UNIVERSITY OF ZAGREB

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS no. 2557

ROBOT CONTROL BASED ON VIDEO INFORMATION ON A CHESS GAME CASE STUDY

Domagoj Kudek

Zagreb, June 2021.

Želio bih zahvaliti svom mentoru, doc. dr. sc. Mirku Sužnjeviću, za sve što se za mene napravili. Pomogli ste mi više od ostatka fakulteta zajedno.

I'd also like to thank Tommy Jonsson and the rest of the Byte Motion AB team for all the help and guidance while working on this project.

Couldn't do it without you guys.

Zahvalio bih se i svojim roditeljima, koji su me uvijek podržavali i pomagali mi za vrijeme studija.

CONTENTS

1. INTRODUCTION

The technology world and its spectrum of smart services and applications based on artificial intelligence are rapidly growing. AI is the main driving force of Industry 4.0 which mainly focuses on fast and smart interconnections between various independent devices that enable us, the developers, to envision new and intelligent systems. The power of machine learning is already changing our daily lives without us even thinking about it. The robotic industry is no exception to this technological change. Factories, assembly lines, processing plants, and other industrial environments that use robots are swiftly accommodating these changes in order to improve their production. With the use of smart sensors, intelligent machinery, and a high degree of automation, we are able to greatly influence a wide range of industries that are in a need of evolution and change.

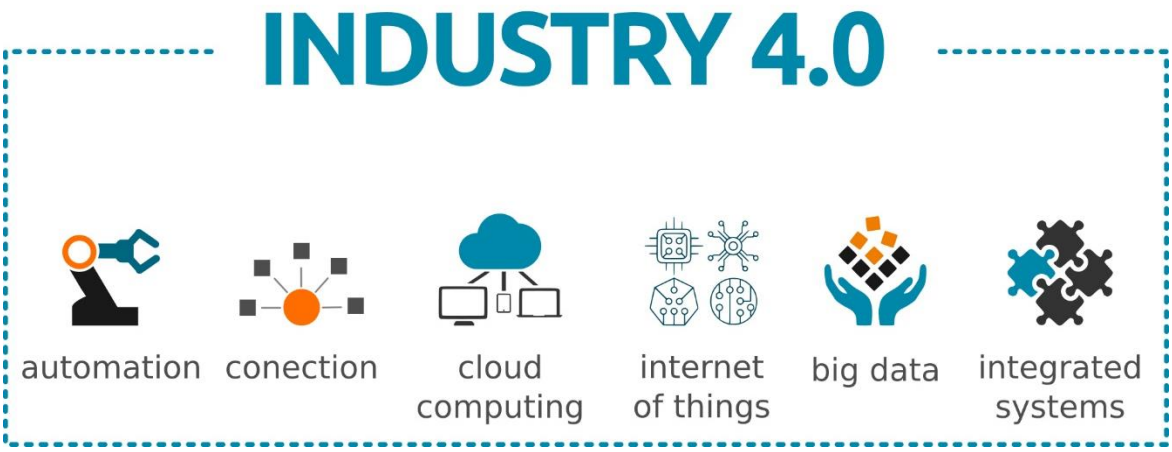

Figure 1.1: Industry 4.0 main focus points

These specially crafted systems are mostly purpose-built, making reuse and adaptation to different use case scenarios almost impossible. The software behind these automated machines develops slower than the hardware that is powered by them. Companies that produce high-grade robotic equipment like ABB and Kuka focus their efforts on making a stable and broad range of products along with offering their maintenance services, while the developers writing classical robot software lag behind both in functionality and the speed of development. The reason for that is that robot languages tend to be very complex to use, which influences the labor price of a limited number of programmers that specialize in that area of work. There is a great demand for a universal solution based on commonly used tools

and programming languages. Robot programming should be simplified in order to attract more potential users and developers to the ecosystem.

Figure 1.2: Siemens AG smart factory powered by Industrial IoT devices

One such idea is to use a game engine to develop an application for robot control. Most popular game engines are based on well-known and established languages like C++ and C#, which would enable a much more efficient way of creating smart robotic systems. The number of potential developers that are fluent in these languages is several magnitudes higher than classical robot software used within the robot's micro-controllers. Classical robot software would become irrelevant for a particular system since all of the heavy lifting would be done in our main application running within a game engine. Creating such a system is no small feat and requires several steps for its completion. Firstly, through the use of different sensors, we need to recreate the real-world inside of a virtual simulation. Without this step, standard functions like path planning and obstacle avoidance are impossible to implement. The virtual world needs to be modeled from the information gathered by precise cameras and other sensing equipment for our system to meet the high standards used in the robot industry. The second step is to create a robot controller inside the game engine. Things such as inverse kinematics, movement speed, and robot commands have to be implemented in order for the final step to work. The support for various robots, APIs, and services

also have to be implemented to make our system universally compatible with as many hardware combinations as possible. Lastly, a real-time API for transferring these controls has to be established between the game engine and the robotic hardware.

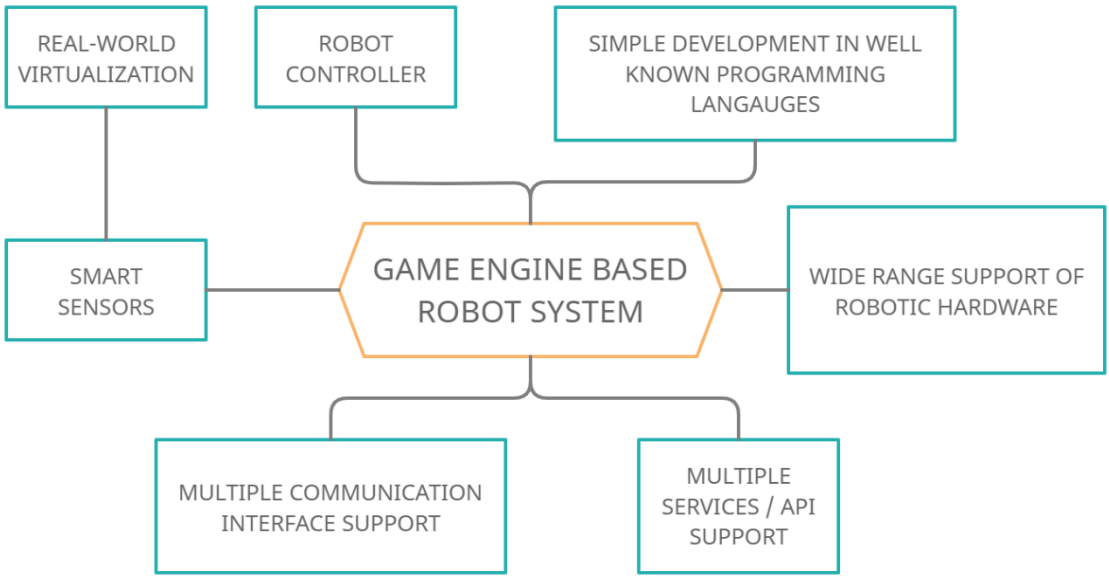

Figure 1.3: System diagram of a game engine based robot system

Making such a system would solve most of the currently known issues linked to robot software. A unified system that works with different kinds of hardware and software, running on a well-known platform that is easier to develop, in popular programming languages would vastly reduce the development time of artificial intelligence-based robot systems. Systems based on this technology would be much more adaptable to different scenarios making them potentially much more useful to a larger number of users. This approach would also create a new community of developers who could exchange and share parts of their developed systems with other members. Currently, there are several companies that actively work on this idea. One of them we cooperate with is Byte Motion AB.

1.1 MOTIVATION

As seen in [10], previous attempts to create a robot controller with popular programming languages like JavaScript have been made. The author used NodeJS and Express.js with the help of the Intel Edison hardware platform to create an intuitively designed interface for robot control and showed that the concept of using cheap and simple solutions can have very successful results in the end. Further attempts have been made for simplification of the robot control process, like the one in [11]. A web interface with basic status information about the robot and the ability to send and receive messages containing directional instructions for the movement of the robot has been accomplished. This shows the importance of simple user interfaces that anybody can use, despite their level of knowledge. Our envisioned system will also need one such web user interface for controlling robot configuration, and other hardware and software parameters.

In [7], a bionic hand finger controller has been developed. It uses Leap Motion sensor technology for finger detection, but more importantly, it uses Unity game engine as a platform that connects the web application, the hardware, and the robot arm. A system that is built in the scope of this work will also be based entirely within Unity. This shows that Unity has great potential for industrial use, especially within the robot and automation sector. Another very similar system was developed in [6], where instead of Leap Motion, the author used depth cameras similar to the ones we are going to use in our project. Real-time tracking and detection of the arm movements were mimicked by the remote controlled robot arm.

As for the use of cameras and sensors for detection purposes, in [26], depth cameras and laser scanners were used to guide and plan a mobile robot’s path through the environment. Even though the system was developed in Robot Operating System, a robot software library and toolbox, the use of different sensors in combination with robotic hardware has great potential for improving its capabilities and functionalities. Lastly, in [23], the author created a robot that is able to execute a sequential transfer of multiple objects in its workspace. This type of pick & place actions will be crucial for our use case of transferring objects in the real world.

Like we previously mentioned, a comprehensive solution for advanced robot control guided by external sensors is needed in the current automation industry to drive the possible productivity forward. That way, robot users don't need to rely on the software support of robot makers, making the whole development process easier and accessible to a larger number of potential developers.

1.2 SOLUTION

In this thesis, we present AutoCHESS, a program based on the previously described idea of robot control from a game engine. An industrial robot with 2 separate arms is going to play a game of chess against itself. The moves of an AutoCHESS game as well as the movement of the robotic arms are fully automated through the use of artificial intelligence and an advanced vision system called Ocellus. Ocellus is implemented inside a game engine called Unity, while the move logic relies on currently best free chess algorithm Stockfish. Robotic arms will know which move is the optimal one based on the current state of the board. Then, they will be able to play that move, guided by an external depth camera that will provide positional information. Chess figure positions will be calculated with the use of object detection powered by machine vision.

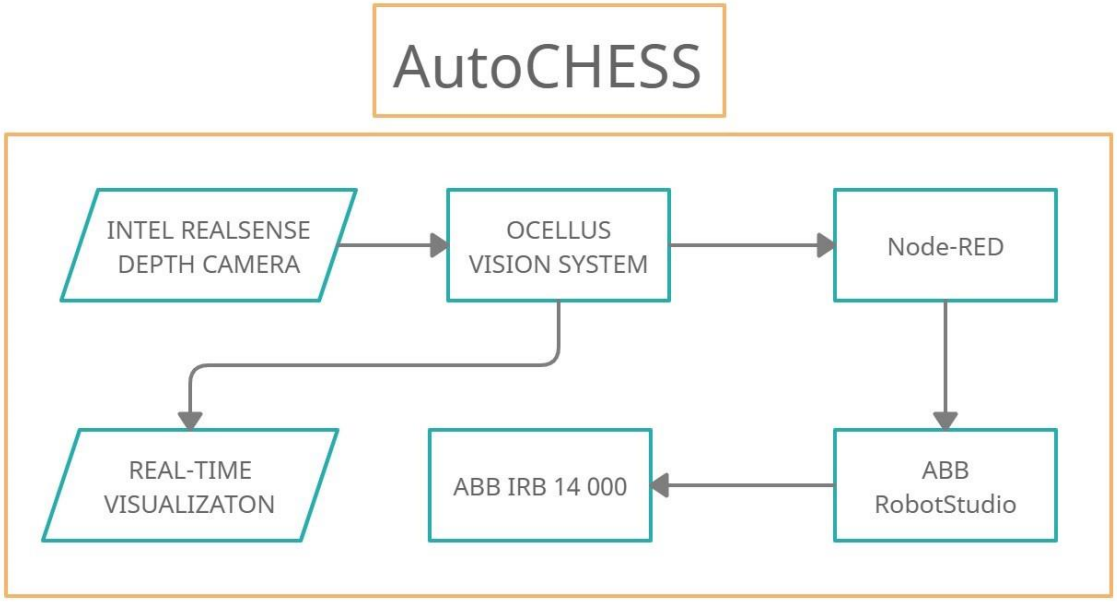

Figure 1.4: AutoCHESS application high-level system diagram

The complete system will serve as a demonstration of the capabilities of game engine-based robot control systems as well as a showcase of the use of advanced artificial intelligence algorithms in industrial robotics.

1.3 THESIS STRUCTURE

The thesis is structured as follows. Chapter 2 gives a detailed description of all the used technologies and services used for the development of the AutoCHESS application. Chapter 3 explains a detailed implementation description focusing on 3 main components of the system. The implementation section is organized as follows. The first sub-chapter deals with the data gathering process, while the second sub-chapter talks about the use of machine vision. The third sub-chapter is dedicated to the Ocellus vision system configuration and usage, fourth speaks about the use of Node-RED as an interconnect between Ocellus and robot hardware. And finally, the fifth is focused on the YuMi robot itself and the use of ABB's proprietary software. Chapter 4 gives a look into the real-life deployment of the AutoCHESS system, while Chapter 5 presents the end results of our accomplished system. The final Chapter 6 gives a conclusion to the work we have done and to the thesis itself.

2. TECHNOLOGIES AND TOOLS

A complex project like AutoCHESS requires many different components to work. To develop this kind of system, we used a wide range of platforms and programming tools that are at our disposal. In this chapter, we are going to examine the most important ones and explain their role in this project.

2.1 Chess

Chess is a well-known board game which in its current state is played for more than 5 centuries across the world [12]. It is first and foremost a strategy game with no hidden information, played on a square board with 64 squares, arranged in an eight-by-eight grid. Each player has 6 different types of figures that have their own rules of movement. The main goal of the game is to checkmate the opponent's king, meaning that the king is under attack and can't be moved to a safe location. Both players start with 16 figures in total, where the player with white figures makes the first move.

Figure 2.1: Wooden chessboard and chess figures

The game itself is divided into three phases where different strategies and tactics are used. In the opening phase, which ranges between ten and twenty moves,

players prepare by moving their figures to useful positions for the upcoming battle. Then goes the middle-game phase, where the main concern for the players is to achieve long-term positional advantages with minimal losses. The endgame is focused on the survival of the king with only a few pieces left. There are countless strategic options on how to approach a chess game and choosing the optimal move each time can be a very strenuous mental task. That is why the development of strong chess engines takes a lot of computing time. Our chessboard is a Tournament No. 6 type. That means that the dimensions of the figures and the fields are valid for chess competition use. Each chess square is 55 * 55 mm long. The squares themselves are white and brown in color which should help to differentiate the black color of the figures from the brown color of the squares.

2.2 Stockfish

Using computing power to calculate the best possible move in a game of chess is already quite an explored idea. During the 90s, IBM's Deep Blue computer has beaten the former world champion, Garry Kasparov, several times, and from that point on, countless new chess algorithms have appeared. One such algorithm that is for many consecutive years at the top of the chess algorithm charts is called Stockfish [13]. It's a free and open-source chess engine developed by a large community that is based on a distributed testing framework.

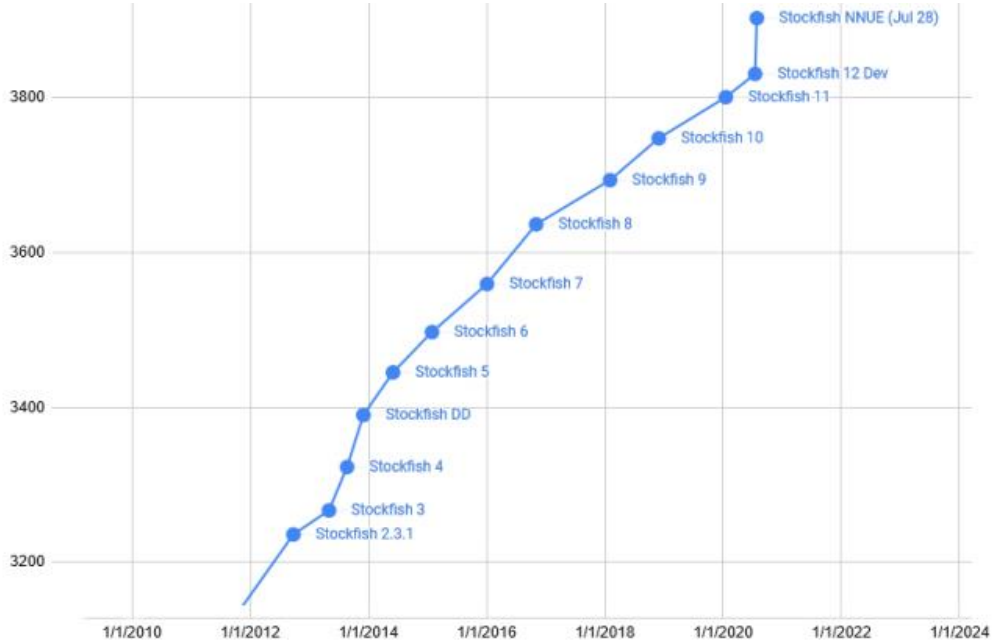

Figure 2.2: Stockfish Elo rating progress over time

Members of the community donate their computer resources in order to further improve on its capabilities. Changes that are proposed to the game flow are then carefully evaluated based on hundreds of thousands of chess games. The standard Stockfish application needs several input parameters in order to calculate the optimal move. The most important one is of course the board state. The board configuration has to be presented in a FEN notation. (Forsyth-Edwards Notation) In a single line of text separated into six different fields, the whole board is presented in a neat manner. The first field focuses on the positions of the pieces, the second one on the currently active player, the third one indicates the possibility of a castling move, and the fourth one the possibility of an attack in passing. The last two fields are game timers. The former is a half-move clock representing the number of moves from the last attack on a figure or the last movement of the pawn. This field is sometimes used in certain game modes to determine the draw state of the game. The latter is the full-move clock that counts the total number of moves of a game. The returned optimal move is written in the algebraic notation that describes which figure has to be placed where on the chessboard.

The second important input for the engine to work is the execution time specified in seconds or the depth of search represented by an integer that will determine how long will Stockfish search for the optimal move in the current game state. Stockfish can be used in various forms and shapes. Since it is written in the C++ programming language, it has to be run as an external process if we want to use it in our application. In the spirit of using Docker containers, the Stockfish executable will be running from a container as a web service whereby sending a simple GET request will grant us the information we need.

Figure 2.3: 4 lines of the starting FEN segment

2.3 UNITY

Unity is a well-known cross-platform game engine that enables the development of highly detailed virtual environments [3]. Even though its primary usage is intended for game development, the software is used in many different industries like film, architecture, and industrial applications. The primary reason why Unity is widely used is its flexibility and adaptability that allows the users to create very complex systems with simple to use, yet advanced tools that can be customized to the users' needs. More than half of current mobile games on the market are made within Unity as well as over 60% of VR and AR titles on several different platforms. Examples of very successful titles made in the Unity engine are shown in the image below. Ori and the Will of the Wisps on the left, and Escape from Tarkov on the right.

Figure 2.4: Example of a 2D/3D game made in Unity game engine

Its main competitor the Unreal engine is mostly used for larger projects targeted for the PC and gaming console markets. The tools it offers though more advanced are vastly more complex to use. Since our system does not need the high-end graphical visualization that Unreal offers, Unity with its emphasis on community collaboration and helpful learning systems imposes itself as a better overall choice [1].

Unity will be our core component in charge of hardware control and communication between other software sub-components. Another thing that our application will be using extensively is Unity's detailed physics simulation system based on the Nvidia PhysX engine. Although Unity is written in the C++ programming language, its primary scripting API is in an object-oriented language called C#. On top of that, the base class from which every unity script derives, that enables scripts to be attached to an object in game is called MonoBehaviour. It contains several key methods used throughout every script like Start, Update, and OnEnable. Every project consists of

one or more scenes where we can arrange our objects and develop their functionalities. These objects are known as GameObjects. Every GameObject is comprised of components that expand their basic behavior. Some of them, like the transform component which describes the position, rotation, and scale of a GameObject are mandatory. GameObjects with their components and specific configurations can also be saved as templates which are called Prefabs. Prefabs can then be used on multiple occasions within one project, or they can even be shared with other projects. There lies the power of Unity's collaborative online platform called Asset Store. Users can share their prefabs, packages, and scripts with other members of the community which can greatly accelerate development.

Figure 2.5: Unity 2019.4.14f1 LTS user interface

The main Unity window consists of several standard components [2]. The hierarchy window describes all instantiated objects within the current scene. The hierarchical relationship between them allows us to neatly organize and group objects similar by type, or by their function. The Project window offers us an overview of all currently used assets within our projects. A folder structure helps us to differentiate types of files that we use like 3D objects, sounds, materials, scripts, and textures. The scene window covers most of the user interface real estate. It shows our newly created two- or three-dimensional virtual environment. It also offers some basic tools for object manipulation mostly focused on the previously mentioned transform component. When the application is launched, within the Game window, we can see the view of our active camera within the virtual environment. The last, but

probably most important window for development is the inspector window. It shows us all the different components attached to the selected GameObject. There we can easily change exposed values and options and add newly created scripts to expand GameObjects capabilities.

2.4 OCELLUS

Ocellus is a next-generation vision system for robotics and machines developed by a Swedish-based company called Byte Motion AB [8]. It simplifies machine vision and robot programming allowing fast prototyping by using many advanced modules and functionalities. Ocellus is a powerful programming tool for connecting hardware devices, different APIs, robots, and services in new and advanced ways.

One of its unique features is the reconstruction of the real world inside a virtual one. Ocellus detects real-world objects using a range of different RGB and depth cameras. These objects are then placed in a virtual environment where they are synced with reality, through the use of sensors, vision, and physics systems. The user can then interact with the objects in the virtual world, for example with a simple pick-and-place command.

Figure 2.6: WEXOBOT waste management robot powered by Ocellus

Another unique feature of Ocellus compared to other robot and vision solutions is that all the robot calculations are done externally by Ocellus. Usually, internal robot controllers have to think about path planning, obstacle avoidance, conveyor tracking, and inverse kinematics. Now, Ocellus can do all these things without using robot software that tends to lag behind in terms of development and features. The Ocellus system is very complex and consists of several key components. The Ocellus Core is the main part of the system responsible for the backend logic and functions. In addition. it acts as a bridge for all the other hardware and software sub-components that Ocellus needs to function properly.

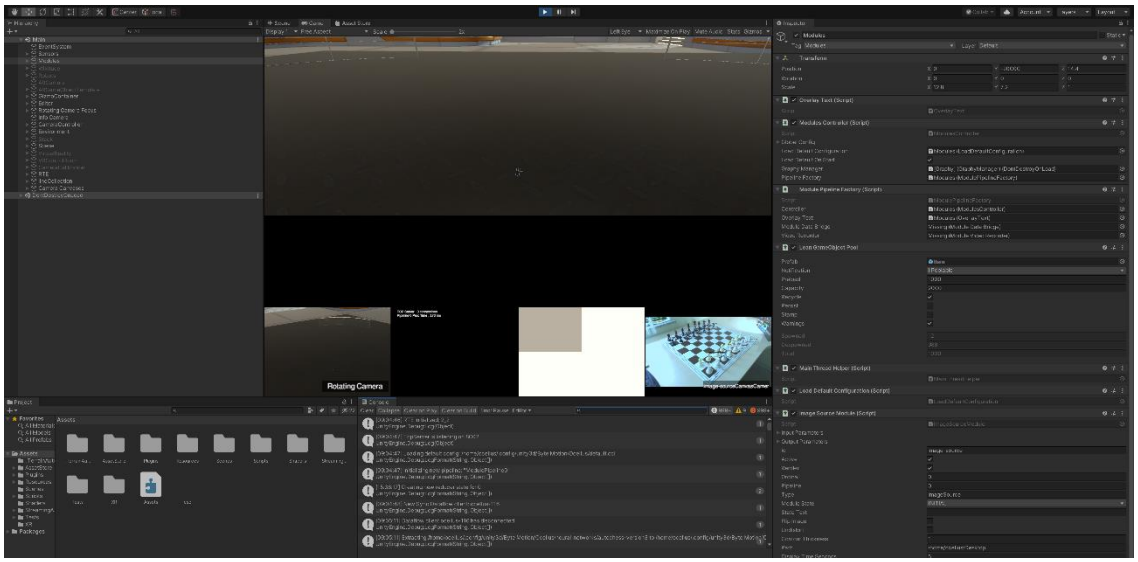

Figure 2.7: Ocellus running inside the Unity editor

The core is developed inside the Unity game engine, written in C# programming language with certain external C++ dynamic-link libraries integrated into its scripts. With a wide range of supported interfaces like gRPC, ABB RAPID, TCP/IP ASCII, and others, Ocellus can successfully communicate with other services and applications expanding its potential use. Unity's powerful physics engine is responsible for simulating the motions and reactions of objects as if they were in the real world. By using cameras and laser scanners in combination with advanced vision algorithms powered by artificial intelligence, it is possible to bring real-world objects into the physics engine in order to solve various problems. By using virtual and augmented reality, users can enter the virtual world of Ocellus to better understand how it sees the reality around him.

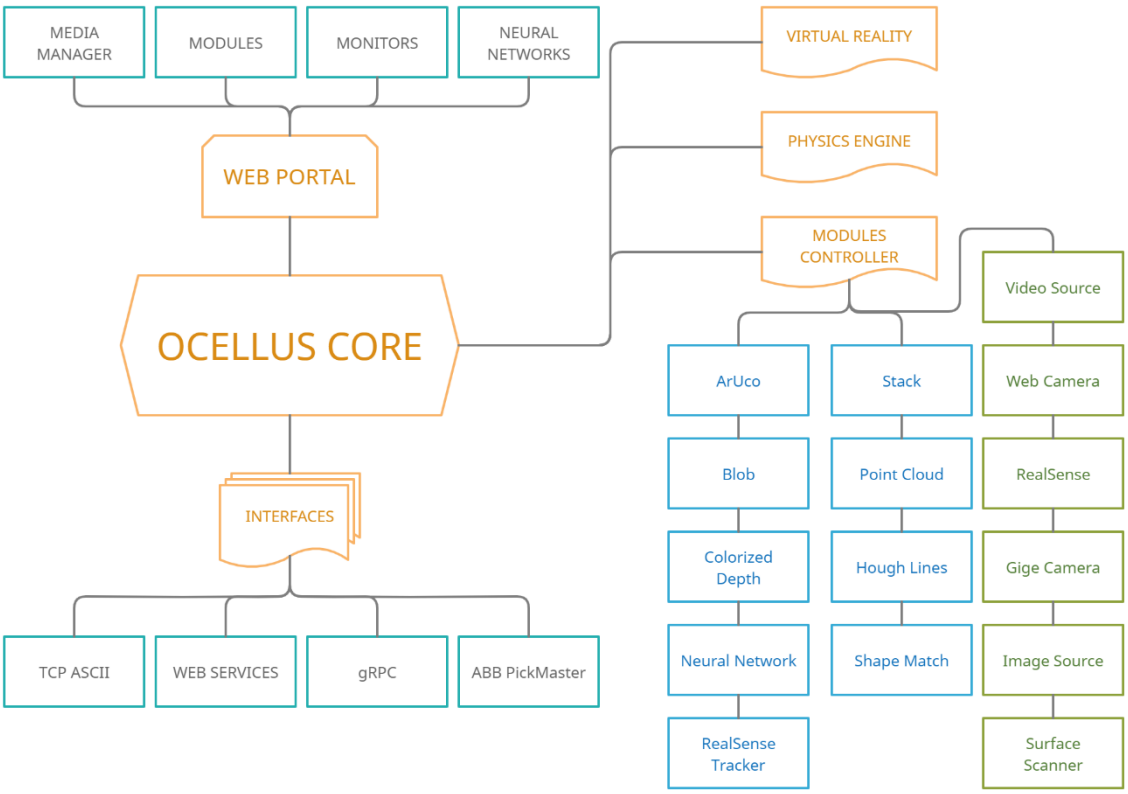

Figure 2.8: Ocellus high-level system diagram

Module controller is an integral part of the system, where different types of modules ranging from imaging sensors to neural network processors enable the user to build a system that can do various jobs. Ocellus offers 16 base modules that can be used in conjunction with custom modules developed for specific customers with more to come. Users can control these modules and other parts of the system through a powerful web interface. Modules can be added and removed, adjusted with a detailed configuration system that also measures the computer resources usage in real-time.

Cameras intrinsics can be automatically calibrated with a help of 2D and 3D markers. Within the monitor's tab, users can customize the arrangement of active cameras from different sensors within a deployed system. The media manager supports recording videos and screenshots from any available source. Collected media can later be organized and inspected to check the performance of the system. The neural networks manager enables the deployment of pre-trained deep learning neural networks through a specific module. Networks can be swapped and adjusted during runtime for faster testing and on-the-fly performance tweaking.

Ocellus will be the main application that we are going to use for machine vision in order to calculate the desired moving patterns that the YuMi robot will perform.

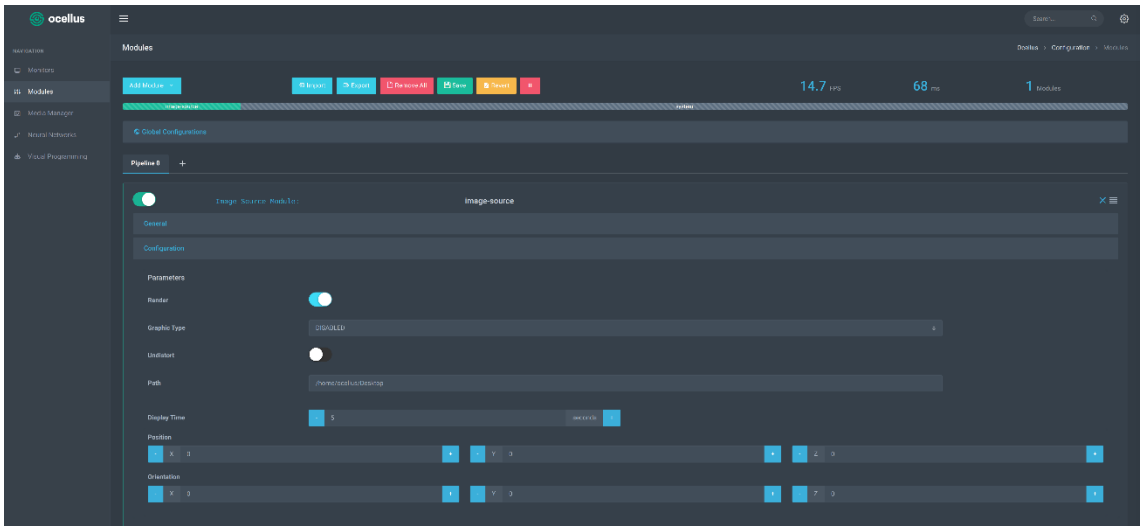

Figure 2.9: Ocellus web user interface modules tab

2.5 DOCKER

Docker is an open-source platform that allows users to create, deploy and run software within a package called containers [14]. Every container is loosely isolated from one another with its own set of libraries and dependencies which enables the application to work on every machine regardless of any specific settings configured on it. That way we can focus on writing code without worrying about the end system where the application will be deployed in the future. It works in a similar way to virtual machines, but unlike a virtual machine that creates a separate operating system, Docker allows applications to use the same base OS and its resources which significantly increases performance and reduces file sizes of applications. Because of its low overhead, we can easily run multiple containers simultaneously on a single machine or virtual environment.

Using Docker will help us to write, test, and share code quickly, which will reduce the delay between creating the application and running it in a production environment. Moreover, sharing containers between team members will help us

during the development process for specific tasks that are already automated and pre-configured further saving on development time.

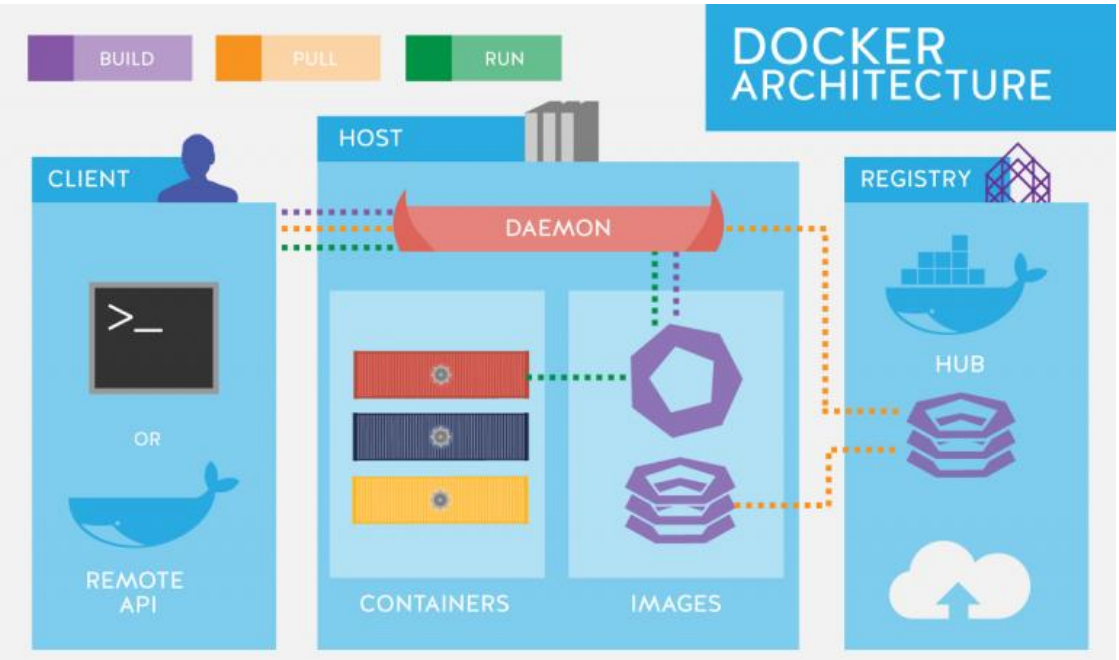

Figure 2.10: Docker high-level architectural view

Docker uses a client-server architecture where the client communicates with the Docker daemon, a persistent process that manages the creation, running, and distribution of containers. Both client and the daemon can run on the same or on a remote system. The client is the main point of interaction between users and containers. Via the provided commands, the client can send various requests over the Docker API to the daemon which will manage and control our collection of Docker objects. The two most important objects are images and containers. Images are read-only templates for the creation of new Docker containers. They can be created from scratch or based on another image with further user customization. Each instruction creates a new layer in the image that when changed, needs to be rebuilt. Other unchanged layers stay intact which improves performance compared to other available virtualization options. Containers, on the other hand, are runnable instances of images. When created, they are isolated from their host machine and

other containers. Even though containers are defined by images, there are multiple configuration options during the process of the creation of new containers.

The last important part of the Docker platform are registries. They are a repository

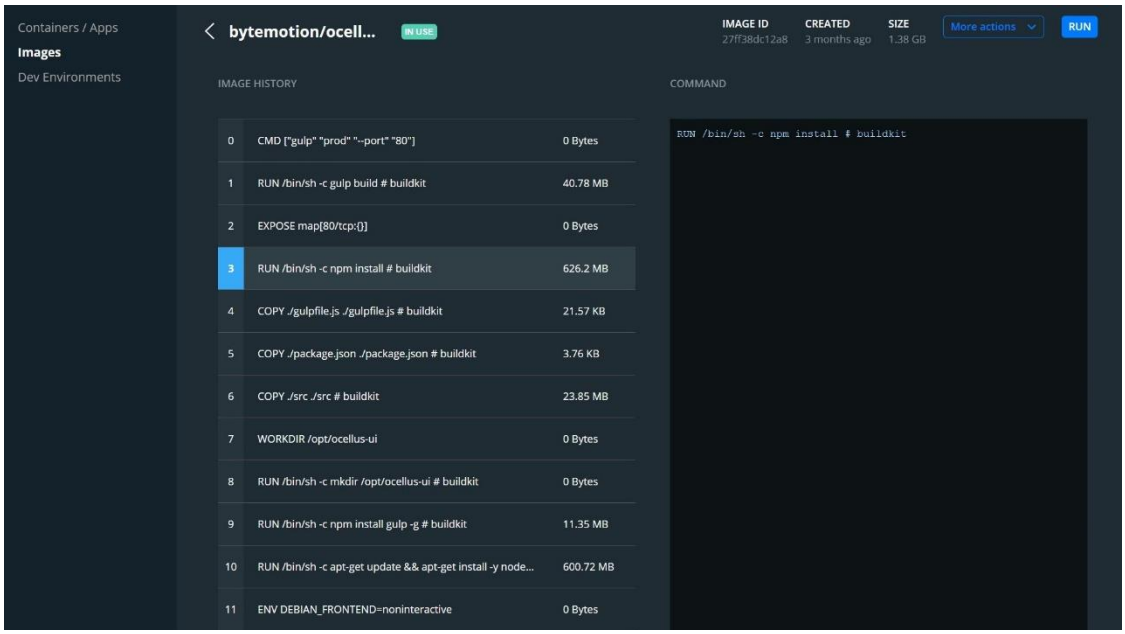

Figure 2.11: Docker image for the Ocellus web UI image

for images that can be stored publicly or privately. That way, sharing and collaboration between teams and colleagues are highly encouraged. The main public registries where Docker images are stored are the Docker hub and the Docker cloud. Every Docker user can access it by default and use all the publicly available images. For the normal operation of our system, we will use several pre-configured Docker images during our development process that are not on public registries but are available on Byte Motion's private repositories [24].

2.6 INTEL REALSENSE D435

Intel RealSense technology is a computer-vision oriented product line designed to enable depth perception capabilities for various machines and devices [4]. It is mostly used in fields that require depth perception and 3D reconstruction of the real world such as artificial intelligence, robotics, and automation. The whole ecosystem is supported with an open-source, cross-platform SDK that allows easier support and uses with third-party software. Every RealSense product is comprised of a vision processor, depth, and tracking modules as well as standard RGB and depth

cameras. For its relatively small entry price, it offers a complete solution for both hobby and industrial use.

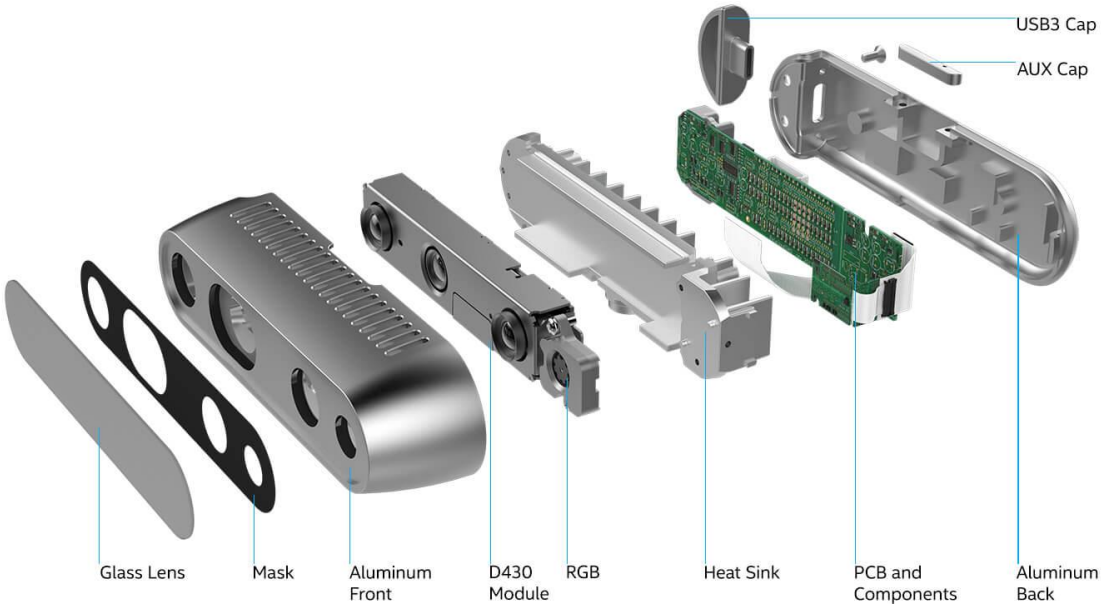

Figure 2.12: Intel RealSense D435 internal components

The model D435 that we are going to use is a part of the D400 family which is equipped with a newer D4 series vision processor based on a 28-nanometer process technology to compute real-time stereo depth data. It has a very wide field of view (FOV) of 87° * 58° and a maximum depth resolution of 1280*720, which will enable us to accurately track and calculate object positions in 3D space. Combined with a global shutter that helps in low light situations and a healthy maximum range of 10 meters, it is a great option for our vision system. When deployed on the field, its small footprint and mass help with the overall setup because it is easily mountable anywhere. In our use case, it will be mounted on a sturdy aluminium photography tripod with a 3-axis level head for precise angle adjustment.

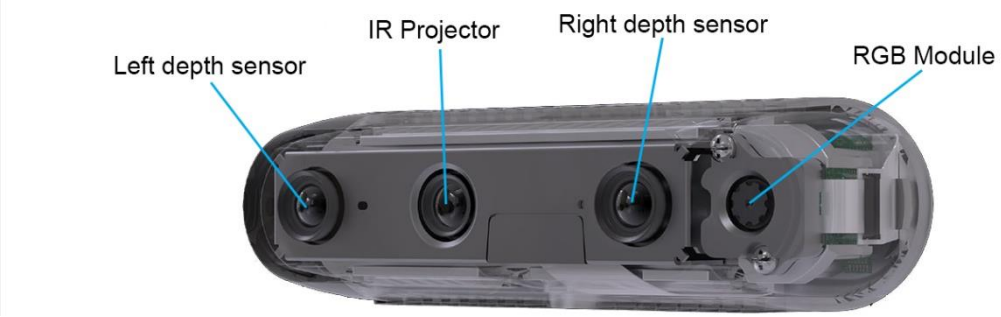

Figure 2.13: Intel RealSense D435 sensor array

To test the D435 camera, we can use Intel's RealSense viewer tool [5]. The tool offers a simple user interface where we can set both our stereo depth module parameters and our RGB camera module settings. Apart from the resolution and frame rate options, we will be using only a handful of its advanced functions such as exposure control and the use of some post-processing filters. When turned on, we can see the results of the depth module placed in an empty 3D environment visualized as a point cloud, or a 3D shaded mesh. If we turn the RGB camera at the same time, color data will be mapped with a vertex color method on the point cloud or on the mesh itself. The available exposure parameter will be important later down the line when we will use the camera to capture images for our dataset and of course in the final product when we will gather object information and use it in our system. The usage of post-processing filters will be explained in detail in the Ocellus Web UI chapter.

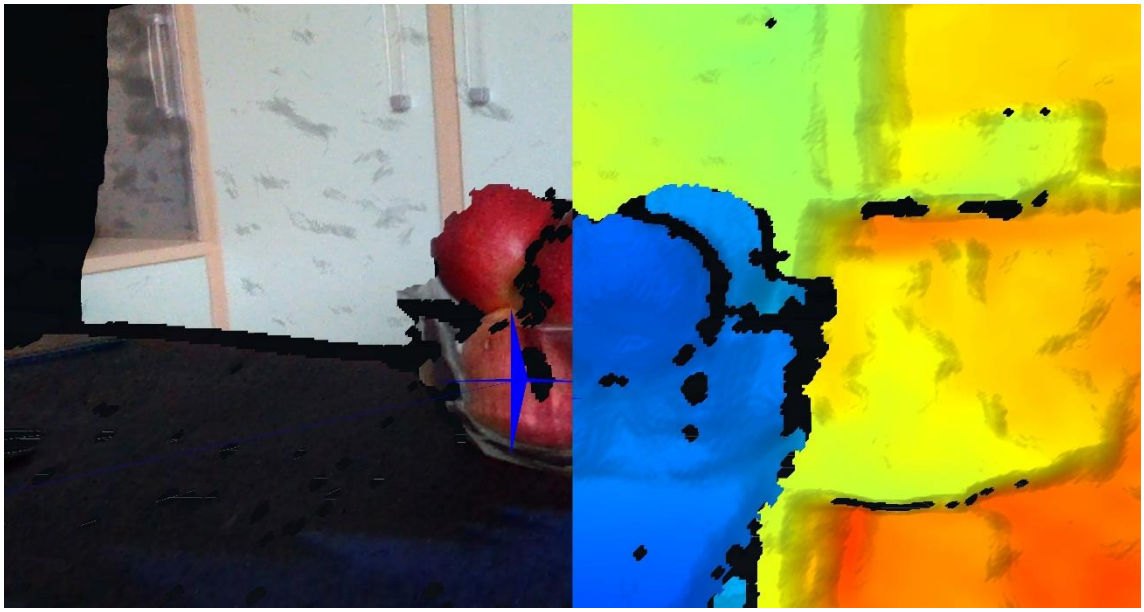

Figure 2.14: Intel RealSense Viewer application, left side is RGB data while the right side represents the depth information

2.7 SUPERVISELY

Supervisely is an online platform for computer vision development that has tools for creating and managing datasets and neural networks [15]. It has many collaborative elements that enable scaling for larger teams where jobs can be outsourced to other team members or even people outside the main team. Datasets in this work will not be extremely large, so it is possible for one person to complete the work on it. One

of the core concepts of this platform is data organization. Working with a large number of files can be really hard if they are not structured properly with a clear sense of hierarchy. To make things easier, first, we need to register a new account on the website.

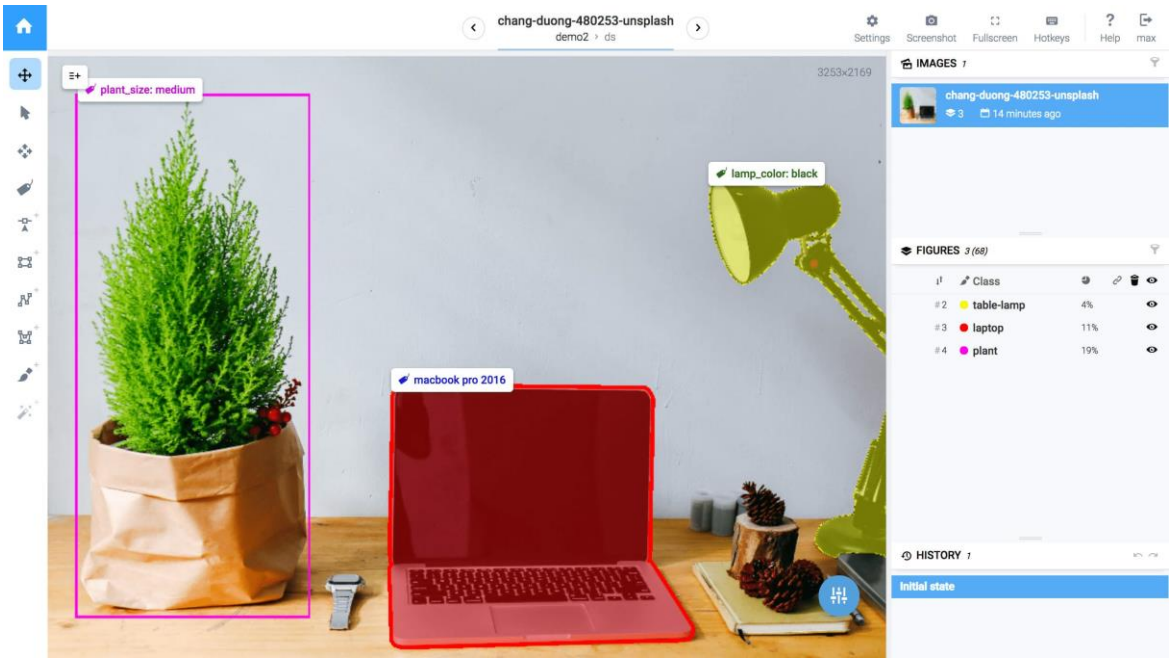

Figure 2.15: Supervisely annotation web user interface

After registering to the platform, we are greeted with a start page for creating new projects. Inside a project, we can import our images structured as datasets. Uploaded images are then loaded in Supervisely's powerful annotation tool, where we can select objects with some of the provided tools. Objects can be selected with points, rectangles, and even user-generated polygons. The latter option is the most useful one since our objects have varied contour shapes and sizes. On the right side, we have the objects tab where we can define what types of objects we will be searching for. The annotation process though very simple takes a very long time to do for a larger dataset. The precision of our marked objects will directly affect the result of object recognition in our neural network. That means we need to put in a significant amount of effort for the dataset to be correctly annotated. Tags can be used to add extra information to every annotation. For example, we can add comments, remarks, and to-do messages for each annotated object. Even though annotating dozens of images can be very time consuming, training a model with only a few hundred images in the dataset will not result in a very efficient neural network.

Figure 2.16: Supervisely DTL code example with computational graph representation on the right

That is where Supervisely's data transformation language (DTL) comes into play. This specially designed language allows us to automate data manipulation from image augmentation to format conversion. The process flow is defined in a JSON based configuration file, where inside of an array, each object represents one transformation. A sequence visualizer is displayed on the right side of the page when writing a new graph. Each transformation object consists of an action that needs to have a source and a destination. Every action has its own additional settings where we can tweak its effect on our data. A few examples of available transformations are cropping, flipping, multiplying, rasterizing and many others with varied end results. When the data augmentation finishes, we can download our dataset in a chosen format directly to our local machine.

2.8 MACHINE VISION

Machine learning is a method of analytical model building though data analysis, where a computer algorithm automatically, through experience, learns and identifies patterns that lead to decision making with minimal human involvement. It is a subset of artificial intelligence that trains a machine how to learn from a set of training data, in order to make calculated decisions without being explicitly programmed to do so. Another smaller subset of that category is computer vision. It is a scientific field that trains computers to interpret and understand information coming from visual media to perform tasks where the human vision is involved. The data that is observed can come in multiple forms such as images, video sequences,

multi camera views or even multi-dimensional data from depth cameras. The end goal is to automatically extract useful information from a visual source that can later be used in an application or service.

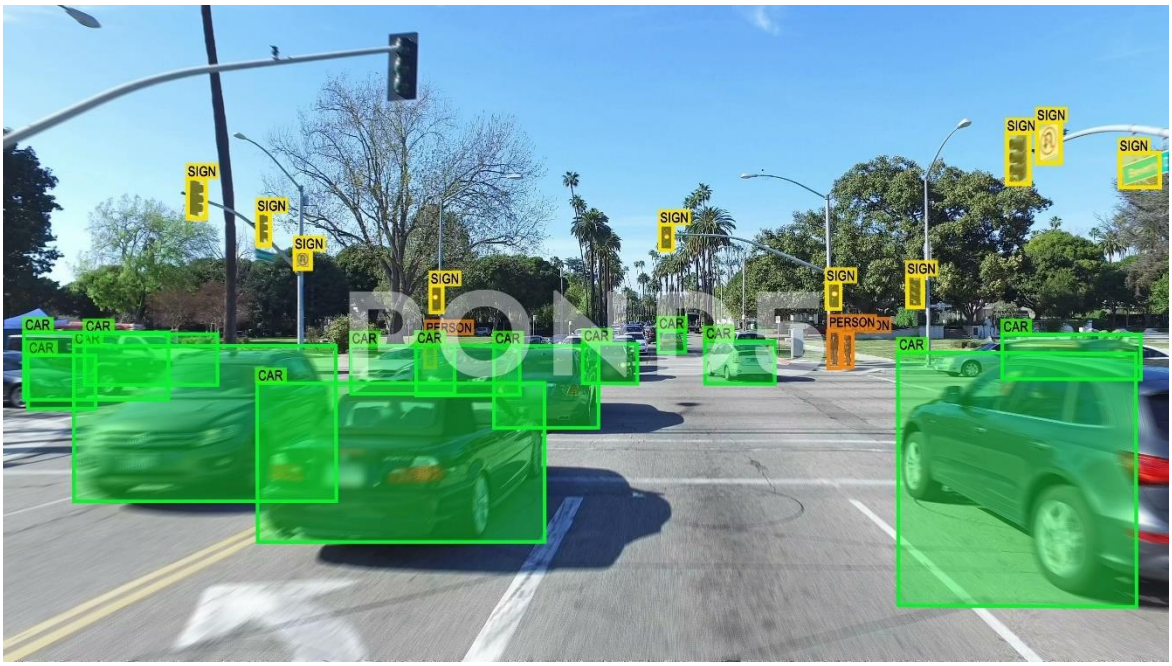

Figure 2.17: Example of object recognition powered by machine vision for autonomous vehicles

This technology has a very broad range of applications in medicine, military, autonomous vehicles, and industrial manufacturing. The standard challenge that computer vision has to overcome is determining whether or not the image data contains some specific object or feature that is expected to be recognized by the algorithm. Here, we are primarily focused on object recognition of pre-specified objects from 2D images. Currently, the fastest algorithms for tasks like that are based on convolutional neural networks. If we use more precise nomenclature, we are dealing with machine vision, an image-based automatic inspection and analysis usually for industrial applications [16]. Apart from object detection, machine vision systems are used for other purposes. Some of them are product and component assembly and packaging, inspection and flaw detection, label identification and processing, position calculation and object measurement, object counting, and sorting. Our application of choice is image-based robot guidance. Machine vision will provide location and orientation information to a robot to enable precise interaction with complex objects. Specific implementations like these require unique

solutions. However, there are some typical functions that are commonly found in machine vision systems.

The first step is always going to be image acquisition. Depending on the type of camera used, the resulting image data can range from classic 2D images to various other physical measurements such as depth. Light plays a vital role here since the pixel values typically describe light intensity in a specific spectral waveband. Raw image data from sensors after that goes through a pre-processing pipeline where certain corrections to the image are applied. Some of them are contrast adjustments, noise reduction, color temperature correction, and lens distortion removal. The image is now ready for feature extraction, where things such as

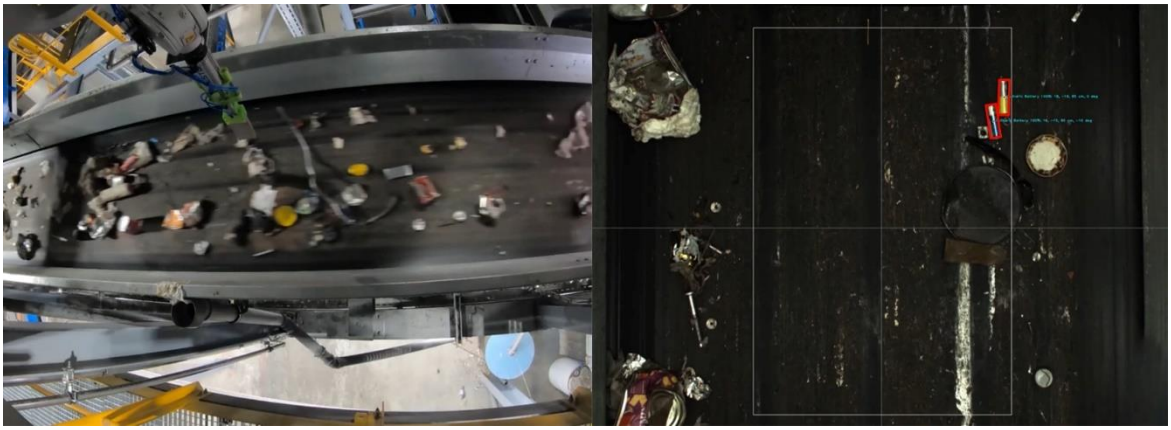

Figure 2.18: Wexobot waste sorting robot on the left, RGB sensor data with detected battery objects on the right

localized lines, edges, and other interest points are detected from the image. Detected interest points are then filtered and segmented into groups that potentially contain targeted objects. Every potential target is evaluated by the assumptions we created at the beginning like object shape and size. At the end of this process, the final decision is required from the algorithm, where the possible target objects are flagged as a positive, or a negative result, with varying degrees of certainty.

2.9 Region-based Convolutional Neural Networks

Deep learning is a subset of machine learning methods that relies on artificial neural networks for its model training. Artificial neural networks are inspired by the information processing found in biological systems. The original idea was to find an algorithm that can solve problems the same way a human brain would. Even though

they share many similarities, neural networks tend to be static and symbolic, which is opposite from most living organisms that have a dynamic and analog neural system. There are many architectural types of neural networks such as graph neural networks, recurrent neural networks, and convolutional neural networks. Networks are organized by layers through which the data is passed and transformed. Each level tries to learn transformations on its input data in order to create a more abstract representation as the layer output.

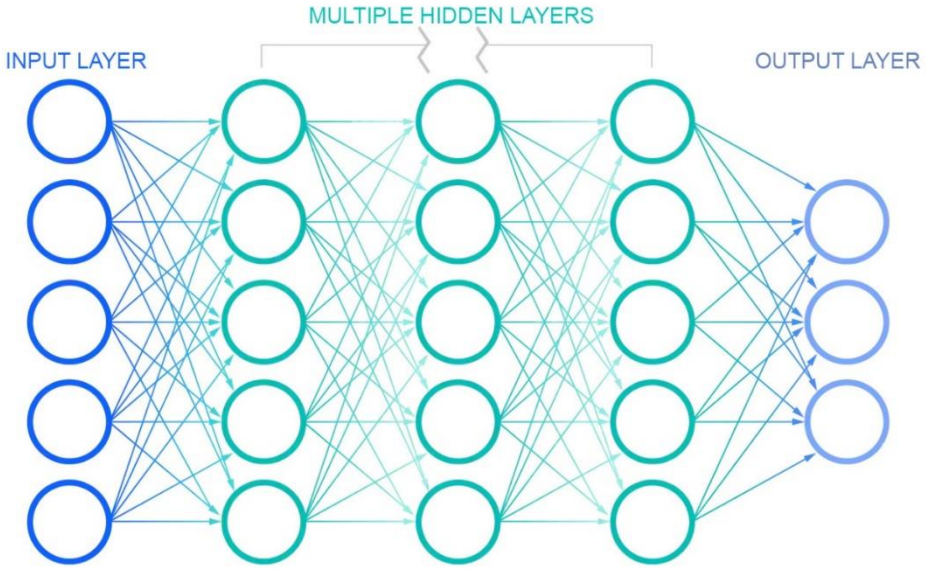

Figure 2.19: Basic deep neural network architecture with 3 standard layer types

During the recognition process on a 2D image, the first layers will deal with matrices of pixels, that by passing through layers will become edges and splines. These will eventually become shapes and objects that we were searching for in the first place. The process learns by itself what feature to place on each layer, and the user can then customize the layer number and size to achieve different levels of abstraction. Each layer is comprised of neurons that can transmit signals to others over a connection called a synapse. Synapses can have weights that influence the strength of the signals sent to neurons, which usually have a state represented by a real number. Deep neural networks have multiple layers between the input and the output layer. They can model very complex non-linear relationships which are exactly why they are used in computer vision.

Previously mentioned convolutional neural networks are today's most commonly used tool for image analysis. Their name comes from the mathematical operation called convolution, which replaces traditionally used matrix multiplication in one of their layers. R-CNN networks, however, are region-based versions of classical convolutional neural networks that focus on object detection inside of a region of

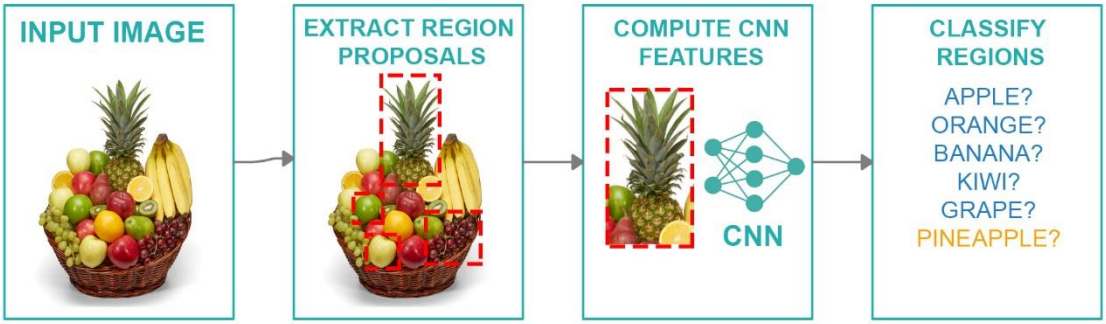

Figure 2.20: Region based convolutional neural networks high-level diagram

interest. At the start, a search is performed on a set of proposed regions inside the input images. All proposed regions generally have different scales, shapes, and sizes. Moreover, the category and ground-truth bounding box of each proposed region is labeled. Then a pre-trained CNN is selected and placed before the output layer. It will transform the proposed region's input dimensions required by the network and use a forward computation to output the features from the proposed region. These features are then used in conjunction with the labeled category of the region for object classification. Features and labeled bounding boxes for each region are used to train a linear regression model for ground-truth bounding box prediction. The main drawback of R-CNN networks is that they are very slow to run. Because of that faster implementations are mostly used in real-world applications [16].

Fast R-CNN networks pass the original image only once to the CNN and then extract its feature maps. After that, a selective search algorithm is used to generate predictions based on those maps. Using these maps, regions of proposals are extracted and reshaped by the region-of-interest pooling layer. Proposed regions are all transformed into a fixed size, so they can be fed into a fully connected network. Instead of using 3 different models, Fast R-CNN networks use a single one that extracts features from regions, divides them into classes, and returns the boundary boxes for the identified classes simultaneously. Faster R-CNN networks

take this concept even further and use region proposal networks for their region of interest generation. Each object in the generated set has an objectness score as output [18].

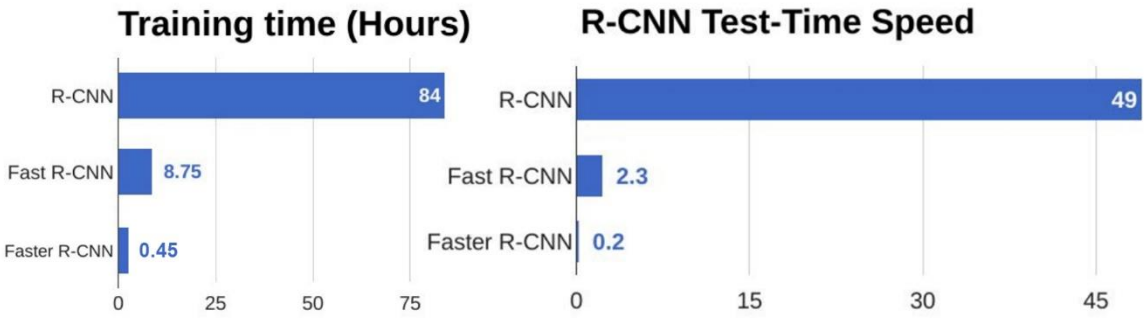

Figure 2.21: Comparison between different R-CNN implementations in training time and test time speed

Finally, we come to the Mask R-CNN networks which we are going to use in our model training. They are built on top of Faster R-CNNs and in addition to the class label and bounding box, they return the object's mask. It is very computationally expensive to search for object contours, but the exact shape of the object is critical for determining the 3D position of the object in question.

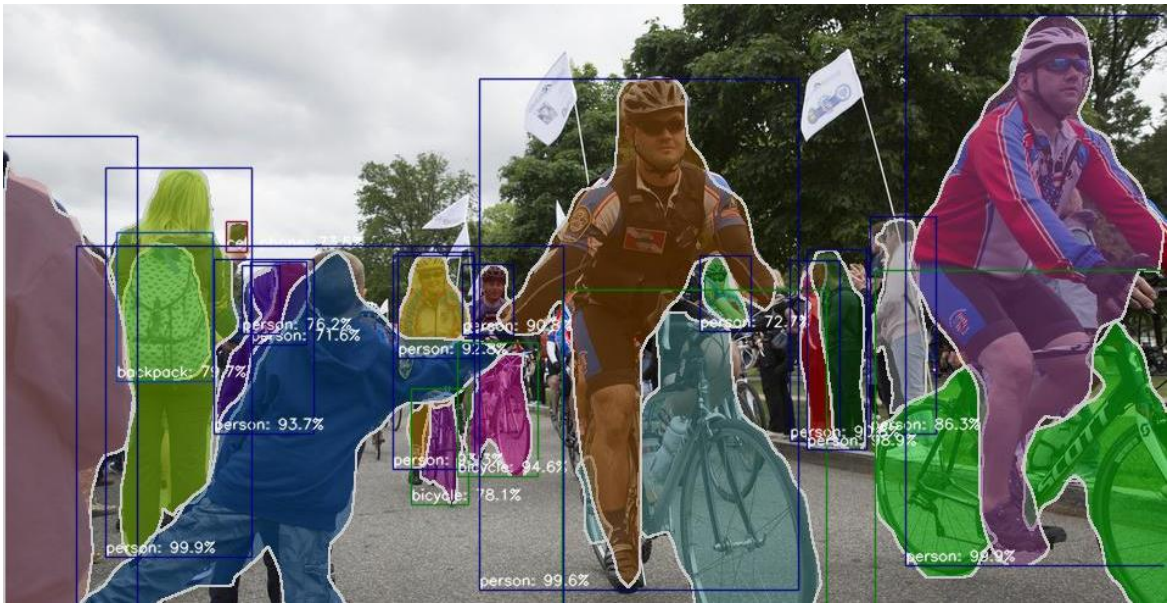

Figure 2.22: Mask R-CNN applied on a bike race image, classes and contours are displayed

2.10 Feature pyramid networks

Detecting objects, and especially their contours is a challenging task. There are several ideas on how we can approach that problem, and the use of a pyramid structure is one of them. We can create a pyramid of the same images at different resolutions to detect objects on them. Unfortunately, that is a very time and memory-consuming task. On the other hand, we can create a pyramid of features and then use them to detect objects. This too has its downside since feature maps closer to the image layer composed of low-level features are not effective enough for object detection. If we combine these two ideas, we get a feature pyramid network [19].

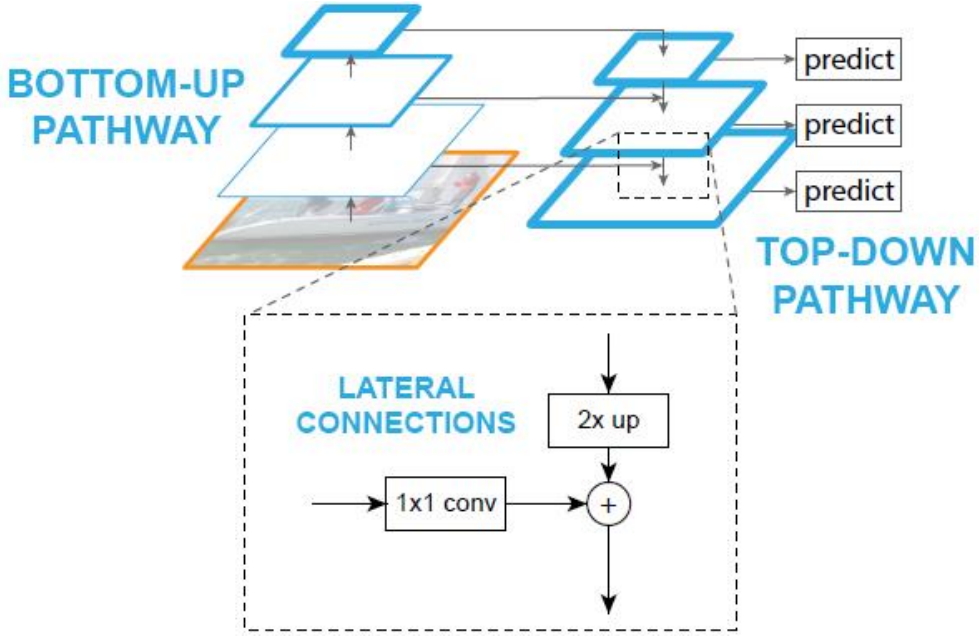

Figure 2.23: Feature pyramid network schematic view

FPN (Feature pyramid network) is a feature extractor designed to work with object detectors. It replaces traditional feature extractors commonly found in Faster R-CNN implementations with accuracy and speed as the main benefits. FPN comprises of a bottom-up and a top-down pathway. The bottom-up pathway is usually a convolutional network for feature extraction. As we travel to the top of the pyramid, the spatial resolution decreases, and more high-level features are detected. We can say that the lower the resolution, the higher the semantic value of the detected features are. The other pathway is the top-down one used to construct higher resolution layers from semantically rich layers. Even though the

reconstructed layers are semantically strong, the precise locations of objects are not known because of the down-sampling and up-sampling processes. Because of that, we use lateral connections between reconstructed layers and the corresponding feature maps to help the detector in finding the locations of the features. The detected feature maps sorted in the pyramid are then later fed into an object detector. This significantly accelerates our demanding process of searching object locations as well as finding out the exact object contours inside our images.

2.11 PYTORCH

PyTorch is an open-source machine learning library, based on the Python programming language, that uses the power of modern graphics processors for its calculations [20]. It is one of the more commonly used deep learning platforms along with Google's TensorFlow. It is built to provide fast and flexible development in an ever-growing active community. Unlike TensorFlow that is better for production models and teams that need good scalability, PyTorch offers an easier-to-learn workflow for smaller projects and prototypes.

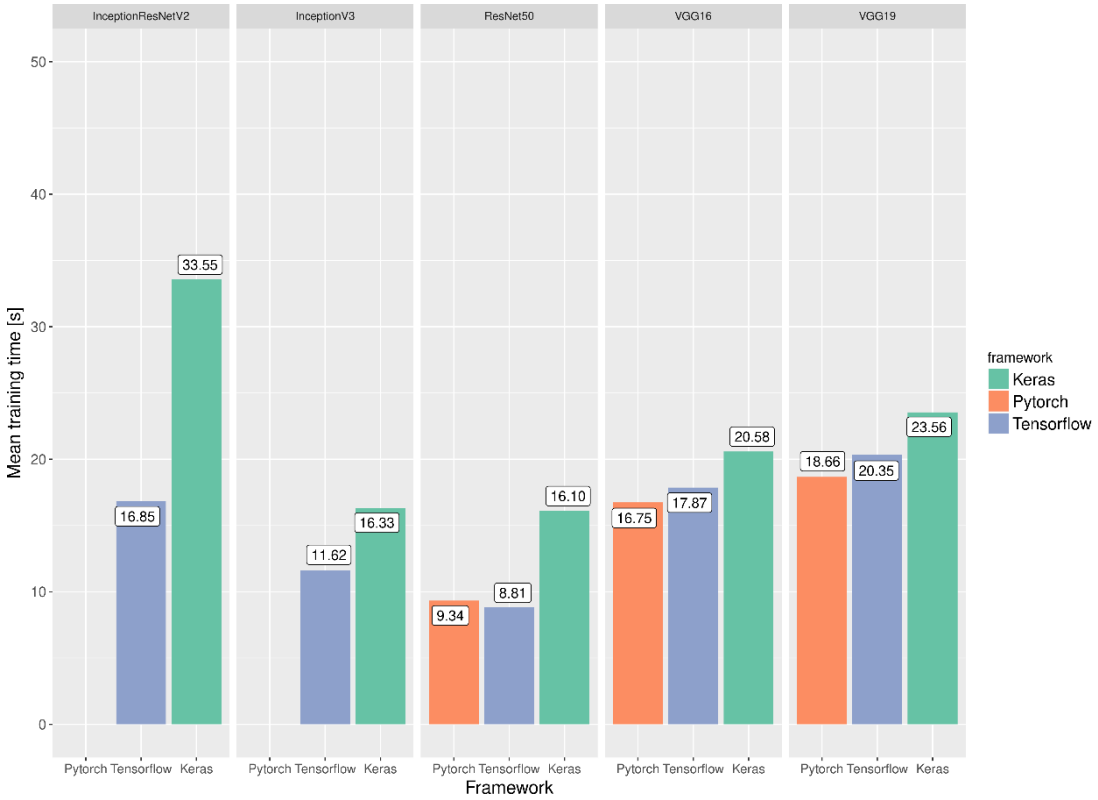

Figure 2.24: Training time comparison between different neural network frameworks

It is frequently used for applications such as natural language processing, self-driving vehicles, and computer vision which is exactly what we need it for. A large number of renowned companies used it in the development of their software like Tesla, Uber, and HuggingFace. Compared to other libraries like Numpy, it is an order of magnitude faster thanks to the strong GPU support baked into the library itself.

It provides two important high-level features, tensor computations with GPU acceleration support and the ability to build deep neural networks on a type-based reverse-mode automatic differentiation system. PyTorch tensors are a data structure that is able to store and operate on homogeneous multidimensional rectangular arrays. Due to its similarities with other data structures, they can be operated on Nvidia's graphics processors with CUDA technology which vastly accelerated the calculations. Automatic differentiation on the other hand is a method wherein order to compute the gradients, operations that have to be performed are recorded and replayed backward. This further saves time when working with neural networks especially during the forward pass of our R-CNN type neural networks.

We will be running a slightly modified PyTorch configuration in a separate Docker container because of the number of dependencies that the user has to have. The environment of a container will provide hassle-free functionality when using different machines. The container will be running locally, and the GPU acceleration through CUDA will be enabled. To track the development of new models with PyTorch, we will need some kind of a ledger where our results and configuration details will be displayed. Jupyter Notebook is a web-based computational environment that enables the use of multiple programming languages including Python. A notebook file is a JSON document containing a list of cells that have code in it. Inside of it, we will determine the configuration of our PyTorch-based neural networks. All of the results and logs will then be saved to the notebook, which is easily accessible later for deeper examination.

2.12 Node-RED

Node-RED is a JavaScript-based visual programming tool developed for wiring together hardware devices and software services in the Internet of Things space. Inside a web browser, Node-RED provides a flow editor which can be used to make JavaScript functions. Flows are made out of different types of nodes that all have different purposes. Its runtime is based on Node.js, which is a back-end JavaScript runtime environment that executes JavaScript code outside of a web browser. Because of it, Node-RED is taking full advantage of its event-driven, non-blocking model. The light-weight runtime allows it to run on low-cost hardware like Raspberry Pi where resources are limited [21].

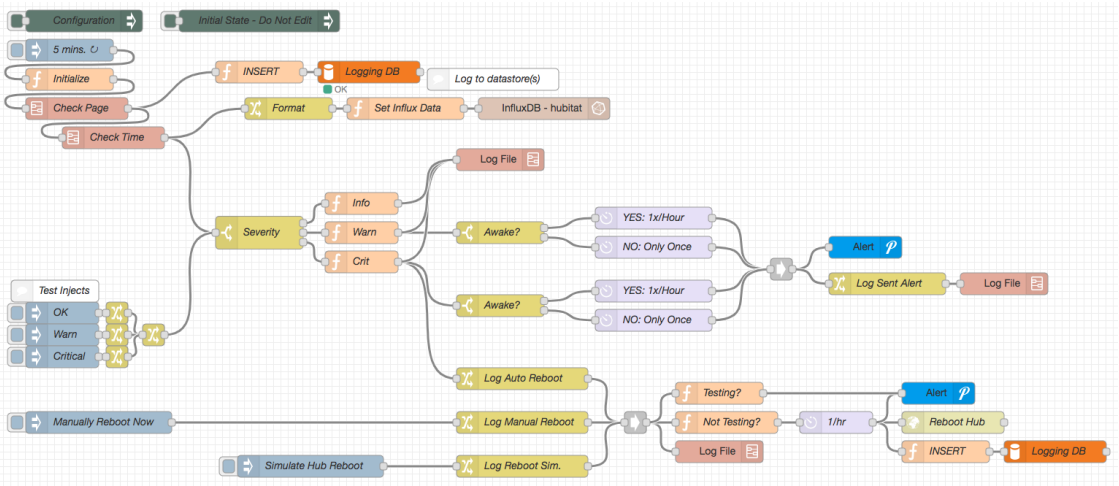

Figure 2.25: Example of a Node-RED flow for a home automation system

The strength of the Node-RED platform is in its focus on social development. Flows and nodes can be imported and exported in JSON format, and then easily shared with others on the growing online library. Currently, there are more than 225 thousand different types of modules in the online library, making the first steps for newcomers much easier. Node-RED can be run on a broad range of devices, from large cloud-type computers to small and affordable microcontrollers. We will be running Node-RED on our local machine and use it to bridge the communicational gap between our main application and the YuMi robot. Current software used for communication between ABB robots is complicated and inefficient. Byte Motion AB is trying to revolutionize inter-robot communication by using Node-RED for transmitting data between different machines. The logic behind that is that if it is good enough for a 5-dollar microcontroller, it should be enough for a robot costing tens of thousands of dollars. The system is still in early development, so there are

still a lot of things that are missing from the final version. Fortunately, there are already custom nodes that enable the communication between our main application Ocellus, and ABB's robots. By using these proprietary nodes, we will be able to send data acquired from our depth camera to YuMi, which will use it to do pick-and-place operations.

2.13 ABB IRB 14000 - YuMi

For many years, there was a need for a faster assembly of high-volume products in the consumer electronics segment. Product life cycles are getting shorter and a growing trend of user customization near the end of the production process is present. In 2015., ABB corporation specialized in robotics and automation introduced IRB 14000, also known as YuMi, as the world's first truly collaborative robot which will tackle this market segment [22]. A cobot, or collaborative robot, is a robot that is intended for a direct human to robot interaction within a shared space. Traditional industrial robots are mostly isolated from human contact with barriers, cages, and safe zones because of safety precautions. In order to meet safety standards when working with humans, cobots have to rely on lightweight materials, rounded edges, and limited speed and force parameters. Their usage varies from robots in public spaces, logistics robots that transport objects, or like in our case an industrial robot that helps to automate a process that is hard for people to do.

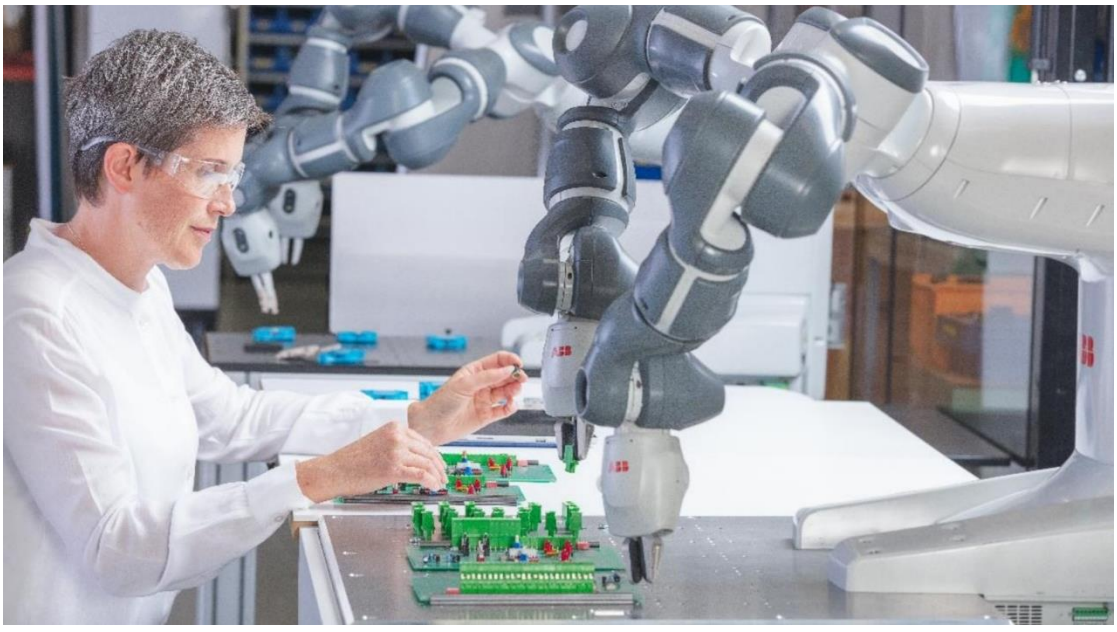

Figure 2.26: IRB 14 000 collaborating with a human on electronics assembly

YuMi is a dual-arm robot designed to be used in a small parts assembly environment. Its name comes from the words “you” and “me,” signifying that human collaboration is one of its key features. It is made out of a rigid magnesium skeleton covered with a plastic casing wrapped in a soft padding material that absorbs any potential impacts. It weighs approximately 38 kilograms, and its grippers can pick up objects that are up to 500 grams in weight. When YuMi senses an incidental impact, it can pause its motion in a millisecond, and restart it after pressing play on the remote control. In addition, its arms are designed so that the robot has no pinch points, thus preventing any harm to sensitive objects while the arm axes are moving. Each YuMi arm has 7 separate axes to work with, opening up a lot of possibilities for motion and positioning.

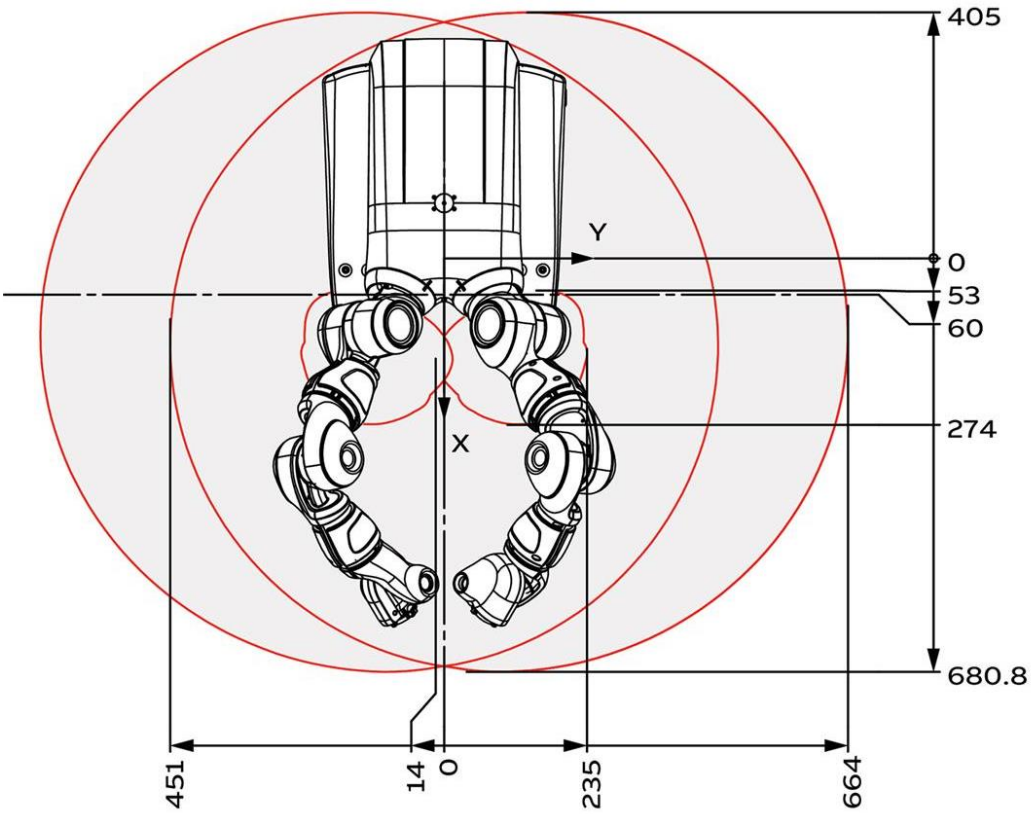

Figure 2.27: IRB 14 000 working range in millimeters top view

When designing a robot, making a very fast one, or a very precise one is somewhat easy, but making one that does both of these things is a challenging task. That is exactly what ABB is trying to achieve with its robot lineup, and YuMi is no exception. The whole YuMi family is also designed to be fairly easy to set up and use. That helps people, who are not specially trained to operate a robot, to easily work on, or with YuMi. Another important design goal for YuMi was a vision-guided assembly

which is exactly the task for which we will use it. Cameras embedded in the grippers in combination with an external depth camera will guide YuMi's movement in 3D space to complete simple pick-and-place tasks guided by the power of artificial intelligence.

Figure 2.28: Official ABB gripper lineup for IRB 14 000

2.14 ABB RMQ protocol

ABB's RMQ is a communication protocol that is built into the Multitask and PC interface of all ABB robots. It is an old and lower level than the standard RAPID protocol that allows messages to be sent from one queue to another [25]. Messages can contain up to 3000 bytes of information and have several components. A source queue that specifies where replies need to be sent, a label designating what data type is being sent, an ASCII coded body of the message containing useful information, and a UserDef parameter which is a 15-bit number containing meta-data about the message itself. A queue on the other hand is a user that wants to receive messages. Each user can be a server, client, terminal, consumer, or producer of messages. A queue exists for each RAPID task and has to be manually created for other users that want to communicate with each other. Queues have a maximum allowed number of messages as well as a maximum size of one message. Even though the maximum message size is 3000 bytes, the web service imposes a 444-byte limit. The queue is cleared each time the robot controller restarts, so we need to create it again in that case. After we create it, we can refer to it in web services and in RAPID. To properly configure the queue for the robot tasks, we need to manually adjust the configuration file to include some RMQ parameters like name, type, mode, maximum size, and a maximum number of messages.

The before-mentioned UserDef parameter consists of two main sections. The two highest bits are reserved for specifying the type of instruction, while the lower 13

bits are used for tracking and identifying messages and their replies. There are 4 possible instruction types. The EVENT instruction triggers an event and doesn't care who or if anyone is listening, EXECUTE runs a specific segment of code, ALLOCATE allows dynamic allocation of large data structures, while the RESPONSE is a response from one of the earlier instructions. We are mostly going to use EXECUTE commands since we need to implement specific behavior in RobotStudio for our Yumi robot.

When the robot receives an EXECUTE message of type COMMAND, it will try to execute a pre-defined code segment and if successful, it will return the same message as the RESPONSE message with the same ID. If the robot on the other hand fails to execute the procedure, it will return an error information data type that describes why the procedure failed.

This is an example of the command data type that we will send from Node-RED to RobotStudio:

```
[["","",0,"","",0,""],"pick_place",["pose","",0,"[[0,0,0],[1,0,0,0]]","",0,""],["pose","",0
,"[[0,0,0],[1,0,0,0]]","",0,""],["", "", 0,"","",0,""],["","",0,"","",0,""],["","",0,"","",0,""]];
```

The command consists of an object that the procedure should be executed on by the robot controller. There are up to 5 arguments that the procedure can be executed with. If the arguments are not used, they can be set to null. The text marker in orange is the command name, text in green is an example of a pose argument, with a specified position and rotation. The text in blue is an empty argument not used by the procedure.

```
PROC pick_place(pose pickPoint, pose placePoint)
    !Code example
ENDPROC
```

The code segment above is an example of how to construct the "pick_place" procedure inside RAPID. The Robot Norway RAPID team has determined that the

RMQ protocol is the preferred communication protocol for communicating internally and externally with asynchronous programs. For that reason, we are going to use it in our AutoCHESS application to send data from our Node-RED flow.

2.15 ABB RobotStudio

RobotStudio software is an offline programming and simulation tool that allows robot programming without the use of real-life robots [25]. It allows the user to develop and test robot software without shutting down the already deployed robotic hardware in order to perform training and testing. The whole tool is built upon ABB's virtual controller. The controller is an exact copy of the real software that runs on robots. The simulation that it performs is very realistic and uses the same configuration files as real robots. Apart from that, it offers many advanced features for collaboration over virtual meetings, virtual commissioning, digital twin creation, augmented reality, and many more. The use of RobotStudio greatly reduces the return on investment and risk for newly created systems. It enhances the start-up process as well as the overall productivity. Users can add different robots, 3D models, and objects to a station, and replicate the real-world situation where the robot will be used. The simulation is very precise, meaning we can expect the exact behavior in a real environment. In addition, there are many tools for creating targets, path plans, and joint configurations. During the simulation, there are different signal analyzers, stopwatches, and I/O logs that can later be examined in order to improve the setup. Apart from the manual creation of virtual objects in robot stations, the other main component of RobotStudio is its programming suite.

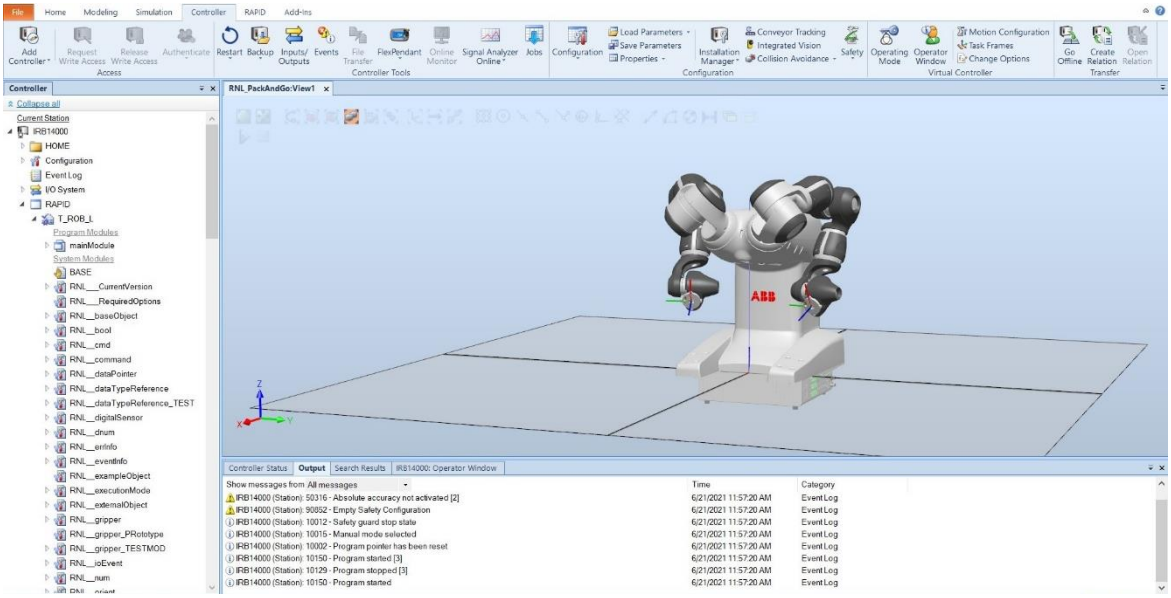

Figure 2.29: RobotStudio main interface

As with all ABB's robots, programming is done in RAPID, a high-level proprietary programming language exclusive to ABB. It was introduced in the 90'ies superseding the previously used ARLA language. RAPID offers many features such as procedures, functions, interrupts and error handling, modular programs, and standard arithmetic and logical expressions. RAPID programs can be run within the virtual controllers in RobotStudio, or after testing, on a real-world robot. The built-in text editor has IntelliSense built-in, meaning code recommendations and syntax error checking can be seen in real-time. RAPID offers a high degree of flexibility when designing a robot system, and that is exactly what this project needs.

3. IMPLEMENTATION

Our AutoCHESS application uses many different components and technologies, so the implementation will be executed in several phases. The first phase will consist of data gathering and dataset building. For the final system to work accurately and reliably, this phase has to be done with a high degree of precision. The second part of the implementation revolves around the training of the deep learning neural network to recognize chess figure objects. Since the training of neural networks takes a significant amount of time, we have to be careful with our configuration in order to repeat the training process the least number of times while still keeping the necessary precision and consistency our system needs. The third part of the implementation will mostly revolve around the use of the Node-RED-based interconnect between Ocellus and our YuMi robot. Here, we will have to construct a flow that will take the information processed by Ocellus and transform that information into pick-and-place actions that YuMi will perform. The last phase of our system will focus mostly on the configuration and use of ABB's YuMi robot. This phase will reflect and indicate any past mistakes and errors that might happen during development.

Finding those mistakes will be quite challenging because of the large number of interconnected components and services. The Ocellus vision system will play a vital role throughout the whole implementation process. Its functions will be extensively used during most phases of the development.

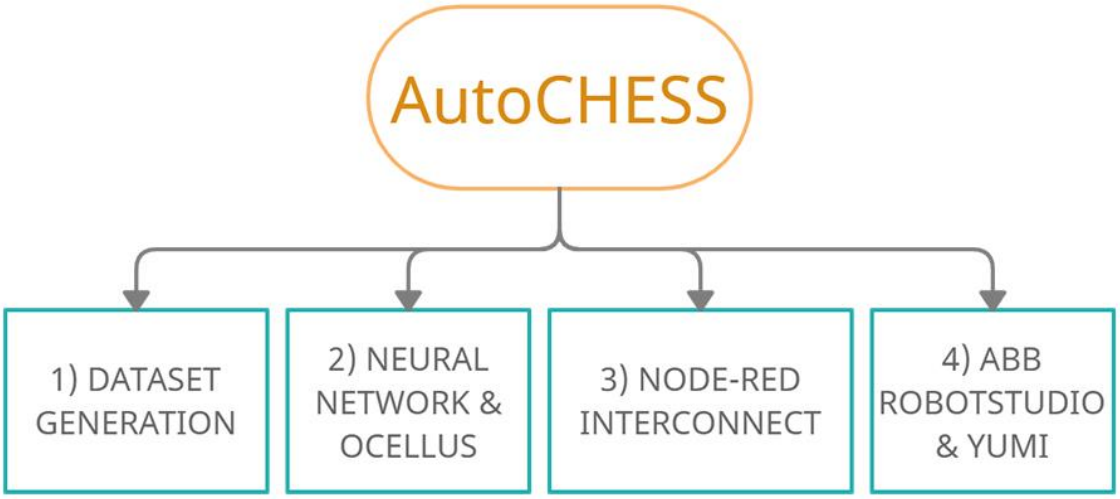

Figure 3.1: 4 implementational phases of AutoCHESS application

3.1 DATA GATHERING

Data gathering is the first part of our journey in the development of AutoCHESS. We will be making some datasets for proof-of-work tests to be sure that our envisioned system can function as expected. Pictures will be taken with Intel's RealSense D435 camera. It is a stereo depth camera with an RGB module. For our dataset, will be mostly using the RGB sensor to capture images at up to 1280 * 720 resolution. Pictures will have to contain all types of figures from both the black and the white team. In order not to represent any of the figures more than the others in our neural network model, we will have exactly one figure per type meaning that we will use 12 figures in total. Board position has to be thought through because of potential occlusion of the figures. Camera position is also a vital factor since object contours start to lookalike when the pitch angle of the camera is too large.

3.1.1 METADATA GENERATION

When training a neural network model, we need to be careful with the data source we provide. If we place some of the figures on the same spots on the board repeatedly, our model will tend to expect that figure more frequently on that spot. Maybe we placed one of the figures more times on the white chess squares instead of the brown ones. To combat all these possible outcomes that shift our model in ways we don't want, we have to try to create a random positioning system for the figures.

We will use a random online FEN generator that will create unrealistic, yet very randomized placement schemes for our figures. We will also use a visualization tool to recreate the visual layout of the board that will later help us when taking images. That metadata will be written in an Excel table with accompanying visualizations in form of JPEG images. We created 200 different board position to be used in the dataset creation process.

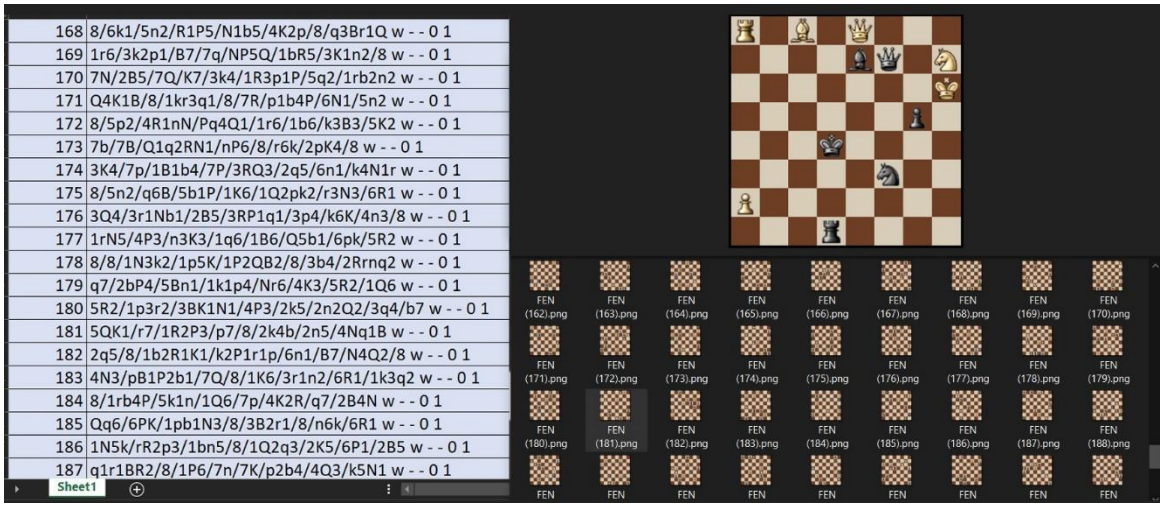

Figure 3.2: Metadata for chessboard figure positions, FEN strings on the left and JPEG visualizations on the right

3.1.2 PHYSICAL SETUP – TEST DATASET

Firstly, we have to break down the physical setup of the image-taking process. Since we are testing only the functionality of detecting chess figures, we don't need high-level precision from a proof of work test. In that case, the lightning setup will be as simple as possible. Two large windows located to the left side of our chessboard seemed to be enough to do the job, but fluctuating sunlight caused constant changes in our exposure levels that we tried to control with white pieces of paper. After some thinking, all the clutter around the board was removed to aid the training in the later stages of this process. Moreover, a small LED light will illuminate the right side of the scene for the figures to be more recognizable.

Figure 3.3: Initial home setup for test dataset creation

After testing a few of the possible board positions, we settled on a 45-degree angle relative to the camera. We positioned the camera on a sturdy, aluminium tripod with a 3-way level head on the included 6mm mounting hole. The height of the cameras was about 120 centimeters while the pitch angle was around -45 degrees. Later it will be clear that the position of the camera and the board was not optimal. The RealSense camera will be connected to our machine through a USB 3.1 cable. That mostly completes our simple, yet effective picture-taking setup for our test dataset.

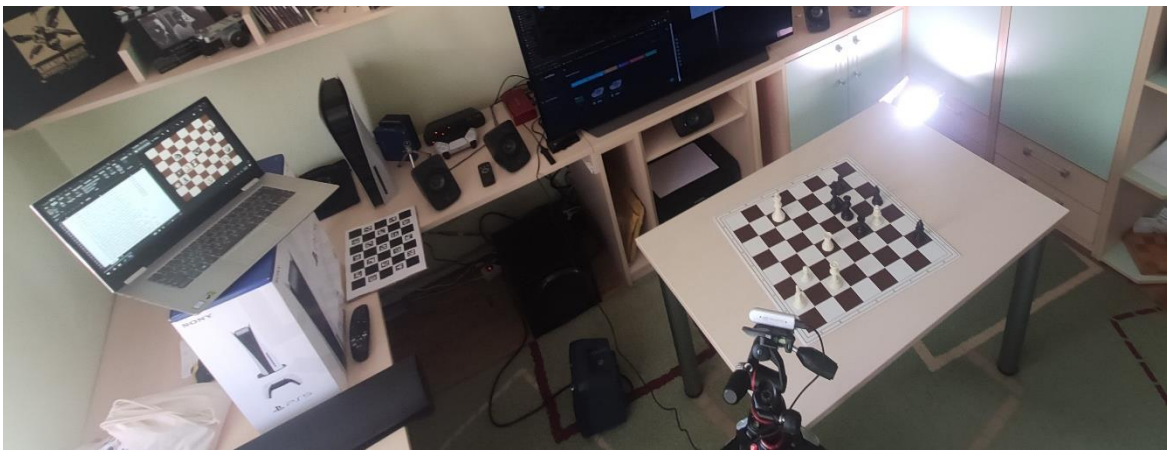

Figure 3.4: Updated home setup for the test dataset

3.1.3 PHYSICAL SETUP – MAIN DATASET

We encountered several difficulties with the lighting setup of the board.

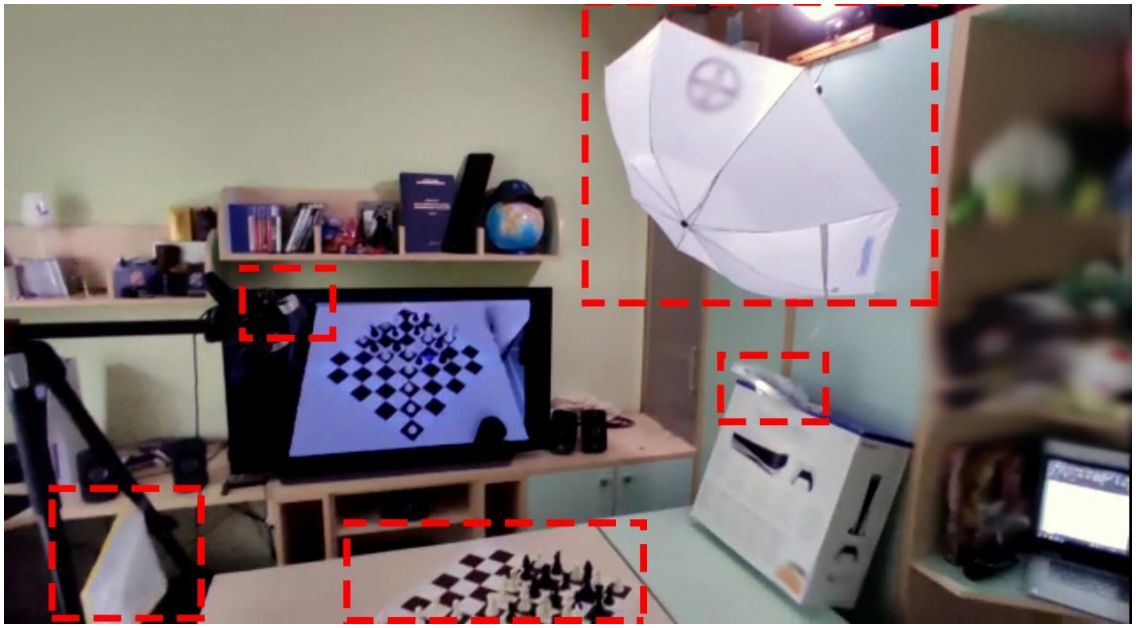

Figure 3.5: Initial home setup for the main dataset

The LED light was moved to the top right side of the board with a white umbrella in front of it to create a more defused lighting. A mobile phone flashlight wrapped in saran wrap was mounted on the bottom right side as well. Papers and white cloth were used to diffuse the light coming from the windows.

Unfortunately, all of this was not enough to create good lighting. The amount of sunlight was constantly shifting because of overcast weather and we were only able to capture 20 out of 200 planned images for our dataset. The only option that was on the table was to use a film club studio to get a much better lighting setup.

Figure 3.6: Studio setup for the main dataset with high power LED lights and controlled conditions

We transferred all the necessary equipment to the studio and started from scratch. The board position was not optimal, so we picked an angle between 50-55 degrees to further reduce occlusion. We increased the height and pitch of the RealSense camera to get a better view of the board. For lighting, we used 7 high-power LED panels arranged in a way to maximally reduce figure shadows. Now, the board was illuminated much more evenly with possible figure occlusion reduced to a minimum.

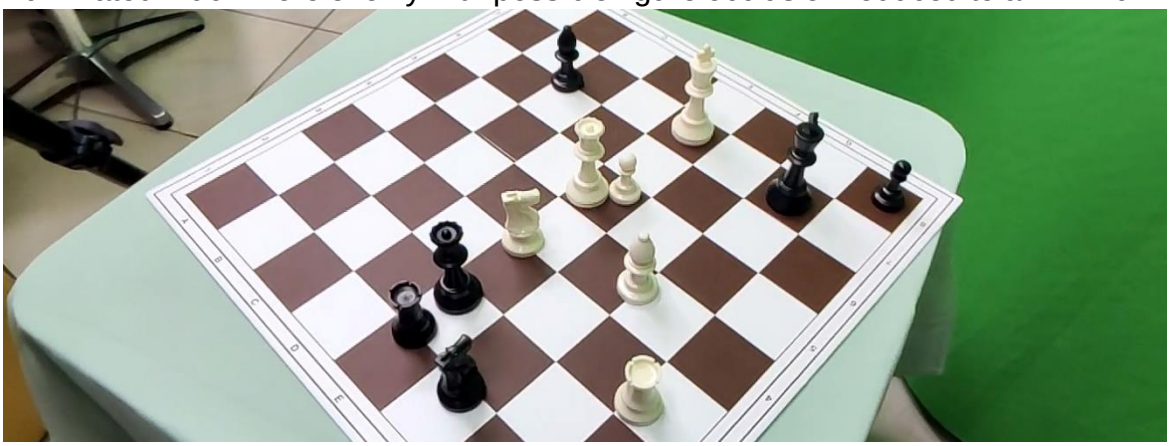

Figure 3.7: Even illumination across the whole chessboard

3.1.4 SOFTWARE SETUP

In all of the datasets that we created, the software setup will be the same. To start, we need to launch our pre-configured Docker container for our web service. Next, we need to startup Ocellus inside of Unity [8]. Inside our web UI, we need to create a new RealSense module. After the module has been created, we need to pick our camera model from the drop-down list and set up our camera settings. We chose the highest possible resolution for both the depth and the RBG sensor and saved our configuration for future use. Now we are able to see the live feed inside Ocellus from both of the sensors.

A very important step before we start is to calibrate the RealSense camera. For that, we will use something called a ChArUco board. Standard ArUco markers are useful due to their fast detection and versatility. However, the corner position accuracy on them is not precise enough for camera calibration. Using a chessboard pattern yields a more accurate result, though finding that pattern can be quite tricky. ChArUco boards try to combine both of the benefits that ArUco markers and chessboard patterns have. That way, our sub-pixel accuracy is adequate enough for calibration use, and the pattern is easily recognizable through ArUco markers placed on the white fields of the chessboard pattern.

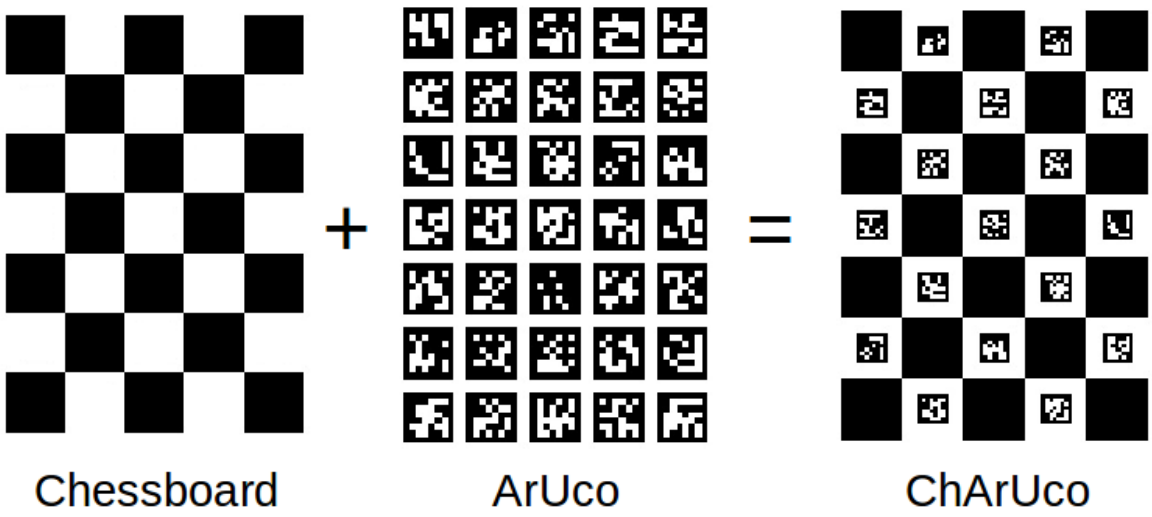

Figure 3.8: ChArUco board, a combination of a chessboard pattern with ArUco markers

Now that we calibrated our camera, we can see inside Ocellus that our point cloud, coming from the depth sensor image, is aligned with the floor inside the virtual scene. RealSense camera position can also be seen within the scene helping us

with the orientation of our real-world objects. Now we are ready to create images for our datasets.

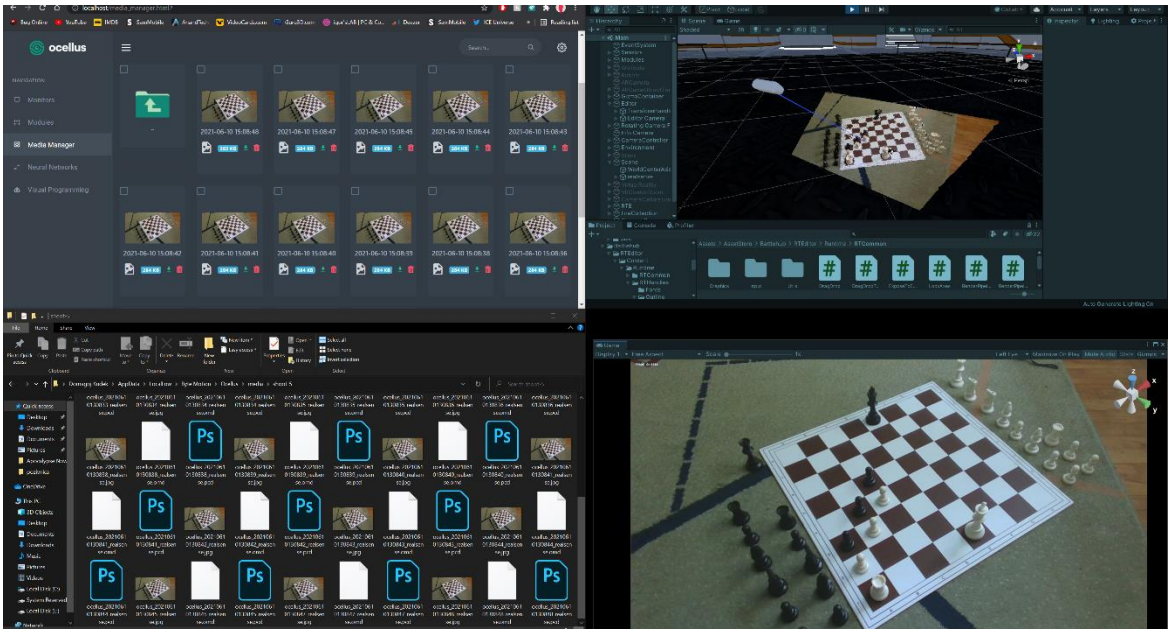

Figure 3.9: Software setup for image capturing, top left is the Ocellus web UI, bottom left is the active folder, top right is the point cloud visualization, and bottom right is the RGB sensor real-time capture

3.1.5 TAKING PHOTOGRAPHS

From inside the media manager, we will be taking screenshots of our currently active RealSense camera. To resolve our previously mentioned problems with figure placement, we will be using our prepared metadata to place the figures in optimal positions. For the test dataset, we will be taking only 30 out of 200 prepared board configurations, and later, full 200 configurations for our main dataset. We need to be careful when working around the tripod since any accidental movement will change camera position, figure contours, and calibration parameters. This process will take quite some time to finish due to the constant switching of the figure positions. When we are out of the way of LED lights, we can press the take screenshot button in the media manager. We can see that we got 4 files from every screen capture. We will be using only the JPEG image from the RGB camera for later dataset creation. We changed the lighting setup and added another 30 images with better conditions having 60 images in total. After we have finished the picture-taking process, we can start working on image annotation.

Figure 3.10: Evolution of the end result with different picture taking setups

3.1.6 IMAGE ANNOTATION

For image annotation, we are going to use the online platform Supervisely. After creating a new project, we need to upload our JPEG images to the service. Shortly after, we can launch the main UI of the web application. The first thing we are going to do is create 12 unique classes for every chess figure that we have. Each class is automatically assigned with a random color. Image annotation is a very repetitive and unintellectual type of work, but we have to be very careful with it since it influences our end result drastically. Our main goal here is to use the polygon creation tool in order to map out each individual contour of chess figures. Each one of them has some distinct characteristic that we need to precisely annotate with points. Points are automatically connected, and a polygon is created. A good practice here is to use as many points as possible to reduce the aliasing of our contours.

Another good practice is to check if we chose the correct classes for our figures since wrongly annotating a few of them can affect the model's ability to recognize them by a lot. For every image, we need to annotate each out of the 12 figures on the board. In terms of time, that will cost us about 8 to 10 minutes for each image. After we are done, we have 280 annotated images ready for training, but that won't be enough. For the model to work correctly, we need at least a few thousand

annotated images which would cost us hundreds of hours in labor. That is where dataset augmentation will speed up the process.

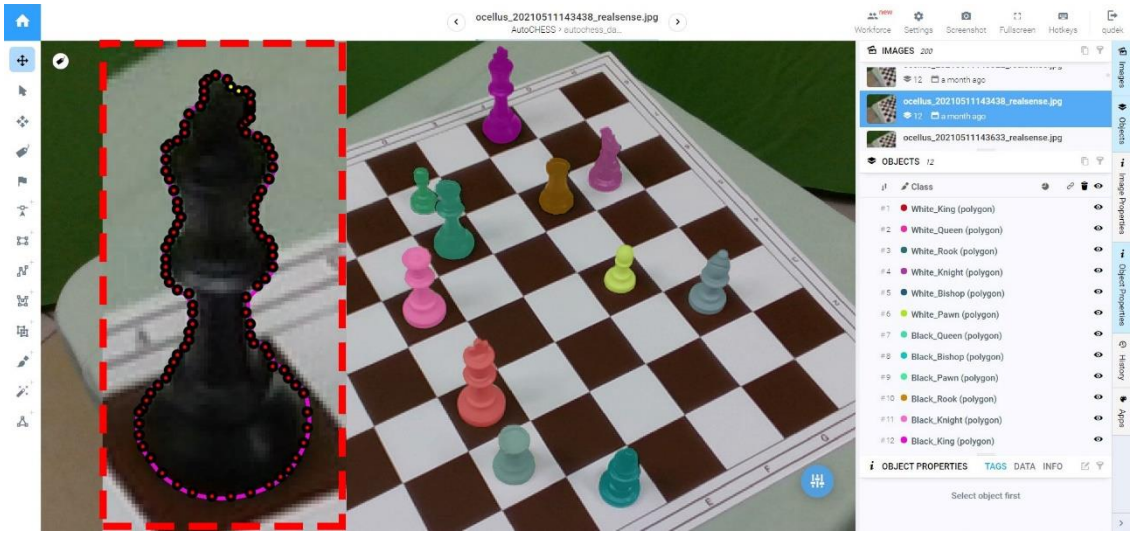

Figure 3.11: Supervisely image annotation tool with annotated figures, black king marked with polygonal shape tool on the left

3.1.7 DATA AUGMENTATION

To get more usable data for the training of our model, we just need to make minor alterations to our existing dataset. By synthetically modifying data, we are broadening our range of conditions in which our detected objects can be found. For this particular job, we are going to use a specially designed language that allows us to fully automate data manipulation. DTL or Data Transformation Language will help us in merging datasets, making various augmentations of images and annotations, format conversion, and more [15].

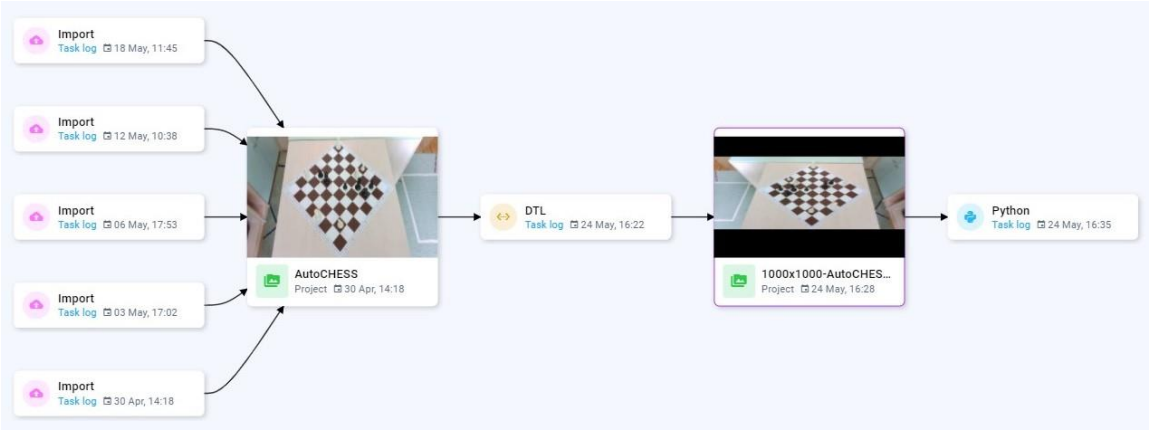

Figure 3.12: Complete Supervisely workflow from data import to dataset download

The process applied to our data is defined with a JSON-based configuration file. Our modifications to the images have to make sense, flipping an image around its horizontal axis doesn't since we don't expect to flip our chessboard upside down.

After loading in our data, firstly we are going to flip the images around their vertical axis. After that, we will join the flipped images with the non-flipped ones. Results from that transformation will branch off in two different directions. One will go directly to the end of the transformation pipeline, the other will go through another set of transformations. Before other transformations take place, we are going to be multiplying our images by the factor of 3. The first transformation on our images is rotation, where we have a random angle chosen between -10 and 10 degrees. The next one is the sliding window in combination with the resize operation. First of all, we enlarge the original image to a higher resolution. Then, we apply a sliding window that will crop the images by sliding from left to right, or from top to bottom. We specify the amount of possible sliding ranges from -200 to 200 pixels. Now within both of the branches we use the resize option and resize every image to 1000*1000 pixels. This is important for our neural network where it will expect images that are exactly that size. The last transformation is the contrast and brightness filter. Again, we will specify a possible range of brightness level changes that will be applied to our image. We do the same thing with the contrast filter settings and merge both of the branches into one. This relatively simple process consisting of several different data modifiers and branches creates 25 images out of one original image.

```
{
  "dst": "$window",
  "src": [
    "$resized"
  ],
  "action": "sliding_window",
  "settings": {
    "window": {
      "width": 1000,
      "height": 1000
    },
    "min_overlap": {
      "x": 200,
      "y": 200
    }
  }
},
```

Figure 3.13: Code snippet for a single DTL transformation

After dialing all the parameters and options we settled on a configuration that produces realistic-looking images that we can use to train our neural network model. We combined the 80 images taken in somewhat ok lighting conditions with the 200 images taken in the perfect lighting to create the input for our DTL transformation pipeline. The resulting augmented dataset consists now of more than 7000 individual images. This is a respectable number that will hopefully yield a good

ending result. It is easy to go overboard with the transformations. One such test yielded more than 80 000 images which is way too much for our use case. Training on that dataset lasted for more than a week and gave similar results to the much smaller datasets previously used. Because of that experiment, we will use the configuration described above for data augmentation.

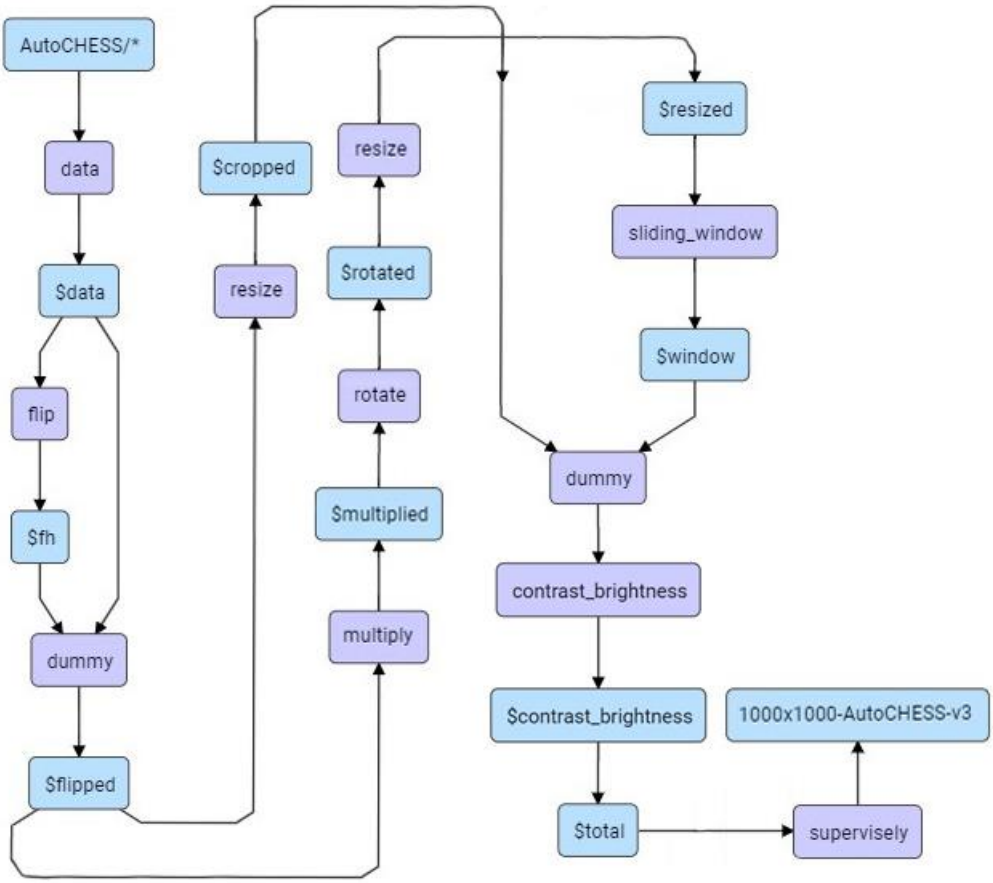

Figure 3.14: DTL computational graph for AutoCHESS dataset

3.1.8 DATA FORMAT CONVERSION

Supervisely uses its own data format which has to be converted to the COCO format that we use to train our models. COCO is large-scale object detection, segmentation, and captioning dataset format with many useful features. For this job, we are going to use a Python script inside of a Python virtual environment [9]. A Python virtual environment is an isolated environment consisting of a Python interpreter and different libraries and scripts used in our conversion script. Similar to Docker's containers, a separate virtual environment with all the needed

dependencies is very convenient to use between different systems and base Python versions.

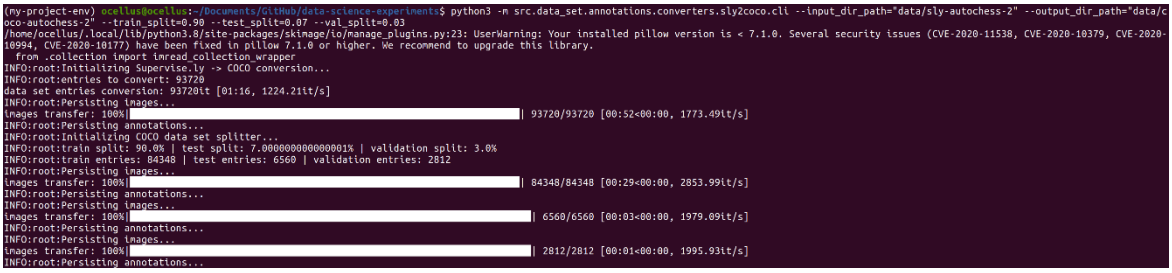

Figure 3.15: Bash shell running the python format conversion script

This script will split our dataset into 3 categories, a training, test, and validation split. Data used for one set can't be used for others, so we have to be careful with our choice of ratios. The balance of ratios between data splits comes from extensive empirical testing and differs from project to project. More training data means that our model will have more examples to learn from thus finding a better overall solution. More validation data help us with making a better decision about which exact model is the best out of the bunch. Lastly, more test data gives us a better view of how well our model generalizes unseen data. We eventually landed with a training split consisting of 90% of total images from the dataset, a test split consisting of 7% of images, and finally a validation split consisting of only 3% of images. Every dataset split has its own JSON document that has all the annotations from the images stored in it.

3.2 AI MODEL TRAINING

Now that we have created, collected, and modified our data, we are ready to prepare our model training setup. This workflow combined with dataset creation is highly dependent on trial and error. It is expected that we will have to do this multiple times to yield an acceptable result. Here, we are going to discuss only the latest version which is also the best one.

3.2.1 PyTorch 1.5 and CUDA 10.1

PyTorch is an open-source deep-learning library that is going to help us to create our model. Its job will be to recognize different types of chess figures placed on the

board. Since PyTorch has better support on Linux-based systems, we are going to use Ubuntu 20.4 LTS for our operating system. Unfortunately, CUDA, Nvidia's technology responsible for GPU acceleration in tasks like machine learning is not well supported on Linux. This means we will have to use some workarounds to get it up and running. This is where our trusted Docker comes into play again. Byte Motion has prepared a Docker container that has both PyTorch and CUDA libraries with all the necessary dependencies ready to go [9]. To run it we have to execute the docker-run command with a few additional flags and settings. Firstly, we have to tell Docker that we want to use our GPU for this container. Also, we need to expose the container's port, in our case we are using port 8888. This exposed port is accessible on the host, and it is available to any other client that can reach it. After that comes the folder structure specification for our used notebooks, source files, data, and configurations. Lastly, the name of the PyTorch image is provided for the creation of the container. This way, we are using PyTorch in the GPU mode that will vastly speed up our learning process. Without the use of Docker containers, due to the lackluster Nvidia support on Linux, we wouldn't be able to use CUDA for our project.

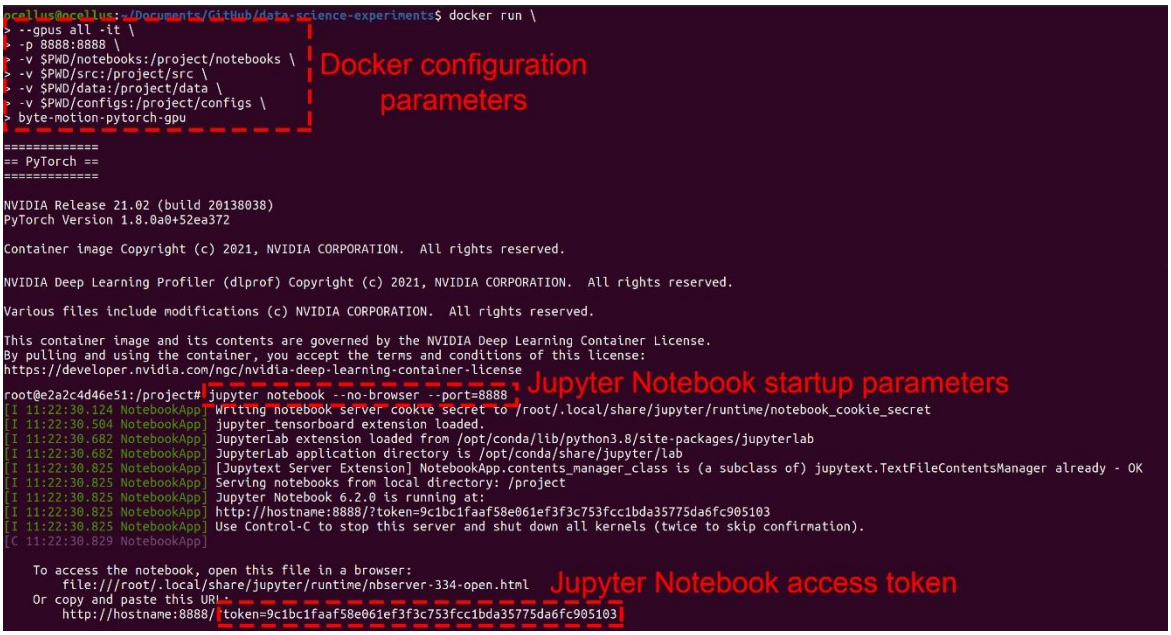

Figure 3.16: PyTorch container and Jupyter Notebook configuration parameters inside a bash shell

3.2.2 Jupyter Notebook

As formerly mentioned, Jupyter Notebook is a web-based environment where we will configure our PyTorch neural network. It will communicate with the host through the previously specified port 8888. To run it inside our newly created container, we have to run our PyTorch container in interactive mode. This means we will be able to execute commands inside of it while the container is running. The command prompt will change accordingly, moving us to a bash shell where we can run Jupyter Notebook. After we run the command, the bash shell will provide us with a token that will connect our browser to a running instance of Jupyter Notebook. Inside the UI, we can browse our previously ran training configurations and inspect the results. Based on these results, we can tweak our parameters and settings to get a better final result.

3.2.3 Training configuration

In our training configuration, the first step is to import libraries that we are going to use, namely the latest version of Python 3 and Torch vision packages. After that we are going to import the Detectron2, a state-of-the-art library providing detection and segmentation algorithms for AI research usage. With its use, training will be faster, and models can be exported to TorchScript or Caffe2 format for immediate deployment. Then we have to specify folder structure settings. We have 3 different datasets for training, testing, and evaluating our model and an output folder that we need to provide. Each dataset has its own JSON metadata file that we also have to link. We are going to run our instance segmentation through a 101-layer Mask R-CNN network that uses Feature pyramid networks for its feature extraction.

Now comes the main configuration part where we have to determine our hyper-parameters. The maximum number of iterations describes how many times we want to search for the optimal model. Making this parameter too large will result in a very long training time while making it too small severely reduces the chance to find the best possible model. We settled on 60000 iterations for our model resulting in a training time of roughly 10 hours making the process easily repeatable. The next one is the learning rate which is the maximum amount of change to the weights that are updated during one training iteration. It controls how quickly our model adapts

to the problem. If the rate is too small, training tends to get stuck in the middle of the learning process. If the rate is too large, our model will converge too quickly to a suboptimal solution and the best one will never be found. It can range between 0 and 1, wherein our case we chose 0.00025 for a steady and continuous improvement. The following parameter called the evaluation period describes the length of the evaluation after a single iteration of training. If we want to validate our model in order to assess its performance, we have to make sure we allocate enough time for this particular job. Assessing the model too shortly won't give enough feedback to the neural network to tweak its weights and states. Again, doing it for too long will stretch the required time to find the best solution. We used the evaluation period of 250 seconds. Last but not least is the number of classes parameter. It represents the total number of different classes or more precisely objects we have in our model. Though trivial, it is very important to set it correctly since a single typo can ruin the whole training process. We have 6 white and 6 black chess figures resulting in 12 total classes. Before starting the training, the last block of code we need to execute is some custom trainer configurations where we define the console-like output that will help us monitor the progress and time remaining of our training process.

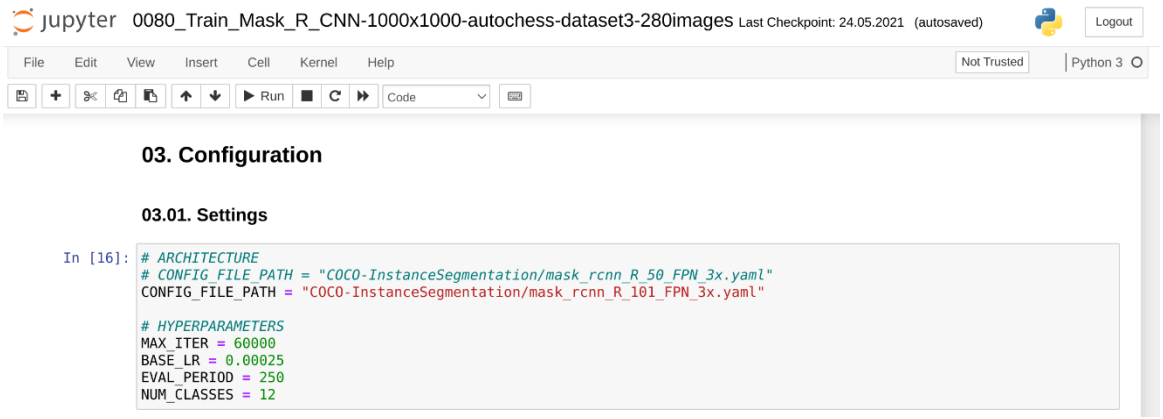

Figure 3.17: Hyper-parameter settings code segment in the train configuration inside Jupyter Notebook

3.2.4 Model training

With that out of the way, we can finally run the trainer and observe the process. After the initial loading phase, training starts in a long loop of intermittent running and testing for a single iteration.

```
[05/24 18:31:46 d2.utils.events]:  eta: 2:27:30  iter: 13999  total_loss: 0.2598  loss_cls: 0.05681  loss_box_reg:
0.1108  loss_mask: 0.07094  loss_rpn_cls: 0.000468  loss_rpn_loc: 0.01631  validation_loss: 0.3128  time: 0.1917  d
ata_time: 0.0010  lr: 0.00025  max_mem: 2125M
[05/24 18:31:50 d2.utils.events]:  eta: 2:27:18  iter: 14019  total_loss: 0.2501  loss_cls: 0.05051  loss_box_reg:
0.1047  loss_mask: 0.06701  loss_rpn_cls: 0.0007104  loss_rpn_loc: 0.0138  validation_loss: 0.3128  time: 0.1917  d
ata_time: 0.0011  lr: 0.00025  max_mem: 2125M
[05/24 18:31:54 d2.utils.events]:  eta: 2:27:18  iter: 14039  total_loss: 0.2713  loss_cls: 0.06303  loss_box_reg:
0.1093  loss_mask: 0.06684  loss_rpn_cls: 0.0005987  loss_rpn_loc: 0.016  validation_loss: 0.3128  time: 0.1917  da
ta_time: 0.0011  lr: 0.00025  max_mem: 2125M
[05/24 18:31:58 d2.utils.events]:  eta: 2:27:07  iter: 14059  total_loss: 0.2591  loss_cls: 0.05307  loss_box_reg:
0.1116  loss_mask: 0.06586  loss_rpn_cls: 0.0009633  loss_rpn_loc: 0.01572  validation_loss: 0.3128  time: 0.1917
data_time: 0.0010  lr: 0.00025  max_mem: 2125M
[05/24 18:32:02 d2.utils.events]:  eta: 2:27:11  iter: 14079  total_loss: 0.2951  loss_cls: 0.05774  loss_box_reg:
0.1201  loss_mask: 0.07217  loss_rpn_cls: 0.0008868  loss_rpn_loc: 0.01655  validation_loss: 0.3128  time: 0.1917
data_time: 0.0010  lr: 0.00025  max_mem: 2125M
[05/24 18:32:06 d2.utils.events]:  eta: 2:27:07  iter: 14099  total_loss: 0.2214  loss_cls: 0.04295  loss_box_reg:
0.1061  loss_mask: 0.06485  loss_rpn_cls: 0.0007665  loss_rpn_loc: 0.01237  validation_loss: 0.3128  time: 0.1917
data_time: 0.0010  lr: 0.00025  max_mem: 2125M
[05/24 18:32:10 d2.utils.events]:  eta: 2:27:09  iter: 14119  total_loss: 0.2415  loss_cls: 0.03893  loss_box_reg:
0.104  loss_mask: 0.0633  loss_rpn_cls: 0.0006129  loss_rpn_loc: 0.01246  validation_loss: 0.3128  time: 0.1917  da
ta_time: 0.0010  lr: 0.00025  max_mem: 2125M
[05/24 18:32:14 d2.utils.events]:  eta: 2:27:06  iter: 14139  total_loss: 0.2763  loss_cls: 0.06155  loss_box_reg:
0.1166  loss_mask: 0.07283  loss_rpn_cls: 0.001141  loss_rpn_loc: 0.01937  validation_loss: 0.3128  time: 0.1917  d
ata_time: 0.0011  lr: 0.00025  max_mem: 2125M
```

Figure 3.18: Jupyter Notebook console output during the iteration phase with details for each iteration

During the evaluation segment, we can see the current per-class bounding box and segment average precision. The first numbers are expectedly very low ranging from 4 to 6 percent. The inference is done on 491 individual images randomly selected from our validation dataset. Now, all it takes is time for our model to finish its gradient descent into the best possible solution based on our current model structure and configuration.

```
[05/24 19:55:04 d2.evaluation.coco_evaluation]: Per-category bbox AP:
| category    | AP     | category     | AP     | category     | AP     |
|:------------|:-------|:-------------|:-------|:-------------|:-------|
| Black_Rook  | 89.212 | White_Queen  | 90.418 | White_King   | 91.611 |
| White_Pawn  | 87.804 | White_Knight | 87.995 | Black_Knight | 87.463 |
| White_Rook  | 88.878 | White_Bishop | 90.806 | Black_Queen  | 92.077 |
| Black_King  | 88.748 | Black_Pawn   | 88.827 | Black_Bishop | 90.192 |
Loading and preparing results...
DONE (t=0.04s)
creating index...
index created!
[05/24 19:55:05 d2.evaluation.fast_eval_api]: Evaluate annotation type *segm*
[05/24 19:55:05 d2.evaluation.fast_eval_api]: COCOeval_opt.evaluate() finished in 0.36 seconds.
[05/24 19:55:05 d2.evaluation.fast_eval_api]: Accumulating evaluation results...
[05/24 19:55:05 d2.evaluation.fast_eval_api]: COCOeval_opt.accumulate() finished in 0.03 seconds.
[05/24 19:55:05 d2.evaluation.coco_evaluation]: Evaluation results for segm:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 87.162 | 97.074 | 96.170 | 53.931 | 90.259 | 90.000 |
[05/24 19:55:05 d2.evaluation.coco_evaluation]: Per-category segm AP:
| category    | AP     | category     | AP     | category     | AP     |
|:------------|:-------|:-------------|:-------|:-------------|:-------|
| Black_Rook  | 88.636 | White_Queen  | 87.217 | White_King   | 88.742 |
| White_Pawn  | 86.097 | White_Knight | 85.930 | Black_Knight | 85.633 |
| White_Rook  | 87.903 | White_Bishop | 88.318 | Black_Queen  | 88.795 |
| Black_King  | 86.470 | Black_Pawn   | 85.558 | Black_Bishop | 86.649 |
```

Figure 3.19: Jupyter Notebook console output during the validation phase with average precisions for each class

3.2.5 Training results

We can see that now, the average precision for chess figures greatly improved and stands at a solid 91.5 percent average. The 2 lowest scoring figures are the white pawn, which due to its reflective plastic material sometimes creates harsh highlights in the dataset images, which then hides a part of an already small figure making it hard to distinguish.

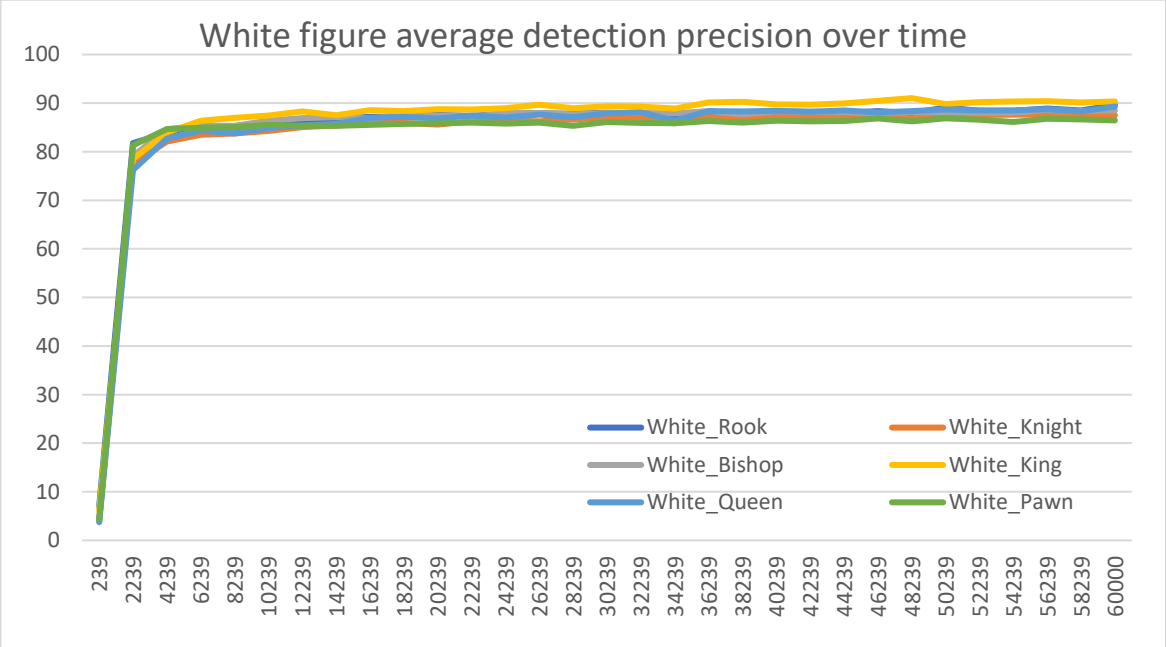

Figure 3.21: Average detection precision of white figures over 60 000 iterations of neural network training

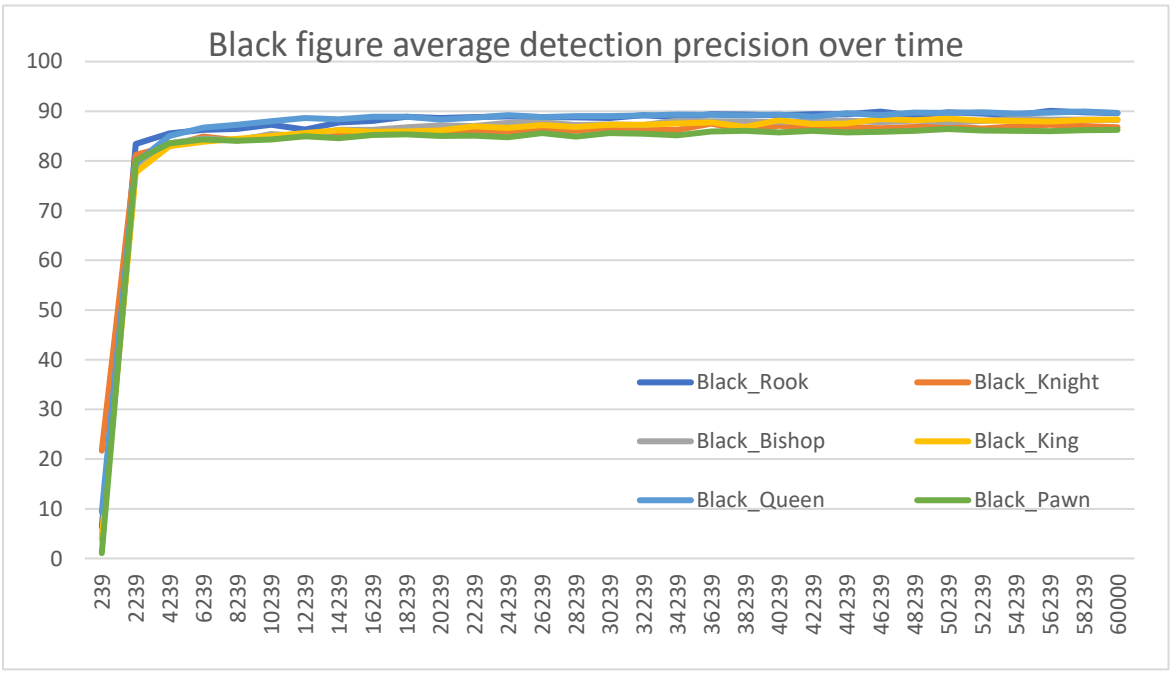

Figure 3.20: Average detection precision of black figures over 60 000 iterations of neural network training

The second one is the black knight whose complex shape observed at an odd angle makes it difficult to pinpoint contours. All in all, through trial and error, after 6 different training sessions with slightly different configurations we got a solid result that is usable for our live demonstration. We can see that the majority of progress happened in the first 5 000 iterations. The rest of the changes slowly happened over the next couple of thousand iterations. Even though it is a small difference, this small improvement in detection goes a long way when the system is deployed. To test its capabilities on new, never before seen examples, we will populate our board with some figures and take a few photographs with a 12-megapixel mobile phone camera in good lighting conditions. We will run inference on that set of images using Detectron2 and the accompanying Python script. In the configuration, we have to give paths for our model, dataset, and output location. Also, we want to specify the all-important number of classes parameter as well as the neural network configuration parameter that describes the number and arrangement of layers inside of it. CUDA is used again to accelerate the process which is relatively short due to the small size of our newly created dataset. The results are in, and we can conclude several different things from them.

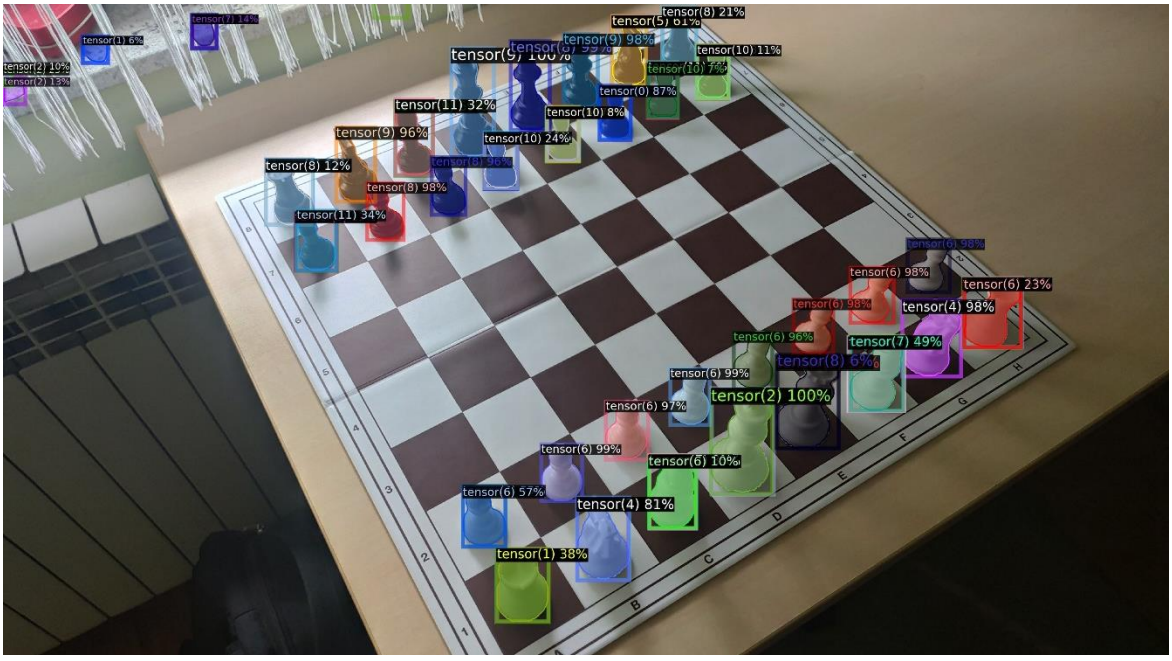

Figure 3.22: Detectron2 image output with labeled classes and object contours

All the figure contours have been correctly labeled, which is very important for determining their 3D location later down the line. We can also see that some figures have very high precision levels. Several figures were registered as multiple ones

with different precision levels. That is an example of unwanted behavior, but we don’t need to worry about them since we will deal with those false positives later in Ocellus. Other false positives around the board are also unwanted, we will have to deal with them in the Node-RED section. To sum the training results up, the detection and the precision of it is on a usable level where we can safely rely on it and continue the work on other components of this system.

3.3 OCELLUS VISION SYSTEM

To try out our finished model inside Ocellus our first task is to convert our model from Detectron2 to Caffe2 format. We will do that we the appropriate Python script that has several arguments we need to fill out. We have to specify the source model, target model, and dataset paths as well as the number of classes and the configuration of the network. This is also a CUDA accelerated task and it should be relatively quick to finish. The conversion is important since we use the model inside of a C++ environment that doesn’t directly support the Detectron2 format for trained networks. Once the conversion is complete, we will wrap our Caffe2 format network in a .tar archive and add an additional JSON document where we will specify all the 12 classes we want to detect. Ocellus is the first major component of our AutoCHESS system. It is very complex by nature and describing it in high detail would take up an entire thesis. To keep the length in check, we will describe only the segments that we use.

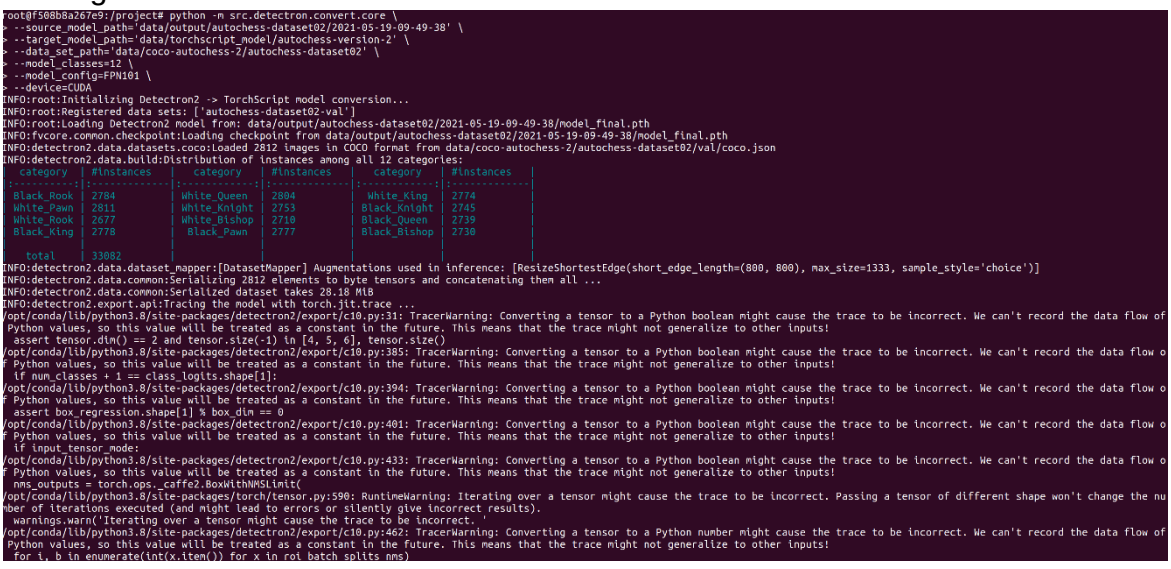

Figure 3.23: Bash shell running the python neural network conversion script from Detectron2 to Caffe2 format

3.3.1 RealSense Module

The RealSense camera is going to be our main image gathering device. It comes with its own SDK and a range of different scripts and software packages [5]. We use those packages as an external C++ library inside our own C# Unity scripts. The main core of Ocellus communicates with a module controller. The controller handles all the communication between the different modules and the core. Modules are separated into frame grabber modules that are able to capture images and other modules that have varied functionalities. The RealSense module belongs to the frame grabber group since it is a camera. It has 3 different output parameters that can be used as inputs for other modules and components. The first one is the RGB image, the second one is the point cloud, constructed out the stereo depth sensors, and finally, a camera output that is used inside Ocellus to monitor real-time feed from the RGB sensor.

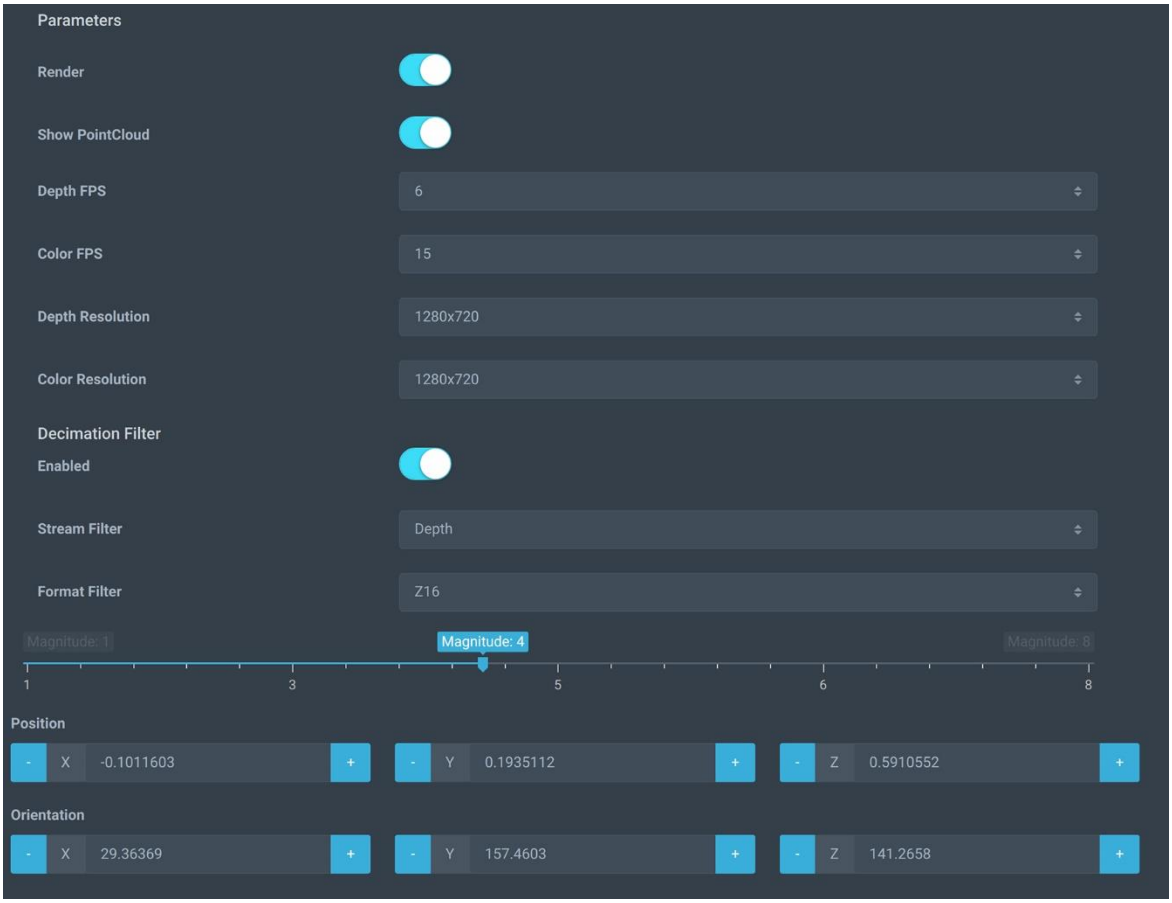

Figure 3.24: RealSense module settings used in the Ocellus web UI

Inside each module, a Reduce function has to be implemented. Its functions are based on the inputs and outputs of the module itself. That function is called each

Ocellus frame, which is a globally synchronized clock that keeps all the systems in the same tempo. It has to be noted that the Ocellus frame is not equal to a Unity frame that happens each second. Inside this particular module, the Reduce function does several things. On the top left corner of the RGB image inside the RealSense camera, there is a frame counter that shows the number of captured frames from the moment the module was turned on via the web UI. There is a segment that keeps the real-world position and orientation of the camera synced with its virtual twin inside the Unity scene. The main part of it is the return state that holds all the previously mentioned output information.

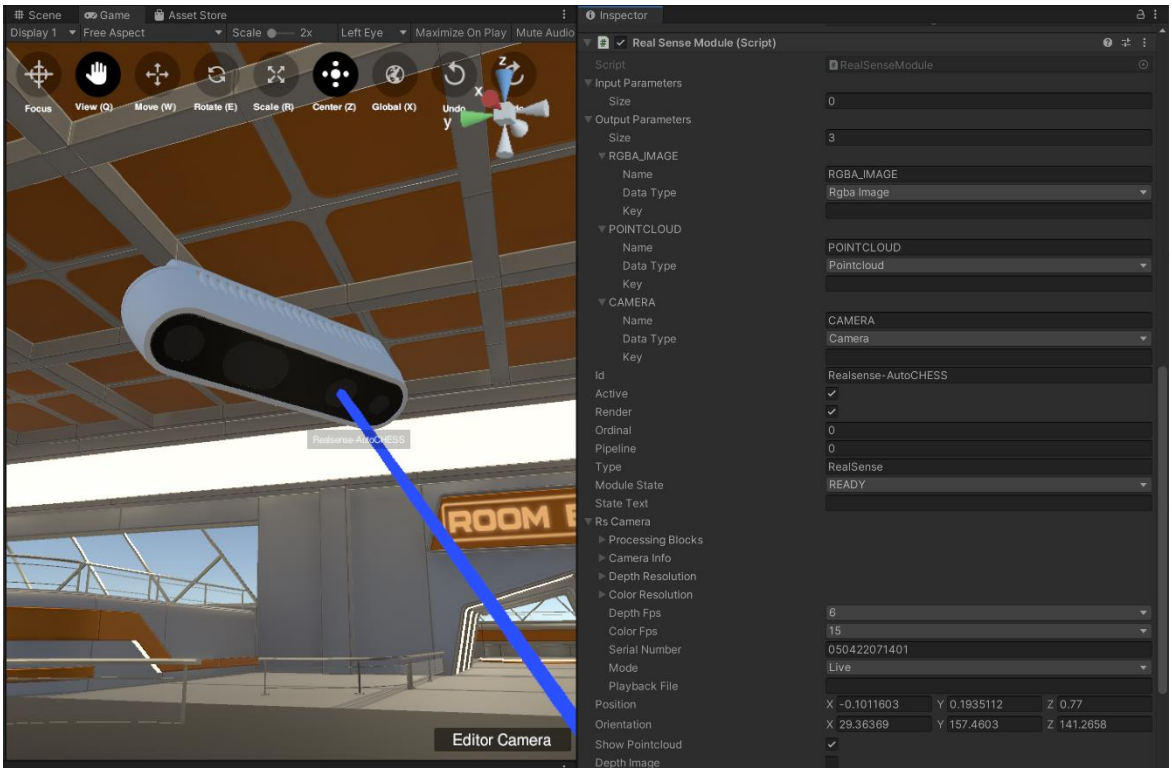

Figure 3.25: RealSense module GameObject on the right and a virtual twin of the camera in a scene on the left running inside Unity's editor

The RealSense SDK 2.0 gives us a wide range of exposed options that can be accessed through the Ocellus web UI. These options are normally available in Intel's RealSense Viewer application, but to use them in 3rd party software, they have to be manually supported by the developers. In the web UI, we can see all the available input and output parameters. In the camera drop-down menu, we can choose between a live feed mode and a playback mode where the video is streamed from a .bag file. In the configuration section, there is a plethora of options and parameters. We can toggle certain visualization methods like point clouds and

real-time mesh generation. There are resolution and frames per second options to choose from. Also, there are filters that can be applied to the image like hole-filling and decimation filter. We will use the decimation filter that reduces the number of points in the point cloud in order to help with the performance of the system. Lastly, there are positional and orientational parameters that are synced with Unity parameters.

3.3.2 DNN Module

The Deep neural network module is the second module we will use in this project. It is built to run a pre-trained neural network from a .tar archive. This module is not a frame grabber, but it belongs to another group of modules that produces items. An item is a virtual object that can represent real-life objects in a Unity scene. Apart from the position and orientation of the item, it can have other more complex data information such as contours, groups, and paths of points, point cloud segments, and generated 3D meshes. Items are used as a simplified visualization of a real-life object. The basic item has a 3D axis object displayed in the correct position and orientation. It gives us helpful information on where objects are in the world since their transform component is precisely matched with the real object. The DNN module produces those items in places where the neural network detects objects.

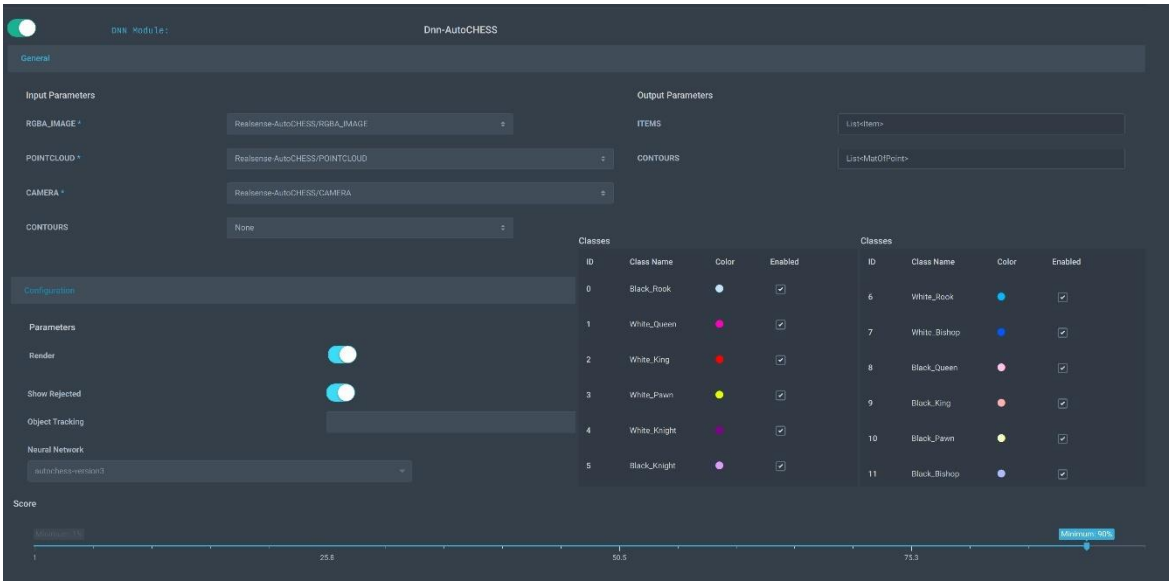

Figure 3.26: DNN module settings used in the Ocellus web UI with class labels

Another important thing to note is that the DNN module Reduce function runs in a separate thread. Unity is still mostly a single-threaded application. Most of the core systems still run on the main thread. This means that making complex modules requires careful allocation of system resources. Multi-threading is available and it works normally like in any C# application, but Unity's API is not thread-safe. That means that we can create threads without any practical limitations, however, we need to manually create safety systems to avoid problems such as deadlock and thread race. In addition, all the thread creation has to be done on the main Unity thread, as well as the use of core Unity components. This is a known limitation of Unity that has to be considered when working on complex systems that require a lot of processing power.

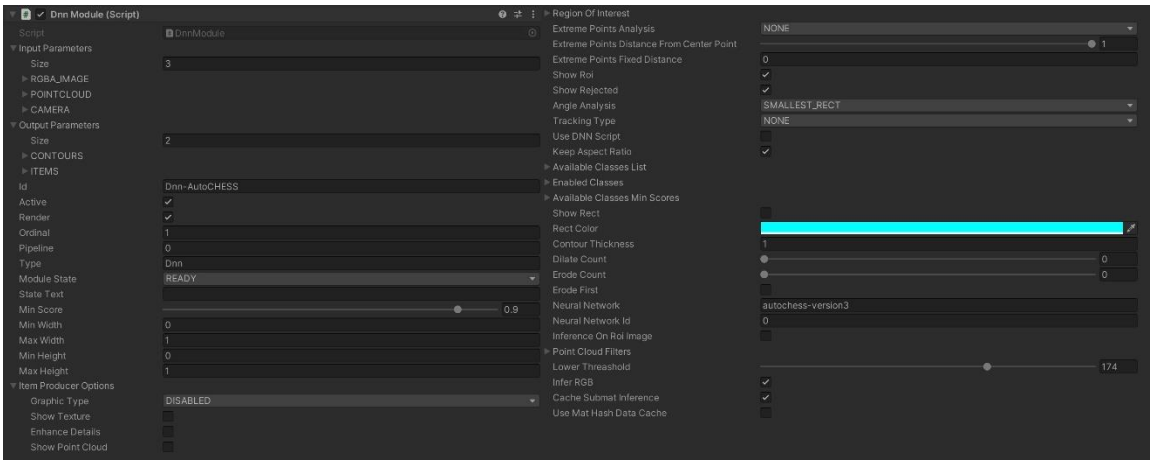

Figure 3.27: DNN module GameObject inside Unity's inspector window

Our DNN module has several input parameters that are mandatory for normal operation. DNN cannot work without an active camera sensor so in the drop-down menus of the web UI, we can choose between the different sources for our RGB image, point cloud data stream, camera input, and contours data stream. As outputs, DNN produces previously mentioned items and their contours. This information will later be fed into Node-RED for processing. There are a lot of configuration parameters available for the user to choose from in the web UI. They are mostly focused on visualization aids for the detected objects. We can select regions of interest, bounding box rectangles, contours, object outlines, and bounds, and manually pick which classes we want to display.

This module is very demanding on the system, so some stuttering and frame dropping can sometimes happen. A RealSense module needs about 30

milliseconds of processing time, while in combination with the DNN module, that number goes to around 260 milliseconds. It is more than 8 times more demanding on the system. Fortunately, our system is equipped with a modern 6-core processor with very high single-thread performance. Even though the module can run in real-time with the Ocellus frame clock, we are going to run it in the synchronous mode. Once we determine our chess figure locations, we don't need a constant stream of data fed into our Node-RED flow.

3.3.3 Ocellus setup

To set up our Ocellus configuration, we are going to use a similar setup to the one used for dataset creation. Firstly, as always, we need to fire up the web service Docker container. This time we need an additional Docker container for the communication between the neural network and Ocellus. Inside the web UI, we need to create a RealSense and a DNN module. After selecting the correct camera and live mode, we will use 720p resolution for both sensors at 6 frames per second. We will use the decimation filter with the factor of 8 and turn on point cloud visualization. Inside of the DNN module, we will select our inputs from the RealSense camera, select a region of interest and pick our latest neural network. Below that we will turn on the item outline and select all 12 classes to be recognized.

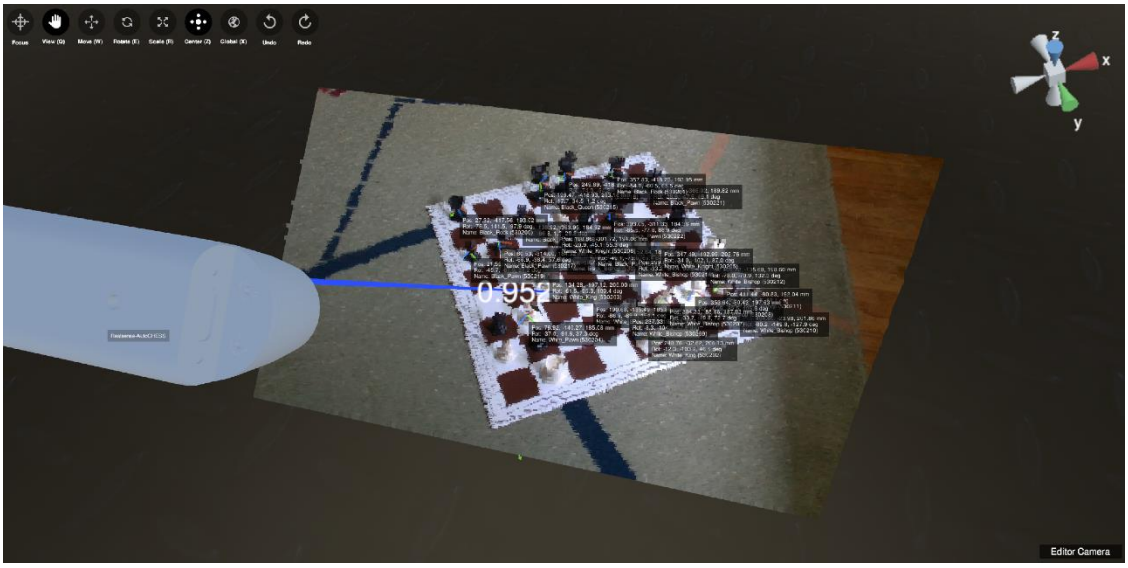

Figure 3.28: Real-time point cloud feed with items displayed on top of detected objects in the editor camera view

To deal with false positives and duplicate readings, we will filter all the detection results below 90 percent of confidence. That way we can be sure that the detected objects are correctly labeled. Since we need precise item positional information, calibration of the camera is needed. We will place the bottom left corner of the ChArUco board on the bottom left corner of the chessboard. Corners from the calibration board and the A1 chessboard field have to align as precisely as possible to ensure correct position calculation. Now we are prepared to try out the real-time detection of the figures.

3.3.4 Real-time object recognition results

Ocellus is now running inside Unity and within the play window, we can observe currently active cameras. We will mostly focus on the canvas camera that displays RGB info from RealSense and has an overlay where we can see detected objects, their class, contours, position, and orientation. As expected, the recognition works very well, but there is still some room for improvement. Black figures positioned on black squares sometimes are not properly detected. White figures exposed to harsh lighting create burned highlight spots that the camera cannot recognize properly.

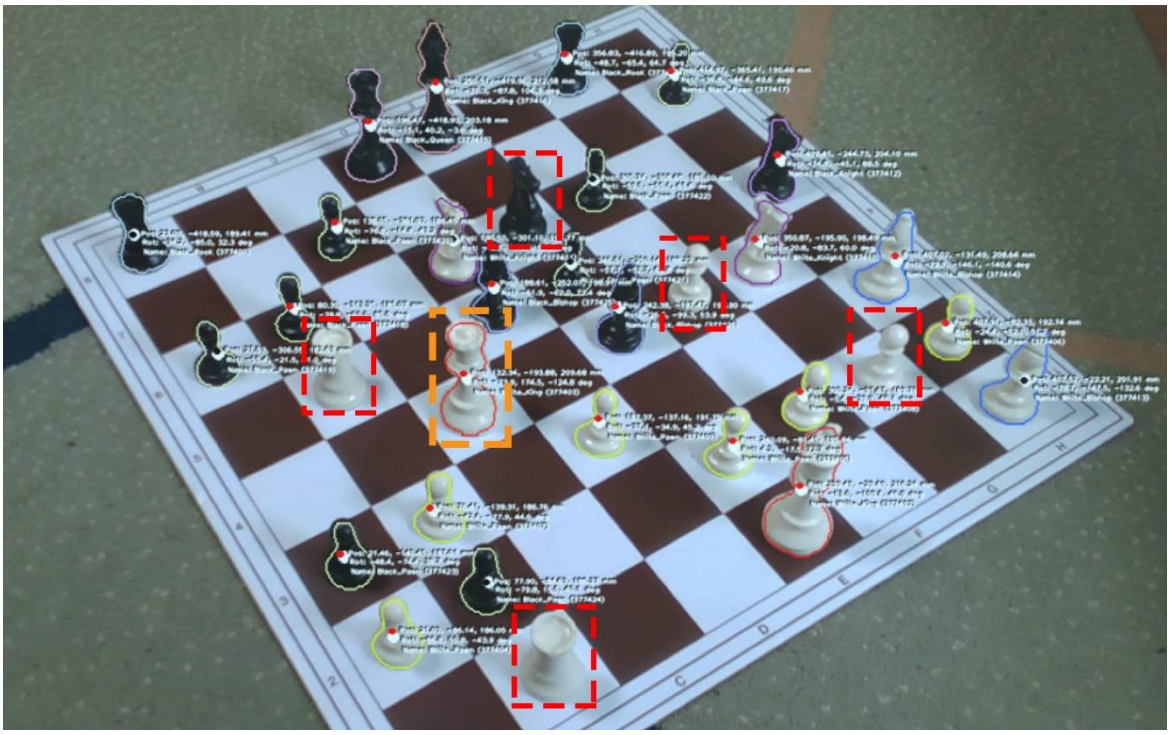

Figure 3.29: Real-time detection results, figures marked in red were not detected, figures marked in orange are falsely labeled

The only larger problem is the possible occlusion. When the board is full of figures, the network has a harder time recognizing all of them due to occlusion.

One solution is to place the camera completely above the board, but the neural network would have a harder time recognizing figures since all the contours would look very similar to each other. Another solution for that issue would be to use multiple cameras from a few different angles. That would greatly help in recognition, however, the implementation would become much more complex since we would need to combine multiple item streams into one. For now, the results are good enough for our live experiment. Since the neural networks that are used are easily interchangeable in the web UI, future iterations with higher levels of detection confidence can be added. We are now ready to start working on the second large component of our system.

3.4 Node-RED

Node-RED is going to be our main interconnect between Ocellus and ABB's software for YuMi robot. Usually, a platform primarily built for the Internet of Things devices is not used with industrial hardware that costs tens of thousands of dollars. Nevertheless, following Byte Motion's philosophy on using popular programming languages and software platforms, we will build the second large component of our AutoCHESS application in it. Node-RED uses visual programming in a form of nodes that communicate with each other via messages [21]. Each node has its inputs and outputs that can be connected with other nodes in order to create more complex functionalities.

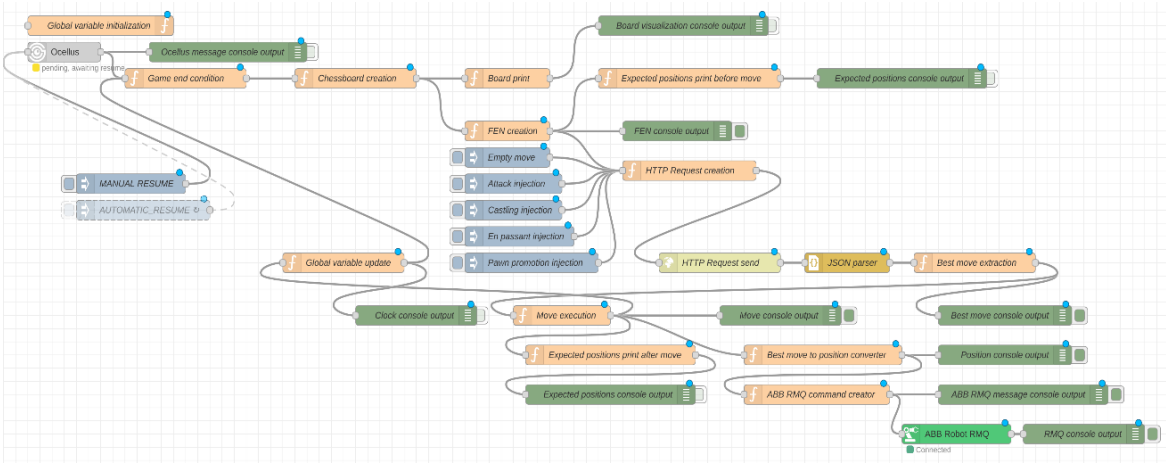

Figure 3.30: AutoCHESS complete Node-RED flow diagram

There are several functions that our Node flow will have to perform. First of all, we need to take the item stream coming from Ocellus and use their positional information to determine where on the board each figure is located. After that, we have to create a FEN string that will be fed into Stockfish. To be sure our neural network's detection works properly, a virtual chess game will be played in the background recording each move. Then, the FEN from detected figures will be compared to the virtual FEN. If those two strings are equal, that means we can proceed with the calculations. Stockfish will return the best possible move generated from the detected FEN and send that information further into the flow. Based on the return value of Stockfish's messages, we need to extract which move needs to happen. We have to be careful here since there are several special moves like castling and attack in passing. We will send that information in two ways. One will be connected to the virtual chess game, refreshing figure positions and global variables that track the game. The other will go into a position calculator that will determine the positions of figures determined by Stockfish. That information will be used to create command strings, that will be sent to the ABB robot node responsible for communication between our flow and the YuMi robot. To explain our flow in detail, lets it break down node by node.

3.4.1 Ocellus node

The entry point of our flow is the Ocellus node. It is a specially designed node that handles communication between a running instance of Ocellus and Node-RED. It is not available in public node repositories, and we have to download it from Byte Motion's private repository. There are several options when the node is placed in the flow. We want Ocellus to run synchronously since we don't need a continuous stream of items at all times. The second option we need to configure is the web address in our local network from which the Ocellus node will communicate with the Ocellus program. We can use a

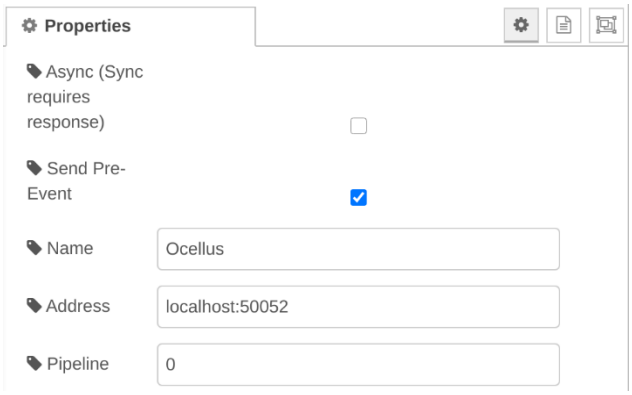

Figure 3.31: Ocellus node properties

debug node and connect it to the end of the Ocellus node to check whether or not we have a connection with the Ocellus system. We can see that within the output message we have a list of items waiting for further processing.

3.4.2 Chessboard creation node

There is a finite amount of default nodes that can be used. To have custom functionality enabled within a node, we can use the aptly named function node. Within it, using plain JavaScript programming language, we can define what exactly we want to do with our data coming from the previous node in the chain. Here, we want to transform positional information into chess coordinates. To achieve that, first, we need to do some measurements. The lower-left corner of the A1 square is 2,75 centimeters away from the chessboard corner. The size of each square on the board is 5,5 * 5,5 centimeters. Within a triple for loop, using these measurements, we will populate our array with detected figures. We also need to add a tolerance of 1 centimeter to have some wiggle room when setting the figures on the board. If there is no detected figure on a chess square, that value will be set to undefined.

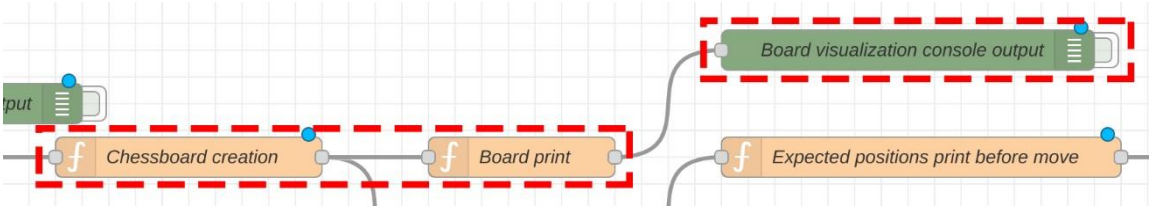

Figure 3.32: Chessboard creation node and its accompanying Board print node

After recording the detected figures in the array, even though empty positions on the board have no figure data, we still need positional data for these chess positions. For that, we will use a separate function that will calculate both the x and the y position of empty spots of the board. This way, figures that are detected, but are not on the board will not affect the game. This will be important later when we need to move figures from the board after they have been attacked. A neatly filled array is then wrapped in a new message and sent to the next node. Here, apart from the connection to the next node, we added another function node that handles board printing. The detected figures are printed in the console window of Node-RED as a visualization aid to help figure out detection issues. Figure names are

displayed on their locations on the board while empty spots are marked as series of dashes.

3.4.3 FEN creation node

FEN creation node is another function node with a special job in mind. It has to create FEN strings for the Stockfish chess engine. There are six different components in a FEN string so logically, we need to go through them one by one. The first section revolves around figure positions. If a figure is located on the chess square that is currently checked, its designated character is added to the FEN string. White figures use upper case letters while black ones use lower case letters. If there is no figure on that specific location, we increment a counter that is added to the string each time when a series of empty positions are interrupted by a figure. Once that is done for each array member coming from the message that the node received, we need to add the currently active player. For that, we will use a global variable that will track which player is the current one.

```
function ExportFEN(switcher, boardPositions)
{
  var freeCellCount = 0;
  var FEN = "";
  for (var i = 7; i >= 0; i--)
  {
    for (var j = 0; j <= 7; j++)
    {
      var test = switcher ? boardPositions[i*8+j].name : boardPositions[i*8+j];
      if (test == "Empty_Slot")
      {
        freeCellCount += 1;
      }
      else
      {
        var figure = boardPositions[i*8+j];
        if (freeCellCount != 0)
        {
          FEN += String(freeCellCount);
          freeCellCount = 0;
        }

        var figureName = switcher ? figure.name.split("_") : figure.split("_");

        if (figureName[1] == "King")
        {
          if (figureName[0] == "White")
            FEN += "K";
          else
            FEN += "k";
        }
```

Figure 3.33: FEN creation node code segment for first FEN segment construction

The next block defines the possibility of castling of both players. Stockfish will never execute castling if it is not physically possible to do it, but somehow, we need to keep track if the king and rook, who participate in this move, haven't yet moved. We will check that condition in a separate function in which a set of global Boolean variables will be used to determine if they had already moved somewhere from their original position. Next is the attack in passing indicator. Later in our flow, each time a pawn moves for 2 chess squares, we will record the attack in passing position in a global variable. Then, in the FEN creation node, we will read the value of that position and add it to our FEN string. Lastly, we

are adding the values of both of our timers to the string before sending our finished string in a new message to the next node.

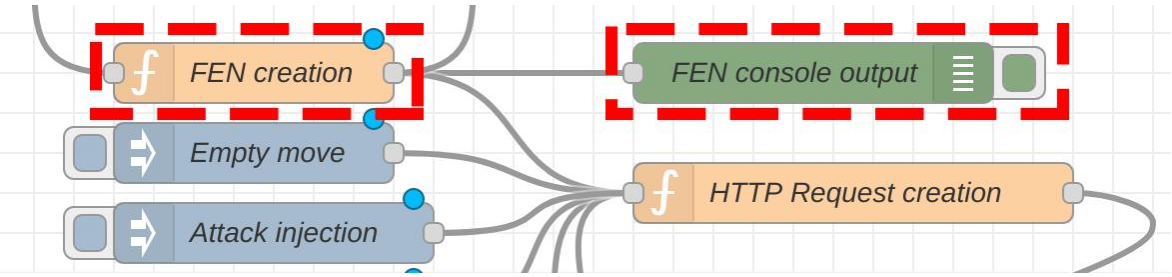

Figure 3.34: FEN creation segment of AutoCHESS Node-RED flow

Again, we can add a debug node to check the value of our generated FEN string. Before sending the message, we will add an if statement where we will compare the generated FEN with the expected FEN from our virtual simulation of the game. If they match, we know that the detection from our neural network is correct and we can proceed with our flow. If they don't match, it means our camera has difficulties in detecting which figures are on the board. That way, we don't change anything in our virtual simulation of the game. We can add more light or slightly adjust the position of the figures and try to detect them again. Hopefully, our results will then change, and our flow will advance to the next node.

3.4.4 Stockfish server segment

To get the best possible move based on our FEN string that describes the current chessboard state, we need to send a request to the Stockfish executable. Stockfish is written in C++ programming language, meaning we can't use it directly with Ocellus which is written in C#, and Node-RED which is based on Node.js. To resolve this issue, we will create a simple application written in Node.js for a chess server that will run locally within a Docker container. We will parse the incoming string from a GET request and search for 2 parameters. The first one is the FEN string, and the second one is the depth of optimal move search represented by an integer. We have to load the Stockfish library into our Node.js script and make a new request based on parsed parameters. When the Stockfish engine finishes its

calculation, the results will be sent as a response together with the depth parameter back.

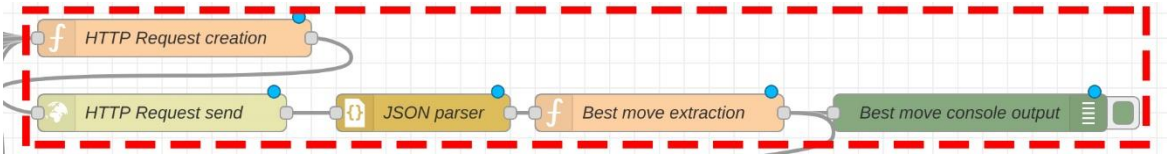

Figure 3.35: Stockfish chess server segment in the AutoCHESS Node-RED flow

Our function node will take the calculated FEN string and add it as a query parameter to the URL string together with the depth of search. For the chess server, we use port 9999 to avoid any conflicts with other applications that are present on our machine. The depth is randomly chosen from a range to spice up the chess game itself. Having the depth parameter always high will result in a mostly boring chess game. This way, robots can make certain mistakes in their tactics that will make the games unpredictable.

```
1  var FEN = msg.payload;
2
3  if(FEN != "DETECTION ERROR")
4  {
5      msg.url = FEN.replace(" ","+").replace("/","%2F");
6      msg.url = "http://localhost:9999/moves/?fen=" + msg.url + "&depth=25";
7      return msg;
8  }
```

Figure 3.36: HTTP request creation node code segment

The finished HTTP request string is sent to a HTTP request node that will carry the job of actually sending the request to our Docker container. A JSON node will parse the incoming message coming from the server and create a new JSON object out of it. That object will be forwarded to the next function node that will extract the best move out of the HTTP response. Another debug node is here just in case something goes wrong. The extracted string containing the move is sent in a new message to the next function node.

3.4.5 Move execution node

The move execution node is one of the function nodes with the longest code in our flow. Because of the size and many functions, it makes, it would be good to create a custom node type for this task. Due to the time constraint of this project, we will leave that for future iterations of this system and write a sizeable JavaScript code segment. We need to determine which move happened so we can create the correct movement pattern for the robot's arms. To do that, we have to separate the start and end chess positions first. Then, calculate the positions of those figures before determining which exact move happened. Castling requires a lot of conditions to happen, so the first major branch in our code determines if castling happened or not. For castling, Stockfish returns a king movement for two chess squares. That is exactly what we need to check to be sure that castling is currently happening. There are two types of castling moves, the kingside, and the queenside castle. To determine which one is happening, we need to check if the king is moving left in the original direction of the queen, or to the right side of the board. Lastly, both players can make the castling move so we need to check which player is the currently active one via the global variable we created.

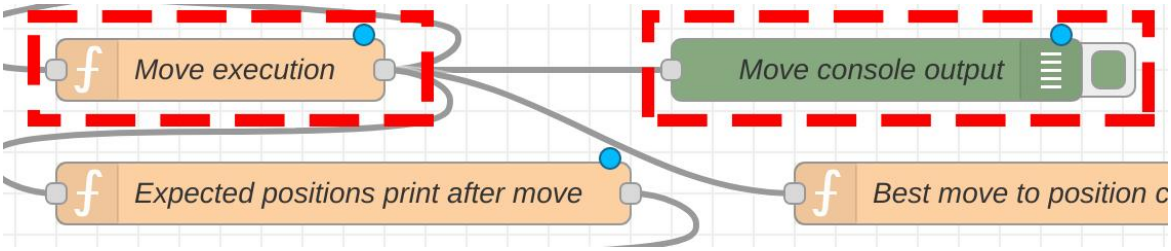

Figure 3.37: Move execution node in AutoCHESS Node-RED flow

If the move is not a castling one, we have four other options to precisely determine what is happening. If the pawn is moving diagonally on an empty chess square and the attack in passing parameter is equal to the end position of that move, we surely have an en-passant move ready for execution. On the other hand, if a pawn reaches the last row of the board without dying, that pawn needs to be promoted and a queen is spawned instead of the pawn. If we reached this part in the code, the move is not a special one and it can only be a normal move or an attack on a figure. We can easily determine that by checking the end position. If there is a figure on the end position, the move is an attack one.

```
// en passant
if(figure.split("_")[1] == "Pawn" && global.get("enPassant") !== "-" && figure2 == 'Empty_Slot' && CheckDiagonalMove(position1, position2))
{
    // brisi black pawn
    newPayload[1] = "EnPassant";
    var pos1 = CreateChessPosition(position1);
    var pos2 = CreateChessPosition(position2);
    var posForCalc =  pos2.split("")[0] + pos1.split("")[1];
    expectedBoardPositions[CalculatePosition(posForCalc)] = "Empty_Slot";
}

if(figure.split("_")[1] == "Pawn" && Math.abs(position2-position1) == 2)
{
    enPassantPosition = position1 - 8;
    var enPassantString = CreateChessPosition(enPassantPosition);
    global.set("enPassant", enPassantString);
}
else
{
    global.set("enPassant", "-");
}
```

Figure 3.38: Move execution code segment for attack in passing calculation

Now that we know the type of move that happened, we will wrap the result and the positions required for that move in a new message and send it to the next node. Before we advance, there are several things we need to update inside this node. Firstly, if we need to check castling in a separate function. If the rook or the king moved, the castling ability is removed, and a global variable is updated to reflect that. Next is the half-move clock, which we have to reset if a pawn figure moved, or a figure was attacked in the last move. If a pawn moved for two chess squares, we have to update the attack in passing global variable. The last thing we need to do, which is probably the most important one, is to refresh the expected positions array to reflect the new positions of the figures. This array is used to check the accuracy of our neural network.

3.4.6 Move positions calculation node

After we know what type of move we have to perform, we have to calculate the positions required to execute that move. Detected figures already have positional information, but apart from the normal move to an empty chess square, all other moves require a more complex robot movement pattern. For an empty move, we only need the start and finish positions that we have from detected figures. As for the attack move, we need an additional location where we are going to put our dead figure away. In order not to create clutter and chaos around the board by dropping the figures on random positions, we created a function that will neatly organize defeated figures around the board. We are going to use two columns on the left and right side of the board to store figures that are out of the game. We need additional global variables to track how many positions are already taken. The queen can

come back to the game via the pawn promotion special move, so we are going to store her on top of those two columns for easy access.

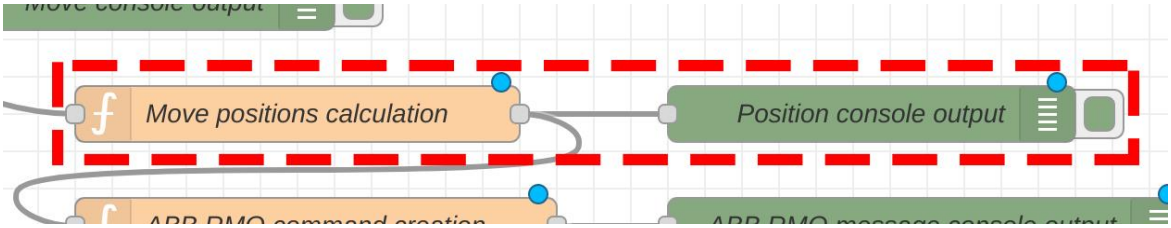

Figure 3.39: Move position calculation node in AutoCHESS Node-RED flow

Attack move requires one such location for a dead figure, completing the position set for that move. For castling moves, we need four total positions to execute it. We have two positions from detected figures, but the other two have to be calculated based on which player is the currently active one and which castling is happening. Since castling has a pre-defined movement, it is easy to calculate the other two positions in row 1 or in row 8. For attack in passing, we also need four locations, two standard from detected figures, one calculated from the positional info of detected figures, and finally one slot for a dead figure on the side of the board. Pawn promotion also needs four positions, but in that case, we need a slot for our pawn on the side, and we need the location of the dead queen resting on top of the two columns of dead figures. All this positional information is stored in an array for each move and sent us a new message to the next node.

3.4.7 RMQ command creation node

We arrived at the very end of our Node-RED flow where we need to construct an RMQ command that will be sent to the ABB RMQ node. As we described the RMQ protocol and its messages in the earlier chapter, commands are constructed with several key segments. The starting segment indicates that we are sending an EXECUTE type command which will trigger a code block inside RobotStudio. After that, commands can have up to 5 different arguments, and there can be an arbitrary number of empty arguments. Since different chess moves require different robot arm movements, the middle portion of the commands will vary depending on the type itself. Before listing all the positions for the arm to move, we need to specify the command name that will indicate which robot arm needs to move. We will designate the left arm to white figures and the right arm for the black ones. After

that, we need to add string segments constructed out of positions from the incoming Node-RED messages that are created in a separate function. Each segment will be one argument in the command.

```
function CreateRmqArgument(position)
{
    var string =
    '["pose","",0,"[[' + (position.x * 1000).toFixed(2) + ','
    + (position.y * -1000).toFixed(2) + ','
    + (position.z * 1000).toFixed(2)
    + '],[1.00,0.00,0.00,0.00]]","",0,""]';
    return string;
}
```

Figure 3.40: CreateRmqArgument function inside the RMQ command creation node that constructs the string that represents a command argument

There, due to the size limit of 444 bytes for each message, we are going to round up all the positional and orientational parameters to 2 decimal points. That way, we will have a command that is shorter than the limit determined by the web service that is still precise enough for our implementation. After we specified all 5 possible moves and their movement patterns, the finalized command with the metadata value of “8193” is sent to the ABB Robot RMQ node.

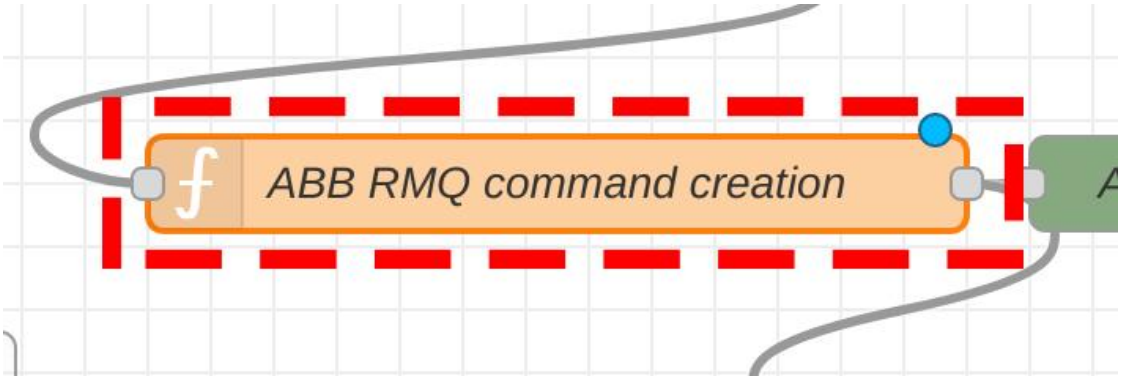

Figure 3.41: ABB RMQ Command creation node in the AutoCHESS Node-RED flow

3.4.8 ABB Robot RMQ node

ABB Robot RMQ node is another custom node developed by Byte Motion and it allows communication between Node-RED and the micro-controller inside the robot [24]. During testing, it will communicate with a virtual controller running inside RobotStudio software on a separate Windows machine. Inside the node, we have to configure the basic settings. We have to specify the local-host URL, username, and password for RobotStudio remote access and target queue name. If we configured the node correctly, and we are running an active simulation of the virtual controller in RobotStudio, we should see the “connected” label below the node itself. Our first implementation had messages that were larger than the 444-byte limit, so we broke the messages down into a few smaller ones. This node is critical for the third part of our system that closely works with the YuMi robot and ABB’s software package.

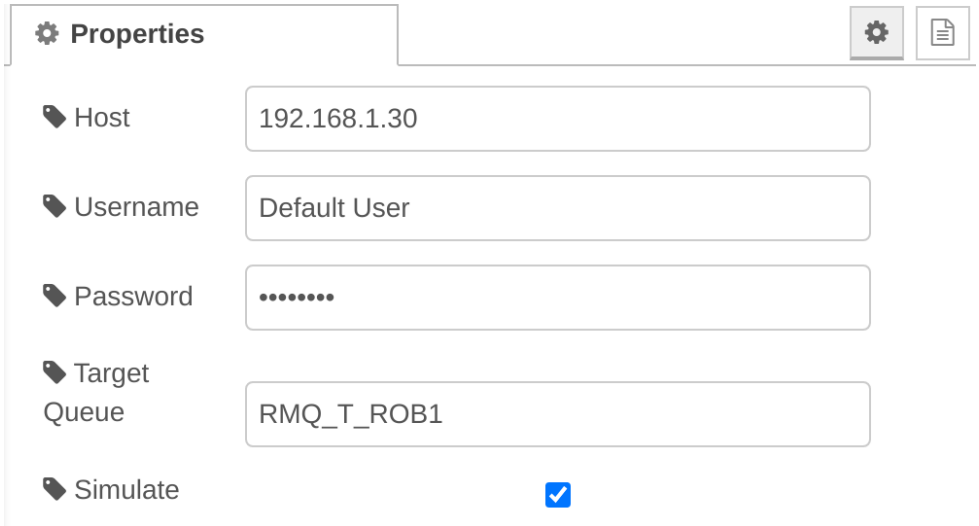

Figure 3.42: RMQ node properties panel at the end of the Node-RED flow

3.4.9 AutoCHESS system update section

The last few nodes are reserved for the initialization and update of global variables, and testing nodes that we used during the development of the flow. After the move is finished, the clock update node has to calculate the half and full move clock values that will be used again by the FEN creation node. We have to switch the currently active player global variable and we are ready for our next move. All the global variables we used throughout the whole flow are defined in a separate

function node dedicated only for that purpose. There we define all of the FEN parameters such as castling indicators, and clock timers. Moreover, we defined expected chessboard positions, dead figure locations, and a game end Boolean flag.

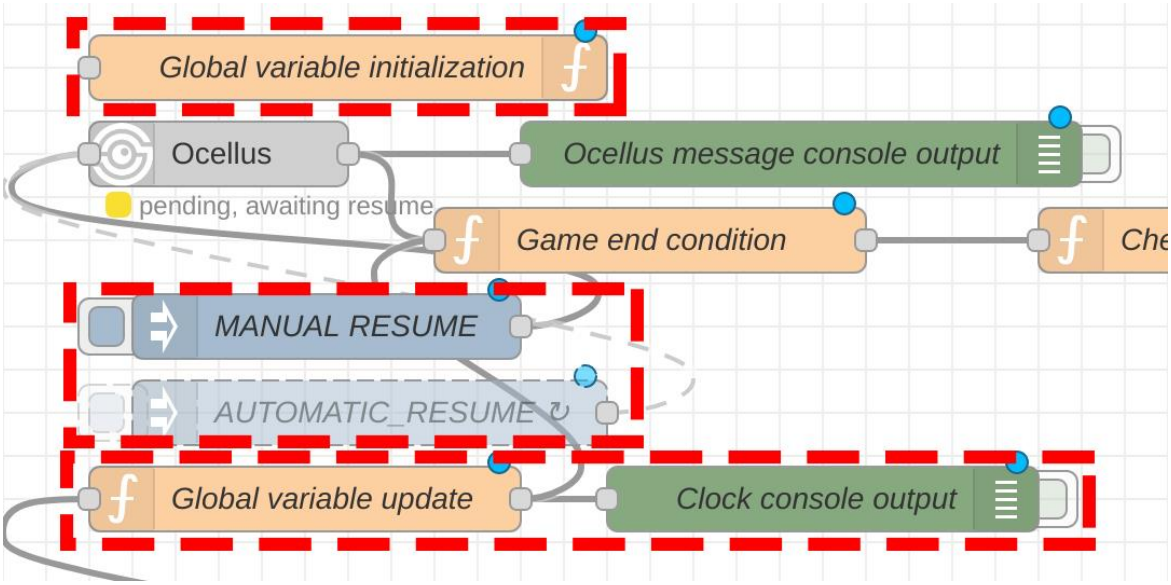

Figure 3.43: System update section in AutoCHESS Node-RED flow

When Ocellus runs in synchronous mode, we have to send him an OCELLUS_RESUME type message to refresh the detected figure positions. To do that, we will create an inject node that will send that message every 15 seconds. To combat possible errors, we will add another inject node that will be manually operated. Finally, we will make inject nodes for every move type to speed up testing and function validation during development. These nodes will manually inject a FEN string that is set up in a way that Stockfish will return that exact move we want to test. That completes our Node-RED flow, and we are now ready to see it in action.

3.4.10 Real-time Node-RED test results

To test both of the currently developed components of the AutoCHESS system, we need to run Ocellus the same way we described it in the previous chapter. Furthermore, we need a running Docker container that has all of the Node-RED dependencies and custom nodes that we downloaded from Byte Motion's private repository. We will turn on the debug nodes that will display information in the console window, namely the detected figure array, generated FEN, best move

output, position information for that move, and lastly, the generated RMQ command that we will send to the robot. After the camera calibration, we can deploy our Node-RED flow and observe the behavior. On the right side of the interface, we can see the console window output.

Figure 3.44: Node-RED console output after one chess turn displaying important information

We placed 13 figures for luck on the chessboard. We can see that based on the detected figures, Stockfish suggests an attack move on the black pawn. All the needed locations are properly calculated and the correct RMQ command is created.

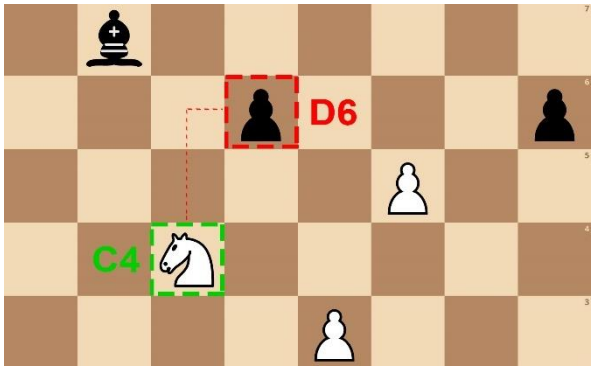

Figure 3.45: Attack move from C4 to D6 by the White Knight

Figure 3.46: Real-time RGB camera view of the detected figures

To test the expected FEN if condition, we will rerun the flow and deliberately cover up a part of the board. We can see in the console window that Node-RED reports a detection error. With that out of the way, we still need to test all the other available chess moves before advancing to the next step.

Figure 3.47: Obscured board with incorrect detection, Node-RED console output displaying a detection error on the top left

3.5 Dashboard

Our Node-RED flow became complicated after a while, so we decided to create a simple dashboard that will display information about the AutoCHESS system. It will consist of several information groups. Moreover, we will also add a bit of character to our robot arms and to the AutoCHESS system itself by creation audio messages that the robots will exchange between each other.

3.5.1 Dashboard section

The dashboard section of our Node-RED flow will be constructed out of special type nodes that we installed into our Docker container from the Node-RED repository. This shows the importance of public repositories and community collaboration when using Docker. Because of these features, we were able to create a working version of it in a matter of days. The core component of the dashboard is the dashboard

node. It offers many options for the general configuration of the dashboard such as layout organization, site information, size and spacing options, and theme and color configuration. Before we configure any of the mentioned options, we need to create elements that will be displayed on the dashboard. We went through the system and found 17 different parameters that are interesting to track. For each of them, we used two types of dashboard nodes, namely the text node and the gauge node. Inside each of them, we configured their size, shape, and appearance. To track the move history, as well as the chat history between robots, we used the table node that we again, manually downloaded from the public node repository. We specified the cell size and column content for each of them.

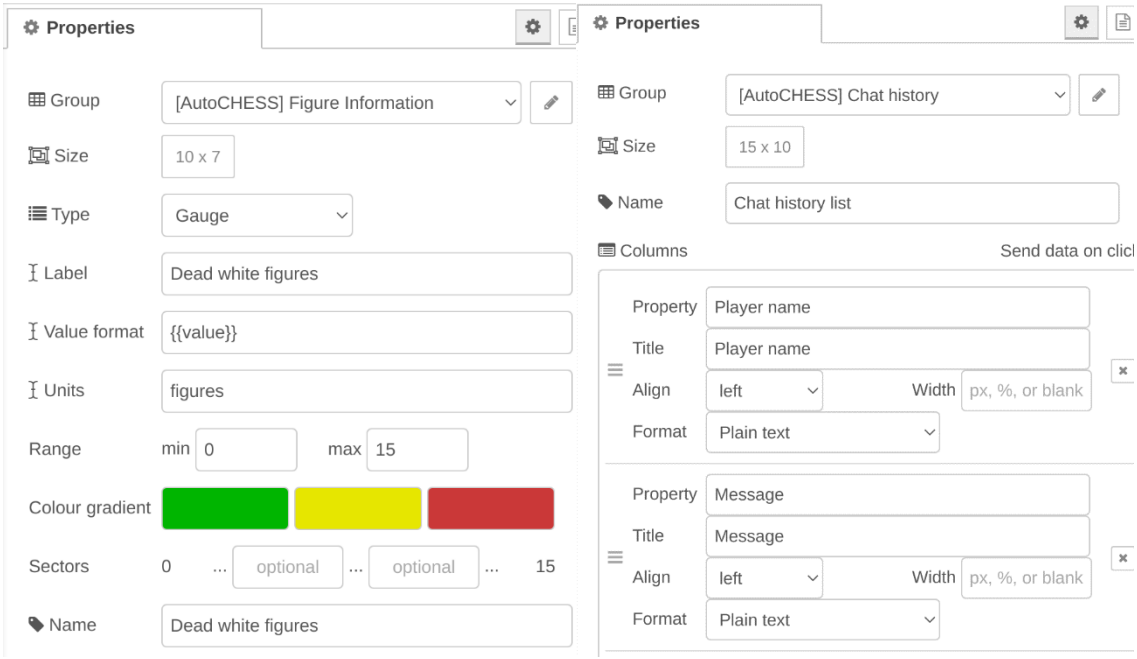

Figure 3.48: Properties window of the gauge node on the left, and the table node on the right

Now that we created all the dashboard nodes that we need, we somehow need to supply them with data that we want to visualize. For that, a large function node with 19 output slots will be added to the flow. The basic idea for fetching the data is to use global variables that are already used somewhere in the main part of the flow.

Some of the data we want to present is missing, so slight changes to the main flow will be added. We will not go into detail about these changes, since most of them boil down to a few extra lines that save a parameter into a global variable. These changes introduced several bugs and quirks that will have to be ironed out before the final release of the software. For the game duration parameter, we are going to

add an inject node that will supply our function node with a timestamp. A starting timestamp will then be subtracted from the dashboard one to calculate the number of seconds that have passed. Now, what is left to do is to convert these seconds into minutes and hours. This will be handled by a separate function. To match the visual appearance of the Ocellus web user interface, we sampled the colors that are used throughout the UI. Also, we arranged our nodes into groups and created a layout for them. The layout is responsive and can even fit on mobile phone screens.

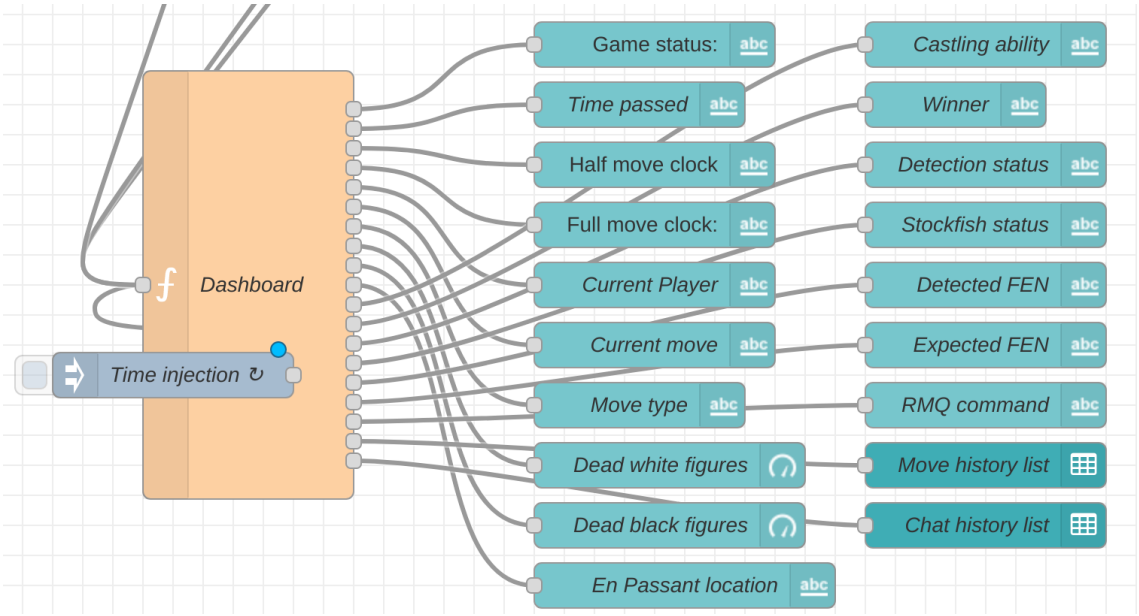

Figure 3.49: Dashboard segment of the Node-RED flow

3.5.2 Chess moves audio section

To make it more clear to the people that are watching an AutoCHESS match what move is currently being played, we added a simple audio constructor that will create audio lines based on the move information. This will add a bit of character to the AutoCHESS system. We created a synthesized voice line for each chess figure, chess location, and move type. We used a female voice that will contrast the male voices of the two robots. Each voice line consists out of 8 sections. Each section works in the same, though some parameters are changed from section to section. A sound preparation function node handles the input information and extracts a portion of it that will be added to the file path string.

We placed all of the sounds inside the Node-RED Docker container via a special flag that links a folder that is outside of the container. After the file path string has been finished, we send it to the file loader node that will read the .mp3 file and output a single buffer object.

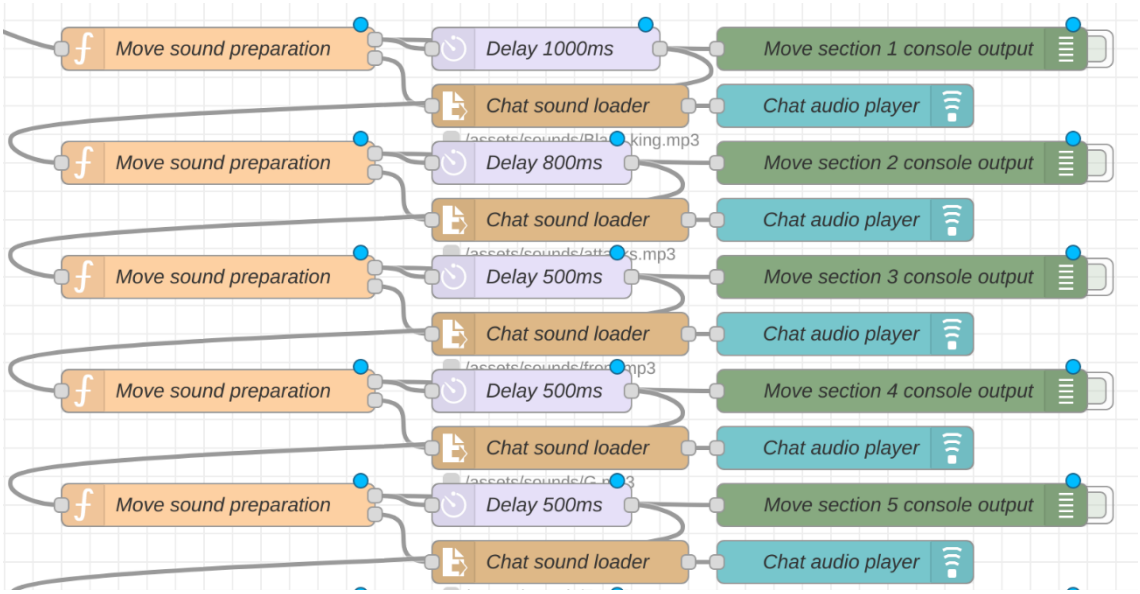

Figure 3.50: A part of the move sound section of the Node-RED dashboard flow

That object will reach the audio player node, and the section of the move audio will be played. To regulate the tempo of reading, we used the delay nodes and created believable pronunciation. After one section finished its audio playback, the next one starts until we reached the end of the chess move line. The input information is sent to the next section via the individual output from the function node. The move information will also be displayed in text format in the move history dashboard section.

```
if(global.get("gameEnded")==false)
{
    var object = global.get("moveAudioObject");
    var newMsg = {payload : object, topic : "Move audio"};
    var string = object["Figure 1"].split(" ");
    var string2 = "/assets/sounds/"+ string[0] + " " + string[1].toLowerCase() + ".mp3";

    var newMsg2 = {filename : string2, topic : "Move audio"};
    return [newMsg,newMsg2];
}
```

Figure 3.51: Sound preparation function node code snippet

3.5.3 Chat audio section

Similarly, we are going to add chat audio lines to each robot arm. We have written and recorded 50 lines for each robot. We chose two different male voices that are different enough to be recognizable. The sounds we recorded will be placed in the same folder as the move audio sounds. A function node chooses a random voice line that is played only 50% of the time in order not to go overboard with this feature.

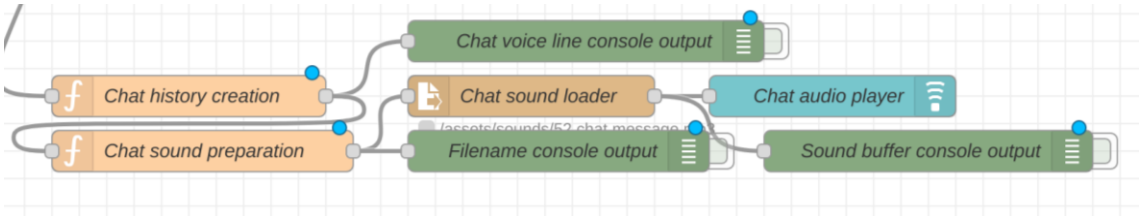

Figure 3.52: Chat audio section in the Node-RED Dashboard flow

That line is also written in the chat history. Again, a sound preparation function node creates a file path string that is being sent to the file loader node. The audio player node then plays the sound after the move has been executing. This way, we can create a funny and entertaining atmosphere where robots tease and make fun of each other. New voice lines can be added easily, we just need to write and record them, add them to the specified folder and then expand the voice line array in the corresponding global variable.

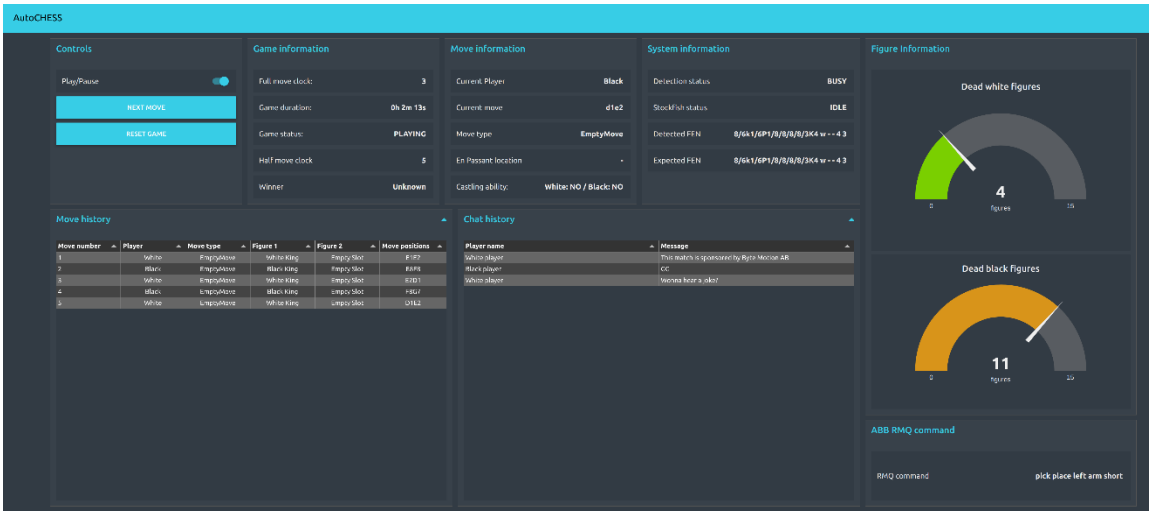

Figure 3.53: AutoCHESS dashboard view

3.5.4 AutoCHESS controls section

Lastly, to simplify the user controls of the AutoCHESS game, we added a pair of buttons and a toggle switch that will enable us to pause the game, advance to the next move, and reset the game. The pause switch controls a global variable that will disable the packages coming from the Ocellus node. Because other nodes are not getting the needed item list from the neural network, the main loop stays still and does nothing until we un-pause the game. The next move button sends an OCELLUS_CONTINUE type message that triggers the Ocellus node when working in synchronous mode. The reset button resets all the parameters of the game and resets the expected figures array to the starting one. With it, users can easily restart the game without tinkering in Node-RED itself. This section completes our simple dashboard.

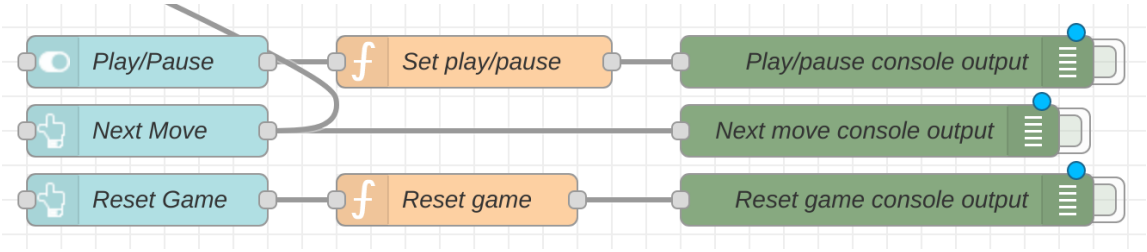

Figure 3.54: Dashboard controls Node-RED section

3.6 RobotStudio and RAPID

Before we start up RobotStudio, there are 2 configuration files located in the Appdata folder that we need to edit. Both of them revolve around the use of RobotStudio web services. We need to enable the remote virtual controller configuration and add a host IP address as well as a port for communication. After that, we need to create a new station with the IRB 14 000 preset and add a compatible virtual controller. Inside the RAPID tab, in the controller window, we have 2 separate RAPID program modules for each robot arm. The Functionality of both of them will be exactly the same. The only difference will be the procedure name which will indicate whose chess turn is it, the left or the right arm. Unfortunately, currently, we are not able to test this code, so possible bugs and errors exist in it. Also, robot and external joint configuration data will be set to 0. We would need some help from a robot engineer to set these parameters correctly. With that out of the way, let's start writing our RAPID modules.

The commands we are sending to RobotStudio have 3 or 5 arguments inside them. This means we need to write 2 separate procedures, one for the short command version and one for the longer one. Both procedures will do exactly the same thing, with very similar code. The only place where they will differ is the length of the target array and the accompanying for loop that will execute movements of the arm. Here, we will only explain the long version of the command with 5 arguments. To start, we will initialize two arrays of type robtarget. Robtarget is a composite data type consisting of multiple standard or other composite types of data.

```
FUNC robtarget CreatePseudoRobotTarget(pose point)
    RETURN [point.trans, point.rot, [0,0,0,0], [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9]];
ENDFUNC

FUNC pose CalculateMiddlePosition(pose point1, pose point2)
    VAR num xCoord := (point1.trans.x + point2.trans.x)/2;
    VAR num yCoord := (point1.trans.y + point2.trans.y)/2;
    !adding 12cm of headroom to the transition position
    VAR num zCoord := (point1.trans.z + point2.trans.z)/2 + 120;

    VAR num aQuaternion := (point1.rot.q1 + point2.rot.q1)/2;
    VAR num bQuaternion := (point1.rot.q2 + point2.rot.q2)/2;
    VAR num cQuaternion := (point1.rot.q3 + point2.rot.q3)/2;
    VAR num dQuaternion := (point1.rot.q4 + point2.rot.q4)/2;

    RETURN [[xCoord, yCoord, zCoord], [aQuaternion, bQuaternion, cQuaternion, dQuaternion]];
ENDFUNC
```

Figure 3.55: Auxiliary functions for targes and middle position creation

The first two types describe the position and orientation of the target. The position consists of 3 num types that describe the position of the target in all 3 axes, namely the X, Y, and Z-axis. The orientation is represented by 4 num types that constitute a quaternion. The third type is confdata type. Confdata represents the axes configuration of the robot. When calculating the corresponding axis positions, there will often be two or more possible solutions. This means that the robot can reach that target with a few different axis configurations, and with this parameter, we specifically pick one of those configurations. The last type called extjoint is used to define the connections of logical and physical axes. If we don't want to use one of them, we are going to disconnect it with the "9E9" value inside the data type.

Once we created the arrays, we need to create robtarget type data to store in them. Our arguments that come from RMQ messages have positional and orientational data. This means that the configuration data as well as the external joint data is missing. Since we don't have enough expertise in the field of robotics, we will leave

these two parameters empty. That way, we can present a pseudo-code version of the module and leave room for future versions of the module that will have these parameters correctly set. To organize the code, we moved the robtarget creation into a separate function. Once we created all 5 targets from the supplied arguments, we need to create transitional positions between them. If the robot for example would grip a chess figure, and then just move to the next designated position, it would knock off many other figures on the board. This means that we need to lift the arm with the gripper a bit to avoid such problems. To do that, we will create transitional positions between each of the targets. A separate function will calculate the middle position and orientation for each pair of targets while raising the Z-axis by 120mm to avoid collisions. What is left to do is to put the targets and the calculated transition targets in the correct order. A for type loop will iterate over each element of the target array, execute a linear move command, and pause the execution of the procedure for 0.5 seconds after each element. Inside the linear move command, we need to specify the speed, the precision level, and the tool we are using. That concludes our procedure that will be executed each time an RMQ message is sent from Node-RED to RobotStudio. We can just copy the code from the left arm and change the procedure names for the right module since we expect both of the arms to work in the same way. Even though we don't have a tested code, we laid a good foundation for future work on the RobotStudio component.

```
PROC pick_place_left_arm_long(pose point1, pose point2, pose point3, pose point4, pose homePoint)
    VAR robtarget middleTargets{5};
    VAR robtarget targetList{10};

    VAR robtarget target1 := CreatePseudoRobotTarget(point1);
    VAR robtarget target2 := CreatePseudoRobotTarget(point2);
    VAR robtarget target3 := CreatePseudoRobotTarget(point3);
    VAR robtarget target4 := CreatePseudoRobotTarget(point4);
    VAR robtarget target5 := CreatePseudoRobotTarget(homePoint);

    middleTargets{0} := CreatePseudoRobotTarget(CalculateMiddlePosition(point1, point2));
    middleTargets{1} := CreatePseudoRobotTarget(CalculateMiddlePosition(point2, point3));
    middleTargets{2} := CreatePseudoRobotTarget(CalculateMiddlePosition(point3, point4));
    middleTargets{3} := CreatePseudoRobotTarget(CalculateMiddlePosition(point4, homePoint));
    middleTargets{4} := CreatePseudoRobotTarget(CalculateMiddlePosition(homePoint, point1));

    targetList := [middleTargets{4}, target1, middleTargets{0}, target2, middleTargets{1},
                   target3, middleTargets{2}, target4, middleTargets{3}, target5];

    FOR i FROM 0 TO 10 DO
        MoveL targetList{i}, v100, fine, gripper;
        WaitTime 0.5;
    ENDFOR
ENDPROC
```

Figure 3.56: The left arm procedure for 5 input arguments

4. DEPLOYMENT

In order to properly prepare the deployment, there are several things we need to test and develop before we can try out AutoCHESS in real life. For the test, we are going to use the newly created CRTA robot facility as a part of the Faculty of Mechanical Engineering and Naval Architecture in Zagreb. Since there are many active projects and people working at that facility in the same time, due to the time constraints of this project, we may or may not be able to use it. Nevertheless, preparing a part of the solution for the future development of AutoCHESS is a good way to go.

Figure 4.1: CRTA robot facility at FSB Zagreb

4.1 YuMi robot evaluation and location test

A good place to start is an appointment with robot specialists at CRTA and an initial discussion of the project as a whole. The IRB 14 000 is mounted on a modular rails system in a horizontal position. We will need some sort of a chessboard reinforcement that will fit on the rails and make the board stable. We turned on the YuMi robot and flipped it into manual free movement mode on the Flex Pendant. Already, we are coming across a problem that we haven't predicted in our initial plan for the project. The maximum horizontal working range of the arms is 406.8mm as stated in the ABB documentation. Robot engineers noted that the number is

probably not realistic when working with the robot in real-life, since that working range can only be reached in perfect conditions and joint configuration.

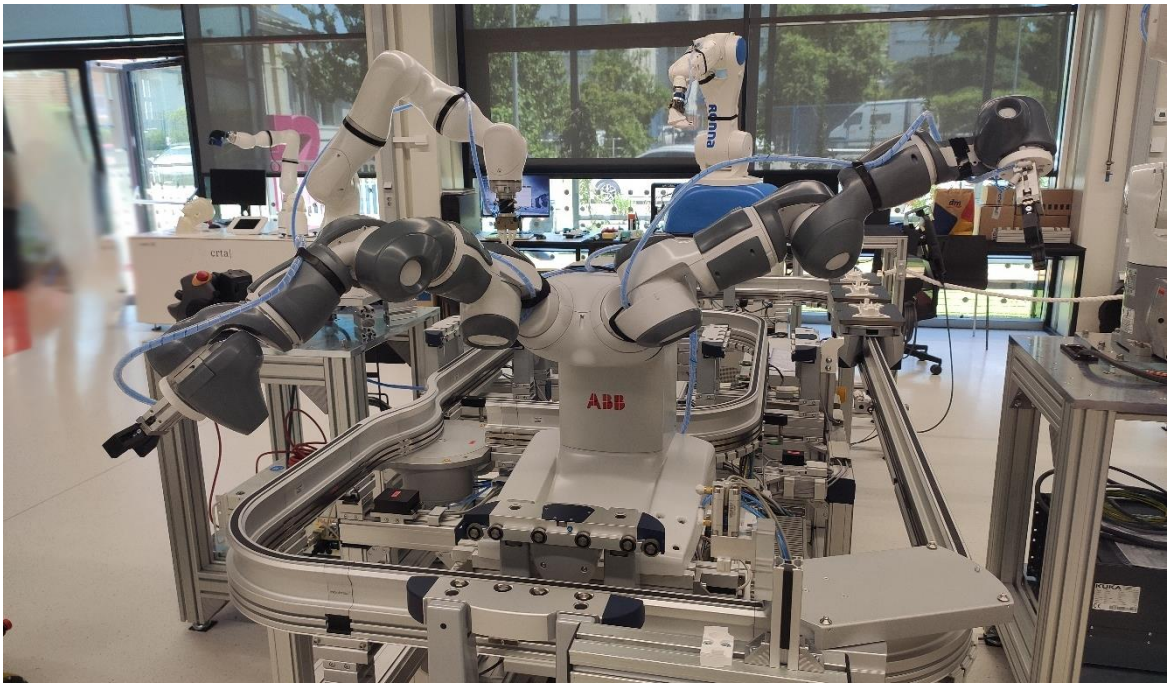

Figure 4.2: IRB 14 000 horizontally mounted on a modular rails system

Our board is 490mm long, which means that the robot will not be able to reach the entirety of the board without a special solution. Mounting the YuMi on the wall would help, but unfortunately, it is permanently fixed in this position. Another solution would be to use magnets and mount the chessboard and the figures vertically on a metal sheet. This option is interesting but requires modification of the chess figures, magnets, and several other components which we simply can't acquire to finish in time.

Another more complex solution would be to mount the chessboard on a system powered by linear motors, that would move the correct part of the board in YuMi's working range. For obvious reasons, this solution is not feasible at the moment. What we can do, for demonstration purposes only is to select a few chess moves within the YuMi's working range that are possible to do. This will drastically reduce the time and complexity of the final step.

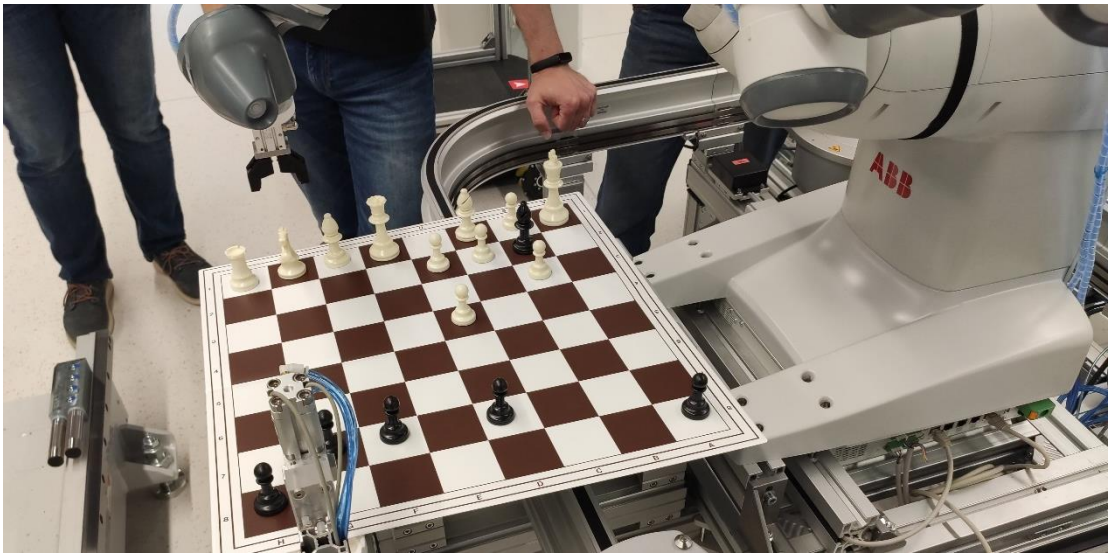

Figure 4.3: Chess figure reachability testing

4.2 Gripper design and manufacturing

To execute chess moves, our robot will need to grip the figures first. Grippers come in various shapes and sizes, but mostly they need to be custom made to fit the task at hand properly. Official ABB servo two-finger grippers cost upwards of 5000 euros, way too expensive for our project. CRTA has its own hydraulic gripper system that is currently fitted to the YuMi arms, the end sections expectedly do not fit our use case.

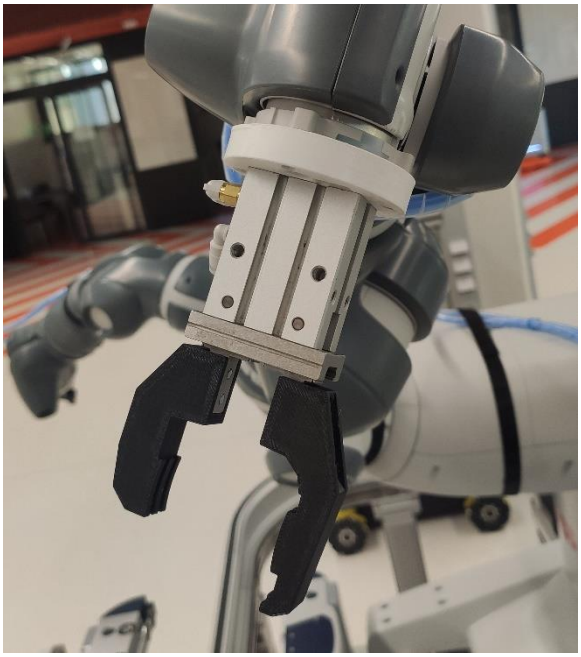

Figure 4.4: Two-finger gripper developed by CRTA engineers

These grippers do not have advanced pressure sensors and cameras so adapting them to each chess figure is going to be tough. Chess figures by themselves vary quite a lot in terms of dimensions and overall shape. In the picture below, we can see that the biggest figure is more than twice the size of the smallest one.

Because of that and the contour shapes, the gripper fingers have to accommodate multiple figures. When talking with the robot engineer at CRTA, we settled on classic clamp style gripper, that will be able to pick up both the smaller and the larger chess figures. The top portion of the gripper will catch the smaller figures, while the bigger ones will be caught deeper in the fork. The first step in the design process is to measure all the figures and figure out the optimal dimensions for the clamp. The gripper design was made in Solidworks, a 3D design tool perfect for creating our chess figure gripper.

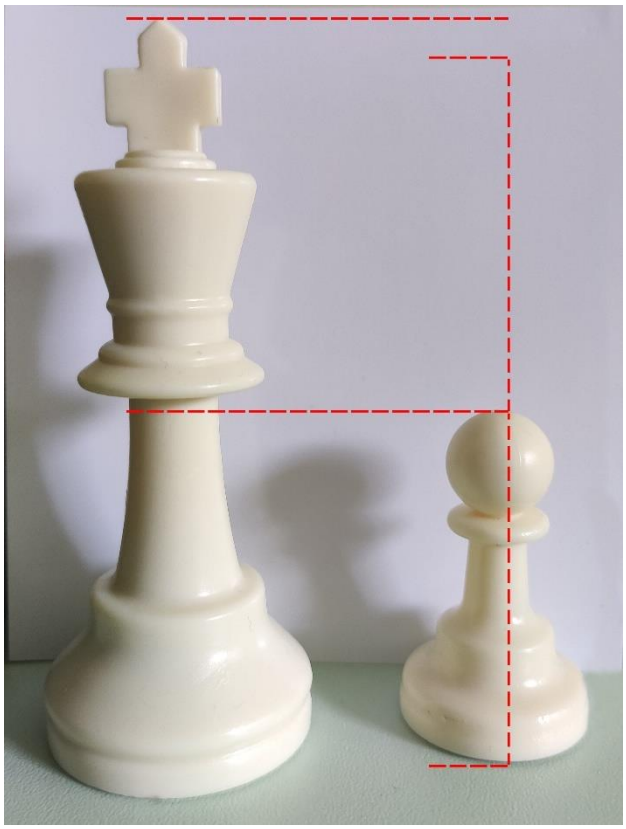

Figure 4.5: Chess figure size difference

Grippers will be printed out of ABS plastic on a FDM (Fused deposition modeling) 3D printer. The Stratasys F170 model that we are going to use has a respectable accuracy of +/- 0.2mm. Printing will take about 2 hours to finish. Each gripper finger has about 10mm of stroke. That will be plenty for our figures that have an average base diameter of 33mm. On the image to the right, we can see the shape and size of the gripper fingers that will be mounted to YuMi's arms.

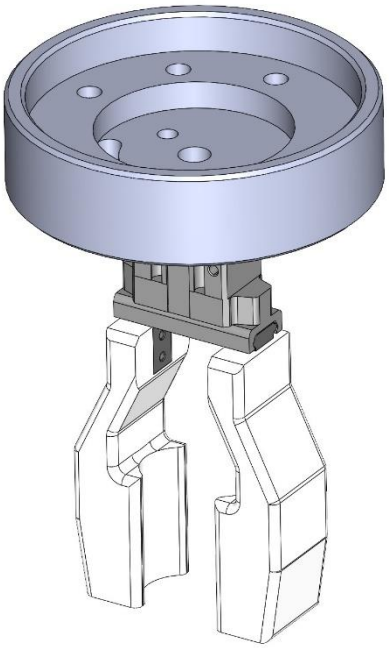

Figure 4.6: Chess figure gripper design in Solidworks

4.3 Pick & place simulation

During the location evaluation, we came across numerous problems regarding the mobility of YuMi's 7-axis arms. We already know that they won't cover the whole chessboard. Because of that, we will create a new station inside ABB's

RobotStudio, and test out pick & place operations on a simplified model. Neither the gripper nor the 3D models are precise enough for a real-world equivalent test, but since the real YuMi is currently unavailable for lengthy testing sessions, we must be satisfied with this simple simulation. Inside RobotStudio, after we have placed YuMi and selected its compatible virtual controller, we need to add 3D models of the chessboard and the chess figures. To do precise 3D models that are equivalent to the real-world ones would take a lot of time which we, unfortunately, do not have. We will measure each individual figure and approximate them with equivalent rectangles that will represent each chess figure. We choose a simple board configuration with only a handful of figures placed on the board. For each figure, we need to add targets on top of the 3D model that will represent the gripping point. We also need to add a gripper to the robot model, our custom one doesn't have a 3D model yet, so we will use the official ABB servo two-finger grippers. Out of the lucky 13 figure positions, the left arm can reach only 6 of them. The right arm however scores much higher with 11 total positions that are reachable. This means that a real-world implementation would have even more problems due to the joint transitions between configurations. YuMi's arms have a very complex movement pattern due to the 7 axes that are placed differently than on standard-looking robot arms like the IRB-120. This test affirmed our suspicions that a truly working real-world test without significant modifications to the robot setup would be impossible to do. We will take this into account when testing the movements of the figures. Future versions of this system will need to find a proper solution for this issue.

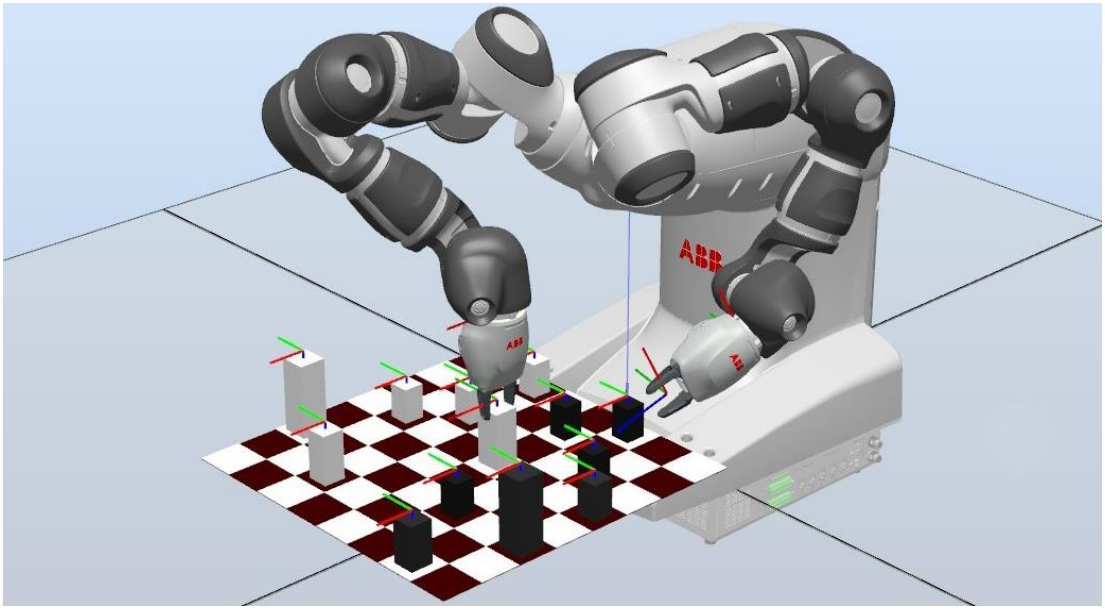

Figure 4.7: Pick & place simulation inside RobotStudio

4.4 Future hardware and software setup

Unfortunately, the CRTA robot facility is completely booked for the rest of the remaining time to finish this thesis. This means we are not going to be able to test AutoCHESS with a real YuMi robot. Still, we reached the point where we can lay out a plan for a future test. The next section describes the planed preparation and setup of the AutoCHESS system. YuMi is conveniently placed at a corner station, meaning we can easily add other components around it. We will reinforce the chessboard with a thick piece of cardboard so it can be mounted on the rail system in front of the robot. A small LED light will be added to the setup to supply enough light for the neural network. Next is the RealSense camera connected to our PC running Ubuntu Linux. We also need another PC for running RobotStudio and communicating with the robot's microcontroller. The custom gripper will be mounted on YuMi's arms. The Ubuntu PC will be running Ocellus, the AutoCHESS neural network, the local chess server, and the Node-RED flow. On the other hand, our laptop running Windows needs only to run ABB's RobotStudio. A robot engineer will help us with powering on the robot as well as configuring everything via the Flex Pendant mounted on the left side of YuMi. Once we will be up and running, the newly created dashboard will help us identifying possible problems along the way. The test will take place in a few weeks, so a basic plan will be of great use.

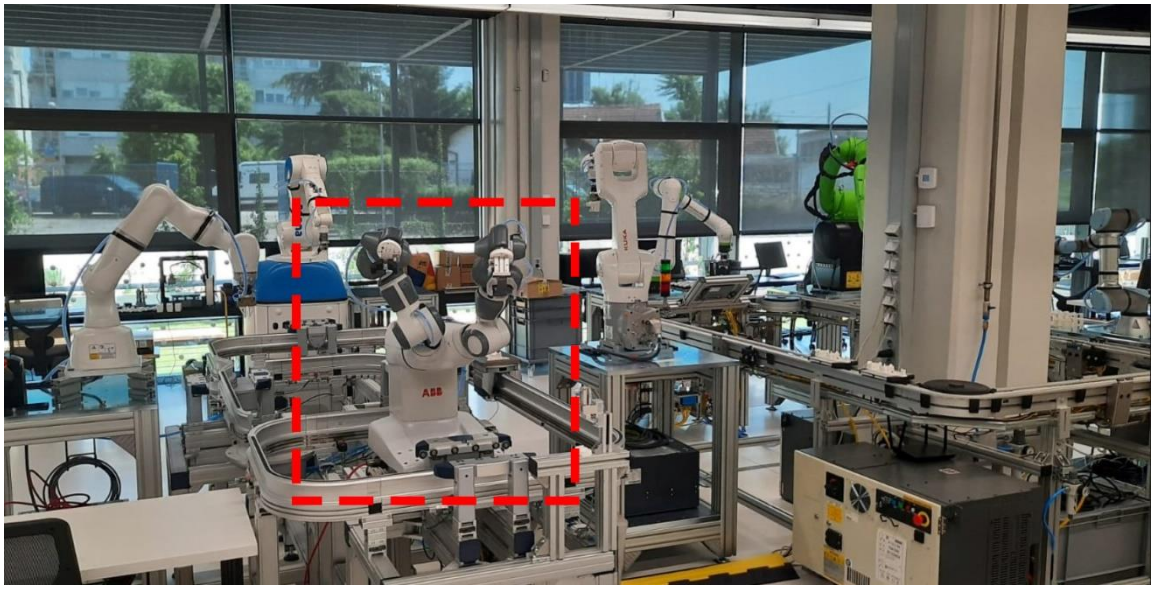

Figure 4.8: YuMi location at CRTA

4.5 Test session with human players

Since we were not able to test our system in full capacity, we are going to create a smaller scope test to try out our system without the robot section. To do that, we will use the hardware and software setup from the previous chapter.

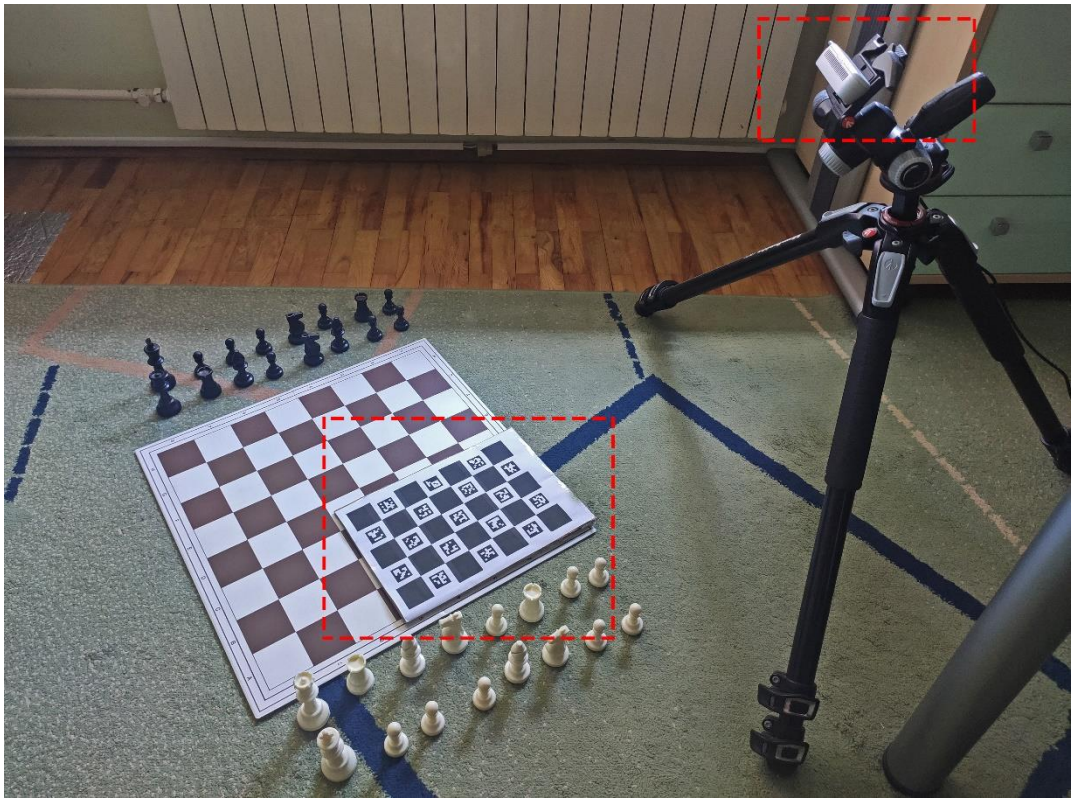

Figure 4.9: Camera calibration with the ChArUco board

We settled on a mid-game scenario where both player lost half of the figures. To aid the detection, we spaced out the figures evenly across the board. A small LED light was placed on top of the board to reduce the amount of chess figure shadows.

Figure 4.10: Chessboard configuration, 2D image on the left, 3D image on the right

We started the test and observed the behavior of the system. We executed 12 different chess moves with varying degrees of success. The main problem was the dynamic sunlight coming from the windows. Because of it, on several occasions, we had figure detection issues that we manually had to resolve. Changing the position of the LED light as well as slight position and orientation changes to the figures helped out during the test. This only shows that we have to improve on the neural network performance for future use.

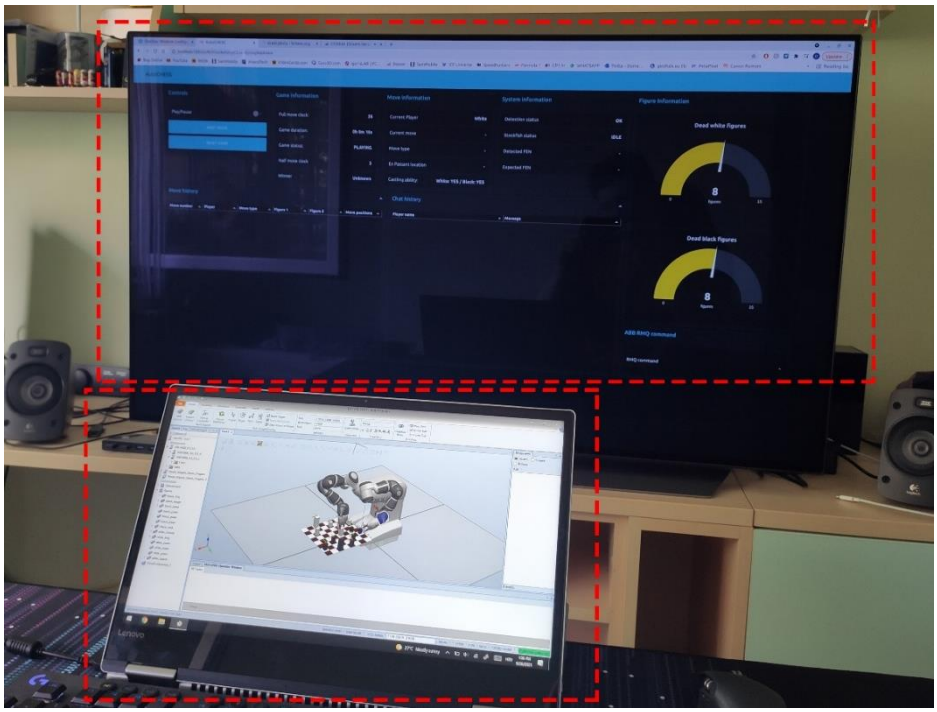

Figure 4.11: AutoCHESS dashboard running in the background, RobotStudio running on the laptop

This is the first time we are properly testing the dashboard component. The information that it provides is very helpful during runtime since we can easily discover and fix issues and bugs in the system. The audio messages give AutoCHESS some character and a semi-human face.

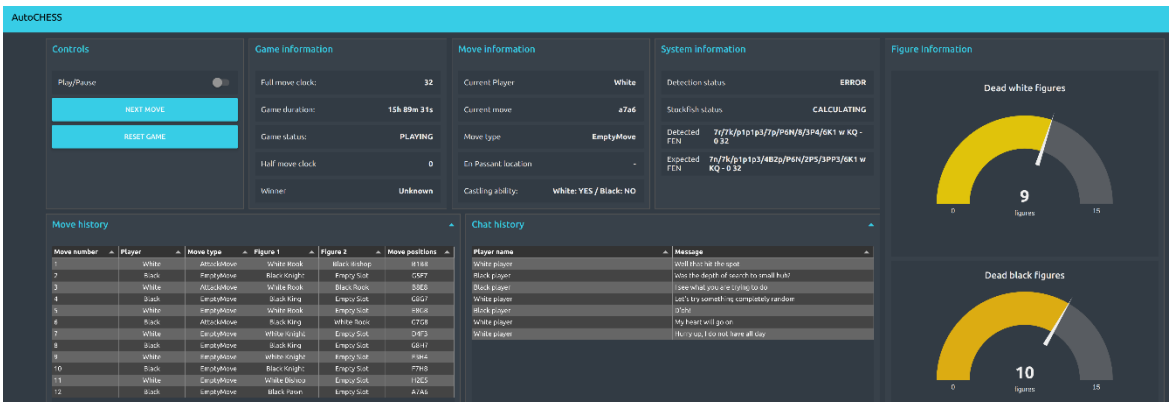

Figure 4.12: Dashboard status after the test

We can examine the details of the chess move history and read through the funny comments in the chat history. The dashboard is also accessible on a mobile phone as seen on the image to the right. Scaling and responsiveness of the elements is not perfect when some of the groups have manual sizing enabled. We would need to make two different dashboard versions, one for the desktop, and one for mobile devices to properly cover different screen sizes.

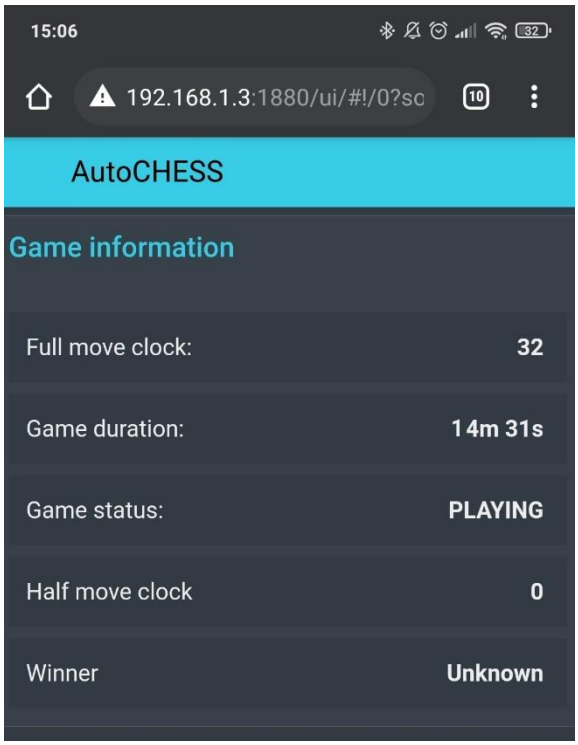

Figure 4.13: AutoCHESS dashboard running inside a mobile phone browser

5. RESULTS

To properly evaluate our AutoCHESS system, we will break it down again into components, asses each of them individually.

5.1 Dataset generation results

Creating datasets for the training of the neural network proved to be a difficult and time-consuming task. The results of our neural network detection precision tell us we did a good enough job with it. Annotating figures with the polygonal tool has to be done precisely. We spent around 10 minutes with each photograph annotating 12 figures. To do it in a more precise manner would take more than double the time we spent on it. In addition, we need much more real photographs, probably in the neighborhood of a few thousand. For that amount of work, we would probably use a remote annotation team to do the work for us. These days, a lot of people are doing annotation on-demand, meaning the price we would pay for the job wouldn't be too expensive. Future dataset versions will rely on outsourcing, to save time that can be used for something more important.

5.2 Neural network results

Chess figure detection during the development and deployment has shown strong results. The reliability of the correct class labeling, and contour detection was always above 90% meaning we could rely on it for our short duration tests. This is one of the components of this system that can be easily upgraded and updated in the future. For perfect operation and use, the precision will need to reach more than 99%. We simply need more data, which needs to be annotated and mixed in with datasets that we already have. We have laid a strong foundation in PyTorch and found good configuration parameters that will help us in the future versions of this system. The process though very complex is now easily repeatable on different datasets. Creating new versions wouldn't take more than a single day including training time, which is great if we want to slightly tweak the detection of specific figures that cause more trouble than others.

5.3 Node-RED results

Node-RED is envisioned as a communication backbone between the robot's microcontroller and the Ocellus vision system. Even though it is primarily used in the Internet of Things space, it showed us that simplicity doesn't always need to restrict functionality. We had all the tools we needed to build our flow exactly the way we wanted. For specific sections, we used custom function nodes with plain JavaScript inside of them. Some of them were more than 200 lines in length meaning we would be better off with developing separate custom nodes for some of them. Since we had a limited development time, we choose not to, but converting our flow to the one with a few new custom nodes shouldn't take much time in the future. Byte Motion AB did an amazing job with their custom Ocellus and ABB Robot RMQ nodes that worked flawlessly and provided us with the much-needed communication bridges between the components of our system. Further code optimization within the flow is also possible with performance and reliability as main goals for future iterations.

5.4 Dashboard results

The Node-RED dashboard was created at the last minute. Even though we had little time to make it work, the end result is satisfactory. Through theme and color settings, we were able to match the appearance of it to the Ocellus web user interface. Now these two services look like the belong to each other. We also added the audio voices to each robot arm and to AutoCHESS itself. The voice lines give some much needed humor and fun to an otherwise very dull system. There are still some bugs here and there, but it works well enough for future use in testing.

5.5 RobotStudio and YuMi results

The last component is probably the only one that lacks the refinement seen in the previous ones. We encountered several physical hardware problems when working with YuMi. Moreover, the availability of working with YuMi was severely reduced to a minimum. There are many active projects on it, and ours is just one of them. The

CRTA robot facility also had its grand opening on the week we needed the robot and prevented any work on it. Robot engineers from Robot Norway were also very busy during that time, so any potential help from them was also reduced. Even though we had all of these difficulties when working on the last component of AutoCHESS, we made some prep work for future work on this project. During the simulation and testing phase, we concluded that a more traditional robot like the IRB 120 from ABB would be much more suitable for this application. YuMi is specialized in small electronics assembly, which mostly happens within a smaller-size confined area in front of the robot. Our chessboard proved to be too big of a playground for the horizontally mounted YuMi, making the whole process difficult. This component needs a proper solution to the range problem. Also, it needs much more time for testing and developing RAPID routines, joint configuration, gripper quality, and move execution. Future versions of AutoCHESS definitely need to focus on this component first, since this is the weakest of the bunch.

6. CONCLUSION

In conclusion, the AutoCHESS project proved to be large and demanding. We had to combine a wide range of technologies, services, and platforms in order to create a working prototype of the system. Most of its developed components worked very well. The chess figure detection, 3D space positioning, and Node-RED flow were extensively tested with great results. The only component that lacked certain functions was the YuMi robot itself, and its implementation and configuration in RobotStudio. We simply didn't have the time to finish the project within the given deadline if we consider the availability of the YuMi robot itself. Even though we haven't fully finished the whole AutoCHESS system, we have learned a lot during its development. The most important conclusion we can draw is that game-engine-based robot control systems have a very bright future in front of them. Ocellus and its advanced AI functions proved to be highly effective. The performance, ease of use, and great compatibility with other platforms positioned Ocellus as a must-have component of next-generation smart robot systems. The use of Node-RED, as a communicational bridge between the robotic hardware and Ocellus, brings a totally new approach for organizing smart factories, assembly lines, and advanced AI-driven robot systems. The whole system is already used for several projects with varying applications. AutoCHESS serves as a good demonstration of what is possible with Ocellus, and other similar game-engine-based robot control systems. In the end, we also learned a lot about AI-driven robot systems. That knowledge and experience we collected will help us in future work and cooperation with Byte Motion AB on AutoCHESS, and other interesting projects and implementations of the Ocellus vision system.

BIBLIOGRAPHY

[1] Unity team, Unity User Manual 2020.3 (LTS), 4.6.2021. , last accessed on 28.4.2021.

https://docs.unity3d.com/Manual/index.html

[2] Unity team, Learn Unity, 2021. , last accessed on 30.4.2021.

https://unity.com/learn

[3] Hocking, Joseph, Unity in Action, Manning Publications, 2015.

[4] Intel, Intel RealSense D400 documentation, Revision 010, February 2021. , last accessed on 14.5.2021.

https://www.intelrealsense.com/download/15409/?_ga=2.201292840.727822122.1623675656-171405354.1621343721

[5] Intel, Intel RealSense SDK documentation, 2021. , last accessed on 16.5.2021.

https://dev.intelrealsense.com/docs/sdk-knowledge-base

[6] Ivan Jurin, System for hand detection and tracking using depth camera, Undergraduate thesis, Faculty of Electrical Engineering and Computing, University of Zagreb, 2015.

[7] Branimir Rudec, Control of Bionic Hand Fingers using Leap Motion Capture Sensor, Undergraduate thesis, Faculty of Electrical Engineering and Computing, University of Zagreb, 2018.

[8] Byte Motion AB, Ocellus Vision System documentation, 2021. , last accessed on 20.6.2021.

https://github.com/tjonsson/ocellus

[9] Byte Motion AB, Ocellus Data-Science Experiments documentation, 2021. , last accessed on 23.6.2021.

https://github.com/tjonsson/data-science-experiments

[10] Maja Vidošević, JavaScript interface for robot control, Undergraduate thesis, Faculty of Electrical Engineering and Computing, University of Zagreb, 2016.

[11] Iva Jutriša, Web based user interface for unmanned ground vehicles, Undergraduate thesis, Faculty of Electrical Engineering and Computing, University of Zagreb, 2019.

[12] Wikipedia, Chess, 9.6.2021. , last accessed on 16.4.2021.

https://en.wikipedia.org/wiki/Chess

[13] Wikipedia, Stockfish (chess), 5.5.2021., last accessed on 17.4.2021.

https://en.wikipedia.org/wiki/Stockfish_(chess)

[14] Docker team, Docker overview, 2021. , last accessed on 20.5.2021.

https://docs.docker.com/get-started/overview/

[15] Denis Drozdov, What is Supervisely, March 2021. , last accessed on 21.5.2021.

https://docs.supervise.ly/

[16] Tom Allen, What is machine vision?, 20.6.2020. , last accessed on 25.5.2021.

https://aijourn.com/what-is-machine-vision-everything-you-need-to-know/

[17] Rohith Gandhi, R-CNN, Fast R-CNN, Faster R-CNN, 9.7.2018. , last accessed on 25.5.2021.

https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e

[18] Dive into Deep Learning, Region-based CNNs (R-CNNs), 2021. , last accessed on 25.5.2021.

https://d2l.ai/chapter_computer-vision/rcnn.html

[19] Jonathan Hui, Understanding Feature Pyramid Networks for object detection, 27.3.2018. , last accessed on 25.5.2021.

https://jonathan-hui.medium.com/understanding-feature-pyramid-networks-for-object-detection-fpn-45b227b9106c

[20] Torch Contributors, PyTorch Documentation, 2019. , last accessed on 20.5.2021.

https://pytorch.org/docs/stable/index.html

[21] OpenJS Foundation, Node-RED User Guide, 2021. , last accessed on 28.5.2021.

https://nodered.org/docs/user-guide/

[22] ABB, ABB YuMi – IRB 14 000, 2021. , last accessed on 13.6.2021.

https://new.abb.com/products/robotics/collaborative-robots/irb-14000-yumi

[23] Ivan Odak, Demonstration of Pick and Place Operation with 3DOF Cartesian Robot, Undergraduate thesis, Faculty of Electrical Engineering and Computing, University of Zagreb, 2018.

[24] Byte Motion AB, Node-RED contribution – ABB robot - documentation, 2021. , last accessed on 24.6.2021.

https://github.com/byte-motion/node-red-contrib-abb-robot

[25] ABB, RobotStudio Operating manual, 2021. , last accessed on 25.6.2021.

https://library.e.abb.com/public/244a8a5c10ef8875c1257b4b0052193c/3HAC032104-001_revD_en.pdf

[26] Nikola Jerković, Planning and Tracking Mobile Robot's Path Based on Given Target and Environment View with 3D Camera and Laser Scanner, Undergraduate thesis, Faculty of Electrical Engineering and Computing, University of Zagreb, 2018.

LIST OF FIGURES

Robot control based on video information on a chess game case study

Abstract

Industrial automation and the use of artificial intelligence in robotics is one of the key focus points of Industry 4.0. Currently, systems used around the world are mostly purpose-built, which means that their reuse in other applications is difficult and time-consuming. To make universal solutions, that are easier to develop, we must construct a system out of well-known programming languages and platforms. Ocellus, developed by a Swedish firm called Byte Motion, is one such system. It is developed in a game engine called Unity and it supports a wide range of hardware and software components. In this thesis, we are going to use Ocellus and its advanced artificial intelligence modules to create a system shorty called AutoCHESS. AutoCHESS is a fully automated system that simulates a game of chess played by two robot arms against each other. With the help of a RealSense depth camera, a neural network trained to detected chess figures will supply information to Ocellus that will calculate their real-world positions. This information will be transferred via Node-RED, a visual programming tool that is mostly used on IoT devices. The moves of the chess game will also be automated and calculated by the currently best open-source chess engine called Stockfish. Constructed RMQ protocol messages will be sent from Node-RED to ABB's RobotStudio for processing. RobotStudio will then execute procedures that will result in movement of the robot arms that will intermittently complete chess moves until one of the players wins the game. The finished system will serve as a demonstration of the capabilities of game engine-based robot control systems as well as a showcase of the use of advanced artificial intelligence algorithms in industrial robotics.

Keywords: Unity, Ocellus, robot controller, artificial intelligence, neural networks, machine vision, Node-RED, ABB RobotStudio, industrial automation

Upravljanje robotom temeljeno na video informaciji na studijskom slučaju igranja šaha

Sažetak

Automatizacija industrijskih pogona i uporaba umjetne inteligencije u robotici jedna je od ključnih točaka industrije 4.0. Sustavi koje se trenutno koriste diljem svijeta uglavnom su rađeni namjenski, što znači da je njihova ponovna uporaba za druge svrhe kompliciranа i vremenski dugotrajna. Kako bi napravili univerzalna rješenja, koja bila omogućila jednostavniji razvoj, moramo razviti sustav temeljen na poznatim programskim jezicima i platformama. Ocellus, kojeg je razvila švedska tvrtka Byte Motion, je jedan primjer takvog sustava. Razvijen je u poznatom sustavu za razvoj igara pod nazivom Unity i podržava širok spektar hardverskih i softverskih komponenti. U ovom diplomskom radu, koristit ćemo Ocellus i njegove napredne module pogonjene umjetnom inteligencijom kako bi razvili sustav koji ćemo nazvati AutoCHESS. AutoCHESS je potpuno automatizirani sustav koji simulira igru partiju šaha između dvije robotske ruke. Uz pomoć dubinske kamere RealSense, neuronska mreža trenirana za otkrivanje šahovskih figura, slat će informacije Ocellusu koji će računati njihove stvarne položaje. Informacije će se prenositi putem Node-REDA, alata za vizualno programiranje koji se inače koristi za uređaje interneta stvari. Potezi šahovske igre također će biti automatizirani i izračunati putem trenutno najboljeg šahovskog kalkulatora poteza pod nazivom Stockfish. Generirane poruke RMQ protokola slat će se s Node-REDA na ABB-ov program RobotStudio na obradu. RobotStudio će tada izvršiti radnje koje će rezultirati kretanjem robotskih ruku koje će naizmjenično odigravati šahovske poteze sve dok jedan od igrača ne pobijedi. Gotov sustav poslužit će kao demonstracija mogućnosti upravljačkih sustava za robote temeljenih na sustavima za razvoj igara, kao i prikaz upotrebe naprednih algoritama umjetne inteligencije u industrijskoj robotici.

Ključne riječi: Unity, Ocellus, kontroler robota, umjetna inteligencija, neuronske mreže, strojni vid, Node-RED, ABB RobotStudio, industrijska automatizacija