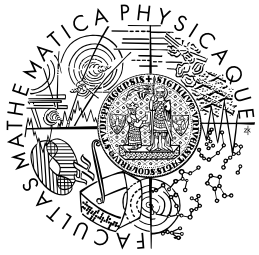


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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 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

Dedication.

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

Contents

1 Introduction

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

The main motivation is to use this project in events organized by the University to demonstrate robotic manipulator play chess against a person using an affordable camera in a showroom. This comes with a number of real world problems which will need to be solved such as configuring the robotic manipulator to synchronize with the positioning of the physical chessboard. Reconfiguring the setup after movements causing fault state. Reconfiguring the chess tracking part with ever changing light conditions.

This application gives us input for our robotic manipulator and lets us 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 which is no longer supported for newer technologies and being motivated to modernize the solution we will look at a proposition on how to integrate seemingly un-integrable part of code while keeping as much efficiency as possible.

This will result in us researching and implementing best possible options for Inter-process communication. We will use memory-mapped files in .NET which will also require parallel programming techniques. Using IPC and memory-mapped files we meet borders of a managed language as we will require memory managment while not being able to use pointers.

All that will be achieved while keeping the code of incompatible frameworks maintainable.

We will also integrate chess engine which will be used to compute moves which will be used by our robotic manipulator to play against the human user.

Executing the moves requires effective path selection in a space which will be achieved with a help of 3-dimensional path-finding algorithms.

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

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

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

1.1 Application specifications

Now that we have are familiar with our chess-playing project we can discuss what we are expecting of our application and create a project specification. Our application will be called ChessMaster.

The goal is to create a user-friendly graphical user interface which is simple to use yet provides all necessary features to correctly setup and use the robotic chess manipulator together with chess tracking sensor. The project aims to be able to be used in a showroom where visitors will be able to play chess against the robotic manipulator or watch a historic match or watch computer play against computer. 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 setup the robotic manipulator. We expect the robotic manipulator to be connected to computer by serial port. We need to select the right connection. After the right connection is selected and established we need to establish where should our robotic manipulator expect the chessboard to be placed. What is the size of the chessboard and where should it 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.

The application provides a way to set the correct location by forcing us 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. To visualise the current location of the robotic manipulator's hand we show the current 3-dimensional coordinations.

1.1.2 Selecting input of the game

After configuring the physical location of the chessboard the application letss 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 strategy. 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. Application asks the user to select a file with PGN record from the file explorer.

Watch AI match lets us watch a stockfish game engine play. The engine is designed not to remember state of the game. The application needs to remember the state and always provide the engine with a correct input. The engine computes 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 chosing version of the engine for 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 file explorer.

Play against AI match let's us use chess tracking provided by Kinect camera sensor and use the input of 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 [1]. 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 placement of the Kinect camera relative to the chessboard as well as the size and colors of the chessboard and chess pieces.

1.1.3 Chess tracker configuration

The application provides user interface to configure the parameters mentioned above and state of the configuration. Numerous parameters need to be configured for optimal results and effect of these changes can be seen in real time visualised in the user interface. This part of 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 Universal Chess Interface (UCI) notation. This part of the application needs to be reconfigured as the light conditions change.

1.1.4 Reconfiguration

The application needs to handle changes in the enviroment such as the chessboard or the robotic manipulator being moved or game being paused. We do not want to reconfigure whole setup each time this happens. The application handles this by providing us option to pause the game and reconfigure the localization of the edges of the chessboard. We can then resume playing without restarting the game.

1.1.5 Main interface

The application lets us see the log of played moves in the user interface. Above this log we provide a menu which lets us pause and reconfigure parts of the current state of the application. We can also see current coordinations of the hand of the robotic manipulator.

1.2 Hardware specification

Application integrates two hardware components that is Microsoft Kinect and custom made robotic manipulator.

1.2.1 Kinect

Kinect is a line of motion sensing input devices produced by Microsoft and 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

As part of the 2013 unveiling of Xbox 360's successor, Xbox One, Microsoft unveiled a second-generation version of Kinect (See Figure 1.4) with improved tracking capabilities (See table 1.1).

Kinect has also been used as part of 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 Microsoft's HoloLens project. Microsoft discontinued the Azure Kinect developer kits in October 2023.

1.2.2 Depth sensor

The depth and motion sensing technology 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 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

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

Table 1.1 Kinect v2 sensor characteristics

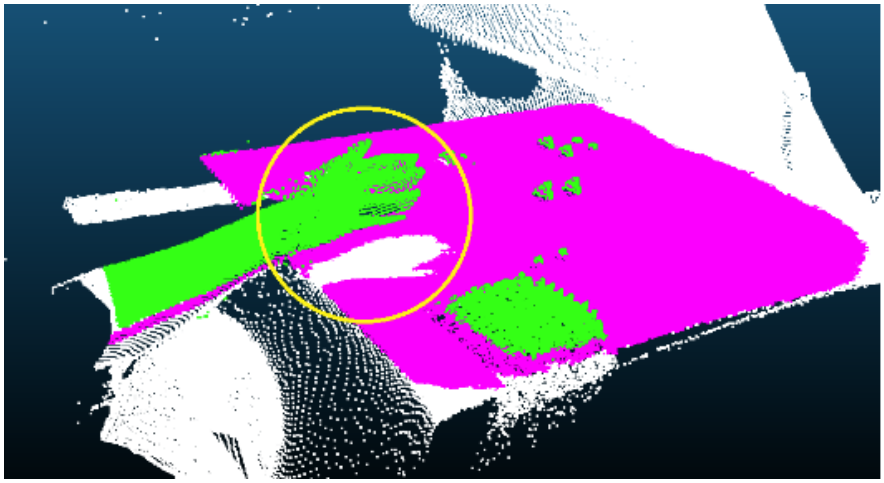


Figure 1.1 Visualization of the multipath interference problem - Purple points are the presumed table surface. Green points are points placed above the surface. In the yellow circle is a hand just above the table (made with [2])

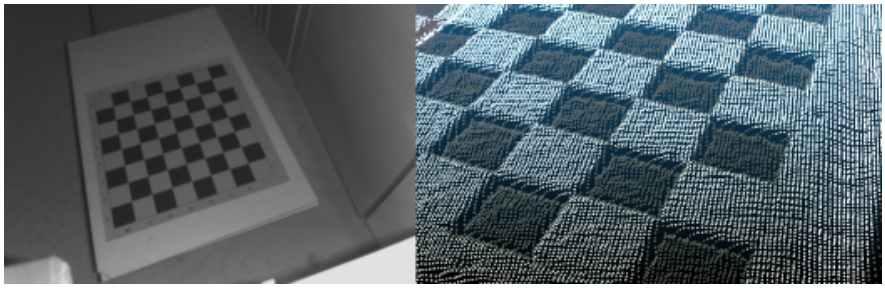


Figure 1.2 Visualization of the unsuitable materials problem. On the left is an image of the amount of reflected infrared light from the scene (brighter = more light). On the right is a detailed view of the chessboard as a point cloud [2]

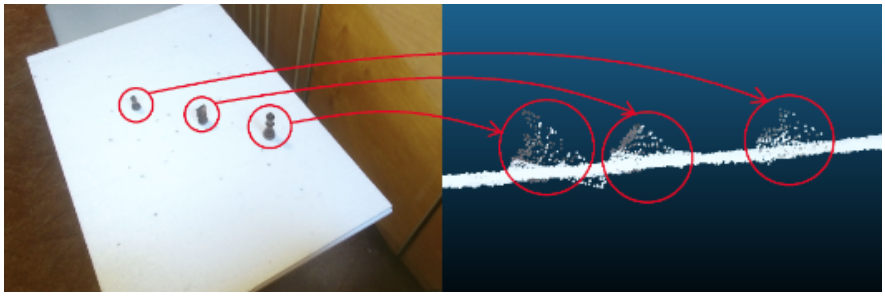


Figure 1.3 Visualization of the flying pixels problem. On the left is an image of the scene from the color camera. On the right is a side view of the point cloud representing the figures [2]

Figure 1.4 Kinect sensor v2

calculated in a shorter amount of time, allowing for more frames-per-second to be detected. [3]

(As discussed in [1]) 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.

However, several drawbacks can be observed. The first significant problem is known as multipath interference (Read more at [4]). 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. (See figure 1.1), white pixels can be observed beneath the plane of the table, which is caused by multipath interference.

Another problem is the observation of materials with poor properties 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 (See figure 1.2), 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. 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, so 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 is the case with the (See figure 1.3).

1.2.3 Robotic manipulator

Robotic manipulator has been custom-made and we do not have a proper documentation, however it is connected via USB and serial port is used for

communication. For communication G-code commands are used. The manipulator works similarly to a crane. It has a grip which can be opened and closed and which can move vertically.

G-code (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.

These are the commands we will use to control our robotic manpulator.

- Move accross X, Y and Z axis respectively - Xnnn Ynnn Znnn (rational numbers are expected)
- Move home - $H
- Open grip - M8
- Close grip - M9
- Reset - $X
- Get current state - ?
- Resume -
- Pause - !
- Set linear movement - G00
- Get info - $#

1.3 Related work

Now that we have a rough idea about our application we will briefly look at other projects focusing on chess-playing robots.

The 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 [5] analyzes pixels near the center of the squares of the chessboard in regards to its red, green and blue values to detect if a square is vacant or not. This work uses OpenCV library for comuter vision using the Corner-based approach for the chessboard recognition.

Another project authored by David Vegas Romero [6] uses different Machine Learning and Deep Learning techniques to perform the recognition of the chess pieces. This approach requires training of a computer neural network model and deep computer neural network model.

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

State of the chessboard is usually transformed to one of the following notations

- Portable Game Notation (PGN)
- Steno-Chess
- Forsyth–Edwards Notation (FEN)
- Extended Position Description (EPD)

State of the game is then used to compute next suficient 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. A robotic manipulator performs this chess move. Robotic manipulators can use electro magnets to hold a chess piece. Magnet is placed on the top of the chess piece and robotic arm uses the electro magnet to pick up the figure and then place it on the right chess square. Another approach is to use a magnet placed on the bottom of a chess piece and held from below the chessboard. Electro magnet is then used to move the piece without a need to pick up the piece above the chessboard and other pieces. This requires moving other pieces from the trajectory of held piece or some kind of a navigation between the pieces.

2 Problem Analysis

In this chapter we mainly look at algorithmic problems encountered in this thesis.

2.1 Computing movement paths

Our robotic manipulator provides option to choose between interpolated or not interpolated movement. Our program architecture is modular and is not dependent on 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 that is only moving to necessary coordinates. As the robotic hand is not a point but rather a solid figure we need to take it's size into consideration as well as 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 to consideration as well. Soon we realize that we are dealing with a path planning problem. Theory about path planning algorithms can be read in the book [7]

2.1.1 Naive approach

The simplest approach would be to set a constant height at which the robot would be moving. When the robot wants to go from point x to point y where it picks up the figure it moves only accross the horizontal plane at fixed height which is above all the obstacles even when holding a chess piece. When the hand wants to pick up or position a piece it moves only vertically. This would ceirtanly work but a lot of time is wasted on an inefficient path.

2.1.2 Shortest path

We would rather find the shortest path to move between two points to save time. As explained in the previous chapter (See 1.1.1) we are only given two edge points of the chessboard in the configuration. For simplicity we construct two dimensional array where each position represents a square on the chessboard and positions for captured pieces. Each square can contain an entity that is chess figure with known height. The size of each square is computed and known at the time of finding a path.

We can not look at the squares as vertices of a graph because the shortest path would cross the edges of the square under an angle which is not a multiplier of 90 unless the two points that is the squares have the same x or y coordinate. On top of that the chess pieces are solid figures as well and their exact size would need to be measured. We can not effectively create a graph because the number of points between any two points on the chessboard is infinite.

We could take into consideration only some points of the space but for that we would need to have a large number of points in the memory if we wanted to

efficiently approximate the chessboard and the chess pieces. A graph created from such vertices would need to be recomputed each time a figure is moved.

As Jihee Han's publication [8] says 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. In previous decades, several methods to overcome the complexity of this problem have been developed. The classical approaches include A*, Dijkstra, PRM (Probabilistic Roadmap Method), and Artificial Potential Field. These approaches were intensively studied in the early days of 3D path planning. 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.

Implementing a standard algorithm for finding the shortest path would take too many computational resources and would be too hard to implement to be useful.

2.1.3 Solution - finding a sufficiently short path

Instead of finding the shortest path we will try to create an algorithm which finds a suficiently short path and which will often result in being the shortest while not using as many computatinal resources as finding the shortest path every time.

Our solution is to find the shortest horizontal path without considering the obstacles. We can do this because we are operating in 3 dimensions and we can move above all of the obstacles because the chess figures are located at the bottom of the chessboard. That is trivial because the shortest horizontal path is a straight line between two points. Horizontally our robotic arm will move on this line and the only problem we need to solve is efficient vertical movement. 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 on to the chessboard.

Each time the path of the robotic arm horizontally crosses a figures it needs to be above the figure. Therefore we can devide 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. But other obstacles can be located on the 3 dimensional line between the position of 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 to be horizontally intersected if the heighest point of the figure is higher than the current vertical position of the arm.

We already know the line between two points but we do not know which chess figures would be horizontally intersected. We esentially need to find which squares on the chessboard are intersected by a line drawn between two points for which

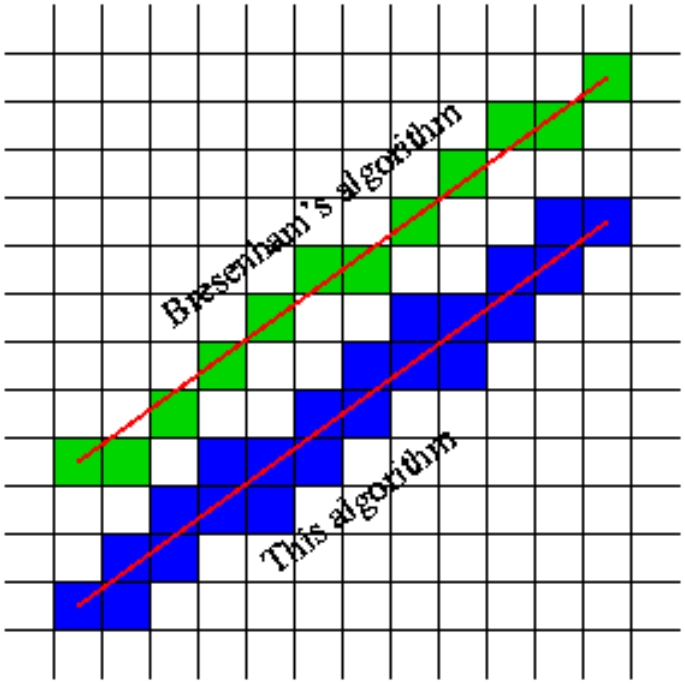


Figure 2.1 Difference between Bresenham's classic and customized algorithm

we need a line drawing algorithm that needs to account for which squares the line passes through. [9] This is similar to the problem solved by algorithms like Bresenham's line algorithm, but with a focus on identifying all the squares that a line intersects rather than drawing the line pixel by pixel. Our problem is similar to a line drawing or line traversal algorithm that's been adapted to identify all the squares (or tiles) on a grid that a line intersects. You can read about Bresenham's algorithm and similar problems in book (See [10]).

2.1.4 Bresenham-based supercover line algorithm

We will use an algorithm created by Eugen Deduv(See [11]). This line is called supercover line and this algorithm might be a particular case of DDA (Discrete Differential Analyzer) algorithm. It may be useful for example when you have to know if an obstacle exists between two points (in which case the points do not see each other).

Figure (See figure 2.1) shows the difference between the Bresenham's algorithm (which draws a classical line) and this one (which draws a supercover line).

If the Bresenham's algorithm does not change the y-coordinate (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 Bresenham's 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:

```
// three cases (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.

Implementation of the algorithm Here is the algorithm (the instructions added to the Bresenham's algorithm are in bold): The difference with Bresenham is that ALL the points of the line are used, not only one per x coordinate.

```
void useVisionLine (int y1, int x1, int y2, int x2)
{
  int i;               // loop counter
  int ystep, xstep;    // the step on y and x axis
  int error;           // the error accumulated during the increment
  int errorprev;       // *vision the previous value of the error variable
  int y = y1, x = x1;  // the line points
  int ddy, ddx;        // compulsory variables: the double values of dy and dx
  int dx = x2 - x1;
  int dy = y2 - y1;
  POINT (y1, x1);  // first point
  // NB the last point can't be here,
  // because of its previous point (which has to be verified)
  if (dy < 0){
    ystep = -1;
    dy = -dy;
  }else
    ystep = 1;
  if (dx < 0){
    xstep = -1;
    dx = -dx;
  }else
    xstep = 1;
  ddy = 2 * dy;  // work with double values for full precision
  ddx = 2 * dx;
  if (ddx >= ddy){  // first octant (0 <= slope <= 1)
    // compulsory initialization (even for errorprev, needed when dx==dy)
    errorprev = error = dx;  // start in the middle of the square
    for (i=0; i < dx; i++){  // do not use the first point (already done)
      x += xstep;
      error += ddy;
      if (error > ddx){  // increment y if AFTER the middle ( > )
        y += ystep;
        error -= ddx;
        // three cases (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);
        }
      }
      POINT (y, x);
      errorprev = error;
    }
  }else{  // the same as above
    errorprev = error = dy;
    for (i=0; i < dy; i++){
      y += ystep;
      error += ddx;
      if (error > ddy){
        x += xstep;
        error -= ddy;
        if (error + errorprev < ddy)
          POINT (y, x-xstep);
        else if (error + errorprev > ddy)
          POINT (y-ystep, x);
        else{
          POINT (y, x-xstep);
          POINT (y-ystep, x);
        }
      }
      POINT (y, x);
      errorprev = error;
    }
  }
  // assert ((y == y2) && (x == x2));  // the last point (y2,x2) has to be the same
}
```

Here we have supposed that if the line passes through a corner, the both squares are drawn. If you want to remove this, you can simply remove the else part dealing with the corner.

Note: The line is symmetric, that is a line from x0,y0 to x1,y1 is the same as a line from x1,y1 to x0,y0.

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 on the vertical movement described in (See 2.1.3). Our robotic arm has a certain width as well as the chess figures which we need to take into consideration and therefore the movement can intersect more than one figure at the same time. 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 inside the group. As we know the source and target coordinates we can use the line equation to compute exact coordinates at which each group is horizontally intersected and finally use these points as an input in the previously mentioned algorithm described in (See 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 figure, chessbooard and plane localization. This section is a sumarization of decisions and explanations of the used algorithms and is written with a help of AI to give the reader a brief summary of the Chess tracking project. (See more about the project in the thesis by Roman Staněk [1]).

2.2.1 Detection introduction

The section outlines the approach for selecting computer vision algorithms for chess analysis with a focus on utilizing Kinect sensor's spatial data. It covers:

- Algorithm Selection (See 2.2.2): Emphasis on algorithms' ability to process spatial data, inspired by similar works that divide tasks into chessboard detection and piece detection.
- Game Plane Detection: Initial detection of the plane (e.g., table) where the game occurs to reduce processing complexity and environmental noise for subsequent steps.
- Chessboard Detection: Utilization of color images to extract features such as edges and corners of the chessboard using edge detectors and filters. The position of the chessboard is determined through a custom algorithm inspired by probabilistic segmentation methods.
- Piece Localization: Verification of chess pieces' presence by analyzing data points above the chessboard squares, addressing spatial data deformations and noise with various filtering methods.
- Continuous Tracking: Maintenance of the chessboard's current state in real-time, adapting to players' moves and environmental changes like sensor movement or scene occlusion.
- Additional Considerations: Determination of piece color and integration of game state knowledge to assist in tracking and application functionality

2.2.2 Algorithm selection

In the quest to identify the game plane within a point cloud for chess analysis, multiple approaches were considered, with the RANSAC 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, Region Growing, and Linear Regression.

RANSAC (Random Sample Consensus) 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 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's 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 sensor's perspective. 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. These methods, while powerful, 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. This plane's model estimation 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.

2.2.3 Looking for the chessboard

Upon determining the game plane, the space where the chessboard might be located was defined, mitigating the impact of surrounding noise and reducing the data volume for further analysis. This step facilitated the precise identification of the chessboard's position and orientation. Various practical approaches to this problem, including those found in related work on determining the state of a chess game from a color image, were reviewed. A particular method was selected to identify significant features of the chessboard and their placement in the image, followed by the application of a custom algorithm to determine the chessboard's position in spatial data.

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's-eye view 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's sensitive to parameter settings, as it may also identify piece transitions, decorative chessboard borders, or external objects as corners (See figure 2.2c).

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't form an entire edge by itself in edge detection. However, unwanted edges may still be identified and subsequently filtered based on length or angle (See figure 2.2d).

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

This binary image served as input for the Canny 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 (See figure 2.2f).

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.

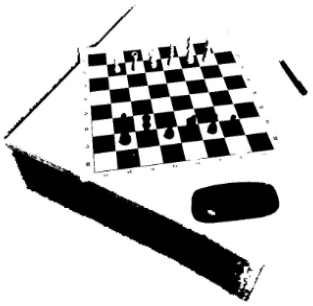
2.2.4 Localization in space

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.

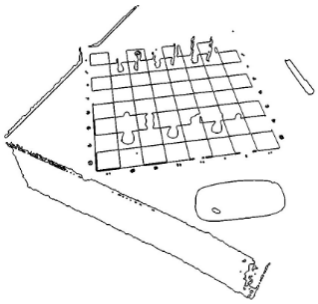
The proposed solution addresses these inaccuracies by selecting key points that carry crucial information, mapping them into space, and fitting them with a grid

(a) Colorful chessboard shot

(b) Gray scale



(c) Threshing



(d) Edge detection

(e) Hough transformation

(f) Edge filtering

Figure 2.2 Steps of edge detection

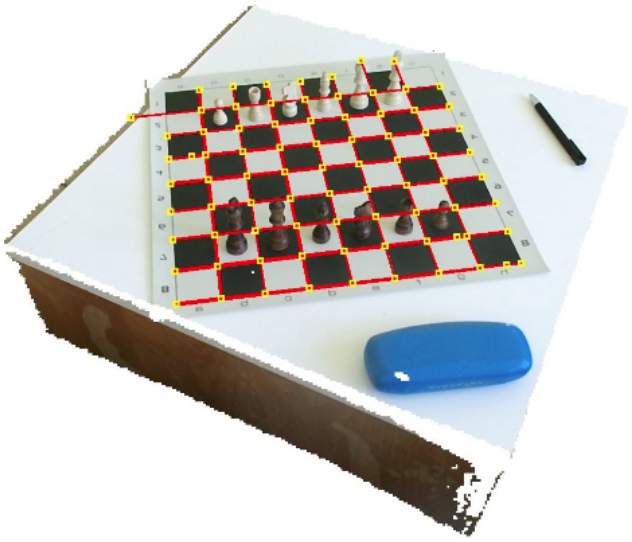


Figure 2.3 Display of edge intersections - red lines represent the edges, yellow squares represent the intersections and ends of the edges.

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 chessboard's vertical and horizontal edges available from the image, which represent the precise corners of the chessboard squares and carry significant information about its position and shape (See figure 2.3). Additional points of interest include the ends of the chessboard's edges, 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 sensor's mapping function 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 algorithm's operation is demonstrated with a smaller problem of fitting a 2x2 chessboard grid with nine points (See figure 2.4).

For each acquired point, referred to as B, several nearest points from the surrounding area 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 chessboard's corners are calculated.

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 neighbors considered for each point (p). The algorithm initializes an empty chessboard model and iteratively selects and evaluates point pairs from the neighbors, 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 chessboard's position in space (See figure 2.4).

2.2.5 Figure localization

In the process of spatial localization, if all edges of the chessboard were identified from the color image, 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 cluster of pieces, rendering the Hough transformation ineffective. 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 color 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.

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 2.5b.

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

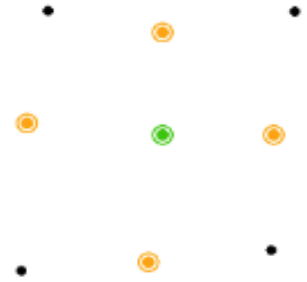
Using the sensor's mapping function between the color image and spatial data, the 3D coordinates of these points are obtained. A RANSAC-like algorithm is then applied to these points, demonstrated on a smaller problem of interpolating a 2x2 chessboard grid with nine points (See figure 2.5c).

This involves selecting the nearest points around a chosen point, B, ideally four, but possibly more to account for inaccuracies. Pairs that are approximately equidistant from B and form nearly right angles are generated from these surrounding points. These pairs are then used to generate all possible chessboard models, evaluating each model's success rate against B.

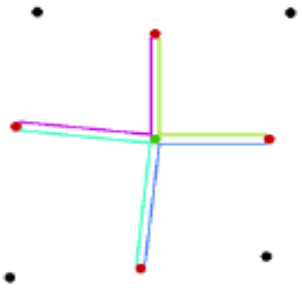
The model with the smallest sum of Euclidean distances between its grid points and the nearest actual data points is considered the most successful, indicating a close match to the measured data and thus an appropriate model for the chessboard's spatial position.

Finally, recognizing the color of the chess pieces is necessary for practical tracking. This complex task is approached by averaging the luminance component of the pixels constituting a piece, setting a threshold to differentiate between black and white pieces. This method, although simple, is not always reliable, especially when shadows on the chessboard could cause black pieces to appear lighter than shaded white ones.

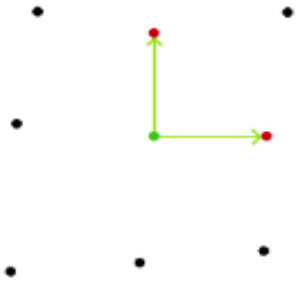
The solution partly relies on the knowledge of the game state, adjusting the threshold to favor the detection of the expected piece's color. This approach works well for moves but not for captures, where multiple opposing pieces could be taken



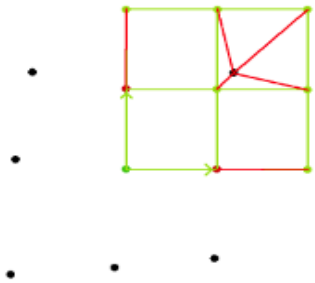
(a) The processed point (in green), and its neighbors (in orange)



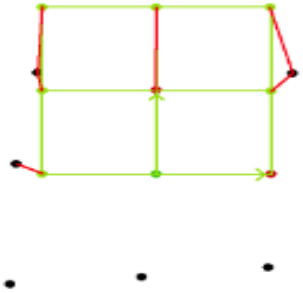
(b) Pair of lines



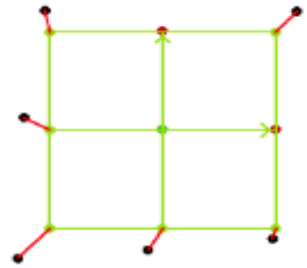
(c) The processed pair



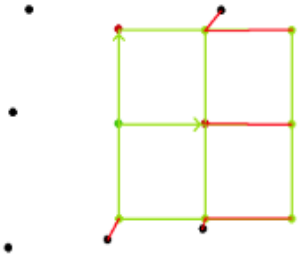
(d) First chessboard model



(e) Second chessboard model



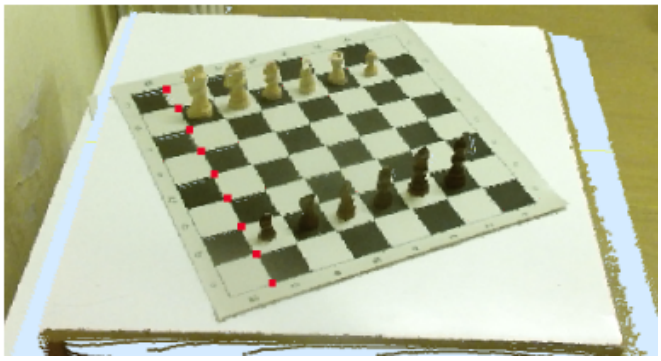
(f) Third chessboard model



(g) Fourth chessboard model

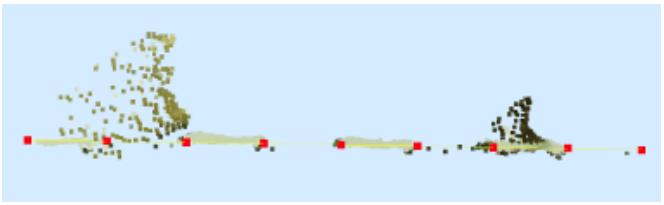
Figure 2.4 Chessboard fitting algorithm with points - green grids are the model of the chessboard, red lines are deviations from the actual data

(See figure 2.6).

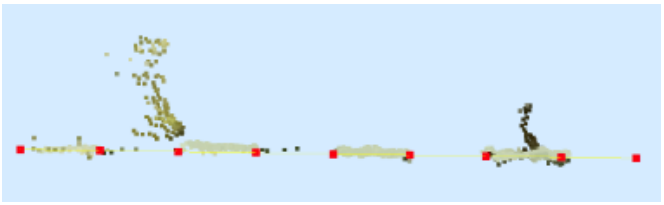


(a) Point cloud containing the chessboard

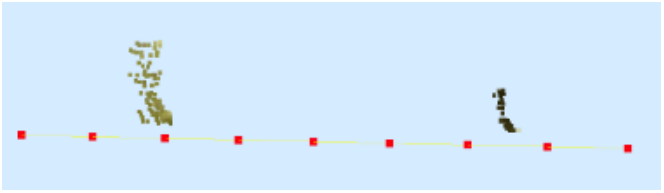
(b) View from the left of the first column on the left containing the chess pieces



(c) Data after removing the black squares



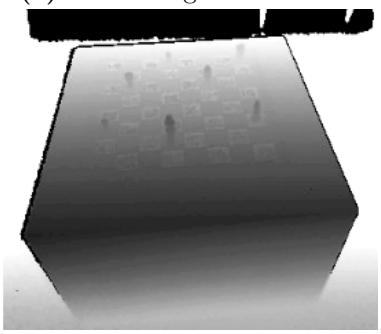
(d) Data after removing the flying points



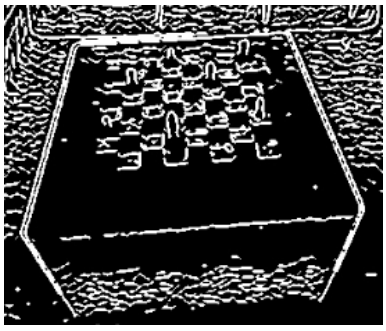
(e) Data after removing points from the squares of the chessboard

Figure 2.5 Individual steps of piece localization

(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

Figure 2.6 Application of the Canny edge detector on depth data

3 Technical analysis

3.1 Technologies used

We are already familiar with the main parts of the application. Now let's look at used programming languages and platforms. Application is implemented in the C# language. Main part of the application that is robotic manipulator control, computing moves, game strategies, chess tracking algorithms and communication with other parts is developed on .NET 7 platform.

User interface is created in the WinUI 3. Kinect input is collected by a program written on paltform .NET Framework 4.8.

Explanation on why we combine two programs each developed on different platform is included in (See 3.2). To connect these two applications we need to use an IPC, which is explained in (see 3.3.2). For the IPC we are using Memory-mapped (See [12]) files in both versions of the .NET platform, which requires using a serializer.

We are using a ZeroFormatter. The application has been developed in Visual-Studio and versioned in Git. Third party libraries are provided by NuGet package manager.

Algorithms used by the Chess Tracking part use Emgu.CV, Accord and Math-Net NuGet packages. As for the Kinect integration we are using Microsoft.Kinect NuGet package and Kinect SDK.

We are using Stockfish as the chess engine which is explained in detail in the subsection (See 3.1.1)

3.1.1 Computing chess moves

There are many chess engines which can beat human players and whose ELOs are bigger than ELO of any human player. This thesis does not focus on resolving chess algorithm problem as that can be considered already solved.

We need to choose between available options of possible chess engines. One of the most famous and highest scoring engines is Stockfish. This is an open source chess engine which provides it's own command line with commands used mainly to set the current state of game and computing next best move in a given time. There are not libraries in .NET which would provide purely programmatical way of using stockfish, however there is a workaround in a form of wrapper of stockfish command line program.

.NET stockfish NuGet package provides simple API to set current moves of the game in a form of either collection of previous UCI moves which lead to the specific game state, or General chess representation which is a format which can store whole state of game in one line.

This NuGet package opens an instance ie .NET Process instance and redirects input and output of the console to our program. Another option would be to use a different engine or use a python program which has API for communicating with stockfish without a need for starting a new process.

We would however still need to configure IPC which would lead to a few problems. The python program would probably be slower than using stockfish

process directly. IPC is not very fast either. There is a whole section dedicated to IPC later in this thesis (See 3.3.2).

3.2 Kinect input

This thesis is mainly focused on other parts of the application and the Chess Tracking part is taken as a foundation for our application. The chess tracking part uses Kinect v2 on platform .NET Framework. There is also version for C++, however rewriting the whole program would be out of scope of this thesis. The decision to use .NET is written in (see [1]): For the second version of Kinect, there are at least two maintained application interfaces (APIs). The first of these is Kinect SDK 2.0. This is the official library directly from Microsoft. The second alternative is the open source implementation of drivers, OpenKinect4, and the open source libraries linked to it. In this work, we decided to use the official Kinect SDK, with which we expected fewer problems with installation and compatibility.

The decission to implement this application in .NET turned out to not be very handy. The Kinect library is only supported for .NET Framework 4.8 and by the time we have realized this limitation much of the application has already been written in .NET Core 7. Reimplementing the Kinect input capturing would be out of scope of this thesis mainly because of time limitation. The structure of the old application is monolitic and it took time to find a border between what can and what can not be changed. The unofficial library may not offer the same API or possibilities.

On the other hand migrating whole software written in .NET Core could be even harder. There might not be the same possibilities because the .NET Framework is deprecated. Whole of the UI would need to be rewritten as we are using WinUI 3.

Input from the Kinect as well as Kinect control is still captured by .NET Framework application while other parts run on .NET Core. .NET Framework part is integrated as a process of the main application and IPC(Inter process communication) is established between them. (See 3.3.2)

3.3 Migrating legacy code to .NET Core

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 modern platform and not produce another legacy code.

There are already many works focused at migrating applications from old version of one programming language to another or older version of a framework to newer version. One critical part of code has been written in .NET Framework which is a legacy framework. This part depends on a NuGet package which does not exist in the newer .NET, that is Kinect for Windows. Many companies and programs might depend on legacy code which is not able to be migrated to more modern enviroments. While this might not be the most straightforward or handy solution to our problem, this is a situation which might happen and is worth analysing. Therefore let's focus on trying to use the best possible option in our situation.

First we need to establish what parts can be migrated to the newer platform. We have decided to use and send raw Data from the Kinect because the application was monolitic and finding out what parts of code can be devided from each other was not easy. If there was more time it would be a much better idea to pre-process the raw data and exchange such data instead of the raw Kinect data. The raw data take up a lot of memory. We have tested average size of all data needed for tracking algorithms which need to be exchanged in between the two processes and the result is around 40MB per frame. The size is not that much to keep in the memory at the time because the data can be processed quickly and not all data needs to be accounted for. We discuss this in more detail in the subsection (See 3.3.2).

Data structures used by the Kinect library are not available for .NET Core and the tracking algorithms partialy rely on them. The .NET Framework was not open-source and the Kinect library wasn't open-source neither. We need to create the same data structures to exchange between the two application processes. (IPC part See 3.3.2) requires a Serializer and IPC classes which need to share the same structures. Rewriting 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 code in a different way it would be harder to keep track of the source code in multiple source-codes. We want to keep the same prescription for both processes the same and only change independent parts without affecting the other process. The same serializer needs to be referenced from both processes.

We can connect these two application into one C# solution. We create a .NET Standard project which serves as a prescription for both applications. .NET Standard can be referenced from both .NET Core and .NET Framework. As is written in the Microsoft documentation (See [13]). .NET Standard is a formal specification of .NET APIs that are available on multiple .NET implementations. The motivation behind .NET Standard was to establish greater uniformity in the .NET ecosystem. .NET 5 and later versions adopt a different approach to establishing uniformity that eliminates the need for .NET Standard in most scenarios. However, if you want to share code between .NET Framework and any other .NET implementation, such as .NET Core, your library should target .NET Standard 2.0. No new versions of .NET Standard will be released, but .NET 5 and all later versions will continue to support .NET Standard 2.1 and earlier.

We can define same structures and use the same serializers as well as the same algorithms in one source-code and use them in both projects even though the project are incompatible.

3.3.1 IPC

As we have mentioned we need two processes which need to communicate with each other. Both are compiled at the same time and when the main part of the application starts the Kinect Input Inter process communication (IPC) needs to be established.

There are multiple options for IPC in .NET. Let's compare some of the available options and decide which one is the right for us.

- IPC over network .NET provides many ways for IPC over a network. We

do not need to share data over network as our scenario is mostly suitable for local communication. We could teoretically 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 much sense. Data which need to be sent are quite large so the network overhead would become a problem.

Nevertheless there are some good network IPC technologies which are atleast worth mentioning.

  - gRPC is a language agnostic, high-performance Remote Procedure Call (RPC) framework. (See [14])

    The main benefits of gRPC are:

    * Modern, high-performance, lightweight RPC framework.
    * Contract-first API development, using Protocol Buffers by default, allowing for language agnostic implementations.
    * Tooling available for many languages to generate strongly-typed servers and clients.
    * Supports client, server, and bi-directional streaming calls.
    * Reduced network usage with Protobuf binary serialization.

    These benefits make gRPC ideal for:

    * Lightweight microservices where efficiency is critical.
    * Polyglot systems where multiple languages are required for development.
    * Point-to-point real-time services that need to handle streaming requests or responses.

  - TCP/IP
  - The advantage of TCP/IP is that the communicating applications can be running on different computers and different locations. Since Internet also works on TCP/IP, remote applications can communicate over the internet. TCP/IP is completely platform independent, standard, and implemented by all operationg systems (and many other devices)

    [9] Setting up and managing TCP/IP networks, especially on a large scale, can be complex. It requires careful planning of IP addressing, subnetting, and routing configurations.

    [9] TCP, in particular, introduces a significant amount of header data and control mechanisms like handshaking, acknowledgments, and congestion control. This overhead can reduce the efficiency of data transmission, especially in high-latency or low-bandwidth environments.

- Pipes (See [15]) Anonymous pipes provide interprocess communication on a local computer. Anonymous pipes require less overhead than named pipes but offer limited services. Anonymous pipes are one-way and cannot be used over a network. They support only a single server instance. 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.

- Files File operations require disk I/O, which is significantly slower than memory operations. Files would need to be removed at the end of the communication as they are not managed by garbage collector. Explicit cleanup mechanisms would need to be introduced to avoid wasting diskspace. Files also come with a significant overhead from the System operations such as opening, closing, reading from and writing to.

  Using Files would certainly not be a good idea.

- Memory-Mapped Files (See [12]) A memory-mapped file contains the contents of a file in virtual memory. 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.

  - Persisted memory-mapped files (See [12]) Persisted files are memory-mapped files that are associated with a source file on a disk. When the last process has finished working with the file, the data is saved to the source file on the disk. These memory-mapped files are suitable for working with extremely large source files.
  - Non-persisted memory-mapped files (See [12]) Non-persisted files are memory-mapped files that are not associated with a file on a disk. When the last process has finished working with the file, the data is lost and the file is reclaimed by garbage collection. These files are suitable for creating shared memory for inter-process communications (IPC).

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

  The best option in our case would be the Memory-Mapped files as we need to transfer data between our processes as fast as possible. [9] Pipes compared to Memory-Mapped files provide a sequential access. Data written to one end of the pipe must be read in the same order from the other end. This means every byte of data has to be copied from the writing process's memory space to the kernel and then to the reading process's memory space, introducing overhead for each read/write operation.

  We do not need to persist data and can dispose of them after our processes are ended. Therefore the best option will be non-persisted memory-mapped files.

3.3.2 IPC solution

.NET offers two variants for managing memory-mapped files, that is MemoryMappedViewStream and MemoryMappedViewAccessor. Both come with their advantages and disadvantages.

The ViewAccessor provides randomly accessed view of a memory-mapped file. This is better suitable for highly random reads and writes of smaller data. ViewAccessor provides more freedom for memory-mapped file (MMF) managment but also requires more work to manage the MMF. The stream is more suitable for sharing larger sequential data between processes. This approach is more straightforward but comes with an overhead of stream managment and is more limiting.

Do we need the random access? Let's quickly look at what exactly do we need. The input from the kinect will be a large structure consisting of byte arrays, arrays of 3-dimensional vectors and ushort arrays. The size of the data is not known in advance. The size depends on the specific frame. The size of the data has been measured and on average comes at around 40MB.

It might seem that the stream managment would be a better idea because would like to share larger sequential data. In our case however we find out that the access is not as straightforward. First of all read/write operations are the most computionally expensive part of chess tracking algorithm even when using shared memory. More frames are captured than what is able to be read/written and therefore we need to throw away 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 localization and later displaying the frames in the UI.

Random access offers us option to partition the data which is about to be read/written and work on each part in parallel which we need for optimalization. ViewAccessor gives us direct control of specific memory segments which makes it easier to synchronize access to different positions in the memory.

We need a lot of synchronization in our case because we need paralelism and synchronizing adds more complexity within streams related to stream positioning.

We have decided that it is better to use MemoryMappedViewAccessors. Let's look at the implementation.

3.3.3 Shared memory multi buffer

Our proposed solution uses parallel reader and writer. When creating a MMF we first allocate desired size of the file. This should be a multiplicator of maximum expected size of the kinect data plus serialization (See more about this in 3.3.4) overhead. As the Kinect input data does not have a constant size we also need to exchange metadata containing the size. 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 which is 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 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 speed differs. This comes with a latency resulting from a finished member waiting for the other to finish.

We can solve this by introducing multi buffer implementation. In our case writer is generally faster so the additional buffer lets us write to a buffer meanwhile another buffer is being read. At the time when the writer is finished it doesn't have to wait for the reader to finish his reading and can instead write to another buffer. When the reader is finished he also has a buffer prepared for reading.

3.3.4 Serialization

C# is a managed language and does not give us much freedom when dealing with unmanaged memory. MMF is an unmanaged memory and any process could use this to write something malicious to our file. MMF are limited to only use value types and arrays of bytes. We can not use pointers so we can't look at an object as an array of bytes and vice versa. The data needs to be serialized.

This serialization comes with more overhead, however .NET offers some really fast binary serializers. The one which we used is the ZeroFormatter (See [16]) which claims to be the fastest serializer for .NET.

This allows us to serialize the Kinect data to array of bytes which adds some speed and memory overhead. This overhead is not very significant. We then write the data to our buffer in parallel, that is we partition the array and as the ViewAccessor makes it easy to randomly access memory we write each part independently. We do the same when reading the data.

3.4 Swapping contexts

Because we want our application to be able to effectively change between game modes (See 1.1.2) and to pause and reconfigure the game we need to handle the effects of the change.

First of all we need to handle the pauses. There are multiple layers to our application (later discussed in 4.1) at which we need to handle the pauses. At the lowest layer we have the robotic manipulator which reacts to changes through a serial port. Once a command is sent to the serial port it obviously can not be cancelled, but we can control the commands which have already been ordered, but which have not been sent to the serial port just yet.

The layer which communicates with the serial port is implemented as a queue but which can only accept one atomic operation which needs to be executed before listening to other commands. The atomic operation can usually contain multiple commands, but which will be executed in the order in which they have been received. Until all of them are executed the queue is not accepting any other commands. These commands must be executed.

The only exception is when we need to immediately stop the execution at which point no received command is executed except for special commands such as resume. When the movement is resumed the ordered commands continue the execution in the initial order.

This layer does not handle any state and only receives atomic commands to execute.

Moving upwards we have another layer. This layer keeps state about entities in a grid. And accepts more complex commands such as move entity from source to target. This layer is not affected by pauses as much as it waits for the lower layer to finish and only orders another execution once the lower level is done. When the game is resumed it finishes it's command.

This approach lets us have a control of what exactly is the state of our game. Thanks to this we can pause the game. Finish the execution of the current move and safely swap the configuration or the game mode knowing exactly where our pieces are and where should the robotic arm be.

Another layer above is responsible for keeping track of the actual game. It's responsibility is to execute actual chess moves. It can be paused and resume execution either by only finishing the current movement and then starting a new game or ordering another commands in the same game.

If we anything goes wrong in the actual physical setup it is possible to reconfigure the position at which the chessboard should be while keeping the state of the game. The game can be resumed even after reconfiguration because we know what is the state of the game and the actual state of the board.

Game modes can also be changed and we can chose whether we want to continue with the same positioning of the pieces or start from the beginning. This is useful if we for example wanted to show a part of historic game or watch a part of ai match and then swap to human playing.

4 Implementation

4.1 Architecture

In this section we look at the architecture of the application. The architecture is devided into two main parts. We first look at the architecure of the ChessMaster which is the main part connecting all parts of the application. Later we look at the architecture of ChessTracking which is the part of the program responsible for collecting input data from Kinect, IPC between .NET Core and .NET Framework processes and chessboard localization.

4.1.1 ChessMaster

This part is made of multiple layers. In this subsection we discuss the meaning of the most important abstractions in these layers.

- **UserInterface** is reponsible for collecting input data for configuration of the setup. It displays information about tracking and played game.
- **ChessRunner** is a class which is referenced by classes from UserInterface. This class works as a mediator between abstractions of the robotic manipulator and chess strategies. It redirects calls from the UserInterface onto another layers. It has a loop which 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.
- **ChessRobot** is a class which is able to accept and call execution of ChessMoves. It inherits from RobotSpace which is a more general class capable of executing RobotCommands. ChessRobot has a ChessBoard field which is an abstraction encapsulating more general Space which holds coordinations of the grid representing the game space. ChessRobot can translate commands, such as MoveFigure and CaptureFigure which are translated from a chess notation, into a form more suitable for robotic manipulator.
- **RobotSpace** is a class which listens to commands such as move entity from source to target. These entities are a generalization for a physical object located on the grid. Grid is a generalization of the space in which our robotic manipulator can operate. It devides 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 which 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.

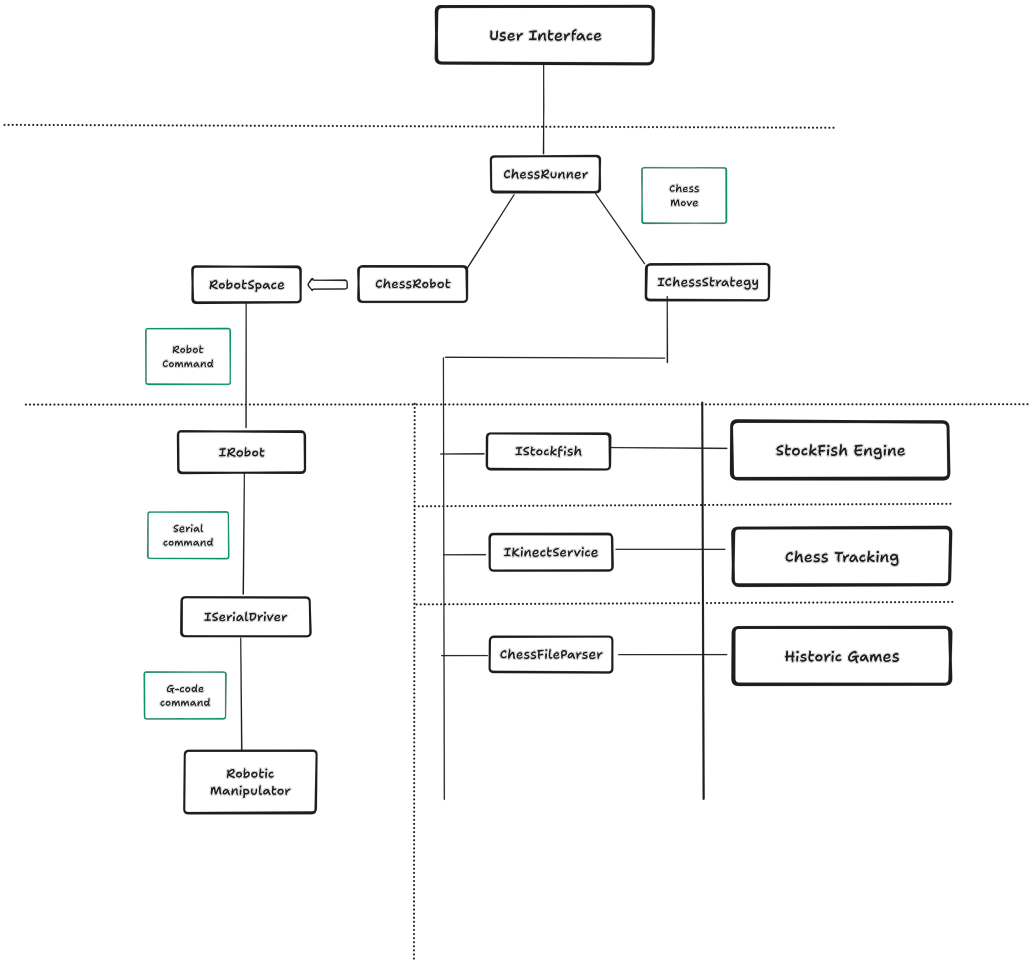


Figure 4.1 ChessMaster architecture

- **ISerialDriver** is an interface which sends SerialCommands to the robotic manipulator. It does not hold any context and serves as a communication layer between higher layers and RoboticManipulator.
- **IChessStrategy** is an interface which accepts calls to compute next chess moves. This interface acts as an iterator which when incremented advances the game by a chess move. This is an abstraction which can be implemented as any of the game modes. The implementations can communicate with the Stockfish or iteratively get next move from a historic game.

4.1.2 ChessTracking

This part of the application consists of multiple projects. This part implements IPC via memory-mapped files. Some parallel techniques are used for performance, the interprocess synchronization is provided by mutexes. We use the named system mutexes which are visible throughout the operating system. We need this because the ChessTracking.Kinect runs as a different process than the rest of the application.

These are the projects creating the ChessTracking part of the application.

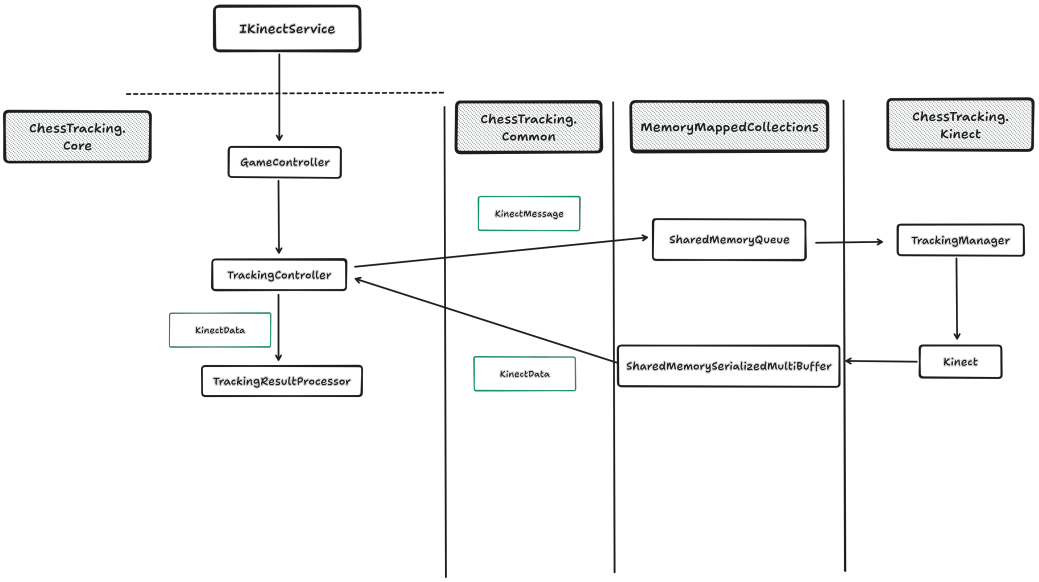


Figure 4.2 ChessTracking architecture

- **ChessTracking.Core** is a project which is responsible for localization of the chessboard as well as the figures on the chessboard. It consists of a pipeline which 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** is reponsible for controlling the game in the context of the Tracking. The tracking algorithms operate in the context of a game. This class starts, stops and loads a game.
  - **TrackingController** reacts to GameController. It can start/stop the tracking and communicates with the Kinect layer via shared memory. It holds 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 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 project which serves as an interface for the actual .NET implementations. It defines KinectData and messages which can be used in both .NET implementations. This is a .NET Standard project which lets us use the common data structures. This project is referenced by both the ChessTracking.Core and ChessTracking.Kinect

- **ChessTracking.Kinect** This layer collects input data from the Kinect. It holds references to the same memory-mapped files as the TrackingController. It listens for the input to start or stop the tracking and controls the Kinect

camera. Kinect class then calls SharedMemorySerializedMultiBuffer to write the data to the shared memory.

- **MemoryMappedCollections** is a .NET Standard project which defines shared memory collections. Two classes are defined here which can be used in .NET Core and .NET Framework projects.

  - SharedMemoryQueue is a simpler shared memory collection which is used for simple messages.
  - SharedMemorySerializedMultiBuffer is a definition for a shared memory collection using multi buffer pattern. It is used for more complex data structures, in our case KinectData. It uses ZeroFormatter serializer to write and read the data to the shared memory. The data is serialized to array of bytes. The arrays are devided into parts which are processed in parallel in order to maximize the performance.

4.2 Application proposal

In this section we discuss some of the mentioned components in more detail. We discuss UserInterface and show the resulting application.

4.2.1 UserInterface

The UserInterface is created in WinUI3. It uses MVVM pattern. ViewModels use Services which act as facades of the interface provided by lower layers. All services are registered in App.xaml.cs as singleton services. The program uses dependancy injection. Services expose event handlers which are subscribed to by the UserInterface.

Application can open multiple windows. Windows navigate between pages. Pages override OnNavigatedTo. In this method dependancy 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 also subscribe to the event hadlers provided by the services. These event handlers are unsubscribed in the override of OnNavigatedFrom.

Application starts and communicates with other processes which are dispoosed of and closed when the ChessMaster application closes.

Application dialog requests user to pick a chess Strategy (See figure 4.3). 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 to pick a Stockfish.exe file.

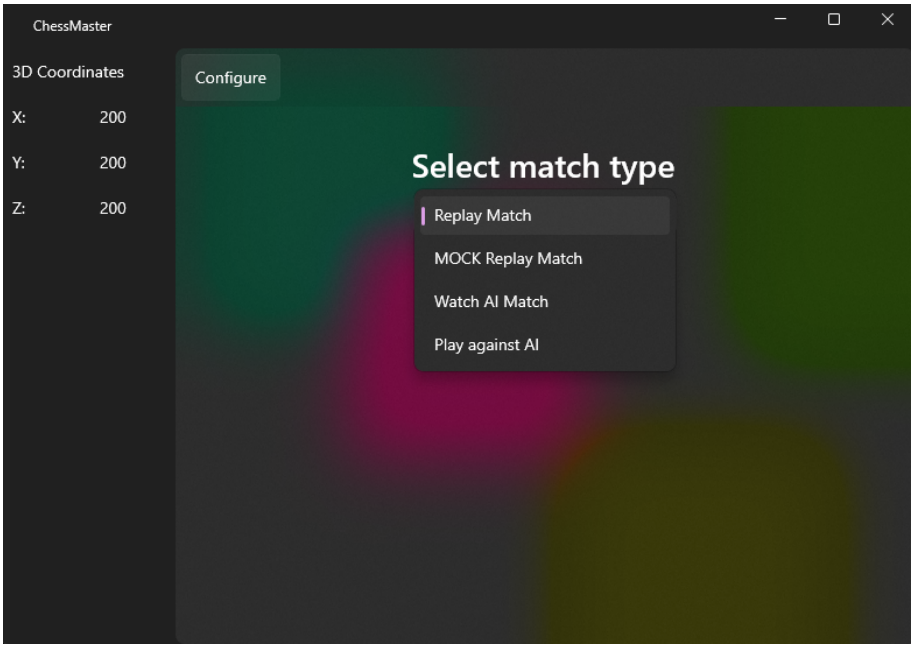


Figure 4.3 Strategy Selection

4.2.2 Play against AI strategy

This subsection discusses a strategy which 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 which are only used to initialize from old game or provide their state for other strategies.

ChessStrategies act as iterators. This particular strategy is interesting because it connects complex parts of the program yet has very simple implementation. When this strategy is initialized it is given IKinectService which is managed by dependancy 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. (See figure 4.5) This window displays results of the tracking and localization. Tracking can be configured to accustom to changes in light conditions (See figure 4.4).

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 the stockfish to compute a chess move or it listens for the input from the ChessTracking.

Strategies fire events when moves are completed. Pages of user interface listen to these events and display the moves in a log. (See figure 4.6)

4.2.3 Context changes

As discussed in (See 3.4) the application needs to handle context changes. The user interface handles this as follows. The page with game log (GamePage) lets us pause the game. This pause is immediate and current move is not finished.

To change a strategy or reconfigure the position of the robotic manipulator

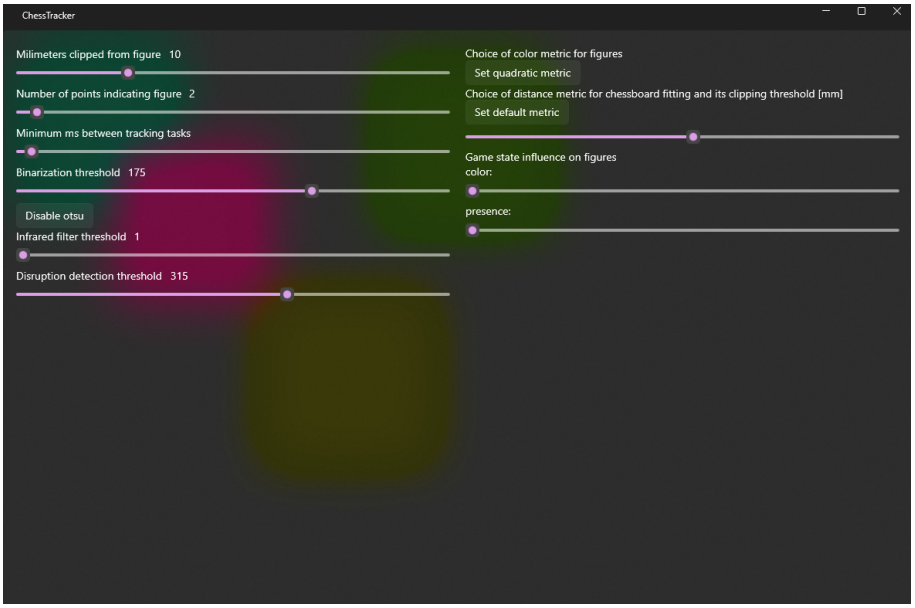


Figure 4.4 ChessTracking advanced settings

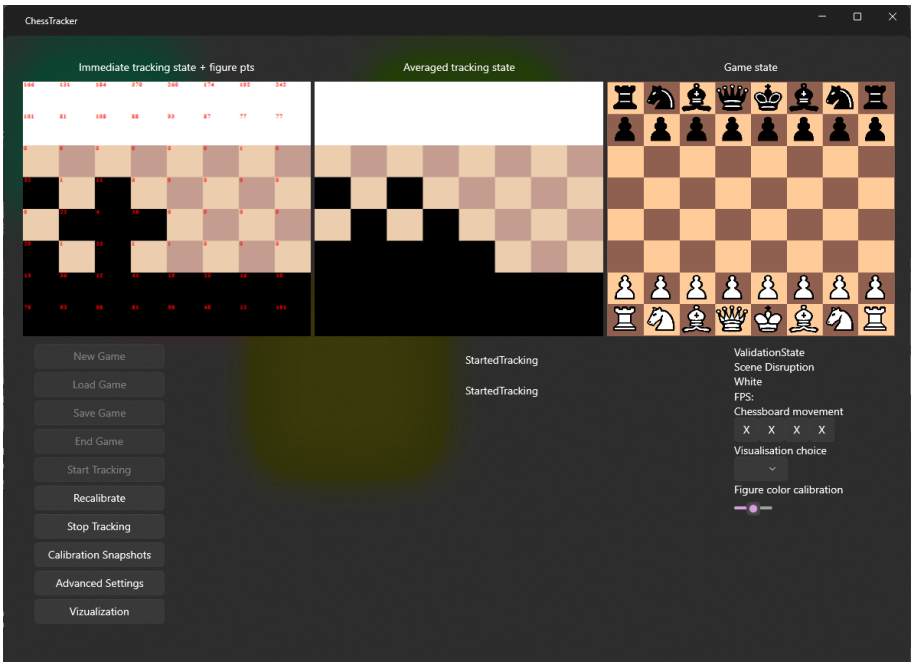


Figure 4.5 ChessTracking main page

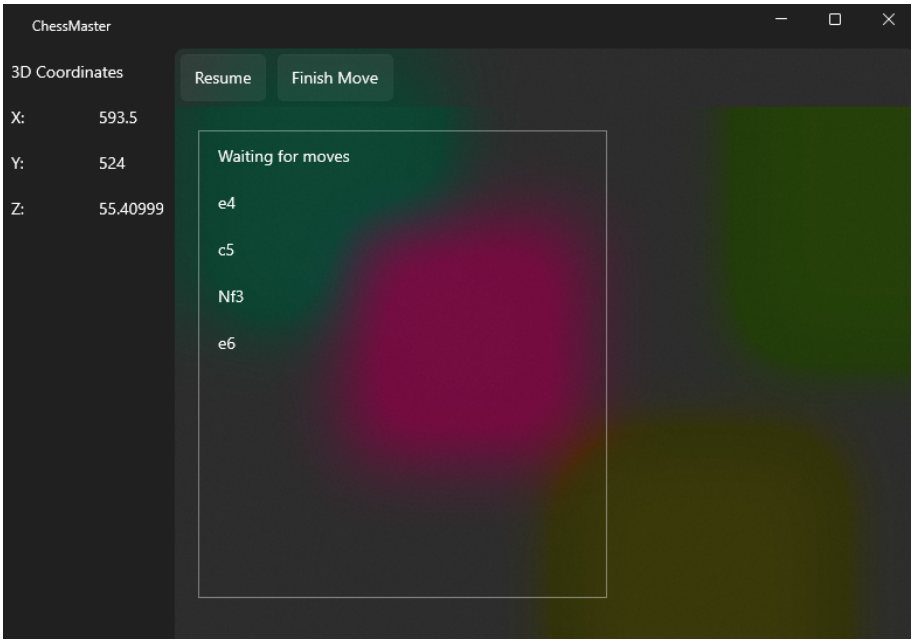


Figure 4.6 Game log - game is paused

user needs to press Finish Move. (See figure 4.6) This finishes move which has already been scheduled. After the move is finished (See figure 4.7) the user can Change Strategy or Reconfigure (See figure 4.8).

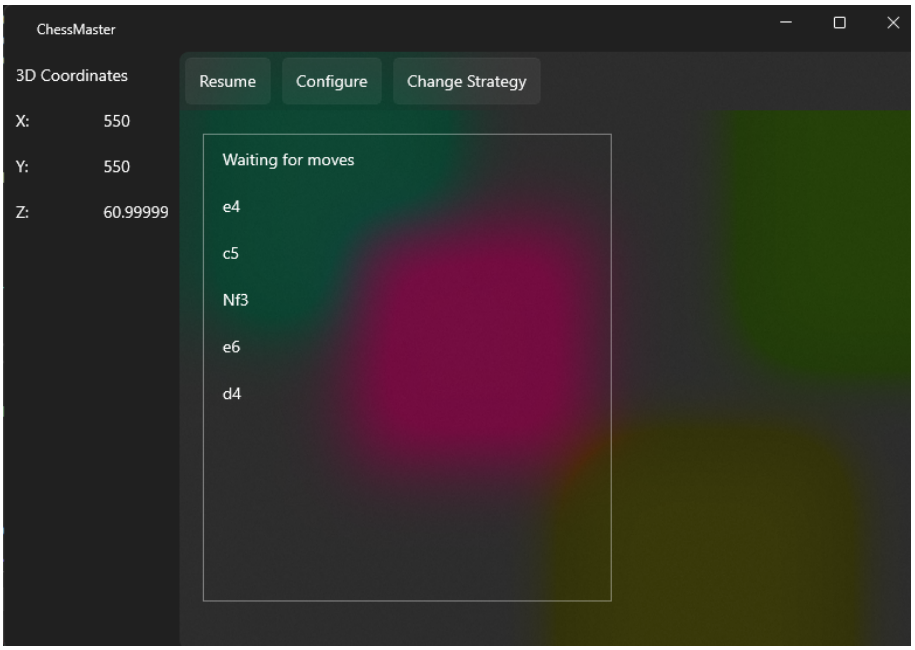


Figure 4.7 Game log - move has been finished

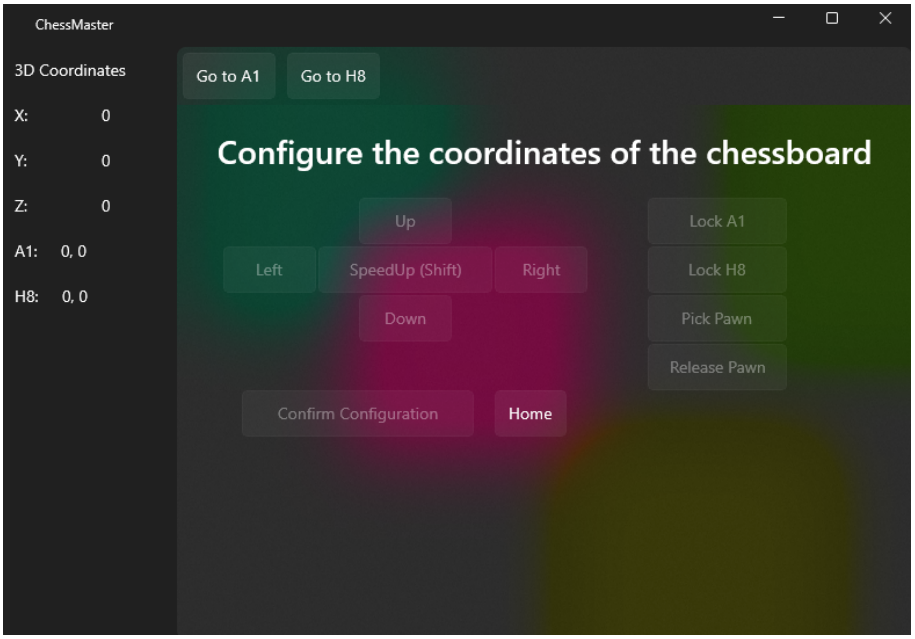


Figure 4.8 Configuration

5 Discussion

5.1 Lessons learned

First of all it is important to research the initial state first, namely the ChessTracking application on which our program was based. More time should have been put into researching Kinect API as there are third party libraries which support Kinect in .NET Core. Knowing this, whole application should have been using .NET Core and there would be no need for IPC and memory-mapped files which slow down the application. Then the User interface could have been using a different framework other than WinUI which would make the application cross-platform.

Another problem with WinUI other than not being cross-platform is that it can not directly draw a bitmap image. The images need to be converted to other formats. Bmp format can be used which is a bitmap image format but it still requires conversion. That is obviously not ideal for an application which works and should draw images in real time. This conversion however was less problematic and takes less resources than localizations and transfering data through IPC.

Even if the application was not rewritten to use the third party libraries for Kinect, the IPC could be made faster by pre-processing the input data from Kinect and taking some of the responsibility of the pipeline to the .NET Framework process. This would reduce the size of the data being transfered through Memory-mapped files.

The simplest chess notation to work with was UCI notation as it is simple to parse and each move is a direct representation of what should happen. UCI notation is supported by Stockfish. On the other hand we have made a chess strategy which accepts PGN notation which is more difficult to parse into form which can be used by our program. PGN notation does not directly tell what is the source and what is the target of a chess move. It often says only what should be the target and expects that the rest is obvious from the context of the game. While that is true, it makes it harder to work with the notation in our application.

5.2 Known issues

Pawn promotions are currently not supported. ChessTracking part does not support this because it does not differ between figures. It can only detect colors of the figure so there is currently no way to localize that the figures of the same color have been changed. It is also problematic to have a figure set with enough placed chess pieces to support all possible pawn promotions and to show such set in a showroom.

Castling is not supported because chess tracking part of the application does not detect castling moves. Unlike Pawn promotions this can be fixed more easily because it is not a matter of detecting individual figures.

Conclusion

We have managed to create a control application which integrates Stockfish, Kinect and robotic manipulator to play chess. Chesstracking application has been integrated and the use of computer-vision algorithms gives us input which we use to control the robotic manipulator. Even though there is some slowdown to chess tracking because of the use of MMF and IPC the application is stil responsible and quickly detects changes on the chessboard. Chess moves resulting from these changes are quickly processed. Application is capable of playing chess matches and replaying historic games. These modes can be changed without corrupting the application. Application is capable of being used with the setup in a showroom.

Application is responsive and simple to use. Application is safe and handles situations when game or hardware pieces are disturbed.

During the development of the application we have created mocks for simulating the process of the game. 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.

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 founds 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 managed to find a solution to effective movements of the robotic manipulator to handle the pieces. This has lead us to research path finding algorithms in 3D in a real time.

Bibliography

1. Staněk, Roman. *Board Games Tracking Using Camera and Depth Sensor* [online]. [visited on 2024-04-02]. Available from: https://dspace.cuni.cz/handle/20.500.11956/109049.
2. Girardeau-Montaut, Daniel. *CloudCompare* [online]. [visited on 2024-04-11]. Available from: https://www.danielgm.net/cc/.
3. Hamed Sarbolandi Damien Lefloch, Andreas Kolb. *Kinect Range Sensing: Structured-Light versus Time-of-Flight Kinect* [online]. [visited on 2024-04-02]. Available from: https://arxiv.org/abs/1505.05459.
4. Girardeau-Montaut, Daniel. *Introducing Project Kinect for Azure* [online]. [visited on 2024-04-11]. Available from: https://www.linkedin.com/pulse/introducing-project-kinect-azure-alex-kipman.
5. HEDSTROM, FREDRIK BALDHAGEN ANTON. *Chess Playing Robot* [online]. [visited on 2024-04-02]. Available from: https://www.diva-portal.org/smash/get/diva2:1462118/FULLTEXT01.pdf.
6. Romero, David Vegas. *Implementation of a Chess Playing robot application* [online]. [visited on 2024-04-02]. Available from: https://upcommons.upc.edu/bitstream/handle/2117/333724/tfm-davidvegas.pdf?sequence=1&isAllowed=y.
7. LaValle, Steven M. *Planning algorithms*. Cambridge University Press, 2006. isbn 0521862051.
8. Han, Jihee. *An efficient approach to 3D path planning* [online]. [visited on 2024-04-03]. Available from: https://www.sciencedirect.com/science/article/abs/pii/S0020025518309332.
9. Available also from: https://chat.openai.com/.
10. Abrash, Michael. *Michael Abrash's Black Book of Graphics Programming (Special Edition)*. Coriolis, 1997. isbn 9781576101742.
11. Dedu, Eugen. *Bresenham-based supercover line algorithm* [online]. [visited on 2024-04-03]. Available from: http://eugen.dedu.free.fr/projects/bresenham/.
12. Microsoft. *Memory-mapped files* [online]. [visited on 2024-04-03]. Available from: https://learn.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files.
13. Microsoft. *.NET Standard* [online]. [visited on 2024-04-03]. Available from: https://learn.microsoft.com/en-us/dotnet/standard/net-standard?tabs=net-standard-1-0.
14. Microsoft. *Overview for gRPC on .NET* [online]. [visited on 2024-04-04]. Available from: https://learn.microsoft.com/en-us/aspnet/core/grpc/?view=aspnetcore-8.0.
15. Microsoft. *Pipe Operations in .NET* [online]. [visited on 2024-04-04]. Available from: https://learn.microsoft.com/en-us/dotnet/standard/io/pipe-operations.

16. NEUECC. *ZeroFormatter* [online]. [visited on 2024-04-04]. Available from: https://github.com/neuecc/ZeroFormatter.

List of Figures

A Attachments

A.1 First Attachment