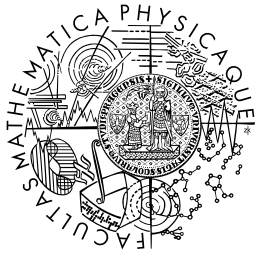


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Boris Kapustík

Controlling a robotic chess manipulator

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: doc. RNDr. Martin Kruliš, Ph.D.
Study programme: Computer Science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I have used a generative AI model for summarizations in parts marked as such. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my family and my friends for being my support and believing in me.

I would also like to thank my supervisor, doc. RNDr. Martin Kruliš, Ph.D., for his patience, guidance and invested time during the development of this thesis. I have gained a lot of experience and new knowledge.

Title: Controlling a robotic chess manipulator

Author: Boris Kapustík

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Martin Kruliš, Ph.D., Department of Distributed and Dependable Systems

Abstract: In this thesis, we will use Kinect v2 (from Microsoft Corporation), Stockfish (one of the highest-ranking chess engines), and a custom-made robotic crane capable of accepting simple commands to move across a 3-dimensional plane. The objective is to integrate software for boardgame (Chess) tracking using an easily accessible camera and depth sensor developed by Roman Staněk with an open-source chess engine to create a simple chess robot. Thanks to the tracing, the robot will have the ability to interact with users. The output will be a desktop application for controlling and configuring the robot, switching between game modes, and tracking the game. It will also require creating a virtual mock of the robotic crane to simplify testing and further development.

Keywords: Kinect, integration, computer vision, memory-mapped files, robotic chess-playing manipulator, inter-process communication

Název práce: Řízení robotického šachového manipulátoru

Autor: Boris Kapustík

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: doc. RNDr. Martin Kruliš, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: V této práci použijeme Kinect v2 (od Microsoft Corporation), Stockfish (jeden z nejlépe hodnocených šachových programů), a na zakázku vyrobený robotický manipulátor schopný přijímat jednoduché příkazy k pohybu po 3-rozměrné rovině. Cílem je integrovat software pro sledování deskových her (šachy) pomocí snadno dostupné kamery a hloubkového senzoru vyvinutýho Romanemem Stankem s open-source šachovým enginem k vytvoření jednoduchého šachového robota. Díky sledování, robot bude mít schopnost komunikovat s uživateli. Výstupem bude desktopová aplikace pro ovládání a konfiguraci robota, přepínání mezi herními režimy a sledování hry. Bude to také vyžadovat vytvoření virtuální napodobeniny robotického jeřábu pro zjednodušení testování a dalšího vývoje.

Klíčová slova: Kinect, integrace, počítačové vidění, Soubory mapované paměti, robotický šachy hrající manipulátor, Meziprocesová komunikace

Obsah

1 Introduction

The goal of this thesis is to implement a desktop application for controlling a robotic chess-playing manipulator and to integrate a legacy control application for chess tracking. The chess tracking application uses Kinect for Windows and computer vision algorithms to track chess figures on a chessboard.

The main motivation is to use this project in events organized by the university to demonstrate a robotic manipulator playing chess against a person using an affordable camera in a showroom. This comes with a number of real-world problems that will need to be solved, such as configuring the robotic manipulator to synchronize with the positioning of the physical chessboard and reconfiguring the setup after movements which would otherwise cause a fault state. Reconfiguring the chess tracking part will also be necessary to adapt to ever-changing light conditions.

This application provides input for our robotic manipulator and allows us to use our manipulator to play chess against a human user in real-time. We base this project on a project written in a legacy framework using a library that is no longer supported by newer technologies. Motivated to modernize the solution, we will explore propositions on how to integrate seemingly un-integrable parts of code while maintaining as much efficiency as possible.

All this will be achieved while keeping the code of incompatible frameworks maintainable. We will also integrate a chess engine that will compute moves to be used by our robotic manipulator to play against the human user. Executing the moves requires effective path selection in space, which will be achieved with the help of 3-dimensional path-finding algorithms.

All integrated parts will be united into one solution while maintaining modularization and a layered architecture according to the best practices used in software engineering.

The application will also provide options to replay an already-played game. This can be used to replay historic games of chess champions. Another option will be to watch the chess engine playing against itself.

The user of the application will be able to configure the setup - that is, position the robotic manipulator to find the chessboard in real space and reconfigure the application while retaining as much of the current setup as possible – that is, without the need to reconfigure the whole scene and state of the game.

1.1 Application specifications

The goal is to create a user-friendly graphical user interface that is simple to use yet provides all necessary features to correctly set up and use the robotic chess manipulator together with the chess tracking sensor. The project aims to be usable in a showroom where visitors will be able to play chess against the robotic manipulator, watch a historic match, or watch the computer play against itself. The application aims to make setting up parts of the project as simple as possible.

1.1.1 Configuring robotic manipulator

The first stage is to set up the robotic manipulator. We expect the robotic manipulator to be connected to the computer over USB. We need to select the right connection. After the right connection is selected and established, we need to establish where our robotic manipulator should expect the chessboard to be placed, what the size of the chessboard is, and where it should expect the chess pieces to be located. We need to set space for captured pieces. We do not want to depend on a specific chessboard and its specific location. We make it possible to use any reasonably sized chessboard and chess figures. The chess figures should be big enough to be picked up by the grip of the robotic manipulator. They need to be structured in a way that the grip can hold the figure without it falling. The size of the chessboard needs to be smaller than the space on which the robotic manipulator operates and big enough to place all the figures. There needs to be some additional space between the chessboard and the edge of the space on which the robotic manipulator operates so that the captured chess pieces can be placed there. It can not be too small, so the Kinect can detect all the pieces correctly.

The application will force the user to use a chessboard and chess pieces that satisfy all of the mentioned conditions, but other than that, it will give the user the freedom to use any such chessboard and chess pieces. It will not require the chessboard to be placed in any special location. Therefore, we need the application to provide a way to find the location of the chessboard and use it in the game. We implement that by forcing the user to move to the edges of the chessboard using manual robotic manipulator movement controls. All other positions, that is, square locations, are then computed by the application.

The movement controls needs to be slow and precise enough to correctly move to the location and fast enough so that the configuration is not too long. We manage this by being able to slow down or speed up the movement, similarly to controlling the direction of the movement. To visualize the current location of the robotic grip, we show the current 3-dimensional coordinates which can also help with the configuration.

The application needs to handle changes in the environment, such as the chessboard or, the robotic manipulator being moved, or the game being paused. We do not want to reconfigure the whole setup each time this happens. The application handles this by providing us with the option to pause the game and reconfigure the localization of the edges of the chessboard. We can then resume playing without restarting the game. We would like to keep the state of the game even after reconfiguring and be able to continue in the game. When reconfiguring, the application will have to keep the state of the game but also the state of the

configuration.

1.1.2 Chess representation introduction

Chess notations are used to record and store a game. These notations record moves in short strings readable by a human and uniquely describe each move of the game. They detail the actions of each player in a way that can be understood and reproduced by others familiar with the notation. Using any of these notations, one can reconstruct the entire game from start to finish. This is essential for analysis, coaching, publication, and archival purposes [1]. Most often used chess notations are:

- Algebraic notation - This is the most common modern notation and is used by FIDE (the International Chess Federation). It identifies each square on the chessboard by a unique coordinate, with moves described by the initial of the piece (omitted for pawns) and the start or destination square (e.g., Nf3, e4). Special moves like castling are noted as 'O-O' or 'O-O-O', and captures are denoted by an 'x' (e.g., Bxf3) [2].
- ICCF numeric notation - Used primarily for international correspondence chess by the International Correspondence Chess Federation. Each square on the chessboard is represented by two digits, where the first digit represents the rank and the second the file (e.g., 52 for e2). Moves are recorded as a pair of these numbers representing the starting and ending squares (e.g., 5254 for e2 to e4).
- Portable Game Notation (PGN) - Used to record entire games along with metadata about the game (players, event, date, result) [2]. It uses Algebraic Notation for the moves themselves but can include additional information, such as variations and textual commentary enclosed in curly braces or semicolons for comments. PGN files are often used to import and export collections of games into databases and chess software.

We will use the Chess notations to get input for our program to replay a record of a historic game. Our application, however, needs more sophisticated way to represent chess moves and work with the state of the game. That is because while the chess notations can uniquely represent each move and the game as a whole, there are several reasons why they are not suitable for working with a computer. They often do not specify the source and destination of the figures used in a chess move. They might not specify the figure that has been moved. They specify castling by special characters, but for the program we need source and destination positions of both the king and the rook. In general, the notations expect the reader to know many other details from the context of the game. This might be trivial for a person but would require the program to always compute many possible scenarios for each move in order to determine source and destiontion positions.

The application will need to be able to parse some of the notations and do the calculations when a game is loaded from a record. But the notation will need to be translated to a more suitable protocol which can be directly used for any computations. We will use Universal Chess Interface (UCI [3]) which is an open

communication chess protocol to represent the state of the game and the chess moves.

1.1.3 Playing the game

After configuring the physical location of the chessboard, the application lets us select a specific game mode, that is, which members will serve as input for the robotic manipulator. We will refer to the game modes as strategies. The architecture of the application is designed to easily implement a chess strategy. We have implemented the following strategies.

- Replay match
- Watch AI match
- Play against AI

Replay match lets us watch a record of a game in PGN chess notation. The application asks the user to select a file with a PGN record from the file explorer.

Watch AI match lets us watch a Stockfish [4] game engine play. The engine is designed not to remember the state of the game. The application needs to remember the state and always provide the engine with the correct input. The engine computes the next move, and the robotic manipulator performs the move. The Stockfish engine is open-source, and we expect it to be updated over time. We want to let the option of choosing the version of the engine be up to the user, and therefore, the Stockfish engine application is expected to be installed separately, and the application asks the user to select the Stockfish application from the file explorer. This will let the user freely decide whether he wants to use a newer version of the Stockfish, and if he decides to update it, he can do so without changing the code.

Play against AI match lets us use chess tracking provided by the Kinect [5] camera sensor and use the input from the camera to localize the chessboard and chess pieces. The chess tracking part of the application uses computer vision to achieve this and is part of a different thesis by Roman Staněk [6]. The algorithms require user-defined parameters to correctly compute the chess state. The application lets us configure these parameters to adapt to current light conditions and the placement of the Kinect camera relative to the chessboard, as well as the size and colors of the chessboard and chess pieces.

User interface provides options to configure the parameters mentioned above and show the state of the configuration. Numerous parameters need to be configured for optimal results, and the effect of these changes can be seen in real time, visualized in the user interface. The application shows us if the configuration is correct, and when it finally detects a move, this move is logged and provided to another part of the application in the UCI notation. The parameters will need to be reconfigured when the light conditions change without interupting the current game.

Independent of the game mode selection, the application lets us see the log of played moves in the user interface. It will give the option to pause the game. When paused the current move needs to be finished so that the state of the setup

is not abrupted and then the strategy can be changed. The play against AI and Watch AI match will be able to be played continuing the unfinished game prior to their selection. For example we can play a record of a game such as a historic game, pause the game and select the play against AI and try to play the game which a chess champion had started playing before.

1.2 Hardware specification

The application integrates two hardware components: Microsoft Kinect [5] and a custom-made robotic manipulator.

1.2.1 Kinect

Obrázek 1.1 Kinect sensor v2 [7]

As Engineers without borders from Oxford say [8], Kinect is a line of motion-sensing input devices produced by Microsoft, first released in 2010. The devices generally contain RGB cameras, and infrared projectors and detectors that map depth through either structured light or time of flight calculations [8].

As part of the 2013 unveiling successor of the Xbox 360, Xbox One, Microsoft unveiled a second-generation version of Kinect, which can be seen in Figure 1.1, with improved tracking capabilities 1.1.

Parameter	Value
Number of models tracker	6
Skeleton joints defined	26
RGB camera:	
Resolution(pixel)	1920×1080
field of view(degree)	84.1×53.8
frequency (Hz)	30
Depth camera:	
Resolution (pixel)	512×424
field of view (degree)	70.6×60
frequency (Hz)	30
minimal operative measure (m)	0.5
maximal operative measure (m)	4.5

Tabulka 1.1 Kinect v2 sensor characteristics

Kinect [9] has also been used in non-game applications in academic and commercial environments, as it was cheaper and more robust compared to other depth-sensing technologies at the time. While Microsoft initially objected to such applications, it later released software development kits (SDKs) for the development of Microsoft Windows applications that use Kinect. In 2020, Microsoft released Azure Kinect as a continuation of the technology integrated with the Microsoft Azure cloud computing platform. Part of the Kinect technology was also used within HoloLens project of Microsoft. Microsoft discontinued the Azure Kinect developer kits in October 2023.

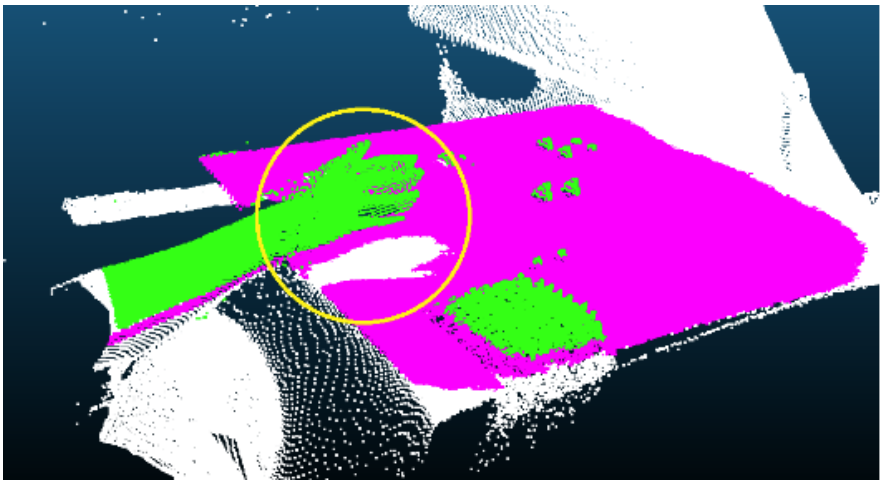
1.2.2 Depth sensor

The depth and motion sensing technology [9] at the core of the Kinect is enabled through its depth-sensing. The original Kinect for Xbox 360 used structured light for this: the unit used a near-infrared pattern projected across the space in front of the Kinect, while an infrared sensor captured the reflected light pattern. The light pattern is deformed by the relative depth of the objects in front it, and mathematics can be used to estimate that depth based on several factors related to the hardware layout of the Kinect. While other structure light depth-sensing technologies used multiple light patterns, Kinect used as few as one as to achieve a high rate of 30 frames per second of depth sensing.

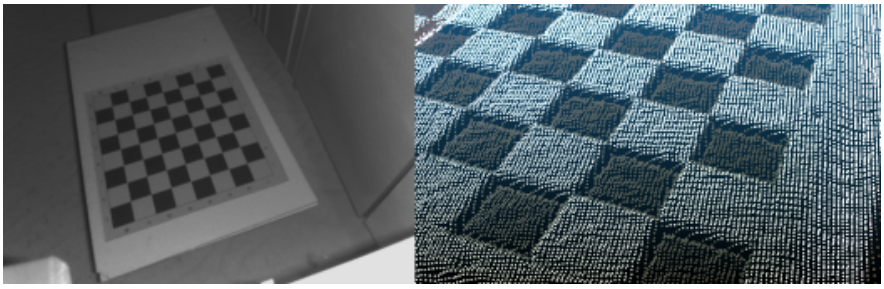
Kinect [9] for Xbox One switched over to using time of flight measurements. The infrared projector on the Kinect sends out modulated infrared light which is then captured by the sensor. Infrared light reflecting off closer objects will have a shorter time of flight than those more distant, so the infrared sensor captures how much the modulation pattern had been deformed from the time of flight, pixel-by-pixel. Time of flight measurements of depth can be more accurate and calculated in a shorter amount of time, allowing for more frames-per-second to be detected. [10]

As discussed in [6], among the main advantages of the technology used is the possibility of using multiple sensors for one scene, as there is a lower likelihood of

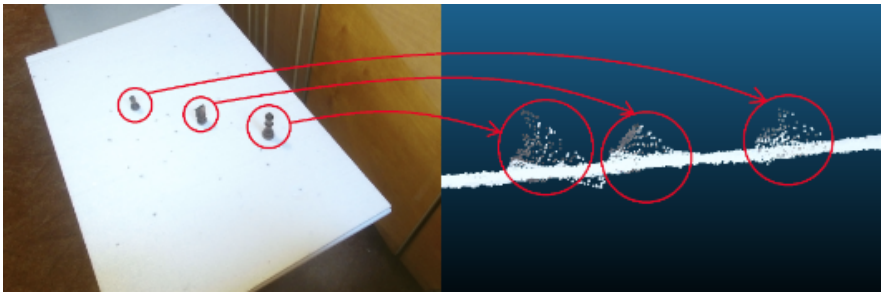
mutual interference. Due to the method of capturing and calculating distances, the sensor is also more stable when observing scenes with other sources of radiation, such as sunlight.



Obrázek 1.2 Visualization of the multipath interference problem [6]



Obrázek 1.3 Visualization of the unsuitable materials problem [6]



Obrázek 1.4 Visualization of the flying pixels problem [6]

However, several drawbacks can be observed [6]. The first significant problem is known as multipath interference [5]. The distance calculation assumes that we illuminate the scene and the reflected light falls directly onto the sensor. However, in the real world, this may not hold true, and reflections can occur, for example, in corners, or light can refract, for example, through transparent materials, and these reflected rays also fall on the sensor. This disrupts the accuracy of the calculation. As demonstrated in Figure 1.2 where white pixels can be observed beneath the plane of the table, which are caused by multipath interference.

Another problem is the observation of materials with poor properties [6] in terms of infrared reflection. For example, high absorption of radiation by the material will mean that very little information returns to the sensor, from which the distance cannot be determined. This effect is clearly visible in Figure 1.3, where the black squares are located a few millimeters lower compared to the rest of the chessboard.

The last problem mentioned is flying pixels at the edges of objects [6]. For example, if we observe an object one meter away with a background two meters away, the pixels on the edges of the object will not be able to receive correct information from just one object but will return a mixed value from both. Thus, the calculated distance of the edge pixels will lie in the interval between one and two meters. This can create non-trivial deformations of objects with a small surface, as can be seen in the Figure 1.4.

We have chosen Kinect partialy because of lack of other hardware options.

1.2.3 Robotic manipulator

The robotic manipulator has been custom-made and we do not have proper documentation, however, it is connected via USB and a serial port is used for communication. The manipulator works similarly to a crane. It has electric motors which enable it to move horizontally using wheels accross the x axis, while keeping space in between the wheels where it operates. Above the free space there is a grip connected to a pole. The grip is moved by an electric motor vertically and horizontally across the y-axis.

For communication, G-code commands are used. G-code [11] (also RS-274) is the most widely used computer numerical control (CNC) and 3D printing programming language. It is used mainly in computer-aided manufacturing to control automated machine tools, as well as for 3D-printer slicer applications. The G stands for geometry. G-code has many variants. The main movement requires 3D coordinates, which represent the destination where it should move. The commands which we use to control our robotic manipulator are defined in 1.2.

Command	**Description**
Xnnn Ynnn Znnn (rational numbers)	Move across X, Y, Z axis respectively
$H	Move home
M8	Open grip
M9	Close grip
$X	Reset
?	Get current state
!	Pause
G00	Set linear movement
$#	Get info

Tabulka 1.2 Robotic manipulator commands

Additionally, the robotic manipulator is equipped with a stop button, which immediately stops it from movement in case of emergency. This stops any command even when not finished just yet. Often after initialization and after stopping via

the stop button homing is required. That means that before executing any other command, the Move home command needs to be called. This moves the grip to what it considers an origin location.

1.3 Related work

1.3.1 Computer-vision and figure detection

Similar projects usually consist of a standard RGB camera or a thermographic camera. Input from the camera is usually used to identify a chessboard within an image based on its characteristics: position, orientation, and location of the squares using either

- Corner-based approach
- Line-based approach
- Heatmap approach

Similar projects use the following approaches to localize chess pieces on the chessboard:

FREDRIK BALDHAGEN [12] analyzes pixels near the centre of the squares of the chessboard in regard to their red, green, and blue values to detect if a square is vacant or not. This work uses the OpenCV library for computer vision using the Corner-based approach for chessboard recognition.

Another project, authored by David Vegas Romero [13], uses different Machine Learning and Deep Learning techniques to perform the recognition of the chess pieces. This approach requires training in a computer neural network model and a deep computer neural network model.

The chessboards usually need to be recognized and located first without the chess pieces on them.

1.3.2 Chess algorithms

The state of the game is then used to compute the next sufficient move. Moves are usually computed by a chess engine. Some of the most often-used chess engines include

- Stockfish
- Leela Chess Zero
- Houdini
- Berserk

The output of the chess engines is recorded in one of the mentioned chess notations 1.1.2.

1.3.3 Chessboard and figure manipulation

A robotic manipulator performs this chess move. Robotic manipulators can use electromagnets to hold a chess piece. A magnet is placed on top of the chess piece, and the robotic arm uses the electromagnet to pick up the figure and then place it on the correct chess square. Another approach is to use a magnet placed on the bottom of a chess piece and held from below the chessboard. An electromagnet is then used to move the piece without the need to pick up the piece above the chessboard and other pieces. This requires moving other pieces from the trajectory of the held piece or some kind of navigation between the pieces.

2 Problem Analysis

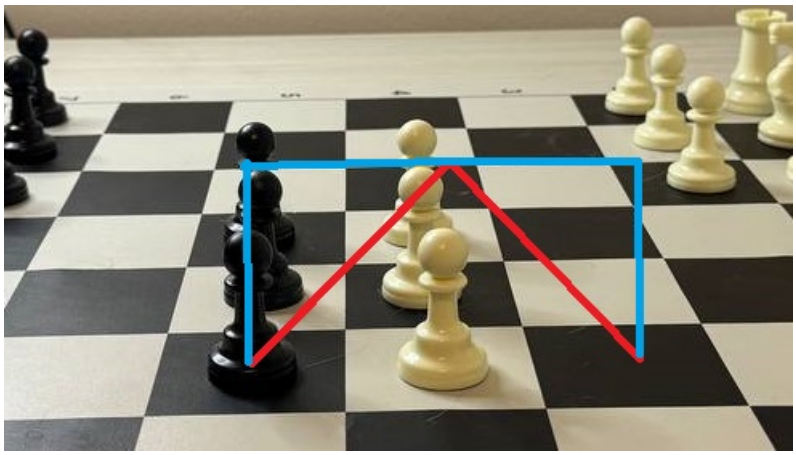
This chapter analyses algorithmic problems relevant to computer vision and robotics.

2.1 Path finding

Our robotic manipulator provides the option to choose between interpolated or non-interpolated movement. Our program architecture is modular and thus is not dependent on a specific API or physical chessboard, but to use different sizes of the chessboard, we need to compute the trajectory of the robotic hand when moving from point x to point y, either to pick up or position a figure. We want the trajectory to be as efficient as possible for the quick game movements, that is, only moving to necessary coordinates. There should also not be too many individual movements as the grip would stop at these positions for a short while, which makes the movement slower. Because, the project is expected to be shown in a show-room, we must also consider the visual effect of the path. As the robotic hand is not a point but rather a solid figure, we need to take its size into consideration, as well as the sizes of all individual chess pieces. The robotic hand must move between or, rather, above the obstacles, that is, the chess figures, without touching them. On top of that, the robotic hand can carry a piece, which must be taken into consideration as well. We realize that we are dealing with a path planning problem [14].

2.1.1 Naive approach

The simplest approach would be to set a constant 'safe' height at which the robotic grip would operate. Moving a piece from position x to point y, it moves only across the horizontal plane at this fixed height, which is above all the obstacles, accounted for holding a chess piece. Upon reaching the vertical alignment with the target position, the arm then transitions to a vertical movement to execute the piece pickup or placement.



Obrázek 2.1 Moving chess figure over another chess figure

While this approach would significantly simplify the pathfinding algorithm and minimize the risk of accidentally knocking other chess figures on the chessboard, it does introduce inefficiencies. The provided figure 2.1 shows a mockup of a simplified potential movement path for the robotic grip. The blue line depicts the path which the robotic grip would follow using this naive approach. The path would be longer than the shortest path, which would result in slower gameplay. The red line shows a path that simply follows the straight line between the source of the figure and the highest point of the figure, which is located between the source and the destination using interpolated motion. While that might not be the shortest path, it is almost as simple to compute while being significantly shorter, thus increasing the interactivity of the system. It also produces fewer individual movements, which adds up to the movement time and makes the visual aspect less appealing.

2.1.2 Shortest path

Finding an efficient path for our robotic chess manipulator is important in creating a seamless and dynamic gameplay experience. Ideally, we would find the shortest motion path. We can store the positions of the squares on the chessboard and the positions of captured pieces. Each square can contain an entity, that is, a chess figure with a known height. The size of each square is computed and known at the time of finding a path.

We could look at the squares as vertices of a graph. But because the shortest path would cross the edges of the square under an angle which is not a multiple of 90 unless the two points, that is, the squares, obtain a non-discrete space.

To manage path-finding effectively, we could consider discretizing the space by selecting specific points that represent potential robot positions on the chessboard. However, the directions of movement between these points can vary infinitely, complicating the pathfinding process. To accurately compute paths at any possible angle, we would need to define a sufficiently dense grid of vertices. This involves identifying and storing a vast number of points to ensure that all potential movement directions are covered, allowing for precise navigation and obstacle avoidance in a dynamic environment.

For that, we would need to store an impractical number of points. A graph created from such vertices would need to be recalculated as the space with the obstacles, that is, chess figures, change. Existing algorithms, such as The classical approaches, include A*, Dijkstra, and PRM (Probabilistic Roadmap Method) usually uses graphs for pathfinding.

As Jihee Han says in his publication [15], in general, given the start and target locations, the goal of path planning is defined as planning a collision-free path from 3D obstacles while satisfying certain criteria, such as distance, smoothness, or safety. As mobile robot path planning is considered to be NP-hard, 3D path planning is also NP-hard with an additional axis for height. More recently, path planning in three dimensions has been studied with heuristic approaches, such as soft computing, meta-heuristics, and hybridized heuristics with classical methods.

Finding the shortest path would be ideal, given that it is an NP-Hard problem in 3D with obstacles. Given the non-discrete characteristics of our space, that is a chessboard, there would be a significant number of vertices which would

need to be taken into consideration when computing the shortest path which would be computationally inefficient and would be difficult to implement. We will thus try to use a heuristic to find sufficiently short path balancing accuracy with computational feasibility.

2.1.3 Sufficiently short path

Instead of finding the shortest path, we will try to create an algorithm which uses a heuristic which might find the shortest path, but even when it does not, the found path would still be short enough for the motion to be efficient.

We will start by finding the shortest horizontal path without considering the obstacles. We can do this because we are operating in 3 dimensions and can move above all the obstacles since the chess figures are located at the bottom of the chessboard.

Finding the line connecting two points is easy; it is the Euclidian distance, but to translate this into the movement on the chessboard, we need to find all the chessboard squares through which the line passes. All intersected positions on a grid passed through by a line are sometimes referred to as a line super cover. Horizontally, our robotic arm would move on this line, and the only problem we need to solve is efficient vertical movement for obstacle avoidance. Vertically, each move consists of picking up the figure from the chessboard and moving it to the lowest carrying position required to reach the destination point without crashing into obstacles, and similarly moving the figure down to place it onto the chessboard.

Each time the path of the robotic arm horizontally crosses a figure, it needs to be above the figure. Therefore, we can divide each move into two parts. First, we find the highest point of the highest obstacle located on our path. Until this point is horizontally reached, the arm only needs to vertically move upwards. Similarly, after reaching the highest point, the robotic arm only moves vertically downwards.

As we want the shortest path possible, we want to use interpolated movement to reach the subpoint that is either the highest or the lowest point. However, other obstacles can be located on the 3-dimensional line between the position of the arm and the subpoint. In the upwards-moving phase, we iterate through all horizontally intersected chess figures, and the robotic arm moves just above the highest point of a figure, which is Next, it will be horizontally intersected if the highest point of the figure is higher than the current vertical position of the arm.

We have managed to simplify the problem into two parts, which is the horizontal and vertical movement. The horizontal movement will be simplified to finding the line supercover, thus finding all the intersected squares. When there is an obstacle on the intersected squares, we will avoid the obstacle vertically.

This solution is simpler than trying to solve an NP-hard problem and takes up less computational resources, while generating an optimally short path sufficient for fast movements of the robotic grid and thus sufficient for the responsiveness of the whole system.

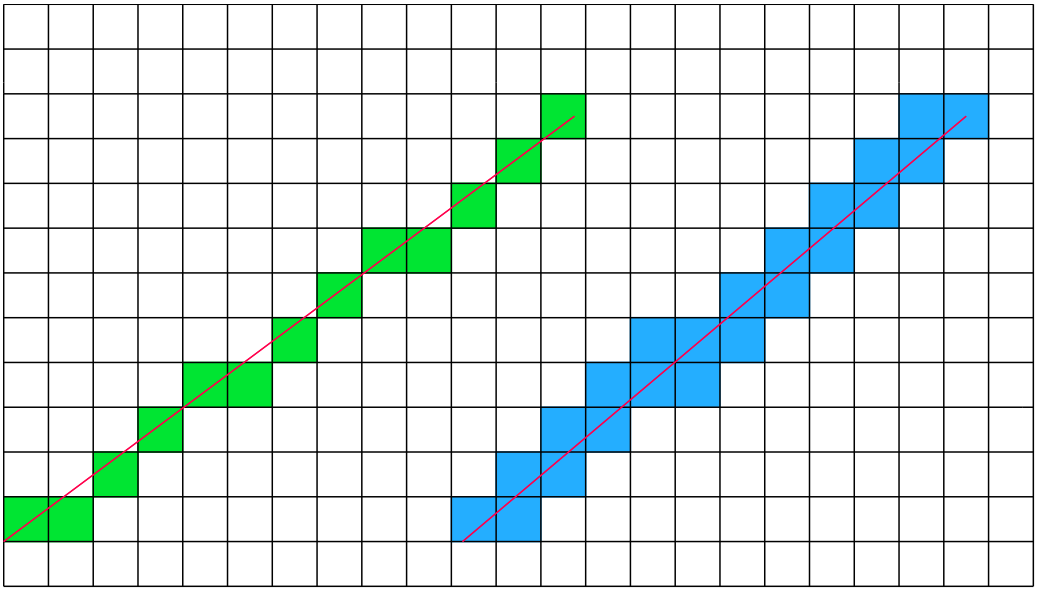
As for the super cover, we essentially need to find which squares on the chessboard are intersected by a line drawn between two points, for which we need a line-drawing algorithm that accounts for which squares the line passes through. This is similar to the problem solved by algorithms like the Bresenham

line algorithm [16] but with a focus on identifying all the squares that a line intersects rather than drawing the line square by square. Our problem is similar to a line drawing or line traversal algorithm that has been adapted to identify all the squares (or tiles) on a grid where a line intersects.

2.1.4 Bresenham-based super cover line algorithm

We will use an algorithm created by Eugen Deduv [17]. This line is called the super cover line, and this algorithm might be a particular case of the DDA (Discrete Differential Analyzer) algorithm.

Figure 2.2 shows the difference between the Bresenham algorithm (in green) and this one (in blue).



Obrázek 2.2 Difference between Bresenham classic and customized algorithm

If the Bresenham algorithm does not change the y-coordinate (the next point is C), this means that D will not be drawn, so we pass directly to C, and we go to the beginning again. The other case is when the Bresenham algorithm changes the coordinate, that is when the next point is B. In this case, both C and D have also to be checked. As seen in the figure, we can know if a point is drawn or not by the following relation:

```
(octant == right->right-top for directions below):
if (error + errorprev < ddx)  // bottom square also
    POINT (y-ystep, x)
else if (error + errorprev > ddx)  // left square also
    POINT (y, x-xstep)
else // corner: bottom and left squares also
    POINT (y-ystep, x)
    POINT (y, x-xstep)
```

Error is the current error (in point B), while errorprev is the previous error (in point A). Remember that the error is the 'distance' (non-normalized) from the

ideal point to the grid line below the ideal point. The difference with Bresenham is that ALL the points of the line are used, not only one per x coordinate.

Here we have supposed that if the line passes through a corner, the both squares are drawn, because the robotic grip would have a non-zero width. Note: The line is symmetric; that is, a line from x_0,y_0 to x_1,y_1 is the same as a line from x_1,y_1 to x_0,y_0.

2.1.5 Final movement trajectory

We have computed all intersected squares on the chessboard, and now we want to combine these squares together with our approach to vertical movement 2.1.3. Our robotic arm has a certain width, as do the chess figures, which we need to take into consideration; therefore, the movement can intersect more than one figure at the same time. We group these figures to make the path shorter and look at them as if they were one obstacle.

The squares are grouped either by their row or column in a grid, that is, the chessboard. If the difference between the row of the source and target is bigger than the difference between their respective columns, we group them by rows and vice versa.

Then, we can take into consideration only the highest point of the highest figure in the group. As we know the source and target coordinates, we can use the line equation to compute the exact coordinates at which each group is horizontally intersected and finally use these points as input in the previously mentioned algorithm 2.1.3.

2.2 Initial state of Chess tracking

The main focus of this section is to give us an idea about algorithms used for chess figures, chessboard and plane localization. This section is a sumarization[1] of decisions and explanations of the used algorithms for the chess tracking project in the thesis by Roman Staněk [6].

2.2.1 Algorithm selection

In the quest to identify the game plane within a point cloud for chess analysis, multiple approaches were considered, with the RANSAC [18] algorithm ultimately selected for its simplicity, flexibility, and adaptability to various parameters. This choice was informed by a comparative analysis of several common techniques for geometrical primitive detection in spatial data, namely RANSAC, Hough Transformation [19], Region Growing [20], and Linear Regression.

RANSAC (Random Sample Consensus) [18] emerged as the most suitable due to its probabilistic nature, allowing for efficient model estimation by iteratively selecting a minimal subset of points to form potential models and evaluating their fit against the dataset. Its advantages include the ability to adjust accuracy and processing speed through iteration control and its inherent design to work directly with input data, minimizing the exploration of implausible solutions. Despite its

[1]This summarization was made with the help of a generative AI model.

reliance on predefined threshold settings and the possibility of not exploring all viable model variations, RANSAC's straightforward implementation and capacity for heuristic enhancements made it the preferred choice.

The selected algorithm implementation involves identifying the largest plane in the captured scene that could feasibly support a chessboard, under the assumption that the chessboard lies on a significant, continuous plane from the perspective of the sensor. This approach simplifies the task by assuming a standard orientation for the sensor relative to the ground, enhancing usability by avoiding the detection of unsuitable planes.

Other considered algorithms included the Hough Transformation, effective for exploring a substantial portion of the solution space but challenged by computational intensity and parameter setting; Region Growing, suitable for contiguous regions but limited by noise sensitivity and computational demands; and Linear Regression, ideal for data closely fitting a model but susceptible to noise interference. While powerful, these methods were deemed less practical for real-time application due to their complex implementation requirements and sensitivity to data quality.

The implementation decision was further refined by the stipulation that the chessboard must reside entirely within a single plane, focusing on the largest continuous area on the table surface as determined by sensor-captured data points. The model estimation of this plane is further refined, when necessary, using linear regression, under the assumption of having identified points reasonably close to the model plane, thereby simplifying the task from a general search to a more focused refinement within already segregated data.

2.2.2 Looking for the chessboard

After figuring out where the game is played, we pinpointed the area for the chessboard. This step helped us cut down on unnecessary data and noise, making it easier to find exactly where the chessboard is and how it's positioned.

We looked at different methods [21] that help recognize a chess game from an image. These included techniques like edge detection, corner detection, and using Hough transformations [19], which are great for spotting the patterns of the chessboard.

In the end, we chose to use corner detection because it is good at spotting where the lines of the chessboard meet at the corners. These corners are crucial for mapping out the chessboard correctly. By identifying these corners, our custom algorithm could accurately place the chessboard in the 3D space.

This method made it straightforward to use computer vision technology to find the chessboard and figure out its orientation and exact position in a busy visual scene.

The challenge of finding the chessboard is multi-faceted, with applications ranging from camera calibration using empty chessboards to adjusting for lens distortions like radial blurring. Although computer vision libraries offer functions dedicated to these calibration tasks, they are not suitable for identifying occupied chessboards. An impractical workaround would involve locating an empty chessboard first, then adding pieces, a process cumbersome to repeat if the chessboard or camera moves.

An alternative approach, capturing the chessboard from a bird perspective to minimize piece interference, demands a specific camera setup that lacks flexibility.

Corner Detection Usage: The task often employs various computer vision methods, such as edge or corner detection. Corner detection transforms the color image to grayscale and applies a corner detection algorithm, identifying rapid contrast changes in multiple directions. This method aims to locate all square corners where two black and two white squares meet. However, it is sensitive to parameter settings, as it may also identify piece transitions, decorative chessboard borders, or external objects as corners as can be seen in the Figure 2.3c.

To accurately find where the chessboard is in 3D space, we use a detailed method to choose points around a reference point, called B. Here is how it works:

- Select Points Near B - We pick several points close to B, aiming for at least four to avoid errors.
- Pair Points - Each pair should be about the same distance from B, helping to form a square shape similar to a chessboard. The angles between each pair and B should be close to 90 degrees, matching the square corners of a chessboard.
- Build Chessboard Models - Using these pairs, we create different possible layouts of the chessboard. Each model represents a potential way the chessboard could be set up based on the data.
- Evaluate Models - For each chessboard model, we check how well it fits with the actual data collected by sensors. We look at whether the corners in the model match up with the real points observed.
- Choose the Best Model - The model that best matches the actual data is chosen as the most accurate representation of where the chessboard is in space.

This method focuses on ensuring that the model chosen is geometrically consistent and closely matches the real-world data, providing a reliable way to determine the position of the chessboard. The model with the smallest sum of Euclidean distances between its grid points and the nearest actual data points is considered the most successful, indicating a close match to the measured data and thus an appropriate model for the spatial position of the chessboard.

Edge Detection Usage

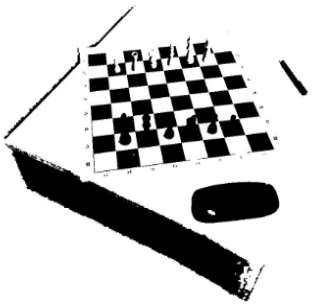
Similar to corner detection, edge detection involves converting images to grayscale or binary (black and white) to detect rapid contrast changes, but in one direction. This method is somewhat more forgiving regarding parameter settings.

For example, while a single piece might create an undesirable corner in corner detection, it wouldn not form an entire edge by itself in edge detection. However, unwanted edges may still be identified and subsequently filtered based on length or angle as depicted in Figure 2.3d.

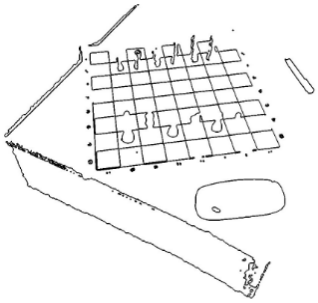
Among the explored methods, edge detection using Hough transformation was chosen for identifying significant chessboard features due to its resilience against

(a) Colorful chessboard shot

(b) Gray scale



(c) Threshing



(d) Edge detection

(e) Hough transformation

(f) Edge filtering

Obrázek 2.3 Steps of edge detection [6]

distortions by chess pieces and ease of implementation. This process required image preprocessing, converting the color image to grayscale, then binary, to highlight high-contrast regions indicative of the grid of the chessboard.

This binary image served as input for the Canny [22] edge detector, which looks for changes in contrast indicative of edges. The subsequent application of the Hough transformation located all significant lines representing potential boundaries of the chessboard, which were then filtered to isolate those relevant to the chessboard grid as depicted in Figure 2.3f.

In summary, by employing edge detection and Hough transformation, critical features of the chessboard were identified, enabling the determination of its position in spatial data using depth sensor information. This approach illustrates the integration of computer vision techniques and spatial analysis for accurate chessboard localization within a physical space.

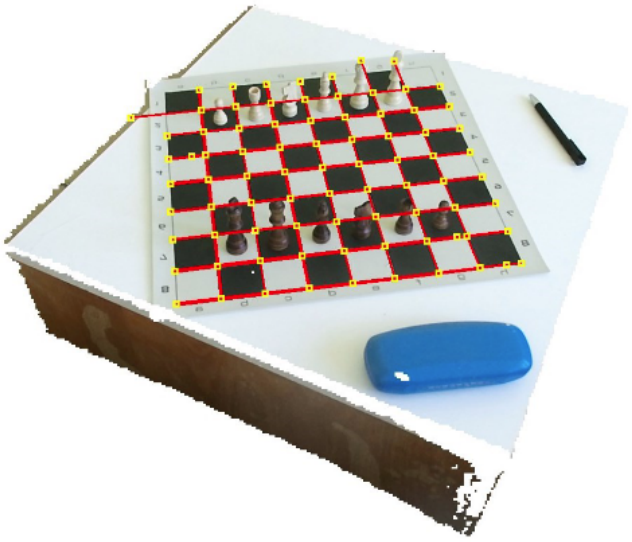
2.2.3 Localization in space

The spatial localization process aims to precisely map the edges of the chessboard from a 2D image to 3D points. This task is complicated by several factors, such as the potential for chess pieces to obscure the edges and the limitations of the Hough transformation to detect them. The subsequent filtering process, designed to eliminate unwanted edges by grouping them into two categories, may not remove all discrepancies. Moreover, mapping a point from the color image that is believed to be an edge might result in a 3D point belonging to a piece rather than the actual edge, leading to incorrect spatial information.

The proposed solution addresses these inaccuracies by selecting key points that carry crucial information, mapping them into space, and fitting them with a grid approximating the chessboard, ensuring proximity to these points while avoiding misalignment caused by incorrectly mapped points. The focal points for this task are the intersections of the vertical and horizontal edges of the chessboard available from the image, which represent the precise corners of the chessboard squares and carry significant information about its position and shape as depicted in Figure 2.4. Additional points of interest include the ends of the edges of chessboard, particularly when an edge is not detected, as these provide information about the boundaries of the chessboard.

Coordinates for these points in 3D space are obtained using the mapping function of the sensor between the color image and spatial data. These coordinates are processed with an algorithm conceptually similar to the RANSAC algorithm, described in an earlier section. The operation of the algorithm is demonstrated with a smaller problem of fitting a 2x2 chessboard grid with nine points, which can be seen in Figure 2.5.

For each acquired point, referred to as B, several nearest points from the surrounding areas are selected—ideally four, but more may be chosen to account for potential inaccuracies. The aim is to choose additional points that share a square with B. From these surrounding points, all possible pairs are generated that satisfy conditions regarding distance from B and the angle they form with B. The selected pairs are then used to generate all 64 potential chessboard models to determine the position of our square within the grid. For each model, the positions of the corners of the chessboard are calculated.



Obrázek 2.4 Display of edge intersections [6]

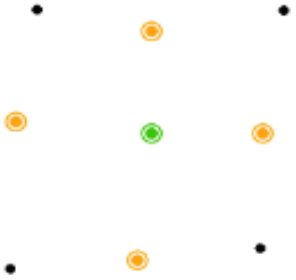
The best chessboard model is determined through an algorithm that inputs the set of points B (intersections), with parameters for deviations in the lengths of vectors defining the chessboard (e1) and the angle deviation from right angles (e2), and the number of neighbours considered for each point (p). The algorithm initializes an empty chessboard model and iteratively selects and evaluates point pairs from the neighbours, refining the model based on the cumulative distance of these points to measured data points. The final model with the lowest cumulative distance is deemed the most successful, indicating a close match to the measured data and, thus, the most suitable model for the position of the chessboard in space, as can be seen in Figure 2.5.

2.2.4 Figure localization

In the process of spatial localization, if all edges of the chessboard were identified from The colour image and direct mapping methods could be used to translate these lines into 3D points. However, the situation is complicated by several factors:

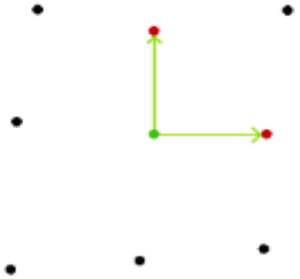
Edge detection might not capture all chessboard edges, especially when obstructed by a a cluster of pieces, rendering the Hough transformation ineffective. [23] Filtering aimed at removing undesirable edges by categorizing them into two groups may not address all the discrepancies.

If there are edges in the scene parallel to one of the groups, such as decorative edges of the chessboard or table edges, they may be mistakenly included as part of the chessboard. Translating a point considered to be a chessboard edge from the colour image to a spatial point could inaccurately map to a piece covering the intended edge, thus providing incorrect spatial positioning. The proposed algorithm aims to resolve these inaccuracies by selecting significant points, mapping them into space and interpolating a chessboard grid that closely aligns with these points without being skewed by inaccurately mapped points.

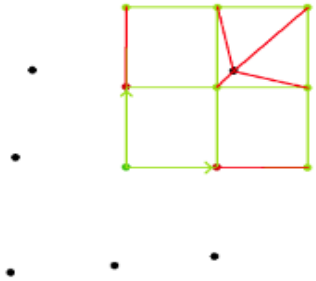


(a) The processed point (in green) and its neighbors (in orange)

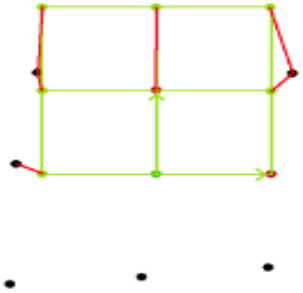
(b) Pair of lines



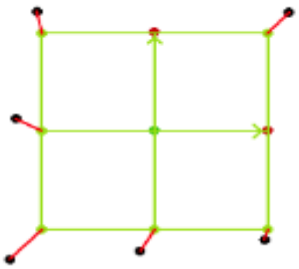
(c) The processed pair



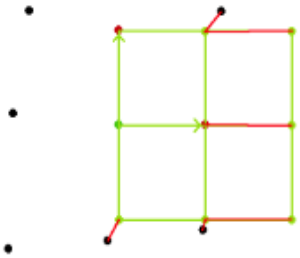
(d) First chessboard model



(e) Second chessboard model



(f) Third chessboard model



(g) Fourth chessboard model

Obrázek 2.5 Chessboard fitting algorithm with points [6]

Focal points include intersections of the vertical and horizontal edges of the chessboard, representing precise corners of the squares, and provide substantial information about the position and shape of the chessboard, as shown in Figure 2.6b.

Points at the ends of the edges of the chessboard are also considered, especially when an edge is not detected, as they provide boundary information.

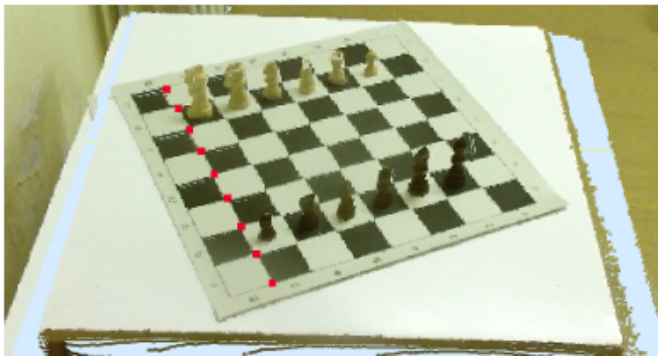
Using mapping of the sensor function between the colour image and spatial data, the 3D coordinates of these points are obtained. A RANSAC-like algorithm is then applied to these points, demonstrated a smaller problem of interpolating a 2x2 chessboard grid with nine points, as shown in Figure 2.6c.

Finally, detecting the presence and the type of chess pieces is enhanced by applying the Canny edge detector on depth data, as depicted in Figure 2.7. The depth data, converted to grayscale, provides a means to identify the height changes corresponding to the depth variations of the chessboard. The Canny edge detector, known for its efficiency in edge detection tasks, outlines the shapes of chess pieces based on these depth variations.

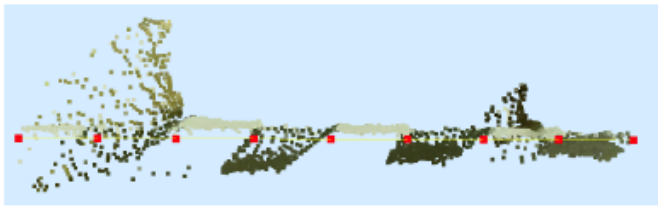
In Figure 2.7, the edge detection algorithm highlights the boundaries of objects within the grayscale depth image. The white pixels, which represent areas of significant depth change, help distinguish the chess pieces from the flat surface of the chessboard. This method is crucial for spatially locating pieces on the chessboard because it reveals the contours of the pieces that are not visible in the conventional colour data due to lighting or colour similarities.

This detection is part of a more extensive process to determine the presence and possibly the type of chess pieces based on their outlines. The contrast between the heights of the pieces and the chessboard surface allows the algorithm to segregate the pieces from the background effectively. The edges identified by the Canny [22] algorithm serve as crucial data points for further computational tasks, such as piece recognition and precise spatial localization on the chessboard.

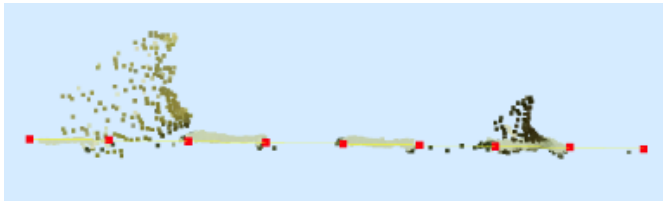
By integrating this edge detection technique, the algorithm significantly reduces the computational ambiguity caused by overlapping pieces or shadows, providing a more accurate and reliable means of analyzing the physical layout of the chess game.



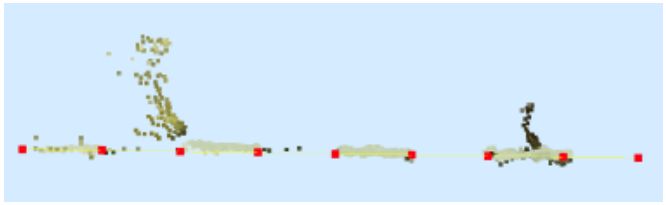
(a) Point cloud containing the chessboard



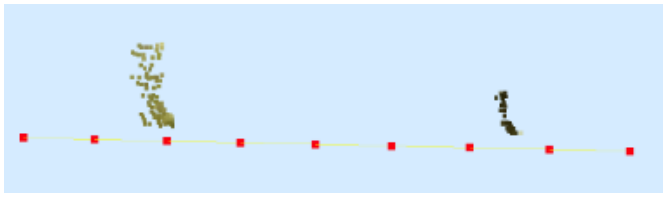
(b) Chess pieces from left



(c) Data after removing the black squares



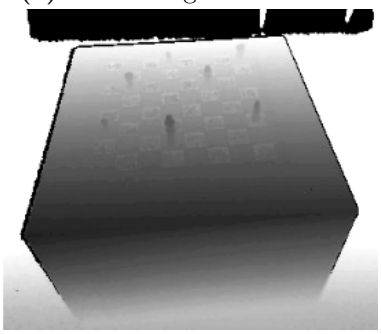
(d) Data after removing the flying points



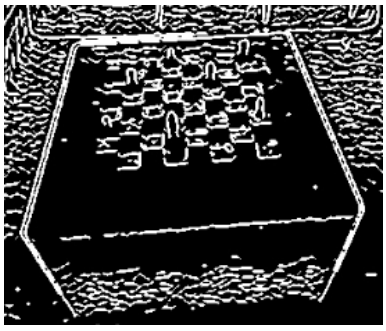
(e) Data after removing points from the squares of the chessboard

Obrázek 2.6 Individual steps of piece localization [6]

(a) Color image of the scene



(b) Depth data as grayscale



(c) Canny edge detector on depth data. White pixels will not be used for piece detection

Obrázek 2.7 Application of the Canny edge detector on depth data [6]

3 Technical analysis

3.1 Technology

This thesis is focused on developing a control application with UI which integrates other technologies, the Chess Tracking application is taken as a foundation for our application. Chess Tracking application provides UI, which enables the user to capture a chess game with the Kinect. The tracking is light sensitive and requires configuration. The effects of the configuration can be seen directly in the UI which lets the user validate the configuration by seeing the effects on the chessboard squares and figures.

Kinect API is available for C++ and legacy .NET Framework [24]. There are also 3rd party library options written in modern .NET. We do not want to rely on a 3rd party library. Chess tracking is only one part of this project which should not lead us to write the whole application in legacy platform. For simplicity we would like to write the application using one programming language. Creating the application, integrating all the various parts and creating a UI in C++ would be out of scope of this thesis. Therefore we will use .NET Framework to create Chess Tracking application, which offers official Kinect API developed by Microsoft. It offers good options for creating a GUI. It makes it easy to integrate other technologies and offers a good compromise between development speed and the actual speed of the program. There are many supported libraries, documentation and community. Thus we use modern .NET to write most of our application and run both programs as separate processes.

We want our application to be integrated as one application to simplify configuration and use it in a showroom. Capturing input from the Kinect will be run in a program as a subprocess of the main application. This program will rely on the old framework and preprocess the data to a form which can be transfered to the application written in modern .NET.

3.1.1 IPC

Main process and Kinect input tracking subprocess need to establish Interprocess Communication (IPC) [25]. Main requirement for the communication is speed. Kinect API does not define the size of the captured data. It is a structure consisting of primitive type [26] arrays, the size of which varies frame by frame. After testing the average size comes at around 40MB per frame. For detection and smooth vizualization of the tracking we need at least around 5 frames per second.

Another IPC parameter requirement is the choice between sequential and random access. Random access is better suited for highly random reads and writes of smaller data. Random access provides more freedom for memory management but also requires more work to manage the memory. The stream is more suitable for sharing larger sequential data between processes. This approach is more straightforward but comes with the overhead of stream management and is more limiting.

Read/write operations are the most computationally expensive part of the chess tracking process. More frames are captured than what is able to be read/written,

and therefore we need to discard some data in order for the algorithm to make sense. Otherwise, we would pile up a lot of data in the stream and we would get a high latency in the results of localisation algorithms and later display the frames in the UI. Random access offers us the option to partition the data which is about to be read/written and work on each part in parallel, which we need for optimisation. Random access memory management gives us direct control of specific memory segments, which makes it easier to synchronise access to different positions in the memory. IPC is resource intensive and given the size of the data and the transfer frequency requirement, we will need to take use of paralelization techniques, therefore random access is the better choice.

We do not want to persist data because the frames will be processed in a pipeline using computer-vision localisation algorithms and shown in the UI, after which they are disposed of. There are multiple options for IPC. We will be using Memory-Mapped files, but we will go through some other options and discuss why Memory-Mapped files are the right choice.

IPC over network

.NET provides many ways for IPC over a network. We do not need to share data over the network as our scenario is mostly suitable for local communication. We could theoretically record Kinect input on one device and work with other parts on a different device, but as the Kinect needs to be physically located close to the chessboard and the robotic manipulator, it would not make sense. The data that need to be sent are quite large, so the network overhead would become a problem.

Pipes

As Gulzar Group of Institutes Khanna states [27], pipes provide IPC by creating a unidirectional or bidirectional channel between them. A pipe is a virtual communication channel that allows data to be transferred between processes, either one-way or two-way. Pipes can be implemented using system calls in most modern operating systems Microsoft provides support for anonymous and named pipes [28]. Anonymous pipes are useful for communication between threads, or between parent and child processes where the pipe handles can be easily passed to the child process when it is created.

Named pipes provide interprocess communication between a pipe server and one or more pipe clients. Named pipes can be one-way or duplex. They support message-based communication and allow multiple clients to connect simultaneously to the server process using the same pipe name.

Pipes only provide sequential stream-based access which makes them unsuitable, considering we require random access IPC communication.

Memory-Mapped files

A memory-mapped file [29] is a form of shared memory which contains the contents of a file in virtual memory. As IBM states [30], Memory-mapped files provide a mechanism for a process to access files by directly incorporating file data into the process address space. This mapping between a file and memory

space enables an application, including multiple processes, to modify the file by reading and writing directly to the memory.

Accessing memory-mapped files is faster than using a type of persisted memory, which uses direct read and write operations for two reasons. Firstly, a system call is orders of magnitude slower than a simple change to the local memory of a program. Secondly, in most operating systems the memory region mapped is actually the page cache (file cache) of the kernel, meaning that no copies need to be created in user space.

.NET offers two variants for managing memory-mapped files, namely `MemoryMappedViewStream` and `MemoryMappedViewAccessor`. The ViewAccessor provides randomly accessed views of a memory-mapped file, unlike the ViewStream, which works sequentially. This gives us a clear choice of using the Memory-mapped files with `MemoryMappedViewAccessor`.

3.1.2 GUI

Developing control application will require chosing between GUI framework options. We seek a framework which is modern and simple to use. Part of our application will need to be developed using the .NET Framework, which only supports Windows. Therefore we can use a Windows-only GUI Framework, which will let us take use of easier integration with Windows features and optimised performance. The following are the available options.

Windows Forms

Windows Forms [31] is a UI framework for building Windows desktop apps. It provides one of the most productive ways to create desktop apps based on the visual designer provided in Visual Studio. Functionality, such as drag-and-drop placement of visual controls, makes it easy to build desktop apps.

With Windows Forms, you develop graphically rich apps that are easy to deploy, update, and work while offline or while connected to the internet. Windows Forms apps can access the local hardware and file system of the computer where the app is running.

This UI Framework has been broadly used and, therefore, provides extensive documentation and many examples. Microsoft does not concentrate on developing this framework, so it can be considered a legacy. Applications can be memory intensive.

WPF

Windows Presentation Foundation (WPF) [32] is a UI framework that is resolution-independent and uses a vector-based rendering engine, built to take advantage of modern graphics hardware. WPF provides a comprehensive set of application-development features that include Extensible Application Markup Language (XAML) [33].

One of the aims of Microsoft was to flexibly couple business logic and user interfaces with the creation of WPF. The framework also makes it possible for you to leverage design patterns like MVVM (Model-View-ViewModel) [34] with WPF.

WPF has a steep learning curve, much steeper than other GUI frameworks [35]. While hardware acceleration is an advantage of WPF, and makes it possible to have smoother animations and improved responsiveness, there is a disadvantage too. This hardware acceleration can be resource-intensive and take too much processing power and memory [35].

WinUI

The Windows UI Library (WinUI) [36] is a native user experience framework for both Windows desktop and UWP [37] applications. It makes it easy and efficient to use Windows API. Similarly to WPF, it uses XAML [33] markup language. It offers support for modern UI patterns, such as MVVM [34].

Final GUI choice

Both WinUI and WPF, unlike Windows Forms, take us of the XAML markup language for UI layout. The advantages are data bindings, adaptive layout and flexible styling. Both offer better UI design and make it easier to use UI patterns, which make the code more sustainable and scalable. WinUI is more modern than WPF, and its design is more polished and intuitive. It is more optimised for memory usage and performance. It is easier to learn than WPF and using the native code is in general faster. We will, therefore, use WinUI.

3.2 Working with chess state

We have already mentioned that we would like to keep the state of the game during reconfiguration, pauses or while changing the game. Representing the chessboard itself is not problematic and we will use a two-dimensional collection. But keeping the current state of the A chessboard is not enough; we also need to represent chess moves that are to be played and that need to be exchanged in the program by several members.

When the Kinect captures input and the move is detected we need to encode this move and then use it as an input for the robotic manipulator and for the chess engine which will react to the move. The moves need to be shown to the user in a human readable way. The encoding of the moves needs to be simple, so that it is easy to calculate a real physical movement as input for the robotic manipulator. It needs to uniquely represent each move and from the squence of these moves whole game must be able to be reconstructed. As has been discussed 1.1.2, classic chess notations, such as the algebraic notation [38] do not satisfy all the criteria, mainly that they need the game context to identify the actual moves and require further computations to use the moves.

3.2.1 Chess protocol

We will use a chess protocol to represent the chess moves. Chess protocols allow for transmitting chess game data between various programs, or between a backend and frontend of applications. In our case, we will be transmitting the data between the output of the Kinect chesstracking, displaying this data in a

chessboard representation in our application. The data will be displayed in a log, and also used for communication between subprocesses of our application. Some of the most often used chess protocols include Universal Chess Interface [3] (UCI), and Chess Engine Communication Protocol [39].

Universal Chess Interface (UCI)

The Universal Chess Interface (UCI) [3] is an open communication protocol that enables chess engines to communicate with user interfaces.

Advantages:

- Stateless - Each command is independent, and the protocol does not require maintaining a state, which simplifies the logic in the user interface [40]
- Popularity - Almost all new and/or strong chess engines support UCI [40]

Chess Engine Communication Protocol

CECP [39] was one of the earlier protocols designed to standardise communication between chess engines and graphical user interfaces. It supports features specific to the XBoard/WinBoard interface.

Advantages:

- Supports features like handling different time controls, variants of chess, and specific tournament settings, which makes it versatile for different types of chess applications.

Disadvantages:

- Complexity - The protocol is more complex than UCI
- Unpopularity - Fewer modern engines support CECP than UCI

Since we do not need to use any advanced features or any special chess variants, we will be using UCI in our application. It satisfies all our needs for chess state communication as it is easy to implement and understand. Being the standard for the chess programming community, any possible integrations would be simple.

3.2.2 Chess engine

There are many chess engines which can beat human players and whose ELOs [41] are higher than the ELO of any human player. This thesis does not focus on resolving the chess algorithm problem as that can be considered already solved. Instead we will choose between available options of possible chess engines and use the engine to compute the moves which will serve as an input for the robotic manipulator to play against the players.

Given our choice of UCI for chess protocol, we want to select a chess engine which supports UCI communication. Computing the best chess move is complex and considering we want to use our project in a showroom, we do not want the engine to win all the time to give the players a chance to win. That is why we would like to support an option to select how much time the engine should spend computing the move. The chess engine would return the best of the computed moves in the selected time frame. We seek an engine which is easy to integrate and offers good documentation.

Stockfish

As the the most popular chess server Chess.com says [4], Stockfish is the strongest chess engine available to the public and has been for a considerable amount of time. It is a free open-source engine that is currently developed by an entire community. Stockfish is not only the most powerful available chess engine but is also extremely accessible. It is readily available on many platforms, including Windows, Mac OS X, Linux, iOS, and Android.

It uses the alpha-beta [42] [43] search algorithm to analyse chess positions and find the best move. The engine generates a search tree, where each node represents a possible move and each edge represents the resulting position. The goal is to find the best path through the tree, which represents the sequence of moves that maximises the chances of winning for the player.

Stockfish supports the UCI protocol. The documentation as well as the community is extensive and the engine is open-source.

Leela Chess Zero

Leela Chess Zero (also known as Lc0, LCZero, and Leela) [44] is an open-source neural network (NN)-based chess engine. Because of its free and open-source nature, it can be run on many platforms, including Windows, Mac, Linux, Android, and Ubuntu. Lc0 is the strongest NN engine available to the public. Unlike conventional chess engines, Leela was only given the rules of the game of chess and became incredibly strong by using reinforcement learning from repeated self-play—as of 2020 it has played over 300 million games against itself.

Leela Chess Zero supports UCI protocol. However the engine requires GPU for optimal performance, which is an issue, because our application uses Kinect which is already GPU intensive.

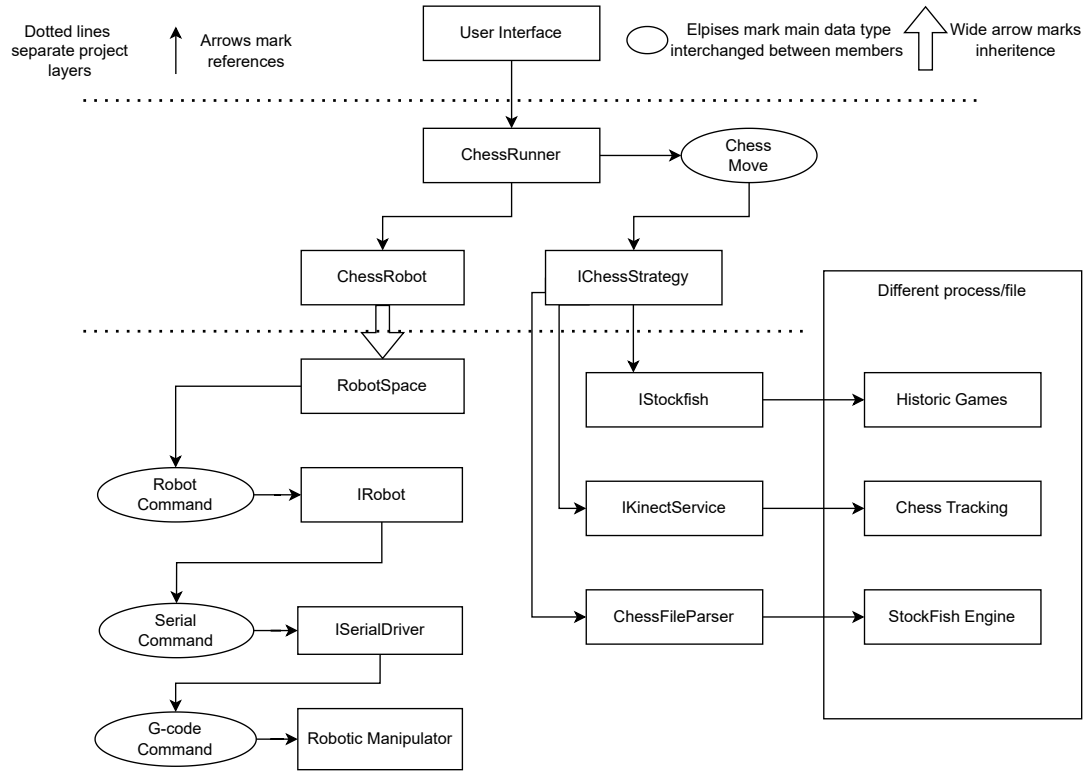
Engine choice

Many of other chess engines, such as Komodo or Houdini offer only limited functionality in the free version. Neural network chess engines are GPU intensive which is an issue considering our application uses Kinect which is already GPU intensive. There are other choices, but they may not be as well documented and do not have such extensive community as Stockfish. We will be using Stockfish, which is open source, free, supports UCI protocol and is not GPU intensive.

4 Implementation

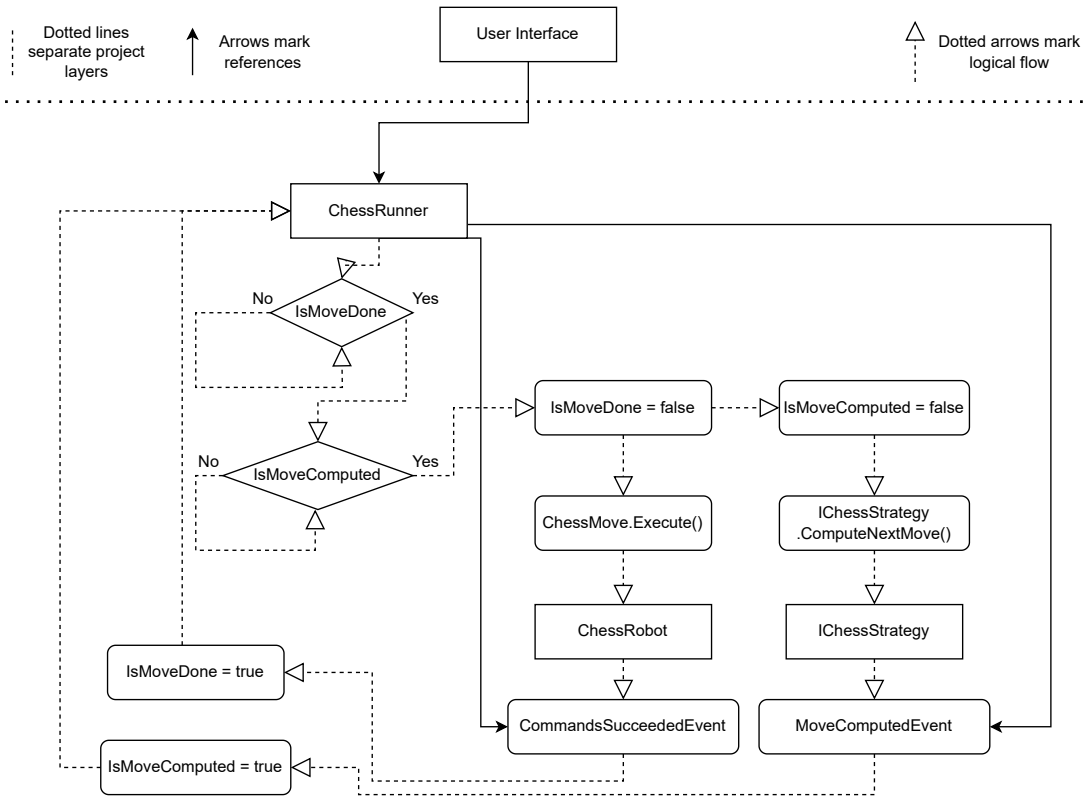
4.1 Architecture

Our application is made of multiple layers. This subsection discusses the meaning of the most important abstractions in these layers. Main members can be seen in Figure 4.1



Obrázek 4.1 Diagram of the main architecture

- **User Interface** collects input data to configure the setup. It displays information about tracking and the game that is played.
- **ChessRunner** works as a mediator between abstractions of the robotic manipulator and chess strategies. It redirects calls from the user interface onto another layers. It has a loop that operates inside a parallel `Task`. According to the state of the game and configuration, it tells the chess strategy that the program is ready to accept chess moves. After computing the move, the `ChessRunner` listens for an event fired by the strategy and, if possible, it asks the `ChessRobot` to execute moves. Data flow can be seen in Figure 4.2
- **ChessRobot** is a class that can accept and call the execution of `ChessMoves`.



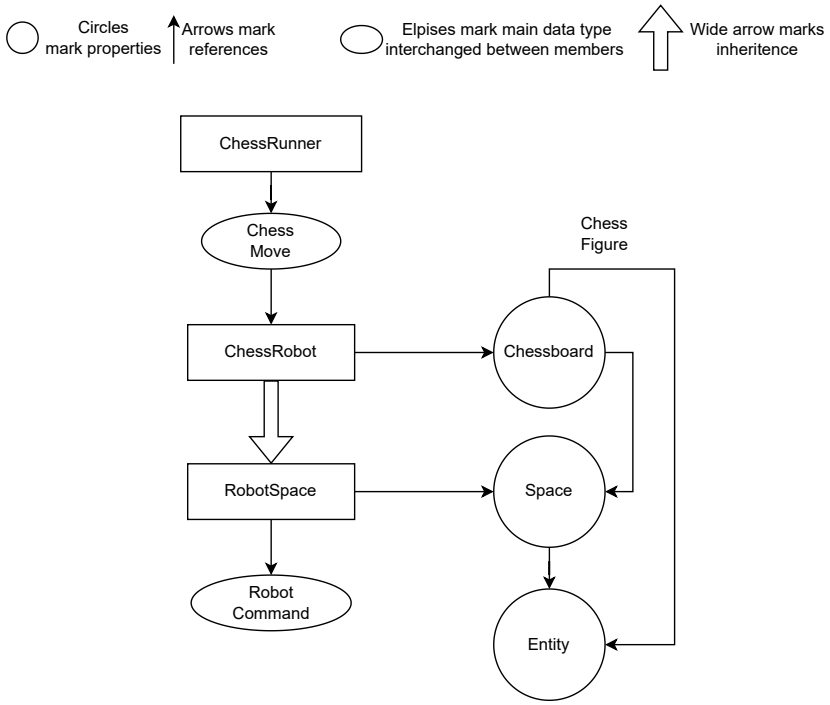
Obrázek 4.2 Diagram of the `ChessRunner` loop

It inherits from `RobotSpace`, which is a more general class capable of executing `RobotCommand`s. `ChessRobot` has a `ChessBoard` property, which is an abstraction encapsulating a more general Space that holds coordinates of the grid representing the game space. `ChessRobot` can translate commands, such as `MoveFigure` and `CaptureFigure`, which are translated from chess notation, into a form more suitable for a robotic manipulator. The relationships can be seen in the Figure 4.3.

- **RobotSpace** is a class that listens to commands such as moving an entity from source to target. These entities are a generalization for a physical object located on the grid. The grid is a generalization of the space in which our robotic manipulator can operate. It divides logical parts of the space into tiles which can contain entities, that is, figures. This class computes the most efficient trajectories through which the robotic hand should move.

- **IRobot** is an interface that serves as an abstraction for initializing and controlling the robotic manipulator. It is responsible for accepting logically atomic collections of commands. These are ordered collections of commands which will be executed in the specified order and, after the execution, fire events. Until all the commands inside the collection are executed, this layer blocks other commands so that the state of the execution is correct.

- **ISerialDriver** is an interface that sends serial commands to the robotic manipulator. It does not hold any context and serves as a communication layer between higher layers and the robotic manipulator.



Obrázek 4.3 `ChessRobot` relationships

- **IChessStrategy** is an interface that accepts calls to compute the next chess moves. This interface acts as an iterator which, when incremented, advances the game by a chess move. This is an abstraction that can be implemented as any of the game modes. The implementations can communicate with Stockfish, with which it exchanges chess data through UCI, or iteratively get moves from a historic game.

4.1.1 ChessTracking

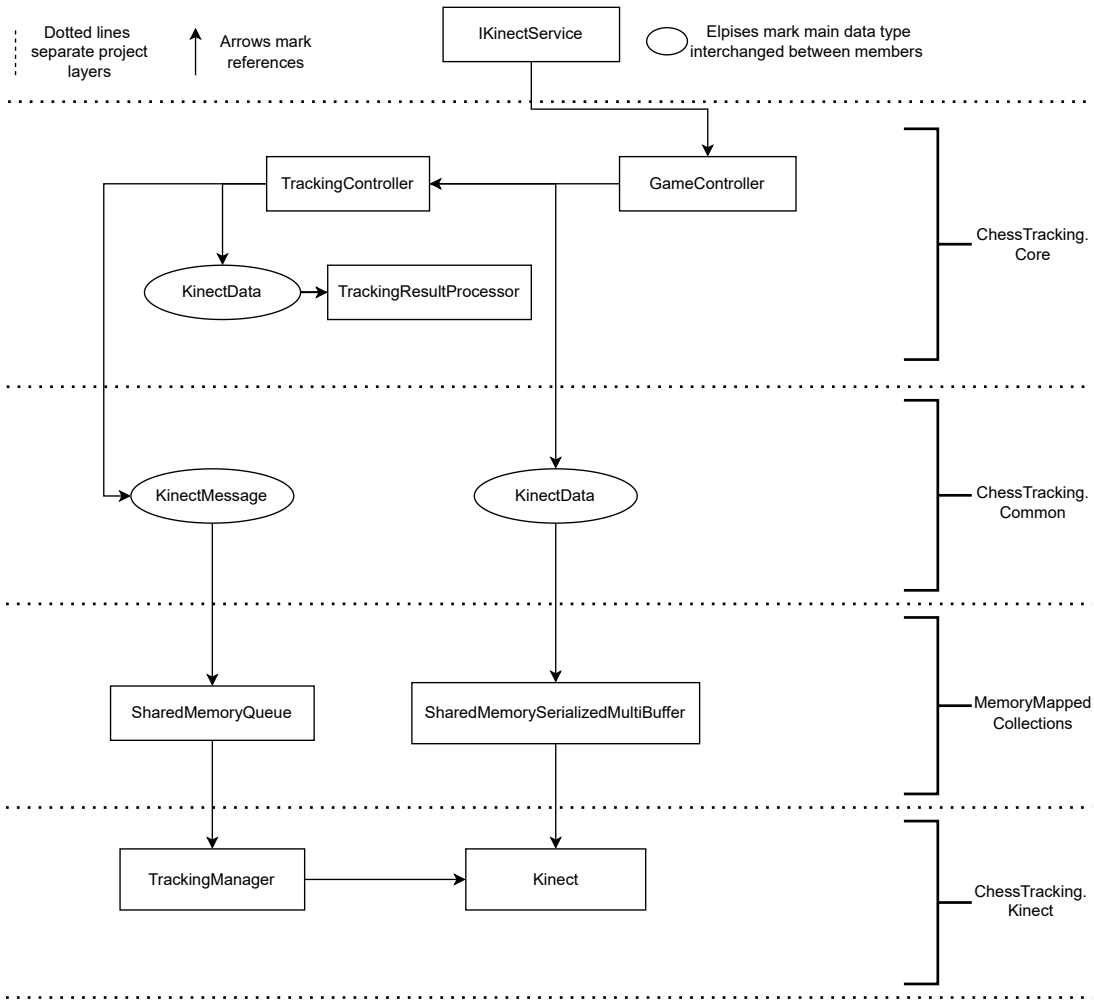
`ChessTracking` consists of multiple projects. It implements IPC via memory-mapped files. Some parallel techniques are used for performance; interprocess synchronization is provided by `Mutex`. We use the named system `Mutex`es, which are visible throughout the operating system and we need to use the `Mutex` in two processes.

Main members can be seen in Figure 4.4. These are the projects creating the `ChessTracking` part of the application.

- **ChessTracking.Core** is a project responsible for the localization of the chessboard as well as the figures on the chessboard. It consists of a pipeline that gets `KinectData` and uses various computer vision algorithms. This layer communicates with the Kinect layer.

  Here we explain some of the most important classes.

  - **GameController** This class starts, stops, and loads a game. It is a mediator between user interface and `ChessTracking` services.



Obrázek 4.4 Diagram of the `ChessTracking` architecture

- **TrackingController** reacts to `GameController`. It can start/stop the tracking and communicates with the Kinect layer via shared memory. It holds a reference to memory-mapped files through which the communication is established. It sends commands to the `SharedMemoryQueue` to start/stop the tracking. It expects input from the Kinect through another memory-mapped file. It actively waits for the input and, when ready, redirects the input to the pipeline.
- **TrackingProcessor** processes localization results from the pipeline and uses the game state to further approximate the results of the tracking to the real state. It fires events based on the results.

- **ChessTracking.Common** is a .NET `Standard` project that serves as a prescription for common data structures used in both processes. It defines `KinectData` and messages exchanged between `ChessTracking` and the main part of the application.
- **ChessTracking.Kinect** This layer collects input data from the Kinect.

It holds references to the same memory-mapped files as the `Tracking-Controller`. It listens for the input to start or stop the tracking and controls the Kinect camera. The Kinect class then calls `SharedMemorySerialized-MultiBuffer` to write the data to the shared memory.

- **MemoryMappedCollections** is a project which defines shared memory collections. Here we define collections which are used for inter process communication.

  - `SharedMemoryQueue` is a simpler shared memory collection. It is a queue implementation which allows for exchanging information between processes. Only primitive types can be used here, but they do not need to be serialized which allows for faster communication. It is used for exchanging simple messages between processes in form of enums, which control start/stop of tracking.
  - `SharedMemorySerializedMultiBuffer` is a definition for a shared memory collection using a multi-buffer pattern. It is used for more complex data structures, in our case, `KinectData`. It uses the `Zero- Formatter` serializer to write and read the data to the shared memory. The data is serialized to an array of bytes. The arrays are divided into parts which are processed in parallel in order to maximize performance.

4.2 Integrations

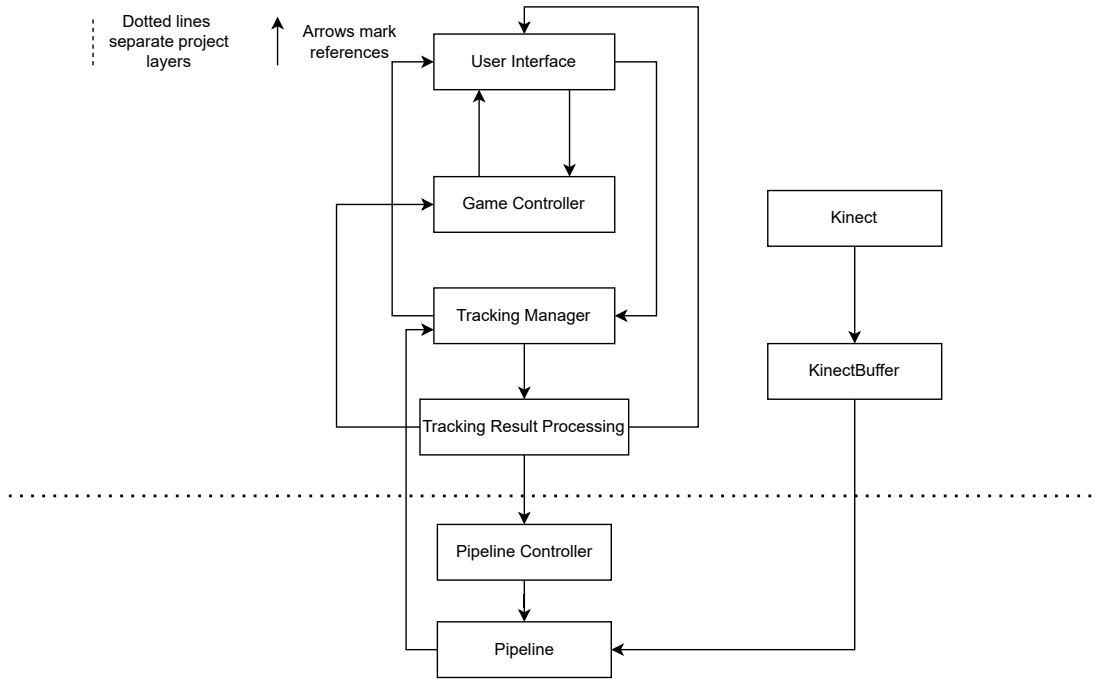
4.2.1 Migrating legacy code

Legacy chess tracking application captures input from Kinect. The input is processed by a pipeline which transfers the data and uses them in various computer-vision algorithms. Results are shown in `WinForm` UI. We have decided to take as much of the code from the legacy application as possible and move it to the .NET Core application so that we can use a modern platform and not produce legacy code.

As the Figure 4.5 shows, the old program had been using two main loops running on two separate threads. One thread was taking input from the Kinect and processing the frames inside a pipeline. Pipeline is a good pattern for processing data using numerous algorithms, however the implementation was not as straightforward.

The architecture was monolithic, meaning the members had been referencing each other. The user interface controls the game via game controller, which transitively controlled the tracking and pipeline. But the pipeline would also change the UI and the tracking. The input data for the pipeline was all in a form of messages. Messages have been abstract and could have been implemented as either `KinectData` or commands to start, stop or pause the pipeline.

Based on the pipeline results, the pipeline controller would change a global state of whole program, taking care of the UI and the state of the tracking itself. The code was coupled and decoupling, refactoring and finding borders between layers was not easy. Given the difficulties, we have decided to use and send raw data from the Kinect.



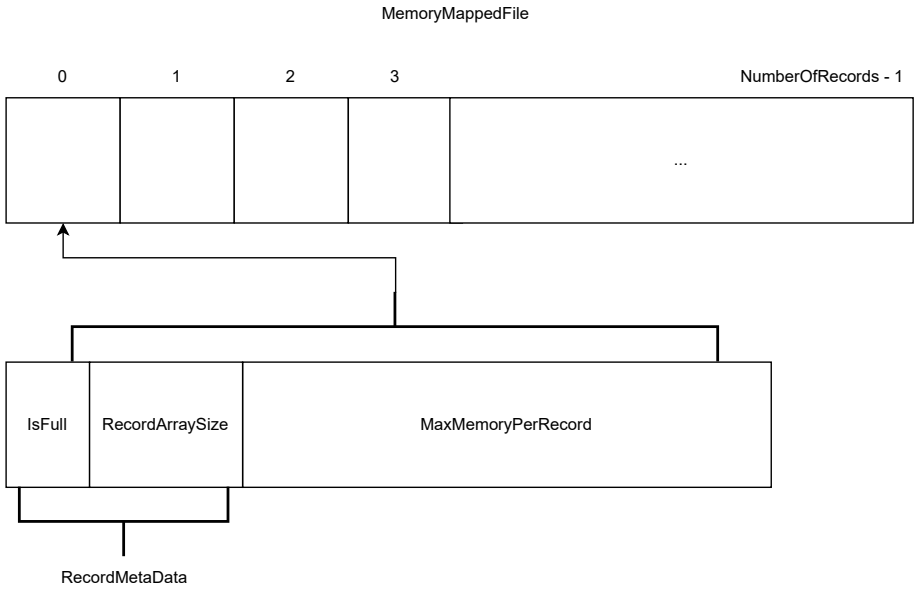
Obrázek 4.5 Diagram of the old `ChessTracking` architecture

Integrating processes running on a different platform

Data structures used by the Kinect library are not available for .NET Core, and the tracking algorithms partially rely on them. The .NET Framework was not open-source, and the Kinect library was not open-source either. Our application runs different processes, but still acts as one executable and both programs can be compiled under one C# solution even though they use a different platform.

Communication between processes is established using the `MemoryMapped-Collections` and `ChessTracking.Common` projects. Here we define data structures which are the same for both .NET implementations. The source code needs to be same for both processes and we do not want to create two separate sources. Copying all the code in two separate source codes would be an anti-pattern and would result in problems, such as changing code in one source code and not in the other. If later it is decided to reimplement this part of the code in a different way, it would be harder to keep track of the source code in multiple source codes.

We can achieve this by creating a .NET `Standard` [45] project which can be added to the solution and be referenced from project files of both platforms, even though they are not compatible. Such project acts as a prescription for both platforms, defining one source code which will be compiled against the compiler of the corresponding platform. This greatly simplifies the source code. We can define the same structures and use the same algorithms in one source code and use them in both projects even though the projects are incompatible. We need to use some 3rd party libraries which have implementation in both platforms. We can define a



Obrázek 4.6 Memory-mapped file structure

`NuGet` reference in a common `Standard` project and the correct version with the corresponding library is selected when compiling a specific project with a specific platform.

When we build our solution, source codes written against both platforms are compiled and we can run our application and start the process which starts the compiled program on the other platform.

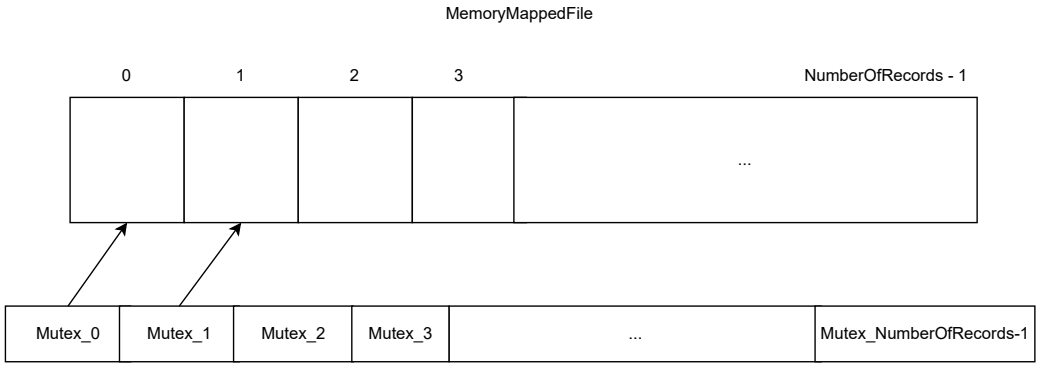
4.2.2 Shared memory multi buffer

The communication between processes is established via collections defined in the project `MemoryMappedCollections`. `SharedMemorySerializedMultiBuffer` is a multi-buffer collection through which Kinect data are transferred. This collection is defined in .NET `Standard` source project. It takes advantage of Memory-Mapped files.

When creating a MMF, as the Figure 4.6 shows, we first allocate the desired size of the file. This should be a multiple of the maximum expected size of the Kinect data plus serialization overhead. As the Kinect input data does not have a constant size, we also need to exchange metadata containing the expected size of a record. For the same reason, we need to store all metadata at a set position in the memory so that all communicating members know where to look for them.

We propose a solution based on a double buffer technique so that while one of the communicating members is reading/writing, the other is not blocked and can do the same. When the producer and consumer are finished with their current operation, the roles of the buffers switch. This works well when the producer and consumer have roughly the same speed, but our operations need to do some additional work and therefore their speeds differ. This results in latency from a finished member waiting for the other to finish.

We can solve this by introducing a multi-buffer implementation. In our case, the writer is generally faster, so the additional buffer lets us write to one buffer



Obrázek 4.7 Memory-mapped file mutex structure

while another buffer is being read. At the time when the writer is finished, it does not have to wait for the reader to finish his reading and can instead write to another buffer. When the reader is finished, it also has a buffer prepared for reading.

For communication, this class is instantiated in both processes. Both processes map the same file by specifying the name and the size of this file. One process acts as a consumer and the other as a publisher. To synchronize with each other, they use `Mutex`es which are provided by the operating system. `Mutex`es are mapped via the corresponding name. All the data required for mapping is known at the compile time by both programs and is defined in the same .NET `Standard` project. This class is generic and the type of the parameter needs to be the same and is known by both programms as well.

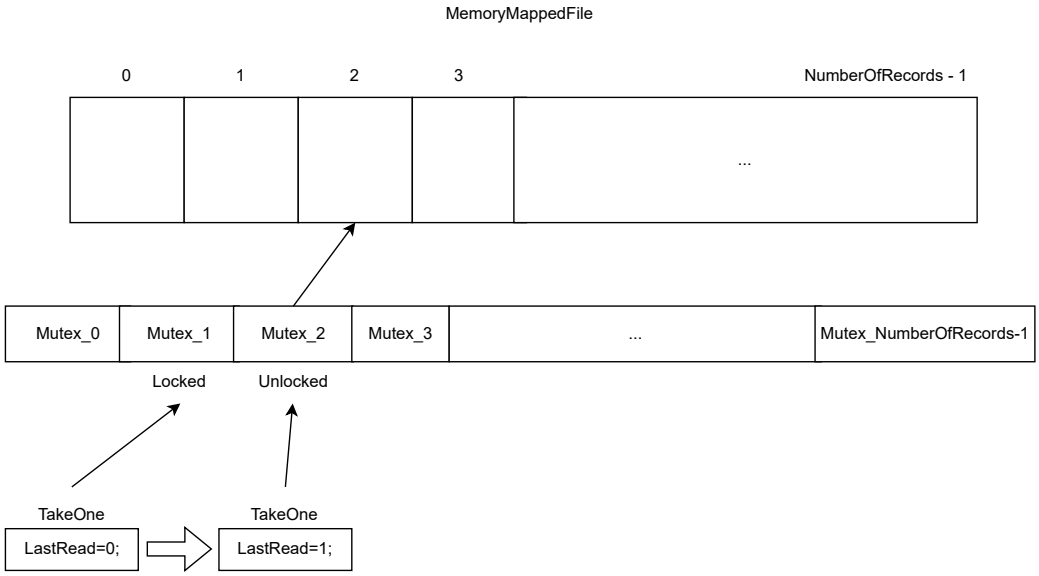
Each time one of the members tries to read/write to/from a position, they need to create a `ViewAccessor` which accesses the shared memory, starting at a specified position, of a specified size. When working with the data accessed by the `ViewAccessor`, starting position and the size of the data needs to be specified again, but this time it is relative to the `ViewAccessor`, not the memory-mapped file. After operation on this data is finished, the `ViewAccessor` is closed and disposed of.

After both instances of the `SharedMemorySerializedMultiBuffer<T>` class are created, they can exchange data. Two main methods are defined, ie. `TakeOne` and `AddOne`. Depending on the number of records storable in this collection, which is also parametrized in the constructor, the corresponding number of `Mutex`es is created, one per each buffer 4.7. Both, `TakeOne` and `AddOne` try to lock a buffer by waiting for the corresponding `Mutex`.

They periodically try to wait for a `Mutex` for a specified time and if unsuccesful try the next `Mutex`.

TakeOne

After aquiring a signal on a `Mutex`, it reads the metadata in the location, corresponding to the aquired `Mutex`. If the location contains data, that is, it has been added data by the writer, it partitions `KinectData` into equal parts. It creates a `Task` for each partition and separately reads serialized data, writing it parallel to one array. It waits for all `Task`s to finish and deserializes the data. The



Obrázek 4.8 Shared memory multi buffer - Unsuccesful operation

buffer is then marked as empty and the `Mutex` is released, returning the data.

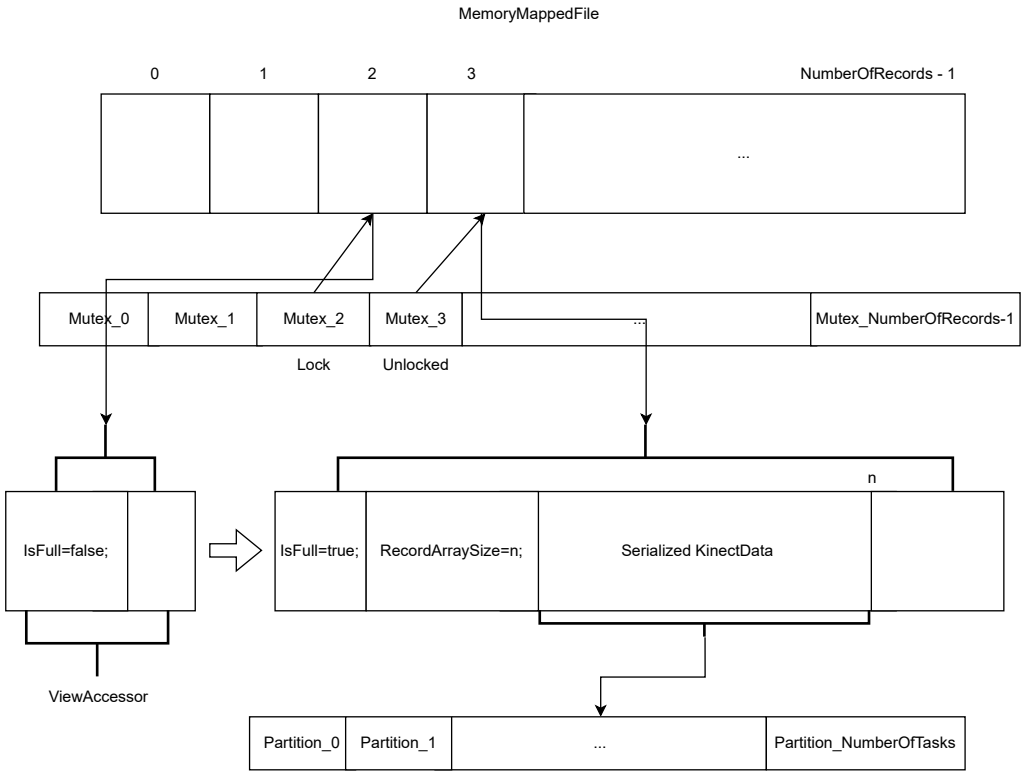
- **Unsuccesful read** - The Figure 4.8 shows the situation, where the `TakeOne`, tries to read the record from the next expected location, that is the record with index 1. It waits on the mutex for a specified time, but the mutex is locked. It advances to the next mutex. It aquires the lock on the record with index 2. As the Figure4.9 depicts, a `ViewAccessor` is created for the record with index 2. However, this record has not been written to yet.
- **Succesful read** - After finding out, that the second record is empty, `TakeOne` advances again. This time the `Mutex_3` is aquired and as the Figure 4.9 shows, this record is full. The serialized record is partitioned and each part is read in parallel.

AddOne

Similiarly after aquiring a signal on a `Mutex`, metadata specified by the `Mutex` is read. If the location is marked as empty, data being added are serialized, and partitioned into equal parts. A `Task` is created for each partition and written in parallel to the corresponding position inside the location for this record. When all `Task`s finish, the buffer is marked as full and the `Mutex` is released, making the buffer ready for reading.

Serialization

C# is a managed language and does not provide much freedom when dealing with unmanaged memory. MMFs are unmanaged memory, and any process could potentially write something malicious to our file. MMFs are limited to using only value types and arrays of bytes. Since we cannot use pointers, we cannot treat



Obrázek 4.9 Shared memory multi buffer - Succesful operation

an object as an array of bytes and vice versa. Therefore, the data needs to be serialized.

Serialization adds some overhead, but .NET offers some very fast binary serializers. One of the serializers we used is `ZeroFormatter` [46], which claims to be the fastest serializer for .NET.

Using `ZeroFormatter`, we can serialize the Kinect data into an array of bytes. Although this adds some time and memory overhead, it is not very significant. We then write the data to our buffer in parallel, partitioning the array. Since `ViewAccessor` makes it easy to randomly access memory, we can write each part independently. The same approach is used when reading the data.

4.3 Implementation

4.3.1 Scheduling commands

Robotic manipulator accepts `G-code` commands sequentially via USB. When the commands are ordered, it returns a response, whether the command execution is valid. If valid, it starts executing, which takes some time. The controller of the manipulator does not have any command queue and therefore we must wait for it to finish executing, because if we ordered another commands, they would be unaccepted and lost. We need to know at what time the execution finishes, so that we can order another commands.

As the Figure 4.1 shows, at the lowest layer we have the Serial Driver. This layer

is only a wrapper for the robotic manipulator, which translates `SerialCommand`s to `G-code`. We ask this member for state of the robotic manipulator and it translates string reponses to objects which we work with.

The class representing communication with the serial port is called Robot, it is the implementation of the `IRobot` interface which we can see in the Figure 4.1. We can not know what is the exact time that an execution will take, so we implement a sort of busy waiting where we order a command and in small intervals ask for the state from the manipulator. When the state says that the manipulator is idle, we fire an event signalizing that we are ready for another command.

The `Robot` implements a queue that can only accept one atomic operation, which must be executed before listening to other commands. The atomic operation can consist of multiple commands, which are logically connected. These commands are executed in order, and until all commands are executed, the queue does not accept additional commands.

`RobotSpace` is responsible for maintaining the state of entities in a grid. It accepts more complex commands, such as moving entities from source to target. It calculates the path using the algorithm defined in 2.1.5, which the grip will follow.

`ChessRunner` orders the `IChessStrategy` to compute chess moves. It orders the `ChessRobot` to execute the chess move. `ChessRobot` translates a chess move, such as move figure to a square, to a more general entity to space in grid, which the `RobotSpace` expects. It is a level of abstraction which helps to decouple ordering chess moves from finding a path in space.

4.3.2 Path calculation

When a `ChessRunner` gets a `ChessMove` from a `IChessStrategy`, one of the following methods on the `ChessRobot` gets executed.

- MoveFigureTo
- CaptureFigure
- PromotePawn
- ExecuteCastling

It then depending on the move, calls `MoveEntityFromSourceToTarget` on the `RobotSpace`. If the move is a castling or a capture, it calls `MoveEntity-FromSourceToTarget` on the `RobotSpace` twice, because two individual figure movements need to be executed. Otherwise the method gets called only once. Pawn promotion is not yet implemented. After each type of move the method `MoveToAHighPoint` on the `RobotSpace` is called, in order to get the grip into a position, where it waits for another move and does not abrupt the chess tracking. When a `MoveEntityFromSourceToTarget` is executed, it creates a collection of moves which will be scheduled.

`RobotSpace` asks for the current position of the grip, from the driver. It then devides the entity move to `GetTakeEntityFromPositionCommands` and `GetMoveEntityToPositionCommands`, where each calls the implementation of the path finding algorithm.

The `RobotSpace` has the figures and their coordinates and sizes saved in a two dimensional array, from which it gets their positions and uses the for path finding algorithm calculations.

If the move is a placement, the final destination is a point just above the picking height of the figure, where it adds `Open` command, it moves a little upward and calls the `Close` command. Similarly if the move is a picking move, the destination of the path finding algorithm is a point just obove the figure, where it adds a `Open` command, it moves a little downwards and calls the `Close` command.

The command collection is then scheduled to the `IRobot`.

Event handling

`IRobot` interface defines event handlers such as `CommandsSucceeded`, which is fired whenever it successfully finishes execution of commands. Until the event is fired, it does not accept another commands. This is beneficial, because the time of execution is unknown and we do not want to block members which order the execution. On the other hand, this could be implemented as `async Task`s, which would complete after the execution is finished, similar to `async Task`s mostly used for database or API calls or reading from a file. Our system, however defines multiple subscribers which subscribe to the events and react to it differently.

Events help with decoupling all the subscribing members. After firing an event, user interface might change the displayed coordinates, the `IChessStrategy` might order another execution, other component in the user interface might unlock a button and display executed move. This way they can all work independently, being notified and reacting if they need to.

4.3.3 Swapping contexts

Because we want our application to effectively change between game modes 1.1.3 and to pause and reconfigure the game, we need to handle the effects of these changes.

Firstly, we need to handle pauses. Our application operates across multiple layers, and at each layer, we need to manage pauses. At the lowest layer, we have the robotic manipulator, which responds to changes via USB. Once a command is sent to the serial port, it can be cancelled, by using the emergency stop button, but we lose the command which has been ordered and the state of the execution would be lost. However, we can control commands that have been ordered but not yet sent to the serial port.

`IRobot` receives atomic commands which may consist of multiple moves. It expectects a collection of commands which are logically connected. It orders the `ISerialDriver` to execute these commands sequentially. When paused, the execution of this collection is paused as well. After resuming, they need to be finished, before changing a strategy, or ordering any other execution.

`RobotSpace` layer is less affected by pauses, as it waits for the lower layer to finish and orders another execution only once the lower level is done. When the game is resumed, it finishes the command.

This approach allows us to control the exact state of our game. We can pause the game, finish the execution of the current move, and safely swap the configuration or game mode, knowing exactly where our pieces are and where the

robotic arm should be. Another layer above is responsible for tracking the actual game. The role of it is to execute chess moves. It can be paused and resumed either by completing the current movement and then starting a new game or ordering additional commands within the same game.

If anything goes wrong in the actual physical setup, it is possible to reconfigure the position of the chessboard while maintaining the state of game. The game can be resumed even after reconfiguration because we know the state of game and the actual state of the board. Game modes can also be changed, and we can choose whether to continue with the same positioning of the pieces or start from the beginning. This is useful, for example, if we want to show a part of a historic game or watch a segment of an AI match and then switch to human play. When we swap the stategies, they exchange collection of executed moves.

The user interface handles this as follows. The page with the game log (`GamePage`) lets us pause the game. This pause is immediate, and the current move is not finished. To change a strategy or reconfigure the position of the robotic manipulator, the user needs to press `Finish Move`. This finishes the move which has already been scheduled. After the move is finished, the user can change strategy or reconfigure.

4.3.4 Play against AI strategy

This subsection discusses a strategy that connects two chess input sources. Chess strategies are polymorphic, and their API provides only simple methods, namely `ComputeNextMove` and an event handler, MoveComputed. It exposes other methods that are only used to initialize from an old game or provide their state, that is the executed moves, for other strategies.

`ChessStrategi`es act as iterators. This particular strategy is interesting because it connects complex parts of the program yet has a very simple implementation. When this strategy is initialized, it is given IKinectService, which is managed by dependency injection. When needed, the IKinectService starts the process responsible for input from the Kinect. The same applies to IStockfish, which also starts a different process.

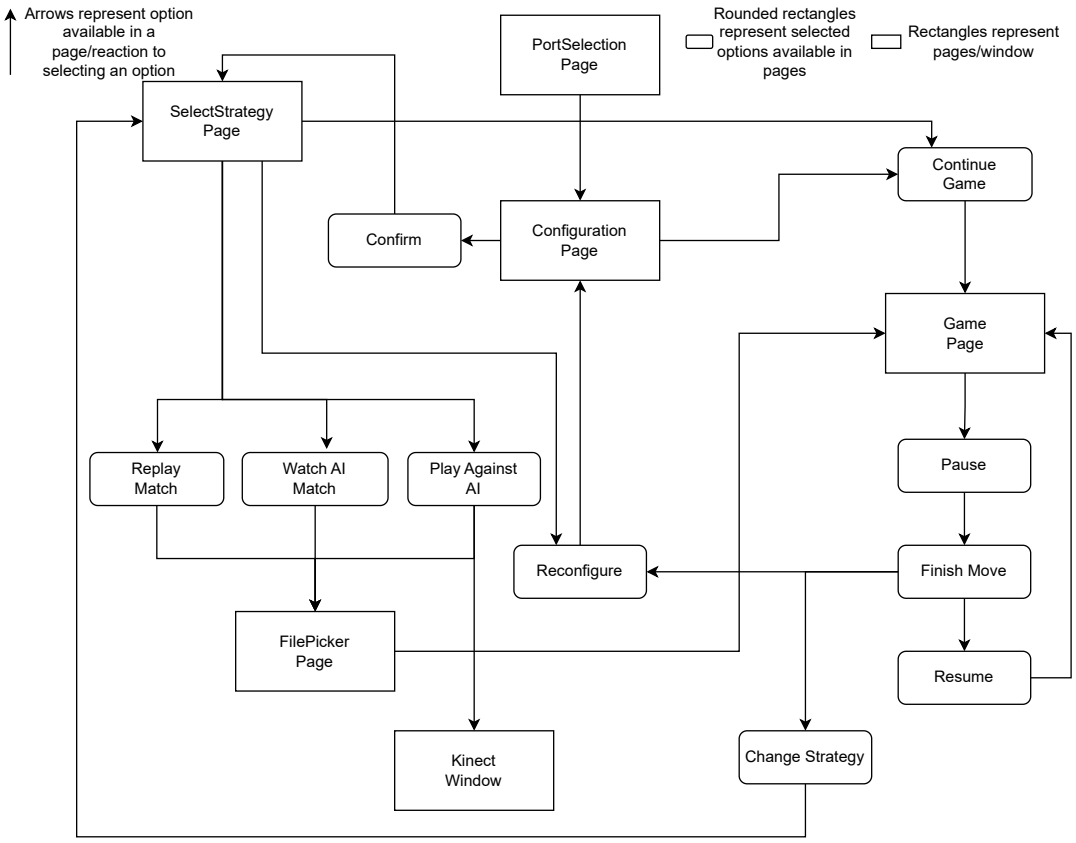
When this strategy is selected, KinectWindow is opened, and MainKinectPage is navigated to. This window displays results of the tracking and localization. Tracking can be configured to accommodate changes in light conditions.

State of the chess game is remembered in a collection of moves in UCI notation. According to which player is expected to perform a move, it asks Stockfish to compute a chess move, or it listens for the input from `ChessTracking`.

Strategies fire events when moves are completed. Pages of the user interface listen to these events and display the moves in a log.

4.3.5 User Interface

The user interface is created in `WinUI3`. It uses the `MVVM` pattern. ViewModels use Services which act as facades of the interface provided by lower layers. Window classes manage which pages will be displayed. When navigated to a page, ViewModel is activated, it asks for data from services and displays the corresponding view based on the created model. All services are registered in `App.xaml.cs` as



Obrázek 4.10 Map of the User Interface and page actions

singleton services. The program uses dependency injection. Services expose event handlers which are subscribed to by the user interface.

Navigation

The application can open multiple windows. Windows navigate between pages. Pages override `OnNavigatedTo`. In this method, dependency injection is applied. It happens in this method instead of the constructor because the constructor is only called once in the lifetime of the application, and when pages are navigated from, the references are lost.

Overrides of `OnNavigatedTo` method on each page also subscribe to the event handlers provided by the services. These event handlers are unsubscribed from in the override of `OnNavigatedFrom` method.

Application dialogue

User Interface is simple and mostly acts as a dialogue between the configuration and the person configuring the project. It leads the user to correctly configure and

use the setup, not allowing operations which would result in a malicious state. The pages of the dialogue can be seen in Figure 4.10; navigation between pages is triggered after firing a corresponding action.

The application dialog requests the user to pick a chess strategy This is provided by a combo box which is mapped to `ChessStrategyFacades`. These facades are used to create an instance of a `ChessStrategy`.

Some strategies require the user to select a file from the file system. For example, match replay requires the user to pick a .txt file which is a record of a chess match written in PGN chess notation. Other strategies require picking a Stockfish.exe file. File system is accessed by Windows native `FileOpenPicker` class.

5 Discussion

5.1 Evaluation

We have created a desktop application, for controlling a robotic chess-playing manipulator and we have integrated a legacy chess tracking application which uses Kinect for Windows and computer vision algorithms to track chess figures on a chessboard. We have created one unified executable program, which connects multiple hardware and software components.

We have not managed to implement every possible movement. The pawn promotion is currently not supported. All the game modes have been implemented, including replaying a historic game and watching the computer play against itself, however, during the development the robotic manipulator got broken and we were not able to properly test the whole setup with the manipulator. Therefore there might be some issues with the movement scheduling which need to be addressed after the robotic manipulator is fixed. Without that the project is not fully usable in a showroom.

However, the code is maintainable and with a decent architecture, which allows for possible future development and for easy resolution of any possible problems.

We have modernized the chess-tracking solution and have created a protocol for communication between incompatible programs.

A chess engine has been integrated, which reacts to the input, provided by the Kinect. The input is collected from the engine and shown to the user. The chess move is essentially translated to commands for the robotic manipulator. 3-dimensional path-finding algorithm is used and the movement is efficient.

The user is able to configure the setup and reconfigure after any physical disruption. Additionally, the user is able to pause the game and change the game mode, where the previous game can be used as a starting point for another game played by computer.

5.1.1 Chess Tracking

Due to transferring data via memory-mapped file, which added time overhead, the resulting evaluation happens at a rate of 4 to 6 frames per second, which is the same as when the tracking was a separate application, tested on a worse hardware. Individual parts of Chess tracking and their respective average execution time can be seen in the Table 5.1. All these operations are executed in parallel.

Operation	**Time (ms)**
Figure localization	7-15
Tracking result processing	5-7
Bitmap conversion	2-4
MMF TakeOne	160-290
MMF AddOne	140-200

Tabulka 5.1 Resulting time of parts of chess tracking

Transferred data has around 40MB, accounting for the overhead of the `Zero- Formatter` serializer, which added around 2-5 MB. Testing was done on a laptop with `Intel Core i7-9750H CPU @ 2.60GHz and NVIDIA GeForce RTX 2060 GDDR6 @ 6GB`.

5.2 Testing

5.2.1 Robotic manipulator

The robotic manipulator has not been properly tested because it was broken during the development and while showing the concept of the application in a showroom. After it is fixed, more tests will be made. We expect, that there might be issues with the scheduled commands and they need to be resolved after the robotic manipulator is fixed. A mock robot has been added as an option to the application to simulate the movement, positioning, and correctness of implemented algorithms. It simulates every movement scheduled by remembering a position in 3D coordinates. The coordinates are shown in the UI, as would be for the real robotic manipulator.

Specifically, we have created `MockRobot`, which implements the `IRobot` interface. It is given commands computed using the path finding algorithm. The coordinates are updated in parts, so that when a movement is ordered, the update does not happen immediately, but rather in parts in a given time.

5.2.2 Robot moves

When a chess move returned from a chess strategy is given to the `ChessRunner`, it is translated to UCI notation and later to real entity being moved from a real position to another position. `RobotSpace` validates whether an entity is really located in the specified position and whether the target position is empty, by comparing the real position of the grip, or the mock to the actual coordinates of the particular chessboard square. Each one of them was correctly played, testing, that our algorithms of translating chess moves to actual movement of robotic grip and picking and positioning the entities on a grid is correct. In these games ranging from 30 up to 88 chess moves have been made. All of these games consisted of all kinds of chess moves, validating normal, capturing and castling moves.

We have tested about 5 different historic games recorded in the PGN notation 5.2.

This also validates other chess strategies which use the same algorithms.

Event	White	Black	White
Ch World (match) - Moscow(Russia) - 1985	Anatoly Karpov	Garry Kasparov	0-1
It (cat.17) - Wijk aan Zee (Netherlands) - 1999	Garry Kasparov	Veselin Topalov	1-0
Memorial Rosenwald - New York (USA) - 1956	Donald Byrne	Bobby Fischer	0-1
75th Tata Steel GpA - Wijk aan Zee NED - 2013	Levon Aronian	Viswanathan Anand	0-1
Paris (France) - 1958	Paul Morphy	Duke of Brunswick and Count Isouard	1-0

Tabulka 5.2 PGN games

Swapping strategies and reconfiguration

Multiple tests have been made, where we have started a historic game and later reconfigured the chessboard, so that the coordinates of expected corner and therefore chessboard square positions have been altered. After continuing the game, the game has been correctly played, validating reconfiguration.

Similiarly a historic game has been played multiple times ranging from 5-20 moves. After this a strategy has been changed to Watch AI match. The context, that is, the collection of already played moves, has been given to the watch AI match strategy. The game then correctly continued and eventually finished, validating swapping of the strategies as the moves were then given to Stockfish, which can be considered as a trusted validator for computing chess moves.

Play vs AI match

We have tested the integration of the Stockfish, Chess tracking and position of the robotic grip. We have played multiple games, trying various move types, including castling. As the robotic manipulator was broken during the development, this integration was tested using the mock robot. When a chess move is played, the chess tracking part correctly detects the move, after which it is logged to the main game page. After this the executed commands are sent to the Stockfish via UCI notation. Stockfish computes the best possible move in a given time and the move is logged to the log in the main page.

The game page was showing the expected coordinates of the robotic grip, but we had to move the chess figure manually. After this, the chess tracker corretly detected the move, logging it to the tracker log.

5.3 Future work

More testing needs to be done with the robotic manipulator as it was broken during the development and we had to wait for months until it was repaired, not being able to test the updated software. At the time of writing, the robotic manipulator is still not repaired, and we expect some movement inaccuracies and potential movement scheduling issues, which we can not resolve without trying if they work.

5.3.1 Chess tracking speed up

There is a space for speeding up the chess tracking process. IPC could be made faster by pre-processing the input data from Kinect and taking some of the responsibilities of the pipeline to the .NET Framework process. This would reduce the size of the data being transferred through memory-mapped files, which would reduce the time needed to read/write data to memory-mapped file, ultimately resulting in more frames being able to be processed per second.

The pipeline which applies computer-vision algorithms could be sped up by carefully splitting it into Tasks. Some of these algorithms do not rely on each other. Parallelization could be implemented, which would reduce the time needed to process the pipeline.

5.3.2 Automatic reconfiguration

Chess tracking relies on user-given parameters which need to be configured to adapt to changing light conditions and different chessboards and chess figures. This could be done automatically based on the current light conditions.

5.3.3 Pawn promotions

Chess tracking algorithms do not differentiate between individual chess pieces but rather remember their previous positioning. This currently makes it impossible to do pawn promotion, because there is no way to know, what has the figure been promoted to. We could add a feature into the control application, where in the user interface, user could set that a pawn has been promoted, setting the promoted type.

5.3.4 Configuration

Currently, the size of the chess figures can only be changed in the code, this could be changed to being parametrized, for example in the User interface or in the configuration when the corners of the chessboard are localized. The location of the chessboard could be computed using the Kinect camera.

5.3.5 Camera

Kinect v2 can be considered outdated, mostly because the API is no longer supported. As our project demonstrates using the Kinect with a modern technology was not easy and we had to come with a few workarounds. The technology of the Kinect is outdated as well. It could be replaced with a more modern camera, which may offer higher resolution, better accuracy and more reliability.

Intel RealSense [47]

An example of a better camera is Intel Realsense D435. This camera offers higher Depth output resolution, Depth frame rate and Depth Field of View. The difference in parameters can be seen in the Table 5.3, which shows a comparison between Kinect V2 and Intel RealSense D435.

Property	**Kinect V2**	**Intel RealSense D435**
Technology	Time-of-fligh	Active stereoscopy
DEPTH Range (m)	0.5 - 4.5	0.2 - 4.5
Resolution	1920 x 1080	1920 x 1080
Frames Per Second	30	30
Field of View DEPTH	70 x 60	85.2 x 58
DEPTH Resolution	512 x 424	1280 x 720
Frames Per Second(FPS) DEPTH	30	90

Tabulka 5.3 Kinect, Intel RealSense comparison [48]

This camera is still supported, offering SDK for .NET core, which would greatly simplify our problem with incompatible platform, ultimately speeding up the application, because of removed need to use IPC. There is also strong developer community, which would make the development easier.

Conclusion

We have managed to create a control application for the robotic manipulator, integrating Kinect and a chess engine, which ultimately lets a player play chess against the computer. Various parts have been unified under one executable. The design is clean and adds up to the simplicity of using the User interface.

The dialog that leads to the correct setup is simple, self-explanatory, and responsive. The application is capable of playing chess matches and replaying historic games and simply changing between these modes. It works well for the initial goal of showing it in a showroom.

During the development of the application we have created mocks for simulating the process of the game. The program is modular, and even when some pieces of the setup are missing, the other parts can be tested and used. The architecture of the program is layered, and the code is sustainable. In the future, this project could be extended to automatically adapt to changing light conditions, offering a layered architecture which makes it simple to make such change.

Some parts of the legacy ChessTracking application have been modernized. During the development we have researched IPC communication and have found a solution for migrating at least a part of legacy application to a modern system. This could be useful if someone finds themselves in a situation where not all parts of a legacy program can be migrated, and it makes sense to migrate at least some code. We have created an efficient protocol for communicating between processes.

In the future, the camera could be changed for a more modern alternative, which would simplify the code, speed up the application and make the tracking more reliable, making the project more appealing in a showroom.

Bibliografie

1. Dostupné také z: https://chat.openai.com/.
2. Wall, Bill. *Chess Notation* [online]. [cit. 2024-05-06]. Dostupné z: http://billwall.phpwebhosting.com/articles/chess_notation.htm.
3. Meyer-Kahlen, Stefan. *Universal Chess Interface (UCI)* [online]. [cit. 2024-04-28]. Dostupné z: https://www.shredderchess.com/chess-features/uci-universal-chess-interface.html.
4. Chess.com. *Stockfish* [online]. [cit. 2024-05-01]. Dostupné z: https://www.chess.com/terms/stockfish-chess-engine.
5. Girardeau-Montaut, Daniel. *Introducing Project Kinect for Azure* [online]. [cit. 2024-04-11]. Dostupné z: https://www.linkedin.com/pulse/introducing-project-kinect-azure-alex-kipman.
6. Staněk, Roman. *Board Games Tracking Using Camera and Depth Sensor* [online]. [cit. 2024-04-02]. Dostupné z: https://dspace.cuni.cz/handle/20.500.11956/109049.
7. Amos, Evan. *File:Xbox-One-Kinect.jpg* [online]. [cit. 2024-05-06]. Dostupné z: https://en.wikipedia.org/wiki/File:Xbox-One-Kinect.jpg.
8. Oxford, Engineers Without Borders. *Application of visual tracking algorithms for human computer interfaces* [online]. [cit. 2024-05-07]. Dostupné z: https://www.ewbox.org/ml-visual-tracking-project.
9. Wikipedia. *Kinect* [online]. [cit. 2024-05-07]. Dostupné z: https://en.wikipedia.org/wiki/Kinect.
10. Hamed Sarbolandi Damien Lefloch, Andreas Kolb. *Kinect Range Sensing: Structured-Light versus Time-of-Flight Kinect* [online]. [cit. 2024-04-02]. Dostupné z: https://arxiv.org/abs/1505.05459.
11. Wikipedia. *G-code* [online]. [cit. 2024-05-07]. Dostupné z: https://en.wikipedia.org/wiki/G-code.
12. HEDSTROM, FREDRIK BALDHAGEN ANTON. *Chess Playing Robot* [online]. [cit. 2024-04-02]. Dostupné z: https://www.diva-portal.org/smash/get/diva2:1462118/FULLTEXT01.pdf.
13. Romero, David Vegas. *Implementation of a Chess Playing robot application* [online]. [cit. 2024-04-02]. Dostupné z: https://upcommons.upc.edu/bitstream/handle/2117/333724/tfm-davidvegas.pdf?sequence=1&isAllowed=y.
14. LaValle, Steven M. *Planning algorithms*. Cambridge University Press, 2006. isbn 0521862051.
15. Han, Jihee. *An efficient approach to 3D path planning* [online]. [cit. 2024-04-03]. Dostupné z: https://www.sciencedirect.com/science/article/abs/pii/S0020025518309332.
16. Abrash, Michael. *Michael Abrash's Black Book of Graphics Programming (Special Edition)*. Coriolis, 1997. isbn 9781576101742.

17. Dedu, Eugen. *Bresenham-based supercover line algorithm* [online]. [cit. 2024-04-03]. Dostupné z: http://eugen.dedu.free.fr/projects/bresenham/.

18. Fischler M. A. a Bolles, R. C. *Random sample consensus (ransac) algorithm, a generic implementation.* Association for Computing Machinery, 1981. ISSN 0001-0782.

19. A. S. Hassanein S. Mohammad, M. Sameer; Ragab, M. E. *A Survey on Hough Transform, Theory, Techniques and Applications* [online]. [cit. 2024-05-08]. Dostupné z: https://arxiv.org/pdf/1502.02160.

20. C Gonzalez R. a E Woods, R. *Digital Image Processing (2nd Edition).* Association for Computing Machinery, 2002. ISBN 0201180758.

21. Nguyen A. a Le, B. *3d point cloud segmentation: A survey.* 2013. ISBN 978-1-4799-1201-8.

22. Canny, J. *A computational approach to edge detection.* IEEE, 1986. ISSN 2160-9292.

23. Koray C. a Sümer, E. *A computer vision system for chess game tracking.* 2016.

24. Microsoft. *What is .NET Framework?* [online]. [cit. 2024-05-02]. Dostupné z: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework.

25. Mullender, Sape J. *Distributed Systems.* ACM Press, 1993.

26. Wikipedia. *Primitive data type* [online]. [cit. 2024-05-02]. Dostupné z: https://en.wikipedia.org/wiki/Primitive_data_type.

27. Institutes Khanna, Gulzar Group of. *IPC technique PIPES* [online]. [cit. 2024-05-01]. Dostupné z: https://www.geeksforgeeks.org/ipc-technique-pipes/.

28. Microsoft. *Pipe Operations in .NET* [online]. [cit. 2024-04-04]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/standard/io/pipe-operations.

29. Microsoft. *Memory-mapped files* [online]. [cit. 2024-04-03]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files.

30. IBM. *Understanding memory mapping* [online]. [cit. 2024-05-01]. Dostupné z: https://www.ibm.com/docs/en/aix/7.2?topic=memory-understanding-mapping.

31. Microsoft. *Desktop Guide (Windows Forms .NET)* [online]. [cit. 2024-05-02]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-8.0.

32. Microsoft. *Desktop Guide (WPF .NET)* [online]. [cit. 2024-05-02]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-8.0.

33. Microsoft. *XAML overview (WPF .NET)* [online]. [cit. 2024-05-02]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/xaml/?view=netdesktop-8.0.

34. Microsoft. *Model-View-ViewModel (MVVM)* [online]. [cit. 2024-05-02]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm.

35. Thattil, Sascha. *Advantages and Disadvantages of WPF?* [online]. [cit. 2024-05-02]. Dostupné z: https://www.software-developer-india.com/advantages-and-disadvantages-of-wpf/.

36. Microsoft. *Windows UI Library (WinUI)* [online]. [cit. 2024-05-02]. Dostupné z: https://learn.microsoft.com/en-us/windows/apps/winui/.

37. Microsoft. *What's a Universal Windows Platform (UWP) app?* [online]. [cit. 2024-05-02]. Dostupné z: https://learn.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide.

38. Chess.com. *Chess Notation* [online]. [cit. 2024-05-02]. Dostupné z: https://www.chess.com/terms/chess-notation.

39. Tim Mann, H.G.Muller. *Chess Engine Communication Protocol* [online]. [cit. 2024-04-28]. Dostupné z: https://www.gnu.org/software/xboard/engine-intf.html.

40. chessprogramming. *UCI* [online]. [cit. 2024-05-02]. Dostupné z: https://www.chessprogramming.org/UCI.

41. Chess.com. *Elo Rating System* [online]. [cit. 2024-05-02]. Dostupné z: https://www.chess.com/terms/elo-rating-chess.

42. Knuth, Donald E.; Moore, Ronald W. An analysis of alpha-beta pruning. *Artificial Intelligence*. 1975, roč. 6, č. 4, s. 293–326. issn 0004-3702. Dostupné z doi: https://doi.org/10.1016/0004-3702(75)90019-3.

43. Chessify. *Maximizing Stockfish's Potential: A Speed Experiment on Cloud Servers* [online]. [cit. 2024-05-01]. Dostupné z: https://chessify.me/blog/stockfish-speed-experiment.

44. Chess.com. *Leela Chess Zero* [online]. [cit. 2024-05-01]. Dostupné z: https://www.chess.com/terms/leela-chess-zero-engine.

45. Microsoft. *.NET Standard* [online]. [cit. 2024-04-03]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/standard/net-standard?tabs=net-standard-1-0.

46. neuecc. *ZeroFormatter* [online]. [cit. 2024-04-04]. Dostupné z: https://github.com/neuecc/ZeroFormatter.

47. Intel. *Intel® RealSense™ Technology* [online]. [cit. 2024-05-05]. Dostupné z: https://www.intel.com/content/www/us/en/architecture-and-technology/realsense-overview.html.

48. Mejia-Trujillo, Jeison; Castano-Pino, Yor; Navarro, Andres; Arango Paredes, Juan; Rincon, Domiciano; Valderrama, Jaime; M. O., Beatriz; Orozco, Jorge. Kinect™ and Intel RealSense™ D435 comparison: a preliminary study for motion analysis. In: 2019, s. 1–4. Dostupné z doi: 10.1109/HealthCom46333.2019.9009433.

Seznam obrázků

A Attachments

A.1 Source codes

Source codes of the ChessMaster are attached to this thesis in the ZIP archive. The archive also contains README.md with user documentation. README references ChessTracking.md, which is user documentation for the chess tracking part of this project.

The ChessTracking.md is located in the Docs folder, as well as this thesis. The Images reference from the user documentation are located in the Images folder. /Data folder contains the tested PGN chess records in .pgn files.
stockfish_20090216_x64.exe is included as the default stockfish version.

All these folders are located in the ZIP archive as well as the GitHub repository.

For the current state of the source codes, check the GitHub repository.

A.2 GitHub

The source codes along with the user documentation and other files located in the ZIP archive are publicly available on GitHub at `https://github.com/bkapustik/ChessMaster`.