures the searching for circles of the correct size.

$$T = P_x \cdot P_y \cdot (r_{max} - r_{min} + 1) \tag{3.6.1}$$

Figure 41 provides an example of how the Hough Space could look if the image's resolution would be 6-by-4 pixels and the circle would have a radius between 2 and 4 pixels.

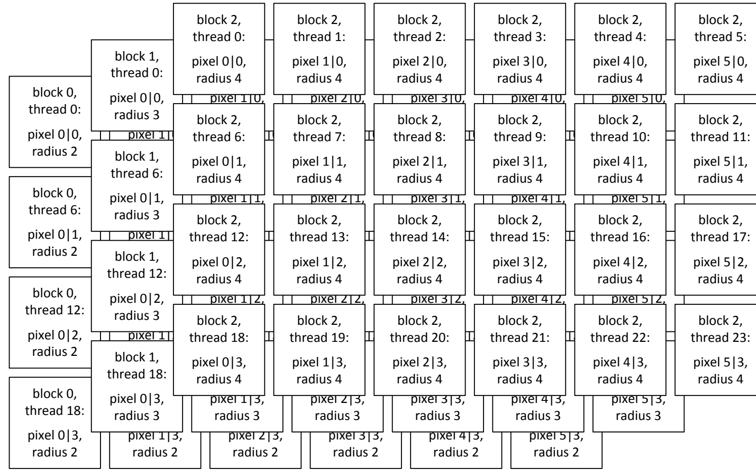

Figure 41: An example of the Hough Space for an image with the resolution of 6-by-4 pixels and a wanted circle with a radius between 2 and 4 pixels.

Each of these threads is assigned to one pixel and starts its procedure by examining the pixel's intensity. If the pixel is an edge point, the hysteresis function will set its intensity to the maximum value. Therefore, examining the intensity of a pixel means checking whether it is an edge point or not. If the pixel is not an edge point, the assigned thread will terminate itself instantly. However, if the pixel is an edge point, an iterative algorithm will start: the thread "goes" in a circular course around this pixel and votes all fields (in increments of 1) in the distance of the thread's assigned radius. The Hough Space is represented by a 3D array (Figure 42) where the multiple planes outline the multiple radiuses. For example: A circle with a radius between 2 and 4 pixels shall be found in the image with a resolution of 6-by-4 pixels. When considering Equation 3.6.1, a Hough Space with 6 x 4 x 3 fields will be needed. For radius 2 every edge point will vote around itself in the zeroth plane (z = 0) of the Hough space array, for radius 3 in the first plane (z = 1) and for radius 4 in the second plane (z = 2). Because of the possibility of two threads "wanting" to increment the same field at the same time, a solution has to be found for dealing with this critical section. Therefore, atomic functions have to be used to fulfill requirements of mutual exclusion. If normal functions were used, the voting process could be totally wrong.

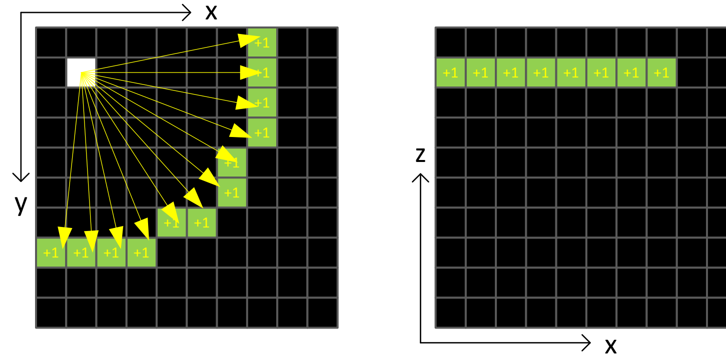

Figure 42: Straightforward Strategy: On the left, some votes in the Hough Space when looking on the xy-plane of the 3D array, on the right, the same votes when looking on the xz-plane.

After all edge points have voted for the multiple radiuses, the maximum value in the Hough Space has to be found since it represents the estimate of the circle's center point. Due to the maximum number of 1024 threads per block [CUD14, 10], many blocks are necessary to have sufficient threads available for the voting process. It is not possible to synchronize different blocks with each other [SW10, 1]. Therefore, it is necessary to terminate the voting kernel after all votes were given and start another kernel for finding the maximum in the Hough Space. The algorithm for finding it will be explained in Subchapter 3.6.4.

It would be also possible to iterate over all radiuses of interest instead of creating so many threads. However, this would lead to a loop that is nested in another one: the outer loop for iterating over all radiuses and the inner loop for going in a circular course around the edge point. This amended version of the voting process would most probably greatly decelerate the procedure. Therefore, it will not be implemented.

An alternative version would be if a kernel counts all edge points and puts them in a queue to let these points vote in the next step. On the one hand, filling up such a queue would lead to more global memory accesses, which takes a bit of time, on the other hand, a lot less threads would be necessary for the procedure. The first kernel invokes as many threads as pixels are in the image. These threads put the pixel coordinates in the queue if it is an edge point. The following kernel invokes as many threads as there are points in the queue multiplied by the radius range (Equation 3.6.2). That means that the second kernel has to process only edge points and not all the pixels.

$$A_t = B_q \cdot (r_{\max} - r_{\min} + 1) \tag{3.6.2}$$

When comparing Equation 3.6.1 with Equation 3.6.2 and neglecting the additional memory accesses, this version of the voting process will be faster when following Equation 3.6.3 is fulfilled.

$$Bq < Px \cdot Py \tag{3.6.3}$$

In other words, it can accelerate the procedure if there are a lot less edge points than pixels in the image. This version will also be implemented and tested to obtain information about the performance differences in the project on hand.

3.6.2 Inverse-checking Strategy of Hough Circle Transformation

As mentioned in the previous subchapter, when pursuing the Straightforward Strategy, every thread that processes an edge point votes around itself for its possible center point. In other words: every edge point "thinks" that it belongs to a circle and "tries" to find its center point.

When using the Inverse-checking Strategy, the voting process is in the opposite direction. Every field in the Hough Space, which represents a pixel with a defined radius, "thinks" that it is the center point of a circle with this radius r and "counts" its assigned edge points (Figure 43). The advantage of this method is that each thread votes only in its own field. Therefore, no atomic operations will be needed to fulfill the requirements of mutual exclusion for critical sections. Avoiding such operations greatly accelerates the voting process.

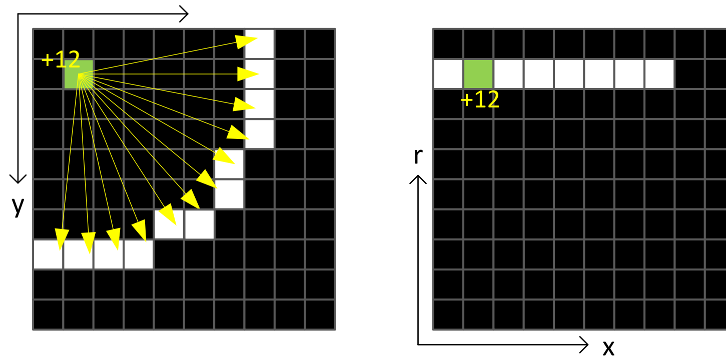

Figure 43: Inverse-checking Strategy: On the left, some votes in the Hough Space when looking on the xy-plane of the 3D array, on the right, the same votes when looking on the xz-plane.

For the Inverse-checking Strategy as many threads are needed as for the Straightforward Hough version. The drawback of this strategy is the fact that every field invariably has to iterate through the various points, which represent the possible circle around. On the other hand, the Straightforward Strategy's kernels only start this loop if their field is an edge point. Studies [AME13, 220][CJ11] stated that the Inverse-Checking Strategy is faster than the Straightforward version. However, I assume that it will be faster only if there are many edge points in the image, so that the Straightforward version would invoke many threads that have to iterate through all points of the circle.

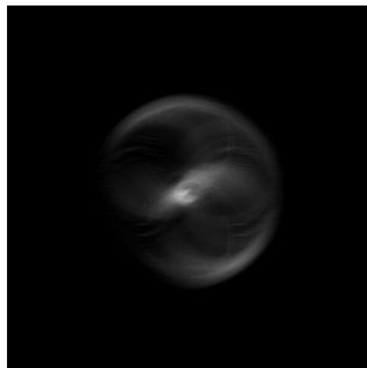

Figure 44: All various radiuses' planes of the Hough Space summed to one 2D image.

Figure 44 shows the Hough Space when all different radiuses' planes of the Hough Space are summed to one 2D Hough Space. In other words, this image shows the Hough Space in a way for giving a better understanding, but should not be used for finding the circles center.

3.6.3 Voting in a circular course around an edge point

As it is necessary to vote, the fields of the Hough Space have to observe or vote around themselves. Therefore, a loop will be started where all coordinates of these circle pixels (for a given radius) will be calculated. There are two different approaches to calculating all these coordinates along a circle: using trigonometric functions or the Pythagoras theorem. No studies have been found that compared these two approaches on a GPU. Therefore, both will be implemented and tested.

3.6.3.1 Drawing a circle with trigonometric functions

The left side of Figure 45 shows that the trigonometric version of "drawing" a circle consists in a loop that iterates over the variable theta (Θ). Equation 3.6.4 and Equation 3.6.5 will be performed inside this loop to calculate the coordinates of circle points based on the angle Θ .

$$x = r \cdot \cos(\Theta) \tag{3.6.4}$$

$$y = r \cdot \sin(\Theta) \tag{3.6.5}$$

However, one important fact has to be considered to guarantee a fair voting procedure: Variables x and y, which represent coordinates, are discrete values and of the integer data type. However, the trigonometric functions return floating point values that have to be rounded to fit the results (Equations 3.6.4 and 3.6.5) in the coordinate variables. The smaller the circle, the more often the same coordinates will be calculated (Figure 45, right). To avoid voting more than once during an iterative step, the newly calculated coordinates have to be compared with the previous ones and ignored if they are the same. Additionally, when the radius is very long, the iteration steps have to be finer than an increment of 1. Otherwise, some pixels could be skipped through roundings (Figure 45, right).

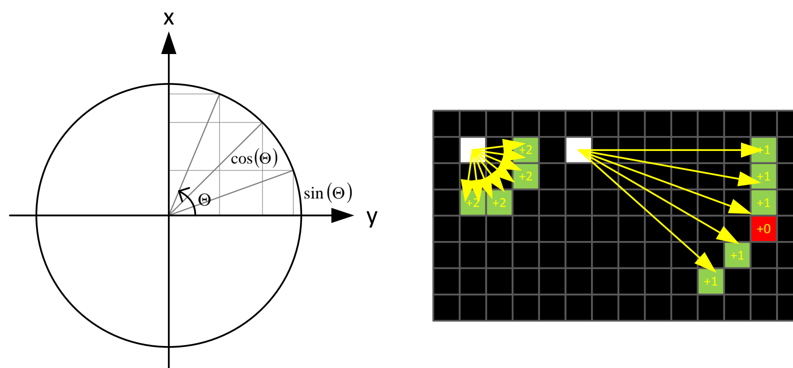

Figure 45: On the left, calculation of the circle point's coordinates, on the right, multiple and skipped votes.

3.6.3.2 Drawing a circle with the Pythagoras theorem

The Pythagoras version consists of a loop, which iterates over the coordinate x. In this loop, the Pythagoras theorem (Equation 3.6.6) calculates the coordinate y for every x to navigate to each of the circle's points (Figure 46, left).

$$y = \sqrt{r^2 - x^2} \tag{3.6.6}$$

Because of the iteration of one of the two coordinates, this version has the advantage of not calculating the same coordinates twice as was the case with the trigonometric functions. However, some fields will be skipped when the circle is big enough so that some adjacent points have the same x coordinates (Figure 46, right). To overcome this drawback, the actual y coordinate has to be compared with the previous one. If the difference of the two y coordinates is greater than 1, additional points have to be "drawn".

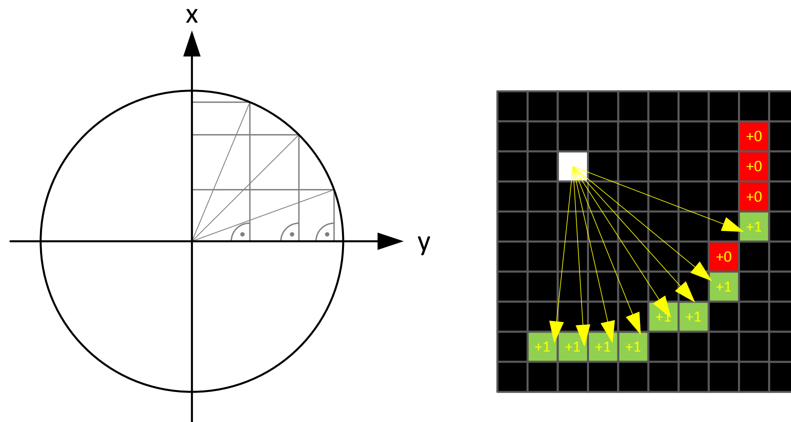

Figure 46: On the left, calculation of the circle point's coordinates, on the right, skipped votes.

3.6.4 Finding the maximum value in the Hough Space

Finding the maximum is the final step of the Hough Circle Transformation and two approaches will be implemented. This step can be done in two different manners: by using the global memory or the shared memory. Since using the global memory alone is easier to implement, the shared memory can greatly accelerate the program [CUD14, 12, 187f].

3.6.4.1 Parallelized maximum search algorithm based on global memory

Since threads from different blocks cannot get synchronized, two different kernels are required. Figure 47 shows the flowchart of the algorithm for an example when searching the maximum value inside the Hough space (with reduced number of threads and blocks). The first kernel starts as many threads as the Hough Space has fields and each of these threads compares its value with a global maximum variable that has been initialized with 0. If the thread's own value is greater than the actual stored maximum, the thread will set this variable to its own value, which represents the actual maximum. An atomic function for comparing and updating has to be used to fulfill the requirements of mutual exclusion.

After each thread had the chance to update the maximum value, the kernel terminates and another kernel starts. It invokes as many threads as the last one and now these threads compare their own value with the maximum variable. The field that has the same value as the maximum variable represents the maximum of the Hough Space and is, therefore, the center point of the wanted circle.

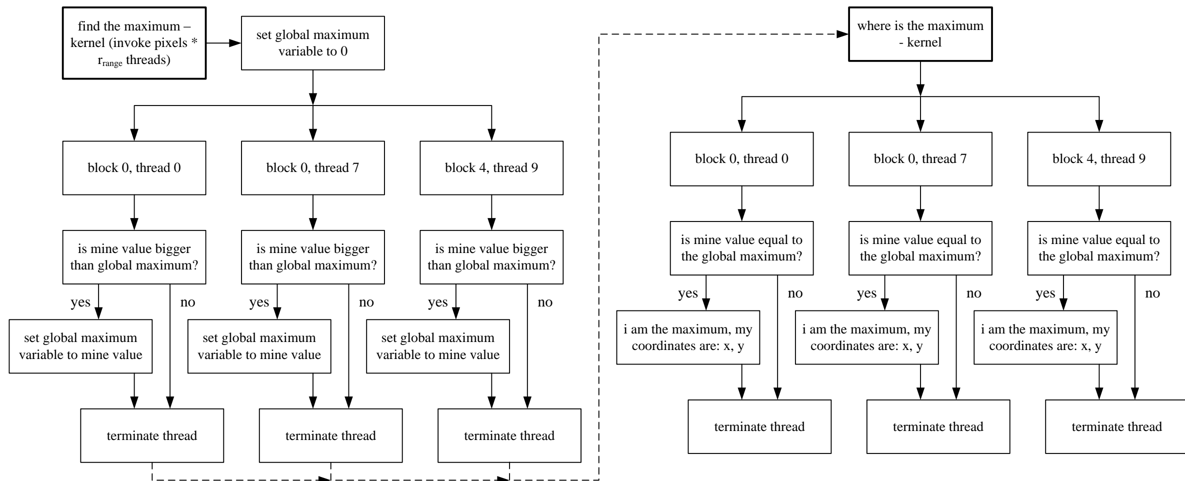

Figure 47: Flowchart of the algorithm for finding the maximum value in the Hough Space (based on global memory). The number of blocks and threads are just examples. In reality, there will be much more of them. The numbering of these items is also arbitrarily chosen, other blocks and threads could be invoked.

3.6.4.2 Parallelized maximum search algorithm based on shared memory

However, this procedure can be easily accelerated through the use of the shared memory (an example with reduced number of blocks and threads is provided in Figure 48). This memory has the disadvantages of being much smaller and only accessible from the same block. Therefore, the shared memory can only be used to determine the local maximum from a block. Then again, it is much faster than the global memory. Every block has its own shared maximum variable which will be set to zero at the beginning of the procedure. Subsequently, each thread compares its own value with the maximum variable in the block's shared memory. If the thread's own value is greater than the shared variable, the shared maximum will be updated. Since the access to the shared variable is also a critical section, comparing and setting must be done in an atomic function as well. Afterwards, a synchronization command forces all threads of the block to wait for each other. This must be done to be sure that every thread of the block had the chance to set the shared maximum variable.

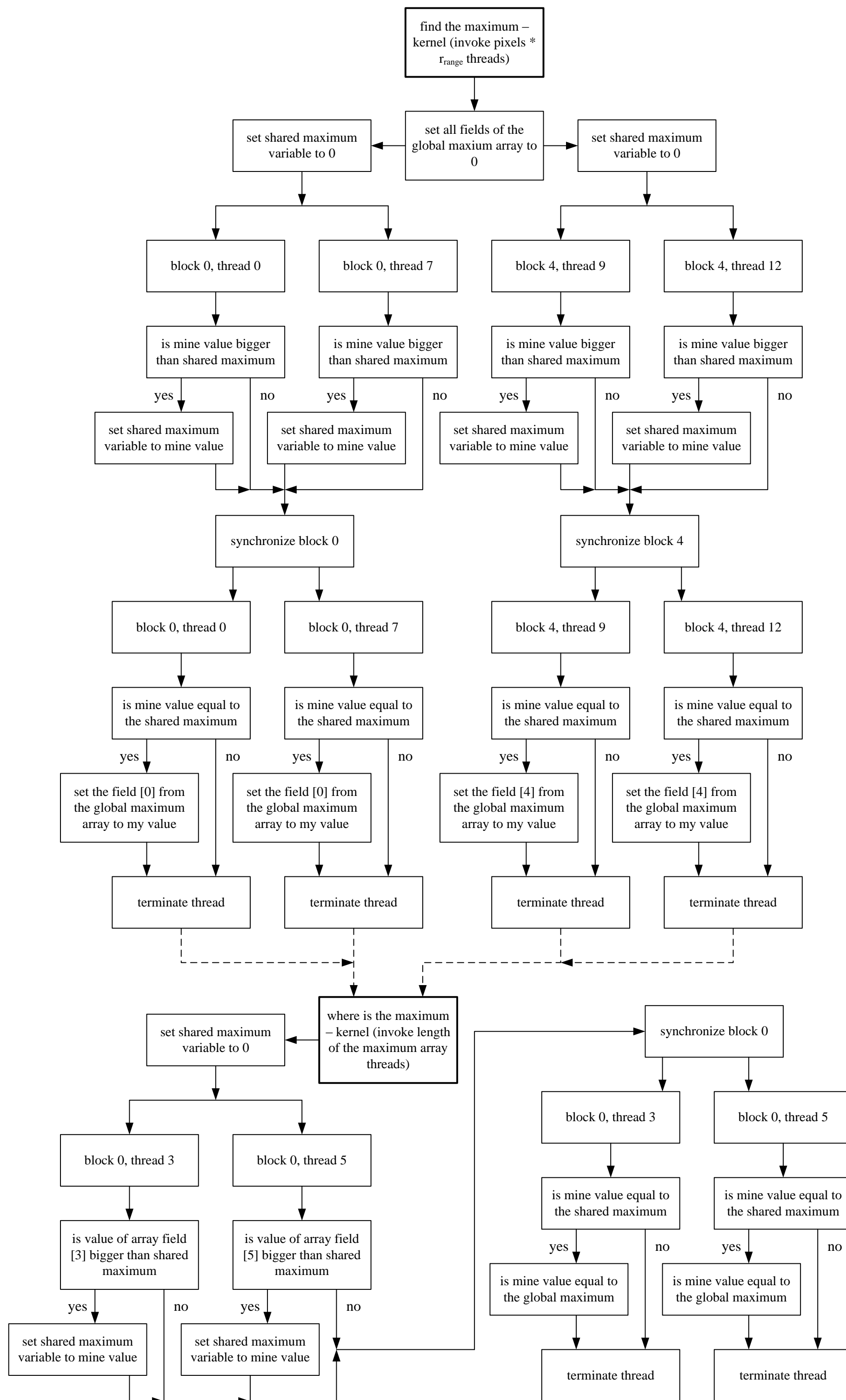

Figure 48: Flowchart of the algorithm for finding the maximum value in the Hough Space (based on shared memory). The number of blocks and threads are just examples. In reality, there will be much more of them. The numbering of these items is also arbitrarily chosen, other blocks and threads could be invoked.

In the next step each thread again compares its own value with the shared variable. If it is the local maximum, it will save its value in a field (with index being equal to the block id) of a global array. That means that block 0 will save its maximum value at field 0, block 1 at field 1, and so on. Subsequently, a new kernel invokes as many threads as blocks had been in the previous step, which is the number of filled fields of the global array. If it is less than 1024, only one block will be created, and the maximum of the entire Hough Space can be found again with the help of the shared memory. If more than 1024 threads are needed, all (new) local maxima will be written in a global array again. This is repeated until all remaining local maxima fit in one block so that the maximum of the Hough Space can be found.

As explained above, the shared maximum variable has also been processed with atomic functions, which serialize some parts of the algorithm and slow down the procedure. However, the various blocks do not share the same variable and, therefore, they can process it totally independently from other blocks. It is assumed this algorithm will be faster than the first one. To validate this assumption, the times of both algorithms will be compared.

3.7 RANSAC algorithm

This subchapter deals with the implementation of another detection algorithm that will also be carried out in the framework of this thesis: the RANSAC algorithm. This detection algorithm requires an edge image as well. Whereas the Hough Transformation can be computationally costly and its processing time depends on the number of edge points present in the picture, the RANSAC algorithm should be much faster and its processing time depends on the number of iterations done to find the best-fitting circle.

The first question to a developer should concern the number of blocks and threads respectively the extent of tasks to be executed in a thread. It can be assumed that the Canny Edge Detector (see Section 3.5) will output edge images of high quality if both threshold parameters are set properly. At least, good edge images will be created if the background of the scene is subtracted (see Section 3.4) from the original image before the Canny Edge Detector begins to work. The better the quality of such an edge image, the less false edge points will be in this picture, the less iterations will be needed to find the ball's center.

As mentioned in Subchapter 2.4.2, a block can consist of up to 1024 threads, and threads of the same block are able to share information among each other with the help of the shared memory. Accessing this shared memory takes a lot less time than accessing the global memory, which is accessible from all threads, independent of the block they are associated with. It seems obvious that the various trials to find the circle should be processed parallel instead of in sequential iterations. Therefore, the term trial will be used from now on. Each of these trials will be performed by a thread of its own, which outputs a possible center point of the wanted circle. After all suggestions have been made, the best-fitting circle will be picked and the coordinates of its center point chosen as those of the wanted circle. Performing the RANSAC algorithm with a number of trials (threads) less or equal to 1024 will lead to a demand for only one block and only one kernel. Therefore, finding the best-fitting circle can be done without using the global memory and will greatly accelerate the entire procedure. In

contrast, the Hough Transformation (see Subchapter 3.6.1) needs as many threads as pixels make up the image multiplied by the range of radiuses. Because of the huge quantity of threads, many blocks will be needed and another kernel has to be started to find the maximum in the Hough Space. This forces the developer to make the algorithm use the global memory. The next subchapters show the implementation of the individual RANSAC steps.

3.7.1 Selecting randomly three edge points

At the start, three edge points have to be randomly selected. Therefore, a queue is needed in which all edge points are registered. This preliminary work will be done by a kernel that invokes as many threads as pixels are in the image. If a thread's pixel is an edge point, it will registered in the queue, if not the thread will be terminated instantly.

indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
0	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	...
1	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	

Figure 49: All edge points' coordinates will be registered in a queue.

Figure 49 shows a 2D array that represents such a queue: one dimension stands for the various edge points and the other dimension for the coordinates of these points. To choose randomly one of these points means to generate a random number between 0 and the highest index of the queue. Generating the random number r leads to picking the r'th point, with the coordinates x_r and y_r, of the queue.

However, it is more complicated to generate a random number on a GPU than on a CPU. First of all, the *curand* [33] library has to be installed and included in the program to use it. Like in "normal" C programs, an initialization function has to be called before it is possible to generate random numbers. More precisely, the initialization function in a host program is not necessary for generating a number, but it enables a quasi-random generation. Without the initialization function, the generated numbers will only be pseudo-random numbers, which means that the numbers and their sequence will always be the same. On the other hand, the initialization function is necessary to get a random number on a GPU, regardless of the type of numbers: quasi-random or pseudo-random. This function, called *curand_init*[6], needs a variable of the *curandState* data type, which is used to save the state of the *curand* function that generates the random numbers. Because of the demand of three random numbers per trial respectively thread, an array of these states will be allocated with as many fields as trials will be performed. Additionally, the initialization function decides on the random-number generator to be used and, in further consequence, on the type of random numbers. Pseudo-random numbers are easier to handle and, therefore, they will be used in this

[6] This has to be done only once when starting the program. The generator will output 2^{67} random numbers before the sequence of numbers will be repeated. The huge amount of random numbers is above and beyond sufficient for testing this algorithm in the framework of this thesis.

experimental setup [33]. However, it is possible to make the pseudo-random generator behave similarly to its quasi-random counterpart. A so-called *seed*, which is an arbitrary value, has to be defined when initializing the random generator. All generated values will be based on this seed, which means that values will always be the same when initializing the generator with the same seed value. To overcome this drawback without using another more complicated quasi-random generator, the seed will be generated by the *rand* function [34] in the host program. The initialization of this random number generator will be based on the actual date and time to guarantee a quasi-random-generated seed for the GPU function.

$$r = (r_{0-1} \cdot 10^n) \bmod P_E \tag{3.7.1}$$

After the initializing step has been completed, the *curand˙uniform* function can be executed to generate a floating point value $r_{0\text{-}1}$ between 0 and 1. Subsequently, this floating point value has to be converted to an integer r between 0 and the queue's highest index that represents the amount of the image's edge points (Equation 3.7.1). First, the random number $r_{0\text{-}1}$ will be multiplied by 10^n, which means that the decimal point of the floating point is shifted n digits to the right. This leads to a random number between 0 and 10^n when neglecting the digits after the decimal point and, therefore, the lower bound is set. To set the upper bound, the result of this calculation has to be divided by the quantity of queue members. However, the remainder of this division is the needed random number, not the result of the division itself. Therefore, the modulo operator will be used, which outputs this remainder.

3.7.2 Calculating the circle that is described by these three points

Each point around the periphery (represented by the coordinates x and y) of a circle is related to the circle's center point (represented by the coordinates x_c and y_c) through Equation 3.7.2.

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \tag{3.7.2}$$

Since it is possible to describe a circle through three points, three random numbers have to be generated to point at three different fields of the queue's array. With the help of these points, a system of three equations with three unknowns can be set up (Equation 3.7.3, Equation 3.7.4, and Equation 3.7.5).

$$(x_1 - x_c)^2 + (y_1 - y_c)^2 = r^2 \tag{3.7.3}$$

$$(x_2 - x_c)^2 + (y_2 - y_c)^2 = r^2 \tag{3.7.4}$$

$$(x_3 - x_c)^2 + (y_3 - y_c)^2 = r^2 \tag{3.7.5}$$

It is obvious that rearranging and transforming these three equations will lead to results for x_c, y_c, and r.

3.7.3 Drawing a circle with these three points

In the next step, a circle with the parameters calculated in the previous subchapter, can be "drawn". More precisely, each thread draws such a circle with different points (Figure 50)

that were selected in Section 3.7.1. However, it is not necessary to save these drawn circles in the memory as was the case with the Hough Transformation. Figure 50 shall only serve for a better understanding.

As explained in Subchapter 3.6.3.1 and Subchapter 3.6.3.2, drawing a circle can be accomplished either with trigonometric functions or the Pythagoras theorem. The same applies to the RANSAC algorithm; both will be tested and the faster approach chosen.

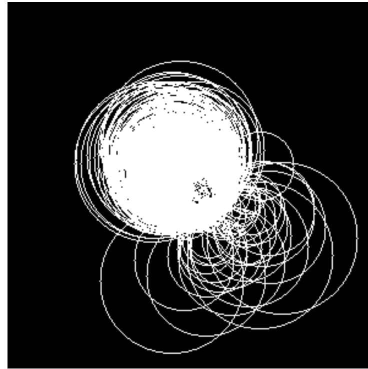

Figure 50: Various possible circles from different trials.

Figure 51 shows the example of an edge image and the circle's dependence on the selected edge points. Because of noise respectively the detection of wrong edge points in the image, the possibility for detecting a wrong circle is higher when the selected points are close to each other (left side of Figure 51). Introducing a minimum distance between the three selected points would reduce wrong detection caused by noise and "unclean" edge images (right side of Figure 51). This would require to calculate the Euclidian distances among all three points: P_1 to P_2, P_2 to P_3, and P_1 to P_3. If the distance between two points is lower than a predefined value, the trial will not be valid.

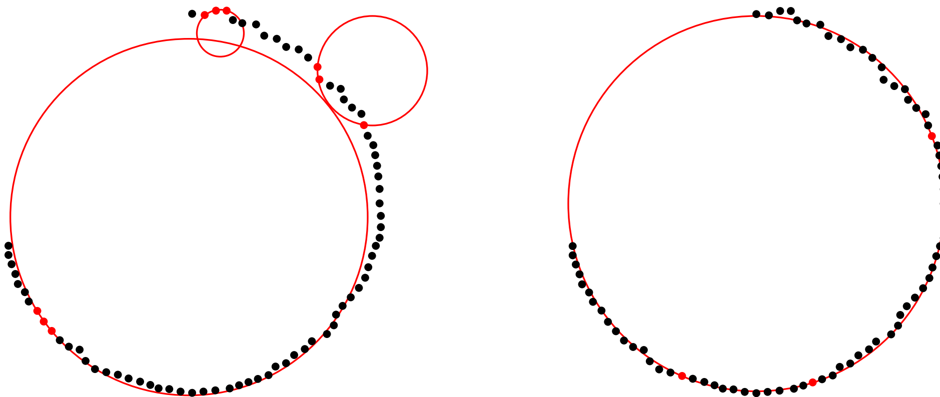

Figure 51: Selecting points that are close to each other increases the likelihood of detecting the wrong circle.

Another approach to circumventing selection of points that describe a wrong circle would be a provision requesting a radius in a predefined range. The drawback of this approach is the need to calculate the center point and the radius regardless of whether the points will be

used. However, because of the amount of edge points and their distribution, it is assumed that the likelihood of selecting three far off points is higher than three points that are close to each other. Therefore, in sum, calculating the Euclidian distances for all three points and subsequently calculating center point and radius will take more time than calculating only the center point and the radius even if it might be in vain. Because of these considerations, the radius will be examined instead of the Euclidian distances to detect the ball as fast as possible.

Furthermore, it would be possible to repeat the process of finding three points if they do not meet abovementioned requirements. However, this could lead to a distinctly longer processing time if one or more trials kept failing to meet these requirements and the thread would be forced to repeat selection again and again. Therefore, a thread that picked wrong points will be terminated instantly in this implementation; with the consequence of losing as many trials as wrong selections have been made. Still, this simplification can be made because the possibility for one trial making wrong selections many times in a row is much higher than all trials making wrong selections given that there will be numerous trials (1000 or more). Nevertheless, the percentage of valid trials relative to the total amount of trials will be examined to ensure that not all trials will be terminated.

3.7.4 Finding the best-fitting circle

The last step of the algorithm is to find the drawn circle that fits best to the circle of the edge image. Therefore, every thread (trial) counts the inliers, which represent the points of the drawn circle that match the edge points. Normally, this is done by calculating the Euclidian distances between all points of the drawn circle and all edge points. As described in Subchapter 2.3.3.5, an edge point will be counted as inlier if the distance is less than or equal to a predefined value. If the distance is higher, the edge point is an outlier. The circle with the most inliers will be chosen.

However, here a further simplification will be introduced. A GPU that features compute capability 3.x would enable calling a kernel out of another kernel. This means that every point of the drawn circle could start as many threads as edge points are contained in the image to enable parallel examinations of the distances. The GPU used in this framework features “only” compute capability 2.1 and, therefore, each point of the drawn circle would have to start a loop to examine all distances to the various edge points. It is obvious that two nested loops would take too much time. Therefore, each point of a drawn circle checks whether it lies directly on an edge point or not. A local inlier count variable will be incremented by 1 for every matching point.

For a better understanding, Figure 52 shows the complete RANSAC procedure as explained in the previous subchapters.

Figure 52: Flowchart of the RANSAC algorithm.

3.8 Obtaining the object' s 3D position

Knowing about the ball's movement in space and time is essential for predicting its further flightpath. Until now, all the previous steps "only" had the task to detect the ball in 2D images. However, this does not predict the ball's movement in space. Nevertheless, the outputs of the previous detection procedures are absolutely important for this step: the so-called triangulation makes it possible to convert the information of the 2D images from the left and right camera to the wanted 3D view of the scene. This procedure is very similar to the way humans see 3D. Since it already exists as a Matlab implementation, it could be readily applied for the purposes of this thesis.

First of all, the coordinates of the ball's center in the left and right images are needed; without them, triangulation will not work. Therefore, the coordinates of the maximum field of the Hough Space (Figure 40) or the center point's coordinates of the best-fitting circle from a

RANSAC trial (Figure 50) are required. Whether Hough is used or RANSAC, triangulation will work either way. However, the results of the 3D position can be slightly different. As stated in Subchapter 2.3.3.6, these detection procedures are totally different and lead to different results, which means that they can be more precise or less. The accuracy of a calculated 3D position depends on the accuracy of the results from the center point detection in the 2D images.

Additionally, one needs to pay attention to the following: the coordinates resulting from both of the detection algorithms describe the ball's position in the small 300-by-300 pixels subimages. To obtain the correct 3D position of the ball, these coordinates have to be calculated back to coordinates that describe the ball's location in the large 2048-by-2048 images. Therefore, the offset parameter values of the subimages (Figure 30) will be required again.

Because procedures such as the background subtraction, the creation of edge images, and the object detection in 2D images include many operations that have to process many datasets with the same instructions, it is possible to carry them out parallel. Therefore, it is reasonable to perform these procedures on a GPU. On the other hand, triangulation only converts the coordinates of the 2D images to the coordinates of the ball's position in 3D space. This procedure does not process multiple datasets with the same instructions, which means that it is not possible to parallelize the required calculations. An algorithm should be executed on the most appropriate hardware. In other words, tasks like the previous ones, which process entire images, perform better on a GPU and tasks, like triangulation, which computes in a serial way, perform better on a CPU. Therefore, the triangulation algorithm will be placed in the host program. The drawback of this approach is in the necessity to move the required data (the coordinates) from the device to the host. On the one hand, this move requires some time, but on the other hand, this extra time will be more or less compensated by the faster execution of the triangulation on the CPU. Additionally, this approach provides useful information about transaction times. For example, if detection and prediction will take too much time when performing them on one GPU, it would be possible to perform these tasks on two different devices. This would also lead to the necessity of swapping the required datasets from one device over the host to the other device.

3.8.1 Triangulation

The triangulation process converts the two positions of the point of interest, which in this case is the center point of the ball, to one 3D position. As shown in Figure 13, the fundamental principle of this procedure is based on simple arithmetic operations (Equation 2.3.2). Because of the imperfect alignment of the stereo vision system, the distortion of the cameras, and other effects, such a simple solution would only lead to extremely inaccurate results. For this reason, the vision system has to be calibrated. This step, which is not part of this thesis, has been carried out with the help of a Matlab toolbox [19]. Following calibration, several parameters needed for accurate triangulation are available: focal lengths, principle points, skew- and distortion coefficients, rotation- and translation vector between the right and left camera.

For an accurate triangulation, a Rodrigues rotation matrix [35] is required that contains the rotation vector, which was calculated by abovementioned Matlab toolbox. This step consists of a matrix multiplication and is only done once during execution time and is, therefore, one of the first steps of the program.

Once the Rodrigues matrix and the other parameters are known, an accurate triangulation can be made. Now, the only factors that impact the 3D position's accuracy are the preceding detection algorithms that were explained in Subchapter 3.6 and Subchapter 3.7.

The first step of the triangulation will be the normalization of the image projection according to the intrinsic parameters. This procedure should remove respectively compensate the distortions of the two cameras and has to be performed for both images: the left and the right. This step requires: the coordinates of the center point, the focal length, the principle point, the distortion- and the skew coefficient. It is obvious that the normalization of the left image has to be calculated with the coordinates and parameters of the left side and, vice versa, the right side with the right coordinates and parameters. The normalization function provides distortion-free coordinates of the images.

Subsequently, with the help of the Rodrigues matrix, these distortion-free coordinates can be calculated to the desired 3D coordinates. More precisely, triangulation will result in two sets of 3D coordinates: for the view of the left and the right cameras each.

3.8.2 Coordinate translation

Since the robot arm catches the ball and not one of the cameras, the resulting coordinates have to be converted to a so-called world coordinate system. This coordinate system describes the view from the catching device, which means that the catching robot lies in its origin. Figure 53 shows an example of a possible alignment of one camera and the catching device. A transformation matrix is required to translate the 3D coordinates to the world coordinate system. This matrix is created as part of the calibration process and translates the coordinates from the left camera to the world coordinate system. Therefore, the 3D coordinates of the right camera will not be required in any further calculation steps.

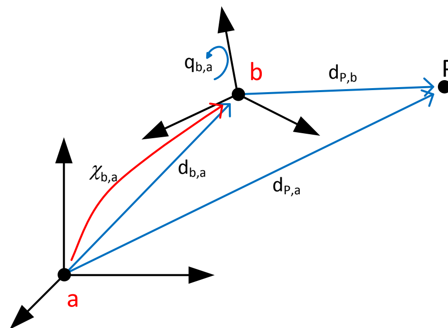

Figure 53: One point has different coordinates in different coordinate systems. Modified from [36]

The catching system can be placed and aligned in various ways: the cameras can be aligned horizontally or vertically, they can be placed behind the catching device or in front, etc. However, for every configuration, a calibration has to be done to obtain all parameters necessary for an accurate triangulation, and a transition matrix for correct coordinate translation is also needed.

3.9 Prediction

Besides detection of the ball, prediction of its flightpath is the other focus of this thesis. A bio-inspired prediction is the basis for this thesis and, therefore, its implementation will be explained in this subchapter. To recap, this prediction system shall estimate the ball's further flightpath based on a wealth of experiences.

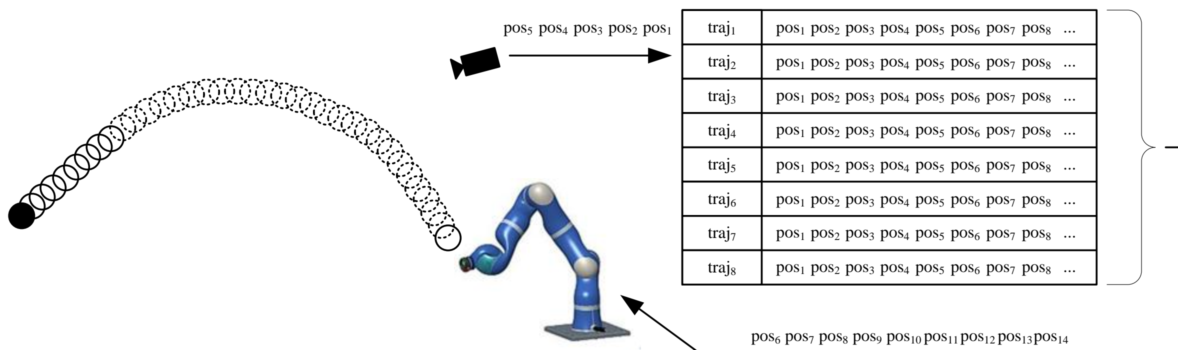

Figure 54: After the ball's actual flight has been compared with the trajectory database, its future positions can be transmitted to the robot arm to enable successful capture.

Numerous flights have been recorded and their trajectories stored in a database for this purpose. After the parameters of the flying ball have been captured, they can be compared with all the stored reference throws (Figure 54). It is comparable to a k-NN searching algorithm, which finds the k-nearest neighbors. Therefore, in the case of a flightpath prediction, the k most similar trajectories shall be found in the database. These trajectories can be averaged to get information about the ball's further movement. If k is equal to 1, only the most similar trajectory will be selected for predicting the flightpath.

The number of actual flight positions required for a reasonably accurate estimation is difficult to determine, but this issue is not part of this thesis. Currently, Pongratz works on his dissertation to answer this question among other things. Probably between 5 and 10 positions have to be known for the first prediction, but this is only an estimation. Nonetheless, it is obvious that the more positions are known, the more precise the prediction will be. Moreover, without any doubt, the wealth of entries in the trajectory database significantly determines the quality of the prediction and the results obtained. In other words, the more entries in the database, the more accurate the prediction will be.

This thesis focuses on the real-time ability of detection and prediction, based on the bio-inspired approach. Such a k-NN search may take a lot of time as a result of the huge amount

of comparisons with all the database entries [MPD14]. A good algorithm has to be implemented that perfectly fits the target hardware. This raises the question: CPU or GPU? Because of the huge amount of comparisons, which means the same instruction is processed with different data, it seems natural to execute this searching algorithm on a GPU.

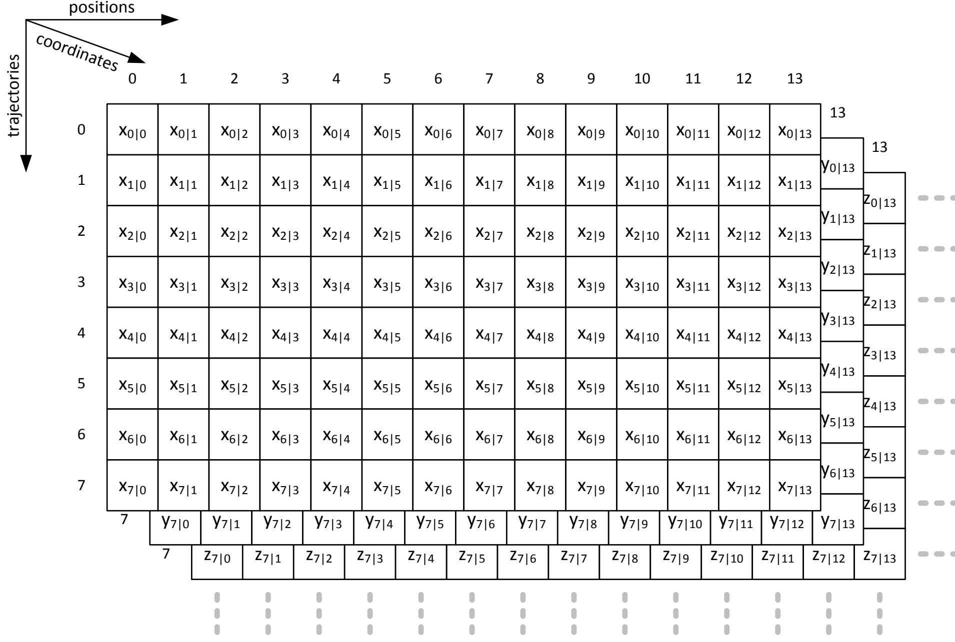

Figure 55: The trajectory database will be stored in a 3D array.

The trajectory database is saved as a CSV-file that has to be read in as part of a program initialization step. Because of the data's structure, a 3D array is the best solution for storing each coordinate of each position from each trajectory. Figure 55 shows the array's structure: one dimension for the trajectories, one for the positions, and one for the coordinates. After all data have been loaded from the hard disk to the main memory, they will be moved to the GPU global memory to make the comparison as fast as possible at runtime. Loading the data just in time would lead to much longer execution times. The trajectory database will be saved in the GPU's memory in the exact same way as it is saved in the main memory: as a 3D array (Figure 55).

Two different approaches exist for estimating a flightpath based on this bio-inspired technique: comparing directly the ball's positions with their counterparts in the database, or comparing the rates of change from one position to the next. This can be done through additional subtractions: the actual coordinate minus the previous one results in the change of the ball's position from its previous to its current position (Equations 3.9.1, 3.9.2, and 3.9.3).

$$\Delta x_i = x_i - x_{i-1} \tag{3.9.1}$$

$$\Delta y_i = y_i - y_{i-1} \tag{3.9.2}$$

$$\Delta z_i = z_i - z_{i-1} \tag{3.9.3}$$

Obviously, rates of change according to the database trajectories must also be calculated to compare them with those of the actual flight. The drawback of this approach is the demand for additional subtractions, but they probably will lead to just a slight increase in execution time. On the other hand, the advantage of this solution would be the independence of the ball's prediction from its launching position. The ball will have a similar movement no matter from where it is thrown. This could lead to more accurate predictions with the same wealth of database entries. However, as mentioned above, the achieved accuracy of the prediction is not part of this diploma thesis. Nevertheless, both approaches shall be implemented to obtain information about the required execution times. The next two subchapters deal with the implementations of both approaches.

3.9.1 Comparison of coordinates

This algorithm shall find the trajectory that best fits the actual flight's positions. For this purpose, the Euclidian distances of all the actual flight's positions (composed of x_{af}, y_{af}, and z_{af}) to their counterparts in the database trajectories (x_{db}, y_{db}, and z_{db}) will be added up to a total distance d_{total} (Equation 3.9.4). The database trajectory that leads to the smallest sum of distances, when comparing it with the actual flight, will be the most similar trajectory and can be selected to predict future movement. To find the minimum distance, an atomic function has to be used to fulfill the requirements of mutual exclusion. Unfortunately, the CUDA's instruction set contains only an atomic function for integer values and is, therefore, not able to handle the calculated rational numbers resulting from Equation 3.9.4. To overcome this drawback, the calculated value has to be transformed to an integer value that describes the distance with the same accuracy. After converting the calculated distances, the minimum of all distances can be obtained.

$$d_{total} = \sum_{i=0}^{n} \sqrt{\left(x_{i,db} - x_{i,af}\right)^2 + \left(y_{i,db} - y_{i,af}\right)^2 + \left(z_{i,db} - z_{i,af}\right)^2} \qquad (3.9.4)$$

However, the challenges of this algorithm lie in the various possibilities to fit the actual flight in a reference trajectory. Figure 56 shows four examples of possible flightpaths compared with one trajectory of the database. While the first example represents a fairly good match between the two trajectories, the second example shows a lack of congruence between the actual flightpath and the database trajectory.

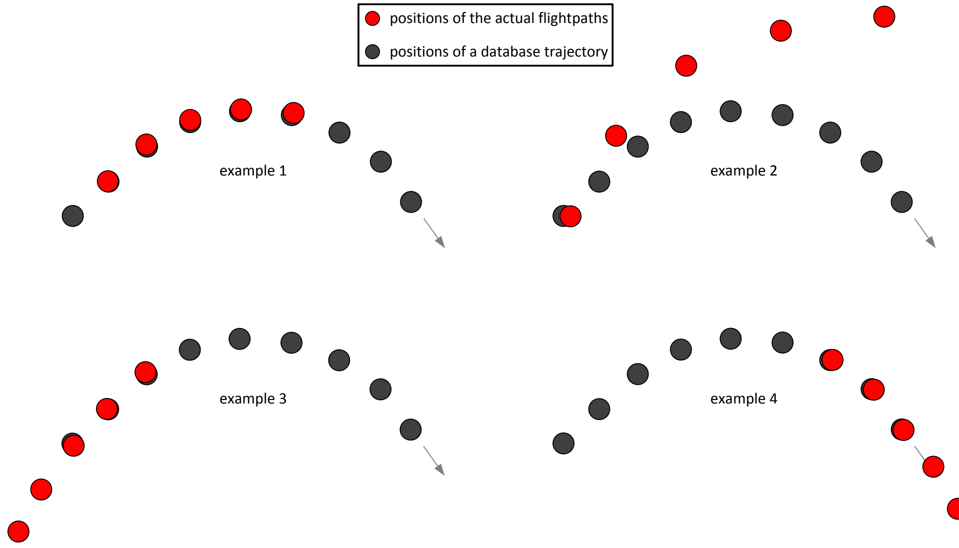

Figure 56: Four examples comparing actual flightpaths with the same database trajectory. While the first example represents a fairly good match between the two trajectories, the second example shows a lack of congruence between the actual flightpath and the database trajectory. Examples 3 and 4 are also good matches, but they do not fit completely into the database trajectory.

However, it is possible that some of the actual flight's positions are not present in the database trajectory that still fits best (example 3 and 4 of Figure 56). Therefore, the actual flightpath must be compared to the database trajectories in a way that some positions can be outside of the array boundaries. In other words, the recorded positions of the actual flightpath have to be "shifted" over the various trajectories of the database, and it must be guaranteed that not all of the actual positions have to lie on their counterpart database trajectory (Figure 57). That way, points of the actual flight not lying on a database position will not be included in the sum of all distances. However, it is logical that a trial that examines fewer positions can have a smaller distance although it fits less than one that examines more positions. An exaggerated example would be a comparison of two trials where one of the trials calculates only 3 distances and the other one 30 distances. It does not matter whether the 30 distances are very small, the sum of only 3 distances will probably be smaller. However, selecting this trajectory for the prediction could be a mistake. To circumvent such errors, all distance calculations have to be normalized, which means that the sum of distances d_{total} of a trial has to be divided by the number of distances calculated in this trial. Therefore, the Equation 3.9.4 will be expanded by an additional division (Equation 3.9.5).

$$d_{total,normalized} = \frac{1}{n+1}\sum_{i=0}^{n}\sqrt{\left(x_{i,db}-x_{i,af}\right)^2+\left(y_{i,db}-y_{i,af}\right)^2+\left(z_{i,db}-z_{i,af}\right)^2} \tag{3.9.5}$$

In this connection, a predefined parameter will determine the number of positions that are allowed to cross the boundaries of the database trajectories. Because the number of these out-of-range positions should be proportionate to the length of the actual flight's trajectory, the parameter will represent a percentage of this length. As an example, 2 out of 10 positions are allowed to be beyond the trajectory's boundaries when the parameter's value is set to 20.

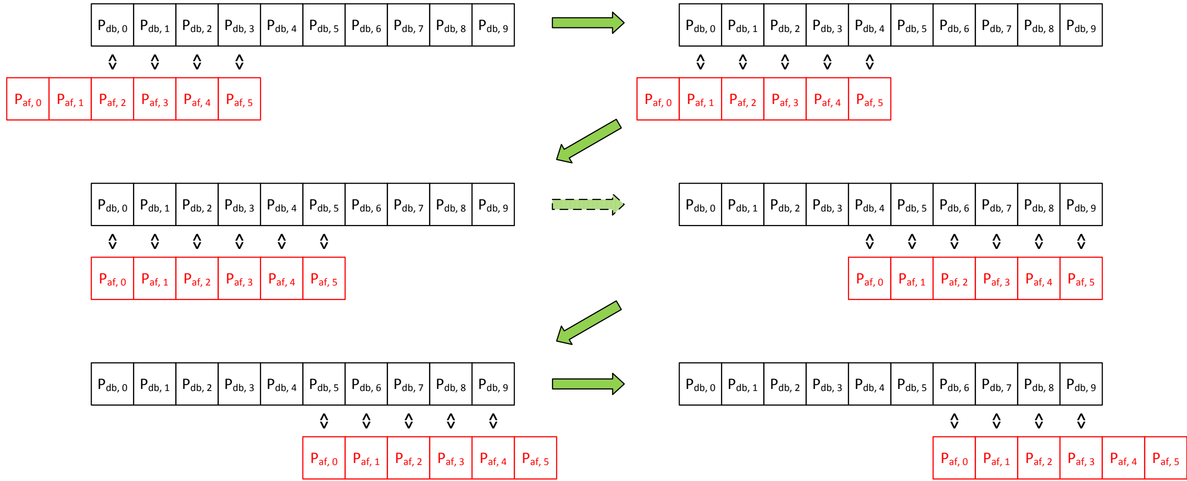

Figure 57: An actual flightpath is "shifted" over a database trajectory to find the place where the sum of all Euclidian distances is the smallest. A predefined parameter decides how many of the actual flight positions (P_{af}) are allowed to be outside of the database trajectory.

A good concept is required, that is, an appropriate amount of threads and blocks must be chosen, to accelerate this procedure with the help of a GPU's parallel architecture. Each database trajectory will be examined in a separate block. Therefore, the number of blocks B will be equal to the number of trajectories n_{traj} stored in the database (Equation 3.9.6).

$$B = n_{traj} \tag{3.9.6}$$

Choosing the right number of threads to be invoked, proves to be a challenge. Each possible alignment of the actual flight with a database trajectory will be examined in a separate thread. Separating these comparisons in different threads is equal to parallelizing the act of shifting the actual flight over a database trajectory. However, the number of possible alignments has to be known to determine the number of required threads. Figure 58 shows an example for a required number of threads T for a given length of the database trajectory l_{traj} and the current length of the actual flight l_{af} when neglecting the out-of-range positions. However, the database trajectories' lengths can vary, but all blocks must contain the same number of threads. Therefore, the required number of threads T depends on the length $l_{traj,max}$ of the longest database trajectory. This interrelationship of T, $l_{traj,max}$, and l_{af} is described by Equation 3.9.7.

$$T = l_{traj,\max} - (l_{af} - 1) \tag{3.9.7}$$

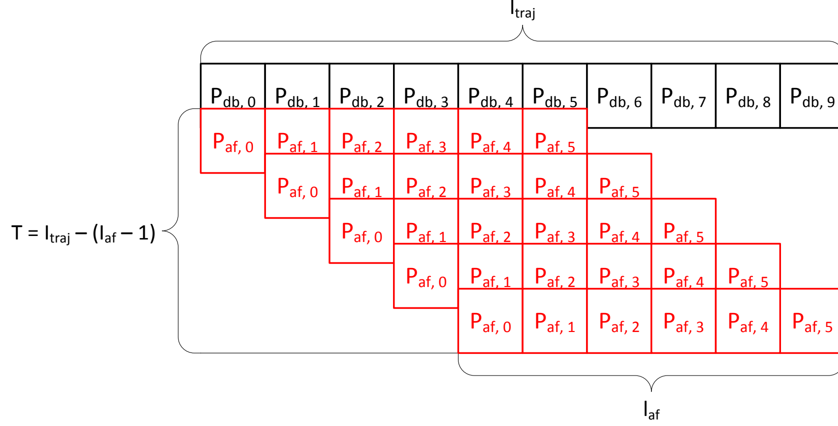

Figure 58: Number of possible alignments of an actual flight with a database trajectory when neglecting the out-of-range positions of the actual flight.

As explained above, a percentage parameter P_{oor} will determine the maximum number of out-of-range positions that will be accepted. This inevitably leads to a higher number of threads T that have to be invoked. The amount of threads has to be increased by the number of these out-of-range positions multiplied by 2 since these positions can lie outside the boundaries either before the first entry of the database trajectory or after it. Therefore, Equation 3.9.7 needs to be adjusted accordingly (Equation 3.9.8).

$$T = l_{traj,\max} - \left(l_{af} - 1\right) + 2 \cdot \frac{P_{oor} \cdot l_{af}}{100} \tag{3.9.8}$$

The number of required threads can deviate from the result (Equation 3.9.8) if the algorithm makes use of the shared memory to accelerate the procedure. In this case, every block loads the recent coordinates of the actual flight's positions and the positions of the block's database trajectory from the global memory to the shared memory to circumvent a huge amount of global memory accesses. More precisely, the first thread from a block loads the first position of the actual flight and the associated database trajectory, the second thread loads the second position, and so forth. If this swapping of data from the global to the shared memory were omitted, the various threads of a block would load the same datasets of the global memory numerous times, which would lead to higher execution times.

However, the more positions of an actual flight are known, the smaller the calculated number of required threads T (Equation 3.9.8). Therefore, it is possible that the number of invoked threads will be smaller than the length of the longest database trajectory if the additional number of out-of-range positions is smaller than the current length of the actual flight (decremented by 1). This would lead to an incomplete move of the positions stored in the global memory to the shared memory and could cause a wrong prediction. However, a sufficient number of threads T has to be ensured. If the calculated value T is smaller than the length of the longest database trajectory or the actual flight, the value T has to be set to the length of the longest database trajectory (Equation 3.9.9) respectively the actual flight (Equation 3.9.10). The number of invoked threads has to be big enough to move all positions

of the database trajectories and all positions of the actual flight from the global memory to the shared memory.

$$T = l_{traj,\max} \quad (3.9.9)$$

$$T = l_{af} \quad (3.9.10)$$

3.9.2 Comparison of rates of change

As mentioned in the introduction of this subchapter, there is another approach to finding the best-fitting trajectory from the database: comparing the rates of change.

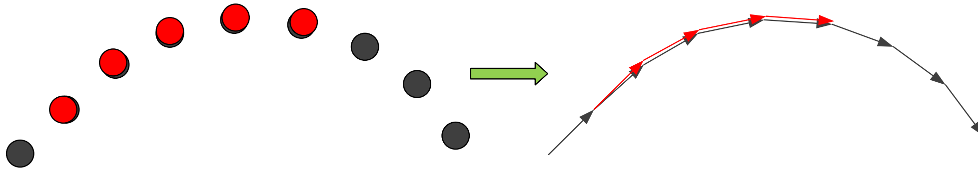

Figure 59: Comparing the vectors of the actual flight with those of the database trajectory instead of their positions.

Figure 59 shows this abstraction from the trajectory points to the movement vectors of the thrown ball. The algorithm is nearly the same as the one that compares the trajectories' points in the previous subchapter. The only difference is the additional step of subtractions from the coordinates (Equations 3.9.1, 3.9.2, and 3.9.3) of one point P_i and its previous point; see Equation 3.9.11.

$$\overrightarrow{P_i} = P_i - P_{i-1} \quad (3.9.11)$$

The assumption here is that this approach will require somewhat more execution time, but this will be tested and described in Section 4.1.6.

4. Results and Discussion

In line with the research goal stated in the Introduction (see Chapter 1), the real-time ability of a GPU-based prediction system was investigated here. For this purpose, related research (see Chapter 2) has been examined to obtain state-of-the-art information about required methods and possible approaches to performing the various procedures. This enabled implementation of the algorithms needed to detect the thrown ball and to perform a bio-inspired flightpath prediction. The deliberations regarding adequate implementation of the various program steps developed in the framework of this diploma thesis have been described in Chapter 3. The following subchapters will provide information about execution times, accuracy, and real-time behavior of the entire program. Each of the following line charts with frame numbers plotted on the abscissa represents an entire flight of the ball. The times represent the average, maximum, and minimum measured at these frames.

4.1 Comparing different approaches implemented

This subchapter deals with the results of several trials that have been examined and tested to find the best-fitting solution as well as to prove the assumptions made.

4.1.1 Performance comparison between CUDA and OpenCL

Conflicting statements have been made by [KDH10, 9f] and [FVS11, 9] regarding performance of the two platforms, CUDA and OpenCL, when programming a GPU. Therefore, a program part was implemented and tested on both platforms. Following examination of the execution times, a decision regarding the appropriate platform could be made. Since the creation of an edge image is among the program's first steps, the Canny Edge Detector was chosen for this comparison.

At the time of performance comparison, the learning process for programming a GPU was still in progress. Therefore, code optimization was omitted. In other words, the execution times needed by the two programs slightly deviate from later versions. However, this was not decisive for comparing the two platforms. Extreme manual optimization of the OpenCL program and pruning of the CUDA program, as done in [FVS11, 6] was not carried out here. Such actions were not considered to be useful when trying to make the right choice regarding the development of a program that will satisfy real-time constraints. Besides the nearly identical implementation of both programs, it was important to make sure that the computer

was in the same state when each of the programs was tested. Therefore, the computer was rebooted each time before program start. Additionally, the Internet connection was shut down during testing. Otherwise updating processes of the operating system or another program might have impaired the time measurements.

To make a clear statement about the performance gap, the execution times of each subframe of a flight, from the left camera, was measured 100 times in a row. The average time the Canny Edge Detector needed to process one image of an entire flight was calculated for every repetition of this flight. Each bar in Figure 60 represents the average execution time needed to create an edge image of a flight.

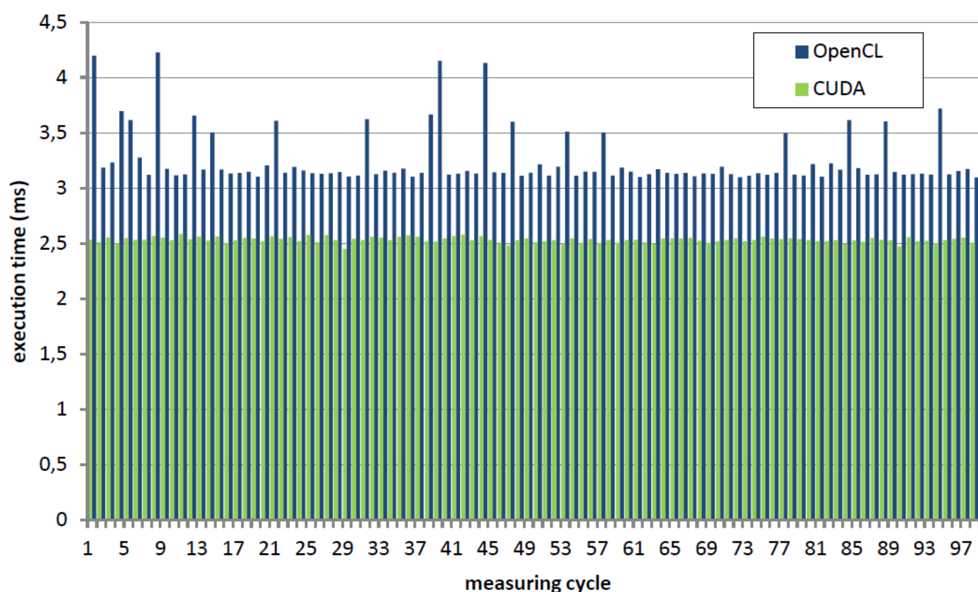

Figure 60: One flight of the ball was processed by the Canny Edge Detector 100 times in a row. The average execution times for one frame of each flight are illustrated in this bar diagram. The green bars are the execution times needed by the CUDA program and the blue bars are those needed by OpenCL.

Figure 60 illustrates the better performance of a Canny Edge Detector algorithm on a GPU when implementing it on the CUDA platform. The OpenCL program was about 28.25 percent slower than its CUDA counterpart. When examining the minimum and maximum execution times, a similar behavior was discerned: 26.34 % respectively 63.64 %. In absolute numbers, the CUDA program took between 2.45 ms and 2.58 ms, with an average of 2.53 ms.

Because of better performance, easier handling of its platform, and the more comprehensive availability of documents as well as tutorials, CUDA was selected as development environment for further implementation steps. OpenCL would only be advantageous with regard to its use on different hardware from different vendors. However, this ability is not required here.

4.1.2 Canny Edge Detector with and without background subtraction

A background subtraction can be done as an additional step to improve the quality of an edge image. The execution times were measured, and the results are shown in Figure 61. One

flight was processed 1000 times in a row to obtain reliable measurement results. The diagram shows the background subtraction's minimum, maximum, and average execution times of these 1000 iterations for each of the 92 frames of a flight.

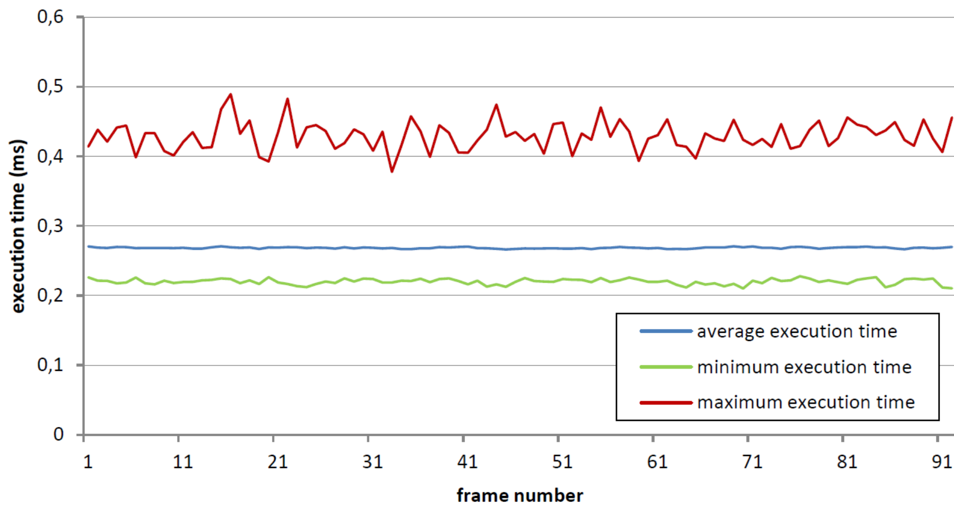

Figure 61: The background subtraction's execution times of an entire flight.

The average execution time for processing two subimages (left and right) is under 0.3 ms and is highly consistent. This behavior proves the independence of the background subtraction procedure from what the images show: whether the ball is near the cameras or far away, whether the ball is big or small, the average execution time is always the same. Background subtraction is a deterministic procedure and, therefore, the required time will always be fairly the same. However, the fluctuation of the maximum execution time is approximately between 0.35 and 0.5 ms and is most probably caused by cache missed.

As explained in Subchapter 3.5.4, a higher number of edge points in an image leads to a higher computation time for the Canny Edge Detector. Therefore, execution times for detecting edges can vary after the background was subtracted. A ball's flight was processed by the Canny Edge Detector 1000 times in a row to get information about the execution times of this procedure.

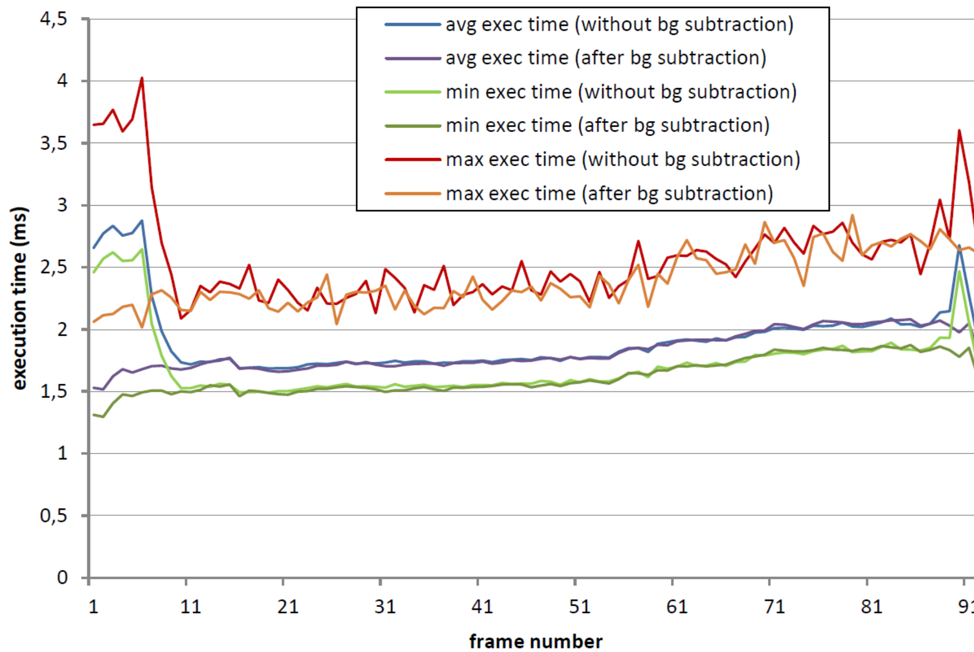

Figure 62: The Canny Edge Detector's execution times of an entire flight.

Figure 62 shows the execution time of the Canny Edge Detector processing left and right images without a background subtraction and after the background has been removed. The slight slope of the graphs is caused by the ball approaching the cameras from frame to frame. The more the flight advances, the larger the ball's shape will be displayed on the images. Because of the larger ball, more edge pixels are present in the frames and the iterative hysteresis procedure takes more time. At the beginning and the end of the flight, the Canny Edge Detector takes much more time to process the images with background than those without background. This behavior is caused by other objects that are present in these frames and lead to more edge points. The Canny Edge Detector takes more time when more edge points are present in the image. These additional edge points are not present when performing the background subtraction and, therefore, the execution time of these few frames is much shorter when subtracting the background.

However, the background subtraction itself also requires time. Therefore, this time should be added to the execution time for detecting edges. Figure 63 shows the ratio of the sum of execution times of background subtraction plus Canny Edge Detector to the Canny Edge Detector without a background subtraction.

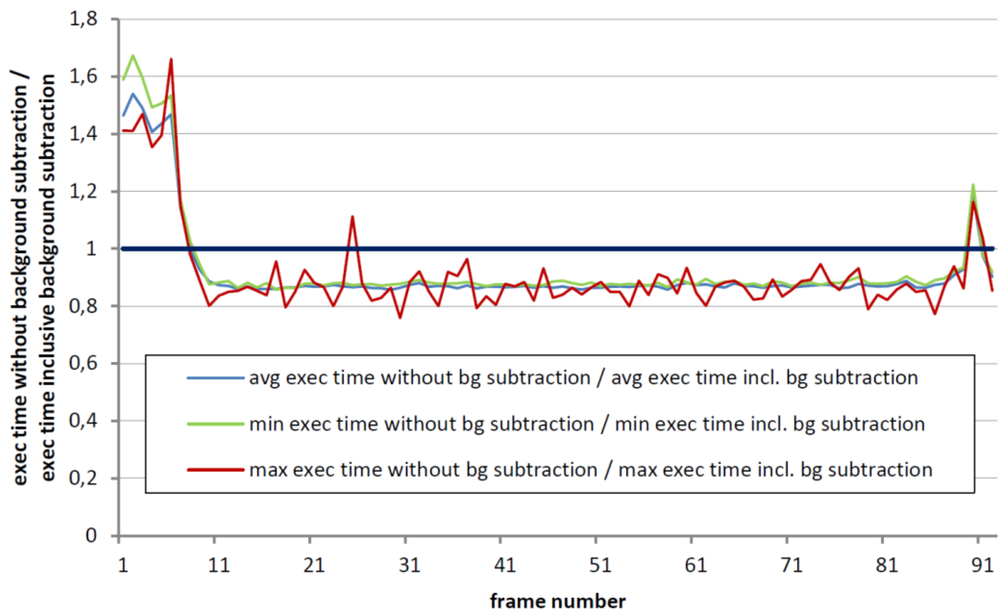

Figure 63: The ratio of the sum of execution times of background subtraction plus Canny Edge Detector to the Canny Edge Detector without a background subtraction.

Performing the background subtraction and the Canny Edge Detector takes slightly more time than creating edge images without subtracting the background when there are no other objects present in the images. In the case of other objects displayed in the image, the Canny Edge Detector without a background subtraction can take up to 1.7 times more time than performing both steps combined (compare Figure 63).

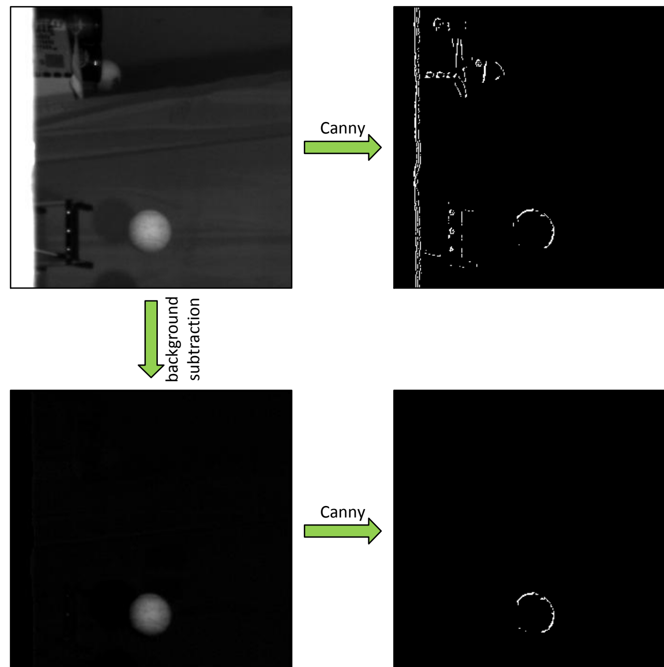

Figure 64: Differences of one edge image when the background had been subtracted.

Expressed in absolute numbers, subtracting the background and detecting edges took between 1.54 ms and 3.42 ms with an average of 2.1 ms when considering all 1000 iterations of the whole flight. In contrast, only detecting edges took between 1.49 ms and 4.02 ms with an average of 1.92 ms.

When subtracting the background, edge images of high quality are still possible when other objects are in the picture (Figure 64) and the execution time jitter is less. Therefore, substracting the background is preferable when considering execution time and accuracy.

4.1.3 Pythagoras vs. trigonometric functions

The Hough Circle Transformation and the RANSAC algorithm have to draw circles as a subroutine. This iterative procedure can be done with the help of trigonometric functions or the Pythagoras theorem. Both object detection algorithms have been examined with both approaches to drawing a circle.

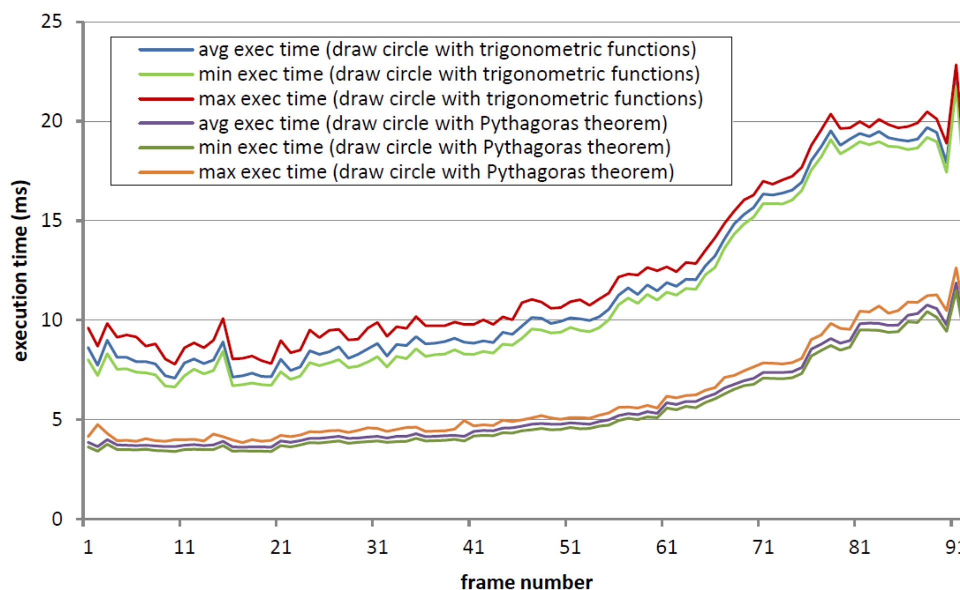

Figure 65: The Hough Circle Transformation's execution times for each frame of an entire flight when drawing circles with trigonometric functions and the Pythagoras theorem.

Figure 65 shows the execution times of both circle drawing approaches when performing the Hough Circle Transformation. Since a trigonometric function will be performed in one of the four SFUs (Special Function Unit) of a multiprocessor [Gla09, 21], the Pythagoras version is much faster. Additionally, it can be seen that execution times increase as the flight advances. Similarly to the Canny Edge Detector results (see Section 4.1.2), the more the flight advances, the larger the ball's shape will appear in the images. Therefore, more edge pixels will be present in the frames and the voting process of the Hough Circle Transformation will take more time.

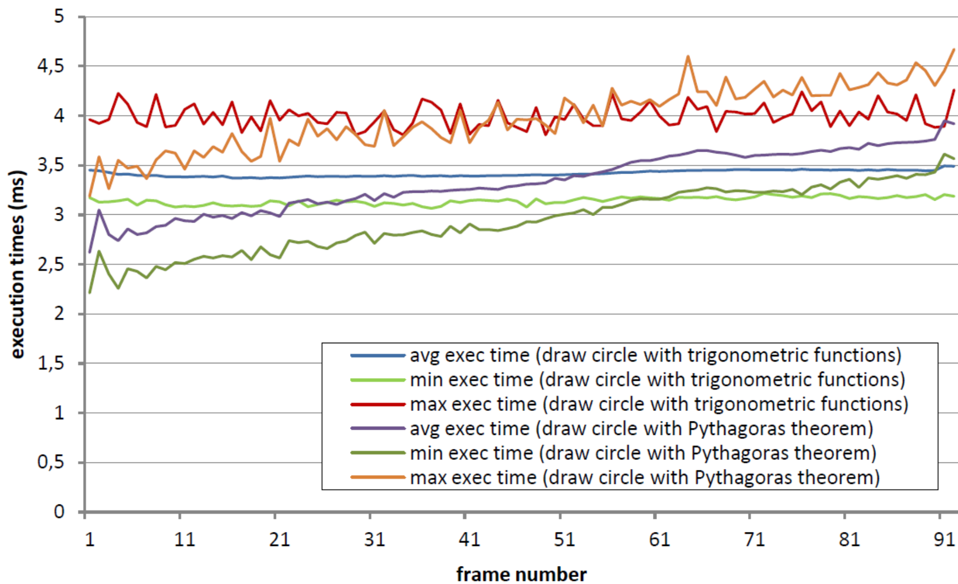

Figure 66: The RANSAC algorithms' execution times of an entire flight when drawing circles with trigonometric functions and the Pythagoras theorem.

Just as the Hough Circle Transformation, the RANSAC algorithm was also tested with both circle-drawing techniques. However, the results illustrate different behaviors. Because the RANSAC algorithm draws considerably less circles than the Hough Circle Transformation, the difference in their execution times is small. Because each iteration is performed over an angle and not a radius, the average time of the trigonometric version is fairly constant throughout the entire flight. However, because the radius increases as the flight advances, the Pythagoras version becomes slower from frame to frame. At the beginning of the flight, the Pythagoras version is faster than its counterpart and at the end, it is the other way round.

4.1.4 Two Hough Circle Transformation Strategies

Referring to Section 3.6, two different approaches to the Hough Circle Transformation have been implemented and compared with each other: the Straightforward Strategy and the Inverse-checking Strategy. For this comparison, frames of the same flight were processed 1000 times in a row and the execution times were recorded to analyze the results (Figure 67). Since the Pythagoras theorem completed the task of drawing circles faster (Figure 65) when performing the Hough Circle Transformation, this approach was chosen.

The Inverse-checking strategy requires considerably more time to detect the ball in both images. This is because each thread must start a loop to draw a circle when performing the Inverse-checking Strategy and only the threads that are assigned to an edge point have to start a loop when performing the Straightforward Strategy. The data obtained do not confirm the statements made in the studies [AME13, 220][CJ11]. Possibly, they had images with a huge amount of edge points, in which case it might be better to perform the Inverse-checking strategy.

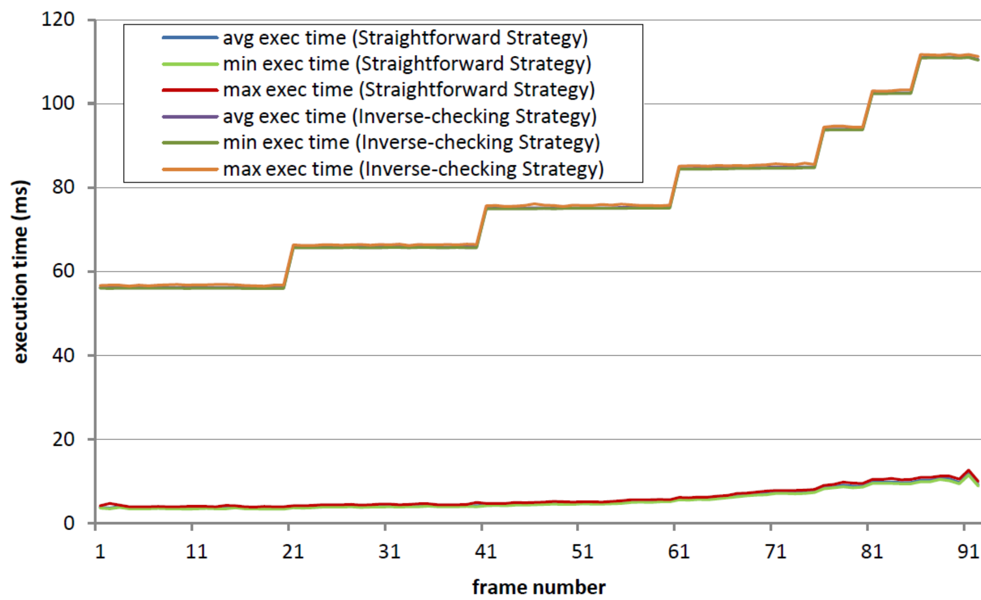

Figure 67: The different execution times of an entire flight when performing the Straightforward or Inverse-checking Strategy of the Hough Circle Transformation.

Additionally, the gradual increase in the Inverse-checking Strategy's execution times is noticeable. This behavior is caused by the radius profile created to save memory and execution time (see 3.6). Circles with a bigger radius have to be searched for when the ball is getting closer to the cameras and, therefore, the iterative procedure of drawing circles takes more time. Since the frames contain relatively few edge points, the Straightforward Strategy draws considerably less circles than the Inverse-checking.

4.1.5 Object Detection with and without background subtraction

Already in Subchapters 4.1.3 and 4.1.4 it was proven that the Hough Circle Transformation requires more time when more edge points are present. Therefore, it was assumed that there would be a difference when performing it after or without a background subtraction. Figure 68 shows both of these cases and illustrates the similarity between the Canny Edge Detector and the Hough Circle Transformation in connection with other objects present in the images (for comparison see Section 4.1.2).

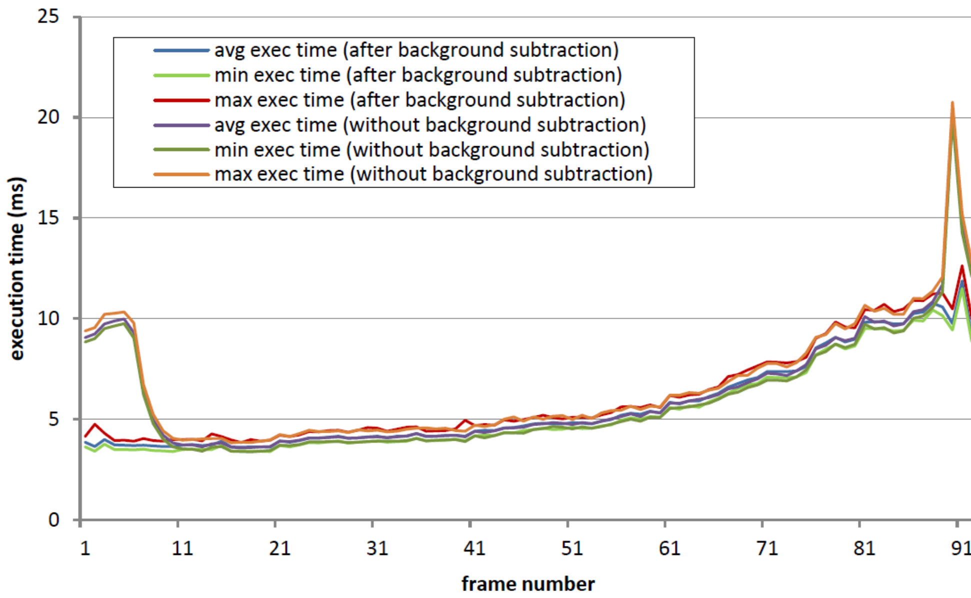

Figure 68: The Hough Circle Transformation's execution times of an entire flight after and without a background subtraction.

In contrast to the Hough Circle Transformation, the RANSAC algorithm's execution time does not depend on the number of edge points. Only the number of trials is responsible for the required time taken by this procedure. However, the detected center point can vary when more edge points are displayed. Two frames were picked out and processed 1000 times in a row to examine this variance in different situations: one with other objects in the frame and one only with the ball.

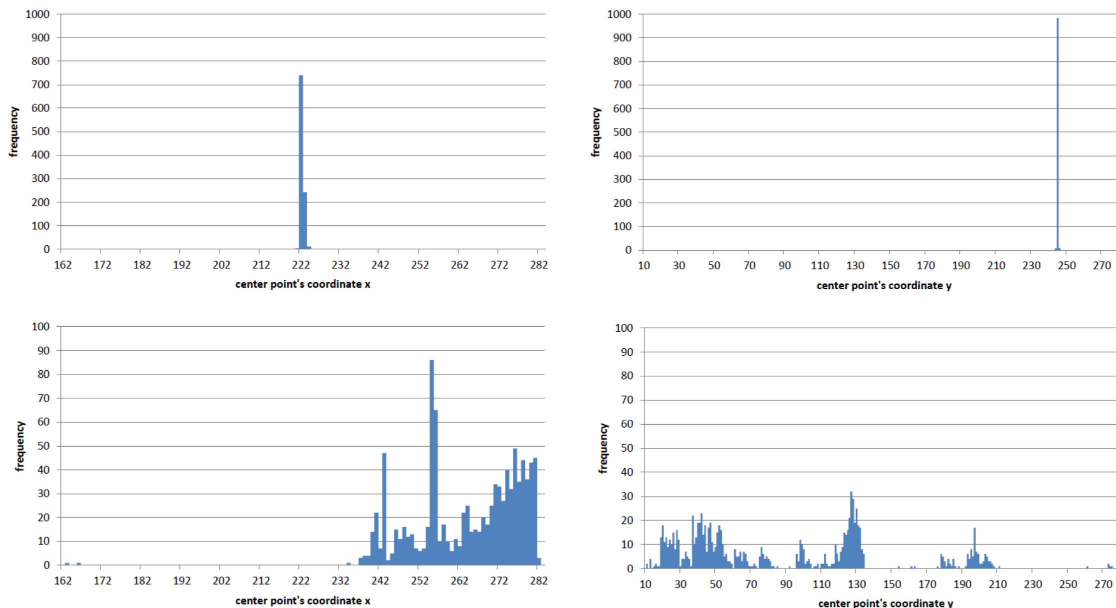

Figure 69: A frame with other objects besides the ball was examined. Above are the coordinates' histograms when the background has been subtracted and below the histograms when the background was not removed.

Figure 69 shows the results for a frame with other objects besides the ball in the frame. There is almost no variance in the coordinates calculated when the background was subtracted. In contrast, if the background was not removed, the resulting coordinates were highly scattered.

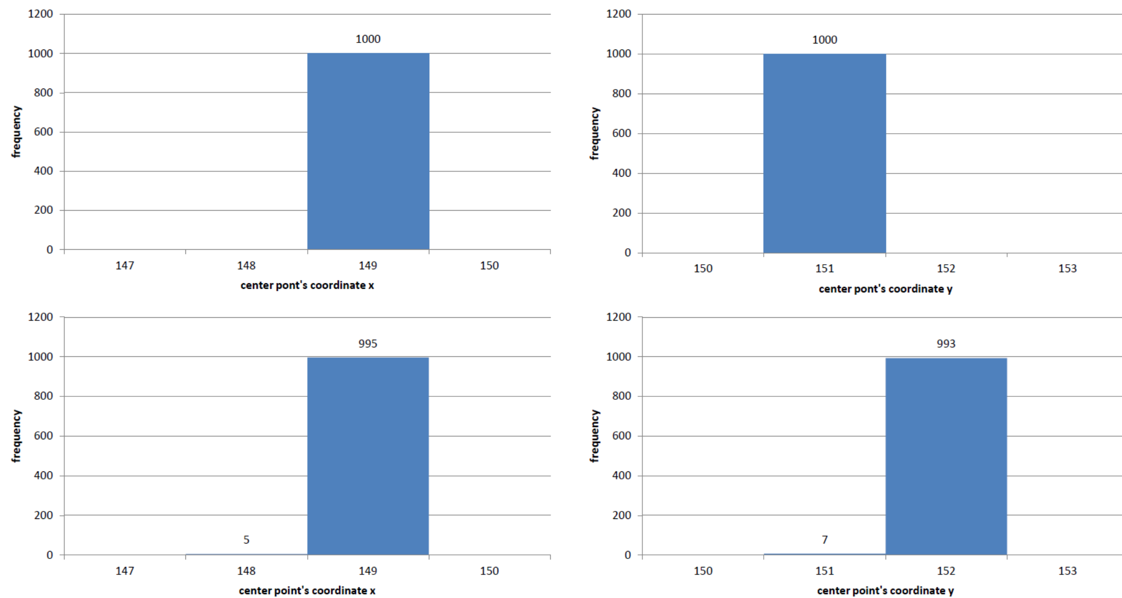

Figure 70: A frame without other objects besides the ball was examined. Above are the coordinates' histograms when the background was subtracted and below the histograms when the background was not removed.

On the other hand, Figure 70 shows the results of a frame that displays more or less only the ball. There is no big difference between the detection in the images with background and those without. However, the background-subtracted image leads to a variance of less than one pixel and that with background leads to a small variance.

detection algorithm	background	average tracking error
Hough Circle Transformation	subtracted	2,73 mm
Hough Circle Transformation	untouched	25,81 mm
RANSAC algorithm	subtracted	2,78 mm
RANSAC algorithm	untouched	25,96 mm

Table 4: The standard deviation from the calculated position in world coordinates to the world coordinates of the smoothed flightpath.

To obtain reliable statements about the average tracking error of the determined and the real center point, the results of the Hough Circle Transformation and the RANSAC algorithm were additionally analyzed with the help of the Rauch-Tung-Striebel filter [37]. More precisely, the results of the detection were analyzed after they had been triangulated (see Section 3.8.1) and translated (see Section 3.8.2) to the world coordinate systems. This filter examines the recorded flightpath from front to back and reverse to create a smoothed flightpath. This means, physically impossible detections will be corrected through this

smoothing process. The average tracking error for the different approaches can be seen in Table 4. The results obtained show that the detection algorithm enables an accurate localization of the ball's center point when the background was subtracted. In this case, the localized position is only about 2 to 3 mm away from the real one away. Because of the better results when detecting the ball after subtracting the background, this additional step is preferable when considering accuracy.

4.1.6 Comparison of the two prediction algorithms

Now let's shift our attention to the last step of the program, which enables a prediction of the ball's further flightpath. The accuracy of two different approaches to this bio-inspired prediction technique shall be examined in future works: comparison between positions and comparison between rates of change. However, execution times of both have been measured to make a statement of their temporal behavior and will be illustrated in the following two subchapters. All of the following measurements have been made by processing one flight 1000 times in a row to obtain reliable measurement results.

4.1.6.1 Dependence on the number of reference trajectories

Because of the execution time's dependence on the number of reference trajectories and the parameter P_{oor} (see Subchapter 3.9.1), two measurements were made to obtain sufficient information about the temporal behavior. The reference trajectories' lengths will affect the execution time as well, but considering these values would exceed the scope of this diploma thesis. Moreover, to provide an accurate prediction, it would not make sense to shorten these trajectories. To make a statement about realistic scenarios, typical lengths were chosen for these examinations.

To analyze the execution time's dependence on the number of reference throws, a database with trajectories of the same length was required. Otherwise, the comparison of the various trials with their different database sizes would not be fair. For this purpose, the database was filled with 200 dummy trajectories all with a length of 92, which is equivalent to the longest trajectories of an already existing database. To conduct this measurement, 21 samples were used, starting with one trajectory, then 10, 20, etc. up to 200. The parameter P_{oor} was set to 0 for the duration of these tests.

Figure 71 shows the execution times of the two different prediction approaches: comparison of positions and comparison of rates of change. Average and minimum time rise fairly linearly as the number of reference trajectories in the database increases. While the minimum execution time increases very slowly, the average time's graph definitively illustrates the impact of including a higher number of reference throws. This behavior is caused by Equation 3.9.6, which leads to a higher number of blocks when more trajectories are in the database. The maximum execution time's graph is also rising and its fluctuation is probably caused by cache misses. The temporal behavior of the two approaches is almost even, but the time required is a bit different. While a database with up to 120 trajectories leads to nearly identical execution times, a small difference is observed when the number of reference throws increases. In other words, a higher number of reference trajectories leads to a bigger difference

between the two approaches' execution times. However, this performance gap, caused by additional subtractions (see Section 3.9), is not very big.

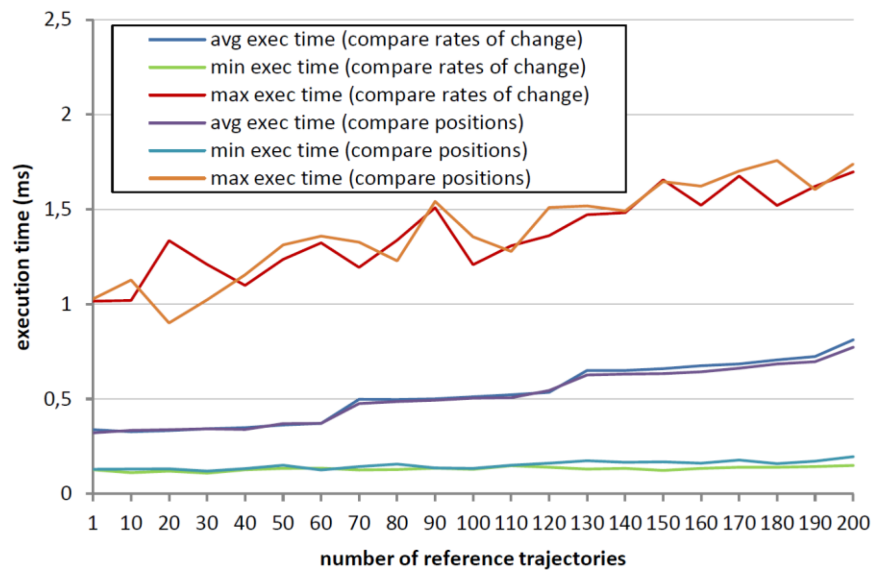

Figure 71: The prediction's execution time as a function of the number of reference trajectories for the comparison of positions and rates of change.

4.1.6.2 Dependence on the number of out-of-range-positions

The execution time's dependence on the out-of-range parameter has also been measured and will be described as well. This parameter determines how many positions of the actual flight are allowed to cross the boundaries of the database trajectories. For this purpose, an authentic database was loaded containing 188 different trajectories with an average length of 74.4 positions, whereby the shortest trajectory had only 55 and the longest 95 positions.

Figure 72 shows the execution times of the two different prediction approaches: comparison of positions and comparison of rates of change. The temporal behavior of the two different prediction approaches is nearly similar. It seems that the minimum execution times are more or less independent from the number of out-of-range positions; the two graphs neither fall nor rise. The linear rise of the average time was assumed and can be explained with Equation 3.9.8. More threads will be invoked when the parameter P_{oor} has been set to a higher value, which leads to a higher execution time.

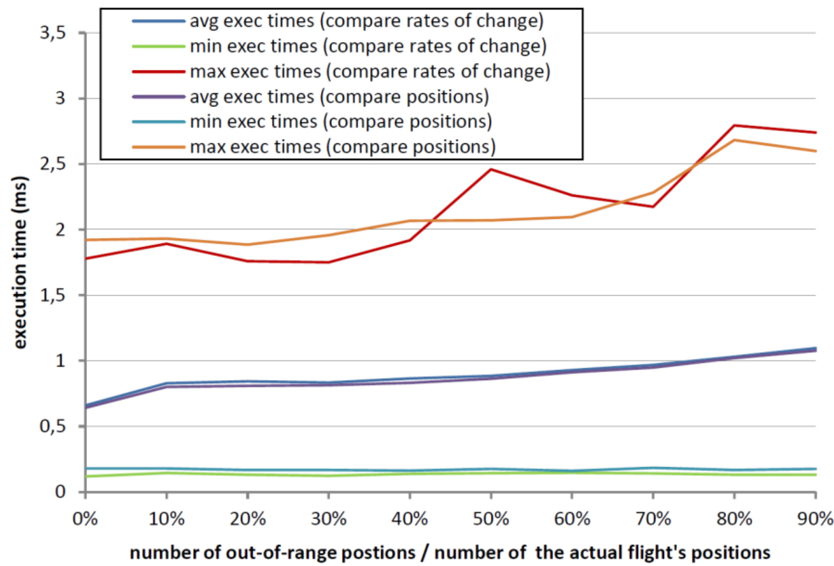

Figure 72: The prediction's execution time as a function of the out-of-range positions for comparison of positions and rates of change.

4.2 Testing worst-case execution times with artificially generated data

Some of the procedure's tasks are independent of the input data and some are not. To get information about the theoretical worst-case execution times of the procedure, such tasks as the Canny Edge Detector and the Hough Circle Transformation have been tested. The prediction task's execution time depends on the number of reference trajectories, the lengths of these trajectories, and the number of out-of-range positions. However, a clear statement can be given without an examination: an infinite number of reference trajectories would cause an infinite high execution time[7]. The execution times of the remaining tasks do not depend on the subimages loaded and the required time will always be almost the same. Therefore, they were not examined here.

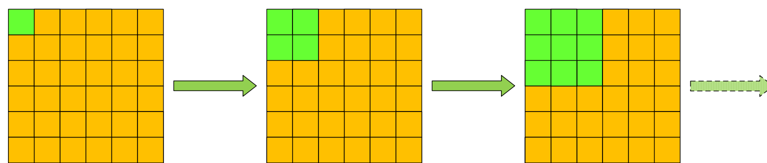

Figure 73: Such an image was injected into the hysteresis function to cause a maximum number of iterations.

Figure 74 shows the worst-case execution times measured with the help of artificially generated data. To be able to examine these times, it is necessary to understand when these cases will occur. The Canny Edge Detector will take the most time if the hysteresis procedure has to iterate as often as possible. For this reason, a special image was inserted into the

[7] The maximum number of reference throws is limited by the GPU's memory capacity.

hysteresis function: one corner pixel had a value higher than the upper threshold and all of the other pixels had values between the two thresholds (Figure 73). Starting with this pixel, until the image's borders are reached, every following iteration step finds neighbors that are above the lower threshold which will, therefore, be registered in the queue of the next iteration. In other words, the hysteresis procedure must go from one corner to the opposite corner. The result of the procedure will be an image full of edge points and can be directly used for the worst-case execution time measurement of the Hough Circle Transformation. Referring to Section 3.6, the more edge points in an image, the more time will be required by the Hough Circle Transformation. As Figure 67 shows, the circle detection also depends on the radiuses that have to be searched for. To examine the theoretically possible worst-case times for the given images, the radius range was set to detect the ball when it is displayed as large as the largest ball in the frames recorded: 50 to 60 pixels. The following worst-case execution times relate to the processing of both images: the left and the right.

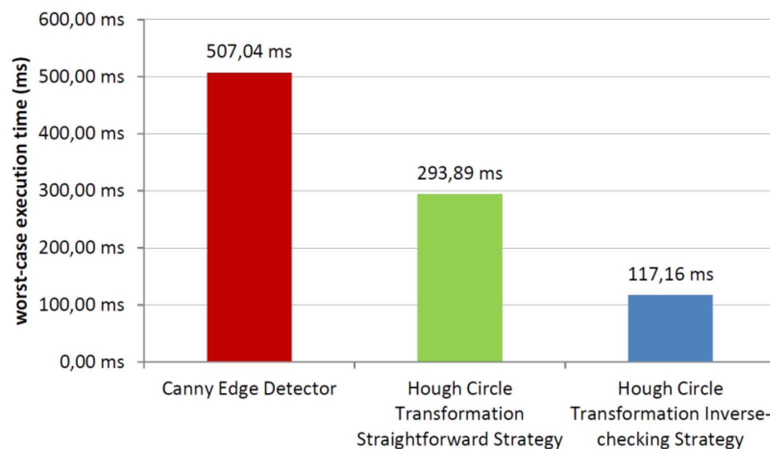

Figure 74: The worst-case execution times of the Canny Edge Detector and the Hough Circle Transformation implemented.

Such an input image provides a totally unrealistic scenario and would never lead to any correct detection or prediction. However, it shows that the Canny Edge Detector depends on the input data. Its worst-case execution time does not provide any information about the maximum time required in a real-time application, but it shows that the execution time can drastically increase if lighting conditions are poor or hysteresis thresholds badly adjusted.

Such a statement cannot be generally applied to the Hough Circle Transformation. When comparing Figure 67 and Figure 74, it can be seen that the worst-case execution times of the Inverse-checking Strategy do not deviate much from the times in a normal scenario. There, it took the Hough Circle Transformation about 111 ms to detect a circle with a radius between 50 and 60 pixels and the worst-case execution time is about 117 ms. In both scenarios, the procedure is the same: a loop, which draws a circle, will be started for every field in the Hough Space. The only difference is that every field of the Hough Space will be voted in the worst-case and, therefore, every value has to be stored. Because of the rise in global memory accesses, the required time is about 6 ms higher. On the other hand, a thread of the Straightforward Strategy only starts such circle drawing loop when its own pixel is an edge

point. Every thread has to start this iterative drawing procedure in the worst-case scenario. Although both approaches draw the same number of circles, the worst-case execution times totally differ from each other. While threads vote only their own fields when performing the Inverse-checking strategy, their Straightforward version's counterparts vote various fields in the Hough Space. Atomic functions, which are needed to fulfill the requirements of mutual exclusion, lead to a partial serialization of the process. In the case of an image full of edge points, a lot of threads want to vote the same field at the same time and, therefore, most of the votes will be serialized. Therefore, the statements made in [AME13, 220][CJ11] can be true if the images used contained many edge points.

4.3 Results of the entire procedure

The question of the time required to detect the ball and predict its further flightpath runs like a golden thread through this diploma thesis. The previous subchapters provided information about different approaches, and the best solution for each step was selected to examine the execution time of the entire procedure. Each pair of frames had to run through the following tasks: clearing the GPU storage, background subtraction, Canny Edge Detector, RANSAC algorithm, triangulation, coordinate translation, and prediction. The RANSAC algorithm drew its circles with the Pythagoras theorem and the prediction consisted in comparisons of the coordinates determined.

Figure 75 shows the combined execution times of five different flights that have been processed 1000 times in a row. The fluctuation of the maximum execution time's graph is caused by cache misses. The shorter time at the beginning of the flight and its fast increase is caused by the throwing device that partially covered the ball. Therefore, the images showed fewer pixels in the first frames. Since the ball's image becomes larger from frame to frame, the execution time of the Canny Edge Detector and of the RANSAC algorithm (Pythagoras) increases as the flight goes on. A kind of light profile was created to set the hysteresis thresholds to values appropriate for the position of the ball. In other words, the ball flew toward the cameras and the light source and, therefore, the thresholds had to be increased in stated intervals of the flight phase. Without such a light-profile, execution times would greatly increase and detection would not work that well.

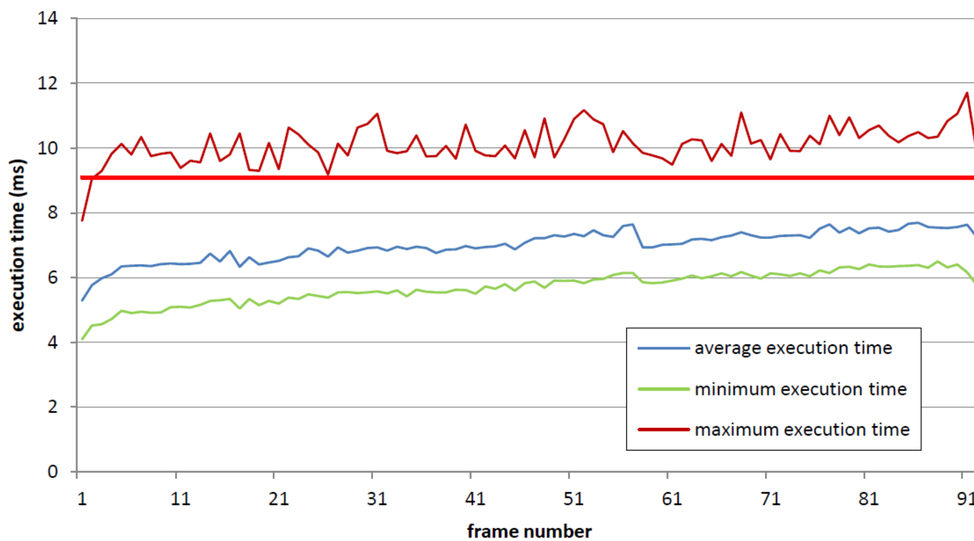

Figure 75: The execution times of five flights were processed, examined, and combined for this chart.

The maximum of the average times of the different flights was about 7.69 ms and enables a frame rate of 130 FPS (Equation 2.4.1). The highest frame rate of the camera system used is 110 FPS, which should not be a problem, but a small buffer is required to compensate for the maximum execution times that were, in rare cases, over 10 ms. Without such a buffer, there would not be sufficient computational power on hand to process the next frames when they appear.

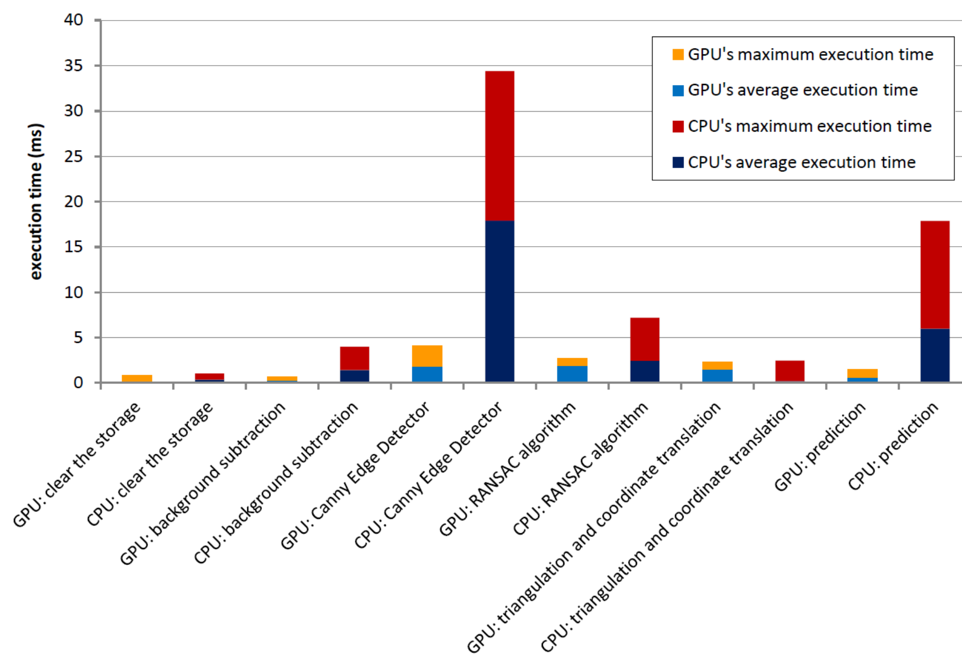

Figure 76: The different bars show the differences between the average and maximum times of the various program tasks when performing it on a GPU and on a CPU.

The last question concerns the speedup achieved through the use of a GPU instead of a CPU. For this purpose, the entire program was rewritten and slightly adapted to be executable on a CPU. To compare execution times, one flight was processed 1000 times in a row and the required times for the various steps were measured (Figure 76). The largest increases in speed were measured for the Canny Edge Detector and the prediction, followed by the RANSAC algorithm and the background subtraction. Clearing the storage also makes a difference, but the difference in performance is not extremely high. Triangulation and coordinate translation was performed on the CPU in both versions of the program, but because of the required data transfer from GPU to CPU, its average time is higher in the GPU version. In short, a single frame was processed 3.46 to 7.17 faster on a GPU than on a CPU.

5. Conclusion and Future Work

In the introduction, the Transport-by-Throwing approach, which consists of robotic arms throwing objects to each other, is described as one possible system for a more flexible type of production line. Catching a ball is still a challenge or benchmark for developing robots and various strategies, and approaches have been tried without achieving a perfect solution. This diploma thesis deals with a biologically-inspired prediction of a ball's flightpath. While the accuracy of this system is beyond the scope of this work, its temporal behavior was examined here.

5.1 Conclusion

One of the main questions of this thesis whether ball detection and flightpath prediction can be performed fast enough to achieve a frame rate of 110 FPS can be clearly answered in the affirmative. It was possible to fulfill and exceed this requirement. The examined execution times enable a computation fast enough for processing data at 130 FPS with a four-year-old GPU: NVidia GTX 560 Ti. Moreover, using a GPU for the required calculations proved to be a brilliant idea. The implemented program was 3.46 to 7.17 faster when running on a GPU instead of on a CPU. On the one hand, the CPU program made use only of one core and, therefore, some optimizations could accelerate execution, but on the other hand, the CPU used was a just one-year-old Intel i7-4770S. Using a newer GPU (Figure 77) and enabling the CPU to make use of its four cores would accelerate the programs, yet, the speedup would most probably remain almost unchanged.

Performing all required tasks in such a short time is easier said than done. Numerous optimizations had to be made to enable such high-speed computation on the GPU used. Well-thought-out algorithms and letting some data make a detour over the faster shared memory were the keys to success. However, some constraints still have to be considered to achieve an execution time short enough for this frame rate: Firstly, a small buffer is needed to compensate for the maximum execution times, which can be slightly too high for 110 FPS. Without this buffer, not enough computational power would be on hand to process the next frames when they appear. Secondly, the Hough Circle Transformation turned out to be computationally too costly and time-consuming. Therefore, the RANSAC algorithm had to be used to achieve the desired execution time.

However, there are more issues to contemplate: Without the additional background subtraction, other objects displayed in the images could lead to a considerably higher number of edge points and, in further consequence, to an execution time that is too high for 110 FPS. This additional computational step also enables a more precise detection. The accuracy of the prediction system was not part of this thesis, but a more accurate detection will probably lead to a more accurate prediction. Therefore, it is strongly recommended to perform a background subtraction regardless of whether a newer GPU is used or not.

Additionally, a good light-profile was created that changed the Canny Edge Detector's thresholds as the flight progressed. Rather low thresholds are necessary to enable detection of the ball in the early flight phase when it is still far away from the cameras and light sources. However, the light reflections on the ball's surface are becoming considerably larger and lead to much more edge points as the ball approaches. Without this profile, the thresholds would have to be set to a rather small value to detect the ball in the early flight phase. Deterioration in the detection in the last frames of a flight would not be the only drawback of this approach. The badly adjusted thresholds could lead to too many edge points and, hence, to an execution time too high for 110 FPS. Bad lighting conditions could also lead to the necessity to lower the thresholds. Therefore a sufficient and homogenous lighting should be present to achieve accurate results and an execution time short enough for 100 FPS.

5.2 Future work

Although this diploma thesis covered a great number of investigations, approaches, and implementations, open questions still remain and appropriate research will have to be carried out in the future. A distinction must be made between research pertaining to the accuracy of the bio-inspired prediction approach and additional research to improve the performance of the program introduced in this work. However, because of the complexity of the Transport-by-Throwing approach, these two domains are often linked with each other.

In the future, the number of required reference trajectories as well as the limitations of this prediction approach will have to be examined. Furthermore, the required number of the actual flight's positions and the differences between the two models compared are of great interest.

Additionally, the database could be filled with artificially generated data or extended with actual flights to improve prediction accuracy. In the future, the transport system should be used to transport goods of different sizes and shapes, not only tennis balls. Therefore, much research still needs to be done to find a solution that provides a prediction as accurate as necessary to catch a thrown object safely and softly. The throwing and catching robots could communicate with each other about their payload and the time of its launching. This communication would simplify some tasks of the prediction system, but a good detection will still be needed.

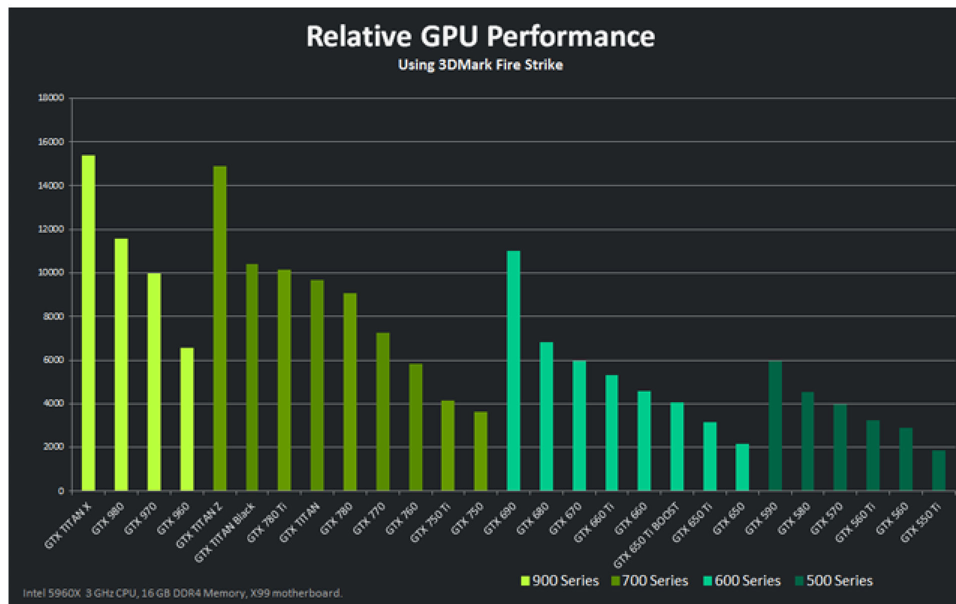

Figure 77: The theoretical speedup when using a newer graphics card can be read in this diagram, which shows the benchmarks of different GPUs [38].

Additional calculations or the use of a more complex detecting algorithm such as the Hough Transformation could overload the GPU used. Figure 77 shows that a more advanced GPU is theoretically four times faster than the one used. In other words, using a newer GPU would be a strategy to reduce execution times and meet these upcoming challenges.

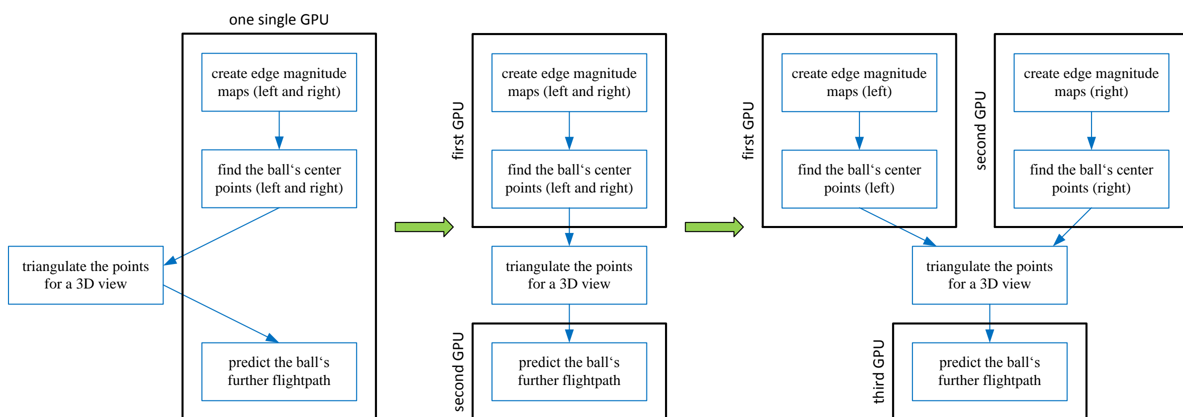

Figure 78: The required tasks could be distributed to different GPUs to increase performance.

To use two or more GPUs to separate the program tasks would be another alternative strategy. Figure 78 shows two possible approaches: separating detection and prediction or, additionally, separating detection from the left side and the right side. Possible future changes in algorithms and approaches notwithstanding, this work has clearly shown that the flightpath of an object can be predicted in a very fast manner.

Literature

[AC13] Al-Qahtani, F. H., Crone, S. F.: Multivariate k-Nearest Neighbour Regression for Time Series data - a novel Algorithm for Forecasting UK Electricity Demand, The 2013 International Joint Conference on Neural Networks (IJCNN), pp. 1-8.

[AME13] Askari, Meisam and Ebrahimpour, Hossein and Bidgoli, Azam Asilian and Hosseini, Farahnaz: Parallel GPU Implementation of Hough Transform for Circles, 2013.

[And98] R.L. Anderson. *A Robot Ping-Pong Player.* MIT Press, 1998.

[ATW07] F. Alam, W. Tio, S. Watkins, A. Subic, J. Naser, P. Jacobs, T. McIntyre, M. Cleary, D. Buttsworth, D. Mee, R. Clements, R. Morgan, and C. Lemckert, "Effects of spin on tennis ball aerodynamics: An experimental and computational study," 2007. [Online]. Available: http://espace.library.uq.edu.au/view/UQ:120801

[BFK08] D. Barteit, H. Frank, F. Kupzog: Accurate prediction of interception positions for catching thrown objects in production systems. In Proceedings on 6th IEEE International Conference on Industrial Informatics, 13 - 16 July, Daejeon, Korea, 2008

[BFP09] Barteit, D., Frank, H., Pongratz, M., Kupzog, F. (2009): Measuring the Intersection of a Thrown Object with a Vertical Plane. Paper is accepted for 7 th IEEE International Conference on Industrial Informatics (INDIN 2009), June 24 -26, 2009, Cardiff, UK.

[BGL01] J. Butterfaß, M. Grebenstein, H. Liu, and G. Hirzinger, "DLR-Hand II: Next generation of a dextrous robot hand," in *Proceedings of the IEEE/RSJ International Conference Robotics and Automation (ICRA)*, 2001, pp. 109–114.

[BSW11] Bäuml, B.; Schmidt, F.; Wimböck, T.; Birbach, O.; Dietrich, A.; Fuchs, M.; Friedl, W.; Frese, U.; Borst, C.; Grebenstein, M.; Eiberger, O.; Hirzinger, G. (2011) Catching Flying Balls and Preparing Coffee: Humanoid Rollin'Justin Performs Dynamic and Sensitive tasks.

[BWH10] Bauml, B.; Wimbock, T.; Hirzinger, G., "Kinematically optimal catching a flying ball with a hand-arm-system," *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, vol., no., pp.2592,2599, 18-22 Oct. 2010

[CAN86] Canny, John, "A Computational Approach to Edge Detection," Pattern Analysis and Machine Intelligence, IEEE Transactions on , vol.PAMI-8, no.6, pp.679,698, Nov. 1986

[CH67] Cover, T. M., Hart, P. E.: Nearest Neighbor Pattern Classification, IEEE Transactions in Information Theory, Vol. IT-13, 1967, pp. 21-27.

[CJ11] Chen, S. and Jiang, H.: Accelerating the Hough Transform with CUDA on Graphics Processing Units, 2011.

[CUD14] NVidia Corporation: "CUDA C Programming Guide", PG-02829-001˙v6.5, August 2014

[DAC02] T. D'Orazio, N Ancona, G. Cicirelli, M. Nitti, "A Ball Detection Algorithm for Real Soccer Image Sequences", IEEE, 1051-4651, 2002.

[DH72] R. Duda and P. Hart: "Use of Hough transform to detect lines and curves in pictures", *Communication of the ACM*, 15(1), pp.11-15, 1972.

[DWS12] Dollar, P.; Wojek, C.; Schiele, B.; Perona, P., "Pedestrian Detection: An Evaluation of the State of the Art,"*Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol.34, no.4, pp.743,761, April 2012

[FB81] Martin A. Fischler and Robert C. Bolles. 1981. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM* 24, 6 (June 1981), 381-395.

[FBH01] Frese, U.; Bauml, B.; Haidacher, S.; Schreiber, G.; Schaefer, I.; Hahnle, M.; Hirzinger, G., "Off-the-shelf vision for a robotic ball catcher," Proceedings of 2001

[FBM08] H. Frank, D. Barteit, M. Meyer, A. Mittnacht, G. Novak, S. Mahlknecht: Optimized Control Methods for Capturing Flying Objects with a Cartesian Robot. In Proceedings on 3rd IEEE International Conference on Robotics, Automation and Mechatronics, 22 - 24 September, Chengdu, China, 2008

[FBW07] H. Frank, D. Barteit, N. Wellerdick-Wojtasik, T. Frank, G. Novak, and S. Mahlknecht, "Autonomous mechanical controlled grippers for capturing flying objects," *Industrial Informatics, 2007 5th IEEE International Conference on, 2007.*

[FLP01] Olivier Faugeras, Quang-Tuan Luong, and T. Papadopoulou. 2001. The Geometry of Multiple Images: The Laws that Govern the Formation of Images of a Scene and some of their Applications. MIT Press, Cambridge, MA, USA.

[FM05] J. Fung and S. Mann, "Openvidia: parallel gpu computer vision," in MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia. New York, NY, USA: ACM, 2005, pp. 849–852.

[FMS09] Frank, H., Mittnacht, A., Scheiermann, J. (2009): Throwing of Cylinder Shaped Objects. Proceedings on 2009 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM 2009), July 14-17, 2009, Singapore, pp. 59-64

[FVS11] Jianbin Fang; Varbanescu, A.L.; Sips, H., "A Comprehensive Performance Comparison of CUDA and OpenCL," Parallel Processing (ICPP), 2011 International Conference on , vol., no., pp.216,225, 13-16 Sept. 2011

[GH03] Günther W.A.; Heinecker, M.: Modulare Materialusssysteme für wandelbare Fabrikstrukturen - Bewertungs- und Gestaltungsrichtlinien für wandelbare Materialusssysteme. (2003)

[Gla09] Peter N. Glaskowsky. NVidia fermi: The first complete GPU computing architecture. Technical report, 2009.

[GSW03] Geusebroek, J.-M.; Smeulders, A.W.M.; van de Weijer, J., "Fast anisotropic Gauss filtering," Image Processing, IEEE Transactions on , vol.12, no.8, pp.938,943, Aug. 2003

[GW07] Gonzalez, Rafael C.; Woods, Richard E.: Digital Image Processing. 3rd Edition. Prentice Hall, August 2007. - ISBN 013168728X

[Hou62] P. Hough: "A method and means for recognizing complex patterns", U.S. Patent No. 3,069,654, 1962.

[HP62] Hough V, Paul C.: *Method and means for recognizing complex patterns*. December 1962.

[HS91] Hove, Barbara; Slotine, Jean-Jacques E., "Experiments in Robotic Catching," American Control Conference, pp.380-386, 26-28 June 1991

[HS95] Hong, Won; Slotine, Jean-Jacques E., "Experiments in Hand-Eye Coordination Using Active Vision," Lecture Notes in Control and Information Sciences, pp.130-139, 1995

[HZ03] Richard Hartley and Andrew Zisserman. 2003. Multiple View Geometry in Computer Vision (2 ed.). Cambridge University Press, New York, NY, USA.

[IK88] J.Illingworth and J. Kittler: "A survey of the Hough transform", *Computer Vision, Graphics and Image Processing*, vol.43, pp.221-238, 1988

[INH04] Y. Imai, A. Namiki, K. Hashimoto, and M. Ishikawa, "Dynamic active catching using a high-speed multifingered hand and a high-speed vision system," Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on, 2004.

[INI96] Ishii, I.; Nakabo, Y.; Ishikawa, M., "Target tracking algorithm for 1 ms visual feedback system using massively parallel processing," *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, vol.3, no., pp.2309,2314 vol.3, 22-28 Apr 1996

[JU97] Julier, S.J.; Uhlmann, J.K.: A new extension of the Kalman Filter to nonlinear systems. In: *Proceedings of AeroSense: The 11th International Symposium on Aerospace=Defence Sensing, Simulation and Controls, 1997*, 1997

[JWD13] Jacobs, L.; Weiss, J.; Dolan, D., "Object tracking in noisy radar data: Comparison of Hough transform and RANSAC," Electro/Information Technology (EIT), 2013 IEEE International Conference on , vol., no., pp.1,6, 9-11 May 2013

[KBS75] C.Kimme, D.Ballard, and J.Sklansky, "Finding circles by an array of accumulators", Proc. ACM 18, pp: 120-122, 1975.

[KDH10] K. Karimi, N. G. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," May 2010.

[Kol02] M. Kolazwole: *Radar Systems, Peak Detection and Tracking*, Newnes/Elsevier, 2002.
[Lin93] T. Lindeberg. Detecting salient blob-like image structures and their scales with a scale-space primal sketch: a method for focus-of-attention. International Journal of Computer Vision, 11(3):283-318, 1993.
[LWT12] Lourenco, L.H.A.; Weingaertner, D.; Todt, E., "Efficient Implementation of Canny Edge Detection Filter for ITK Using CUDA," Computer Systems (WSCAD-SSC), 2012 13th Symposium on , vol., no., pp.33,40, 17-19 Oct. 2012
[MGV09] Theo Moons, Luc van Gool, and Maarten Vergauwen. 2009. *3D Reconstruction from Multiple Images, Part 1: Principles*. Now Publishers Inc., Hanover, MA, USA.
[MGW11] Martinez, G.; Gardner, M.; Wu-chun Feng, "CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures," *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, vol., no., pp.300,307, 7-9 Dec. 2011
[MHM04] R. Mori and K. Hashimoto and F. Miyazaki (2004), *Tracking and Catching of 3D Flying Target based on GAG Strategy*; Proceedings of the 2004 IEEE Int. Conf. on Robotic & Automation, April, 5189-5194.
[MPD14] K. Mironov, M. Pongratz, D. Dietrich, "Predicting the Trajectory of a Flying Body Based on Weighted Nearest Neighbors"; in: "Proceedings ITISE 2014", Copicentro Granada S.L, 2014, 12 S.
[MS81] L.Minor and J.Sklansky, "Detection and segmentation of blobs in infrared images", IEEE Trans. SMC 11, pp: 194-201, 1981.
[NI03] A. Namiki and M. Ishikawa (2003), Robotic Catching Using a Direct Mapping from Visual Information to Motor Command; IEEE Int. Conf. on Robotics and Automation, September, 2400-2405.
[NI05] A. Namiki and M. Ishikawa, "The analysis of high-speed catching with a multifingered robot hand," Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on, 2005.
[NII03] Namiki, A.; Imai, Y.; Ishikawa, M.; Kaneko, M., "Development of a high-speed multifingered hand system and its application to catching," *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, vol.3, no., pp.2666,2671 vol.3, 27-31 Oct. 2003
[NIN97] Nishiwaki, K.; Ionno, A.; Nagashima, K.; Inaba, M.; Inoue, H., "The humanoid Saika that catches a thrown ball," *Robot and Human Communication, 1997. RO-MAN '97. Proceedings., 6th IEEE International Workshop on* , vol., no., pp.94,99, 29 Sep-1 Oct 1997
[NNI99] Namiki, A.; Nakabo, Y.; Ishii, I.; Ishikawa, M., "High speed grasping using visual and force feedback," Proceedings of IEEE International Conference on Robotics and Automation, 1999, pp.3195-3200 vol.4, 1999
[OC76] F. O'Gorman and M. Clowes: "Finding picture edges through collinearity of feature points", *IEEE Transactions on Computers*, vol.25(4), pp.449-456, 1976.
[OCL09] NVidia Corporation: "OpenCL Programming Guide for the CUDA Architecture", Version 2.3, August 2009
[OIN10] Ogawa, K.; Ito, Y.; Nakano, K., "Efficient Canny Edge Detection Using a GPU," *Networking and Computing (ICNC), 2010 First International Conference on*, vol., no., pp.279,280, 17-19 Nov. 2010
[OLG07] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrϒUger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," Computer Graphics Forum, vol. 26, pp. 80–113, March 2007.
[PKH10] Pongratz, M.; Kupzog, F.; Frank, H.; Barteit, D., "Transport by throwing - A bio-inspired approach," Industrial Informatics (INDIN), 2010 8th IEEE International Conference on , vol., no., pp.685,689, 13-16 July 2010
[PMB13] Pongratz, M.; Mironov, K.; Bauer, F.; "A Soft-Catching Strategy for Transport by Throwing and Catching", International Conference "Information Technologies for Intelligent Decision Making Support", Ufa, Russia, 2013, TU Wien, April 2013
[Pon09] M. Pongratz, Object Touchdown Position Prediction: A Stereo Vision Based Approach, Diploma Thesis, TU Wien, Wien 2009.
[PP12] M. Pongratz, KOROS Initiative: Automatized Throwing and Catching for Material Transportation, TU Wien, Wien 2012.

[RA02] M. Riley and C. G. Atkeson, "Robot catching: Towards engaging human-humanoid interaction," Autonomous Robots, vol. 12, pp. 119—128, 2002.

[RFQ03] Ali Ajdari Rad, Karim Faez, Navid Qaragozlou, "Fast Circle Detection Using Gradient Pair Vectors", Proc. VIIth Digital Image Computing: Techniques and Applications, December 2003.

[SC07] C. Smith and H. I. Christensen (2005), *Using COTS to Construct a High Performance Robot Arm*, In Proc. IEEE International Conference on Robotics and Automation, April 2007, Roma, Italy, pp. 4056-4063.

[SHB07] Sonka, Milan ; Hlavac, Vaclav ; Boyle, Roger: *Image Processing, Analysis, and Machine Vision*. 3rd Edition. Cengage-Engineering, March 2007. – ISBN 049508252X

[SHH07] S. S. Stone, H. Yi, J. P. Haldar, W. mei W. Hwu, B. P. Sutton, and Z. pei Liang, "How gpus can improve the quality of magnetic resonance imaging," in In The First Workshop on General Purpose Processing on Graphics Processing Units, 2007.

[SNI05] Shiokata, D.; Namiki, A.; Ishikawa, M., "Robot dribbling using a high-speed multifingered hand and a high-speed vision system," Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, (IROS 2005)., pp. 2097-2102, 2-6 Aug. 2005

[SPV05] Scaramuzza, D.; Pagnottelli, S.; Valigi, P., "Ball Detection and Predictive Ball Following Based on a Stereoscopic Vision System,"*Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, vol., no., pp.1561,1566, 18-22 April 2005

[SSL01] Slabaugh, Greg ; Schafer, Ron ; Livingston, Mark: Optimal Ray Intersection For Computing 3D Points From N-View Correspondences. (2001)

[SW10] Shucai Xiao; Wu-chun Feng, "Inter-block GPU communication via fast barrier synchronization," *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on* , vol., no., pp.1,12, 19-23 April 2010

[Sze10] Richard Szeliski, "Computer Vision: Algorithms and Applications", September 3, 2010 draft

[TKB99] Toyama, K.; Krumm, J.; Brumitt, B.; Meyers, B.: Wallower: principles and practice of background maintenance. In: *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on 1 (1999)*, S. 255-261 vol.1

[VSH04] VanDyke, M. C. ; Schwartz, J. L. ; Hall, C. D.: Unscented Kalman Filtering for Spacecraft Attitude State And Parameter Estimation. In: *2004 AAS/AIAA Space Flight Mechanics Meeting, Maui, Hawaii*, 2004

[Wei08] J. Weiss: „Real-Time Feature Detection Using the Hough Transform", Proceedings of the ISCA 21st International Conference on Computer Applications in Industry and Engineering, pp.168-173, Nov 2008

[WL12] Suping Wu and X. Liu, "Parallelization Research of Circle Detection Based on Hough Transform," *IJCSI International Journal of Computer Science*, vol. 9, 2012, pp. 6.

[Wri93] A. Wright, *A high speed low latency portable vision sensing system*, SPIE, September 1993

[Yak87] Yakowitz, S.: Nearest-Neighbour Methods for time series analysis, Journal of Time Series Analyses, Vol. 8, No. 2, 1987, pp. 235-247.

[YLJ10] Wang Yingshi; Sun Lei; Liu Jingtai; Yang Qi; Zhou Lu; He Shan, "A novel trajectory prediction approach for table-tennis robot based on nonlinear output feedback observer," *Robotics and Biomimetics (ROBIO), 2010 IEEE International Conference on*, vol., no., pp.1136,1141, 14-18 Dec. 2010

[YR08] Yuancheng Luo; Duraiswami, R., "Canny edge detection on NVIDIA CUDA," Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on , vol., no., pp.1,8, 23-28 June 2008

Internet references

[1] http://www.profil.at/home/autoindustrie-wirtschafts-motor-86604
May 2015

[2] http://history1900s.about.com/od/1910s/a/Ford--Assembly-Line.htm
May 2015

[3] http://www.gracesguide.co.uk/Oliver˙Evans
May 2015

[4] http://www.live-counter.com/autos/
May 2015

[5] https://hbr.org/1997/01/the-four-faces-of-mass-customization
May 2015

[6] http://www.quest-trendmagazin.de/artikel-archiv/einsatz-von-robotern-steigt-2011.html
May 2015

[7] http://www.britannica.com/EBchecked/topic/638321/weather-forecasting/49626/Numerical-weather-prediction-NWP-models
May 2015

[8] http://www.theweek.co.uk/world-news/middle-east/59368/iron-dome-how-israels-missile-defence-system-works
May 2015

[9] www.nennstiel-ruprecht.de/bullfly/index.htm.
May 2015

[10] http://graphics.stanford.edu/˜jplewis/lscourse/SLIDES.pdf
May 2015

[11] https://www.cs.unc.edu/˜welch/media/pdf/kalman˙intro.pdf
May 2015

[12] http://homes.cs.washington.edu/˜todorov/courses/cseP590/readings/tutorialEKF.pdf
May 2015

[13] http://ffden-2.phys.uaf.edu/webproj/211˙fall˙2014/Max˙Hesser-Knoll/max˙hesserknoll/Slide3.htm
May 2015

[14] https://www.linkedin.com/pulse/20140920022202-82311677-virtual-tennis-academy-lesson-1-slice-topspin-in-tennis
May 2015

[15] http://www.extremetech.com/extreme/197262-its-2015-self-driving-cars-are-more-than-a-promise
May 2015

[16] http://www.hawkeyeinnovations.co.uk
May 2015

[17] http://www.intorobotics.com/fundamental-guide-for-stereo-vision-cameras-in-robotics-tutorials-and-resources/
May 2015

[18] https://courses.cs.washington.edu/courses/cse455/09wi/Lects/lect16.pdf
May 2015

[19] http://www.vision.caltech.edu/bouguetj/calib˙doc/htmls/parameters.html
May 2015

[20] http://www.rmm3d.com/3d.encyclopedia/keystone/keystone.html
May 2015

[21] http://www.sensoray.com/support/appnotes/pixjiter.htm
May 2015

[22] http://www.mathworks.com/matlabcentral/fileexchange/40737-canny-edge-detector
May 2015

[23] https://github.com/oleander/ransac-and-hough-transform-java
May 2015
[24] https://www.khronos.org/opencl/
May 2015
[25] http://www.vision.caltech.edu/bouguetj/calib˙doc/
May 2015
[26] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-560ti
May 2015
[27] https://developer.nvidia.com/cuda-toolkit-65
May 2015
[28] https://developer.nvidia.com/opencl
May 2015
[29] http://www.cplusplus.com/reference/cstdio/
May 2015
[30] http://www.opencv.org
May 2015
[31] http://docs.opencv.org/doc/tutorials/introduction/display˙image/display˙image.html
May 2015
[32] http://docs.opencv.org/modules/imgproc/doc/miscellaneous˙transformations.html#cvtcolor
May 2015
[33] http://docs.nvidia.com/cuda/curand/#axzz3Zxtl60vR
May 2015
[34] http://www.cplusplus.com/reference/cstdlib/rand/
May 2015
[35] http://mathworld.wolfram.com/RodriguesRotationFormula.html
May 2015
[36] http://chronoengine.info/mediawiki/index.php/ChronoEngine:Coordinate˙transformations
May 2015
[37] http://becs.aalto.fi/en/research/bayes/ekfukf/
May 2015
[38] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-560ti/performance
May 2015