UiA University of Agder

Master

Deep ArUco: AI/ML-based Real-Time Marker Pose Tracking

SILJE WETRHUS HEBNES

SUPERVISORS
Kristian Muri Knausgård
Andreas Klausen
Sondre Sanden Tørdal

University of Agder, 2022
Faculty of Engineering and Science
Department of Engineering and Sciences

Acknowledgement

This master thesis is a part of the master study in mechatronics at the University of Agder, Campus Grimstad. This master thesis is provided by MotionTech AS for the spring semester of 2022.

A big thank you is directed to MotionTech AS with Sondre Sanden Tørdal and Andreas Klausen, for the opportunity to work on this exciting project and for the guidance along the way.

Another thank you is directed to the supervisor from the University of Agder, Kristian Muri Knausgård, for good feedback and follow-up throughout the whole time period of the project.

Abstract

Machine learning is used in various types of machine vision. The convolutional neural network (CNN) excels in finding difficult patterns to program in images. As CNN models can be trained in different lighting, motion blur, colors and changes, they are used in unstructured tasks of machine vision. The requirement for training data is the biggest disadvantage of this approach, but since fiducial marker designs are known, it is easy to generate training data that represent real life. To use these markers in marine applications, where distances may be great, an ArUco marker may be desirable to use because it can be used as a single marker. This report will go through methods used to generate training data and the detection of an ArUco marker in different lighting, motion blur and other interruptions in the scene by using machine learning and artificial intelligence. The already existing detection algorithm for ArUco markers in OpenCV is not good enough, and has trouble detecting the marker in poor lighting. DeepTag is a proposed method for marker detection which will be used to detect the markers in this project. The results from the experiments in this study will be compared to two other studies to evaluate the detection efficiency in this study. The results showed that the detection algorithm is usable in images with contrast and lighting, but should have more training on small contrast, motion blur and lack of light.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Fiducial markers are artificial landmarks that are added to different scenes to facilitate the locating points that correspond between images. An ArUco marker is a binary square fiducial marker. ArUco markers are used to estimate the camera pose and are robust and fast in their detection, and are also simple to use. In this report, ArUco markers are generated and used to create a dataset with images of the marker in different lighting and motion blur. The proposed method uses the detection scheme from DeepTag to detect and locate the markers' position in the image and the marker family. Chapter 2 describes the theory used in this project. Chapter 3 goes through the the method, while Chapter 4 shows the results. Chapter 5 contains the discussion and Chapter 6 has the conclusion.

1.1 Project Objective

Since the already existing detection algorithm for ArUco markers in OpenCV does not work well enough, the objective of this project is to generate training data to test another algorithm to detect the ArUco markers. A part of this study is to use artificial intelligence and machine learning to do a Real-Time pose tracking of an ArUco marker. The project goals are as follows:

- Propose an improved solution that would work well on smaller/single markers, for example an ArUco marker.
- Generate training data of the marker in Unity.
- Propose a Machine Learning pipeline for detection of markers.
- Compare the project to ChArUco paper results.

1.2 Background

Several studies on detecting different markers have been performed before, also on ArUco markers. Therefore, this project will use existing knowledge to detect the ArUco markers that are generated and put them into a virtual environment. In the virtual environment, different parameters will be added and the marker will be moved around to different positions and rotated in front of the camera. This is to evaluate if the existing technology on ArUco marker detection is of sufficient quality to detect the marker with different parameters. In addition to this, the detection will be made more difficult by adding more and more interruption in the camera view. The required knowledge needed to start this project can be found in section 1.3 below.

1.3 State of the art

The results from this study will be compared to the paper "Deep ChArUco: Dark ChArUco Marker Pose Estimation" [5]. The ChArUco paper is a previous study done on tracking ChArUco boards. ChArUco boards are used in robotics and augmented reality for pose verification, monocular pose estimation and camera calibration. These types of fiducial markers can be detected via computer vision by using OpenCV. However, there are some negative aspects of this technology since the environments needs to be well-lit or the classification methods will not function. In poor lighting and extreme motion blur, the classification method fails. This study presents a real-time pose estimation system, Deep ChArUco. The system combines two custom deep networks which use the algorithm Perspective-n-Point (PnP) to estimate the 6 DoF pose of the marker. The custom deep networks that are used are ChArUcoNet and RefineNet. ChArUcoNet outputs 2D point locations and jointly ID-specific classifiers. It is a two-headed marker-specific convolutional neural network (CNN). After getting the 2D point locations, they further use RefineNet as they are refined into subpixel coordinates. The networks are trained using a combination of different data. These data are synthetic subpixel corner data, target marker auto-labeled videos and extreme data-augmentation. Deep ChArUco is evaluated in different lighting and motion blur to demonstrate that this approach is better than the traditional OpenCV method used for detection and pose estimation of ChArUco markers [5]. The report about Deep ChArUco is extremely relevant for this report about Deep ArUco, since the Deep ArUco study experiences the same challenges as the Deep ChArUco. The difference between the studies is that ChArUco boards take a big portion of the camera view to decipher the details of the ArUco since the board is large. This is a big downside in marine applications where distances may be great to the markers. Therefore, the Deep ArUco study will look into a better way to use this method for real-time marker pose tracking so that it can be used in, for example, marine applications.

Another study done on ArUco markers is on detection of binary square fiducial markers by using an event camera. The event camera is a new type of sensors for imaging, where the output changes the light intensity values instead of having intensity values that are absolute. The cameras have a high dynamic range and temporal resolution. In this study, a method to detect and to decode different binary square markers is proposed. This method uses an event camera to detect the edges of the markers. By detecting line segments in the image that are created from the events, the edges are detected. To form the marker candidates, the line segments are combined. By using the events on the marker's border, the marker cell's bit value is decoded [6]. The method used is able to run in real-time on a single CPU thread. This study is relevant for the project in this report since it detects ArUco markers when they are in motion, which is one of the tasks in the Deep ArUco study, to detect ArUco markers with motion blur.

Previous work with one of the topics on ArUco markers, is a study done on design, detection and tracking of fiducial markers that are customized. This study uses the markers ArUco, QR codes and AprilTag to compare the detection and tracking possibility of customized markers. ArUco markers, AprilTag markers and QR code markers are easy to detect and are robust, but they have an industrial appearance which do not have a visually appealing look, as for example in a company logo. This is where customized markers are relevant. The result from this detection study allows the markers to be customized with some restrictions to the design. Some positions must be indicated to the template where each marker will get a unique identifier from the bits encoded. After the different markers were made, a method for detecting and tracking the markers was applied. This method was also used to compare the detection of the customized markers with the detection of ArUco markers, AprilTags and QR codes [4]. This study is relevant for the work in this report since markers will also be detected and tracked here.

A report about speeded up detection of square fiducial markers has an improved method for the ArUco detection system, which is one of the most reliable system for detection of fiducial markers. The proposed method speeds up the computing time used in the video sequences by applying a multi-scale approach and exploiting temporal information for the detection and identification of the markers. The method is very attractive for mobile devices since no parallelization is needed and mobile devices have computational power that is limited. Squared markers is a popular method for post estimation in autonomous and unmanned application. With high robustness, high speed and low cost, the markers makes it possible to estimate the position of a monocular camera [7]. This report is relevant due to the desirability of quick marker detection in this study.

A related study done on marker detection in RGB images and IR images for an augmented reality marker based on an ArUco marker, proposes a physical topology that enables detection in IR images and RGB images. This method used retroreflective materials in the marker, and was therefore tested with two different types of retroreflective materials. This was used to evaluate the impact this type of material had on the data from the image by two different cameras. The method was focused on the detection quality when the camera had different distances to the marker. A negative effect of the IR cameras is that different lighting on the marker can cause disturbance in the detection since these cameras rely on the reflection of the marker. The redundancy of the parallel processing architecture stabilized the detection in the IR images and the RGB images. Since the different materials that are retroreflective has an influence on the quality of the image depending on the camera, the performance of the Intel RealSense D435 and the Kinect V2 regarding the probability of detection depending on the distance between the camera and the marker was examined [8]. The detection based on ArUco markers is relevant for this study about Deep ArUco since the markers also need to be detected in different lighting here.

Another study in this field is done on infinitesimal plane-based pose estimation. This study was investigating better solution for estimating the plane pose since this is a core problem in computer vision in many applications such as camera calibration, augmented reality and 3D scene making. In mobile application, there is a need for a more efficient solution since the budget on run-time is critical. This study proposes an analytical solution which is much faster and accurate than the existing methods on solving pose from n point (PnP). The approach used in this study uses a new method of exploiting redundancy in the homography coefficients [9]. The pose estimation done in this study can be used for comparison to the one in this report.

There are existing studies performed on framework of the fiducial markers' design and detection. One of the studies is DeepTag, a general framework for the detection and design of fiducial markers. This study proposes a procedure that is effective to synthesize the training data without using manual annotation. Thus, the study shows that DeepTag better facilitates a broader application. The techniques for the object detection estimate the location and determine the category of an object. Different Convolutional Neural Network (CNN) detection schemes have been used for detection of objects. Two types of neural networks, Fast R-CNN and Faster R-CNN, are used to generate a high number of proposals for the region, then they classify the object and predict the bounding boxes. One technique that uses a single CNN to predict the class probabilities and the bounding boxes is the Single Shot Multibox Detector (SSD). This technique achieves a much faster convolutional speed without re-sampling pixels or the different features for bounding box hypotheses, than Fast R-CNN and Faster R-CNN. Based on this, research in this study uses SDD to generate marker hypotheses for the detection of keypoints. The marker detection was also tested in noise and motion blur. This showed that pose error increased when motion blur or Gaussian noise became critical for the methods. The results showed that under the same level of noise and motion blur, DeepTag performed better than the original methods. The marker's 6 DoF pose is estimated when the physical size of the marker is known. The 6 DoF pose is estimated with the algorithm Perspective-n-point (PnP) [10]. The study done on the framework of the fiducial markers design and detection has useful information and results for this report about Deep ArUco since this study also has detection of markers and motion blur.

A report about Fast R-CNN proposes a new algorithm for training. This algorithm is based on R-CNN and SPPnet and fixes the disadvantages in these algorithms, while also improving their accuracy and speed. Fast R-CNN is comparatively fast when doing the tests and training, and therefore is called Fast R-CNN. The advantages with this method compared to R-CNN and SPP-net, is that it has a higher detection quality. It also has a single-stage training process and uses a multitask loss. The training is able to update all the layers of the network and there is no requirement of disk storage for catching feature [11]. This report is relevant for the Deep ArUco project since Fast R-CNN is an algorithm that can be possible to use for this project. However, one of the disadvantages is that it requires a large amount of computational power.

After reading the report about Fast R-CNN, another study which goes further on this topic was investigated. The study upgrades the Fast R-CNN method and this new method is called Faster R-CNN. Faster R-CNN is used to do real-time object detection by using region based networks. This is a single unified network that is used to do object detection. It is composed of two different modules of networks, the first one is a deep CNN that proposes regions and the second is the module that proposes regions from the Fast R-CNN method. The RPN model uses terminology of neural networks with mechanisms for attention to tell the modules of the Fast R-CNN network where to search. Given the computation of the detection network, the computation is almost free of cost, and gives an effective and elegant solution of the computing proposal with deep CNN by changing an algorithm. To generate region proposals, region-based detectors that use convolutional feature maps can also be used. In addition to the convolutional features, by adding a few additional convolutional layers, RPN is constructed. These layers simultaneously regress the region bounds and the sources of objectness at every location that is on a grid. To effectively predict the region proposals with a wide range of different aspect ratios and scales, RPNs are designed. Faster R-CNN is a scheme of training that alternates between fine-tuning for object detection and for region proposal [12]. For the same reasons as for the Fast R-CNN report, this report is also relevant for the Deep ArUco project. The Faster R-CNN algorithm seems to have a better performance then the Fast R-CNN algorithm, and is faster. Thus, the same disadvantage is for this proposal as it uses a large amount of computational power.

Another study about Single Shot Multibox Detector (SSD) shows a method for object detection that is faster than the Fast R-CNN and Faster R-CNN and also YOLO. The SSD method uses a single deep convolutional network for the object detection to predict the class probabilities of the bounding boxes and the bounding boxes it selves. It uses less computational power than the two other algorithms mentioned above, without re-sampling pixels or the features for the hypotheses of bounding boxes. Even with a smaller input size of the image, SSD has a much better accuracy than other single stage methods it is compared to. SSD can be used for keypoint detection. To achieve the highest detection accuracy possible, predictions of different scales and aspect ratios are used. This leads to high accuracy and simple end-to-end training, improving the speed vs the accuracy trade-off even in bad quality images. Models with different sizes of input evaluated on COCO, PASCAL, ILSVRC and VOC are used to do the experiments on training. These models have shown to be computational intensive for embedded systems and also too slow for applications running in real-time. SSD have improvements that use a convolutional filter that predicts the category of the object and the offsets in the bounding box locations. This is chosen by using separate predictors and filters, for detection in different aspect ratios and then applying these to multiple feature maps. This makes it possible to do the detection at multiple scales [13]. This approach is relevant for this study, Deep ArUco, since this approach detects the marker in a fast way even with poor images. Since the detection of the ArUco marker will happen in different lighting and motion blur, the images might be poor.

Chapter 2

Theory

2.1 ArUco marker

ArUco markers are used to do post estimation in computer vision. An ArUco marker is a binary square fiducial marker which can be used in various types of machine vision tasks, such as; robot navigation, augmented reality, elevator buttons, etc. A fiducial marker is an object that is placed in front of a camera or in a visible place in front of an imaging system, so that it is visible as a reference point when an image is taken. Post estimation on ArUco markers is based on finding correspondence between the 2D image projection and the real environment of points. This step is normally a difficult step, but by using synthetic or fiducial markers it makes it much easier [1], [2]. An example of an ArUco marker is shown in Figure 2.1.

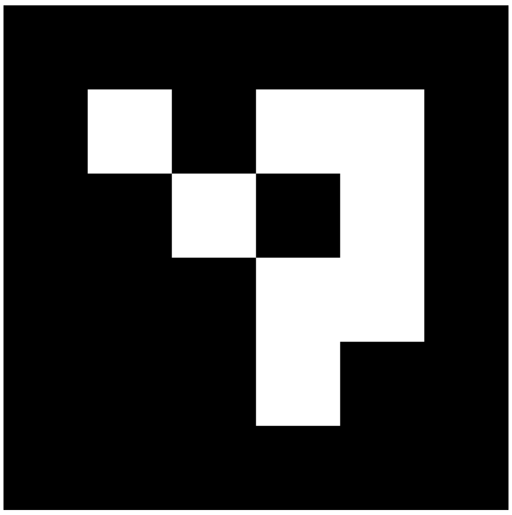


Figure 2.1: Example of an ArUco marker [3].

Binary square fiducial markers are commonly used. These markers have large benefits, since a single marker provides enough correspondence in its four corners to obtain a camera pose [1]. An ArUco marker is in the category of a synthetic square marker. ArUco markers have a white inner binary matrix surrounded by a black border. The inner binary matrix determines the identifier (ID) of the specific ArUco marker. Since ArUco markers are well known, the black border simplifies fast maker detection and determines the binary code identification inside the border. The internal matrix is fixed by the size of the marker. If the marker has a size of 4x4, it is composed by 16 bits. Even though the ArUco module has different pre-defined dictionaries containing many different marker sizes, this report will only concentrate on one ArUco marker [1].

To be able to detect an ArUco marker, at least four points are required to obtain a good camera pose estimation and detect the marker. This is why the marker needs to have four corners. When the ArUco marker is placed in a real environment, it is important that there is enough contrast between the marker and the background. In some cases, if the background is dark, a white border around the marker might need to be added [4].

2.2 Deep Learning

Deep learning is when a neural network consists of three or more layers, which is a subset of machine learning. The neural networks in deep learning attempt to simulate the human-brain behavior, by training an algorithm and learning from a large amount of data. A single layer of neural network can make approximate predictions, but when hidden layers are added it will refine and optimize the prediction and accuracy. Deep learning is used in many applications for artificial intelligence, such as application for automation, robots and analytical performance. Many everyday products use the technology of deep learning. These are products that are commonly used, such as TV remotes with voice control, digital assistant, mobile phone, autonomous vacuum cleaners and many more applications [35]. In this project, deep learning is used to detect the markers in the real-world environment created in Unity.

2.3 Convolutional Neural Network (CNN)

A Convolutional Neural Network (CNN) is an algorithm used for deep learning. CNN is a type of artificial neural network and is used to recognize objects in images and image processing [14]. This algorithm takes an image as input and assigns the importance of different objects and aspects in the image. The CNN algorithm is also able to see the difference between each object and find its position [15].

A Convolutional Neural Network has convolutional layers. On a layer that is fully connected, each output of the neurons is a linear transformation of the previous layer which is composed with an activation function that is non-linear. This means that the output of a neuron that is in a convolutional layer is a function of a subset of the neurons on the previous layer in the CNN [16]. An example of a convolution can be seen in Figure 2.2.

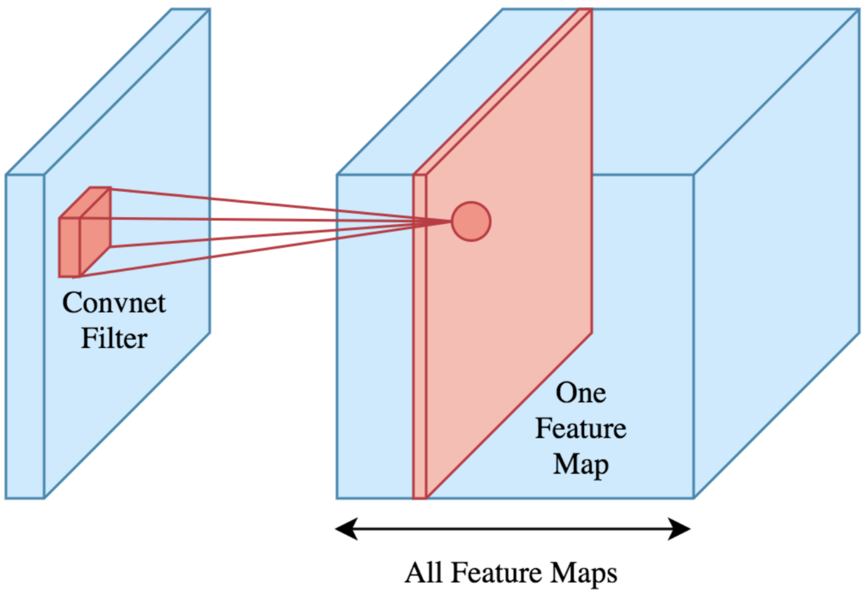


Figure 2.2: Example of a convolution [16].

CNN also contains filters and pooling. A filter is used when a convolution takes the sum of the elements close to each other and combine them to, for example, sharpen an image. The pixels' values that are different from the adjacent pixels' values, are boosted by the pixels added together. The network is able to learn the filters when the CNN is trained, which can help extract information needed from the image. When the convolution has made a feature map, a pooling layer can be applied. The most used type of pooling is max pooling. Max pooling is defined as the maximum value taken from an array of given numbers. Figure 2.3 shows an example of pooling where the feature map is split into n x n boxes. The maximum value is then chosen from all of the boxes [31].

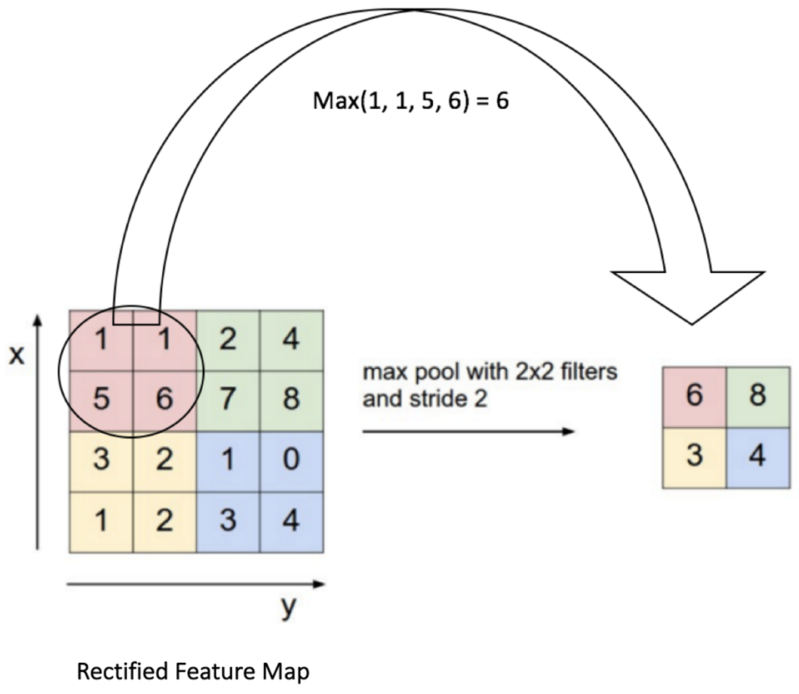


Figure 2.3: Example of max pooling [31].

The fully connected layer is the final layer of a CNN. Very deep CNN normally has many layers of pooling. Max pooling layers can be stacked upon each other to give a better result. The most common use of convolutional neural networks are for classifications of images or videos. They are ideal when objects are going to be detected in different positions in a dataset of images [31].

To get a CNN algorithm to fit perfectly for the task, the algorithm needs to be trained. This is done by first creating a dataset. A dataset can be a set of images where there are enough images to use for both training, validation and testing. Normally, this is distributed 1/3 to each part. When the dataset is created, it needs to be prepared for training. This is where the objects are labeled and the images are resized. Next, training data is created. Then random images are either drawn from the dataset or images are chosen from the dataset used for the training. The chosen images are labeled and assigned features. Now the algorithm runs the 1/3 of the dataset that is chosen for training, to train the algorithm to detect the desired object in the image. Once finished, 1/3 of the remaining images of the dataset will be used for validation. These images are run in the algorithm to validate that the training was successful and that the algorithm can detect the desired object. The last 1/3 of the dataset can then be used for testing of the algorithm and further work.

2.4 Data augmentation

Data augmentation can be used to create more data from the existing data for the process of training the model for detection. This means that different perspectives of the image can be used to create a "new" image with the marker in a different position of the image. For example, if the marker is in the middle of the image, a perspective of the image with the marker in the corner can be used. Thus, this will create a new image with the marker in the corner which can train the model to detect the marker both in the middle with the original image and in the corner with the image created with data augmentation.

Chapter 3

Method

3.1 Introduction and design

In the method section, all the practical parts of the project are described. At first it goes through the workspace and the generation of ArUco markers and explains how this is done. When the marker generation process is described, the virtual environment is created. This environment is designed to simulate real-time, where the marker will move in front of the camera view. The marker's position and rotation will change for each new screenshot, and there will be a shadow-plane moving to create a shadow on the marker in different positions and with different angles. This is where the dataset is being created. Gradually, different brightness, motion blur and other interruptions on the camera lens will be added.

After the marker generation and creation of the virtual environment, marker detection is next. Since there already exists technology on detection of ArUco markers, the DeepTag algorithm is used here [10]. This algorithm is used to find the ROIs, boxes and corners, image with pose and direction of marker, keypoints and marker. The DeepTag algorithm is chosen after research of several studies, where DeepTag appears to be best suited for this project. The algorithm will be tested for its detection accuracy with only the marker moving with shadow, with different brightness, motion blur and other interruptions on the camera lens. More information on how this algorithm is used can be found in section 3.7.1. When the algorithm is able to detect the markers with acceptable certainty, the brightness in the image will be changed and motion blur will be added. After this, more interruptions will be added to investigate how good the detection quality of the algorithm is, and when is it necessary to train it for more difficult tasks of detection.

3.2 Workspace

The chosen workspace for this project is fully digital. This is due to covid-19, and with a possibility of shutting down the country again, the work will be able to continue as planned. The scene with the marker in is created in Unity and all the movements of the objects in the scene are scripted in C#. Python is used to generate the ArUco markers and to detect them. The training and validation of the detected ArUco markers also uses Python as the programming language. Another option for this project would be to physically print out the ArUco marker and take images in a physically created environment. However, when this is done fully digitally, it is easier to create the wanted scenes with a much greater variation of parameters and a combination of them.

3.3 Generation of ArUco markers

First, different ArUco markers had to be generated by using Python as the programming language. To use Python, the program PyCharm was downloaded [17]. When PyCharm was installed, the code for generating ArUco Markers could be written, and is called "generateArUco.py" [18]. The code used to generate the different ArUco markers can be seen in Appendix A. Even though only one type of ArUco marker was used in this study, it was created like this in Python. Knowledge of how to generate the markers in Python, makes it easier for further work, if it is desirable to test this proposed method of detection and pose estimation with different kinds of markers. Listing 3.1 below shows the two lines for deciding which ArUco marker that will be generated. By changing the first number in the second line, where it says 20 here, the marker that is generated will be different.

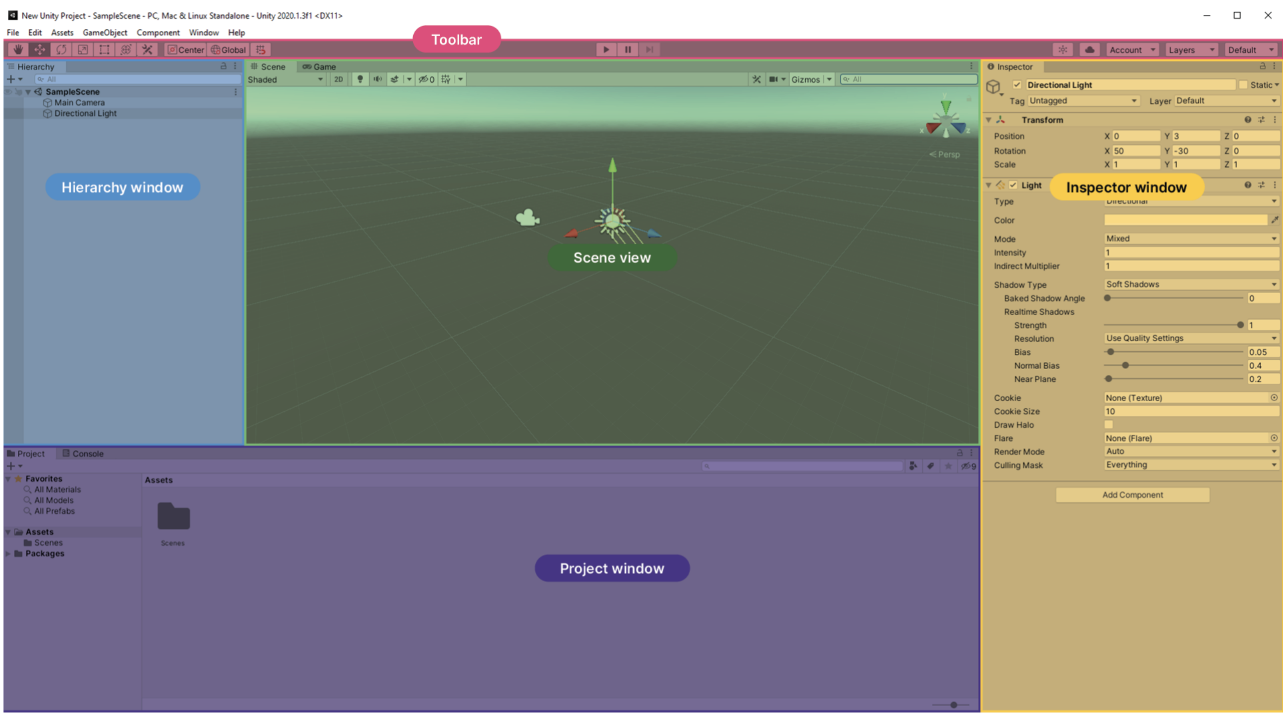
Listing 3.1: Two lines of codes for generating the ArUco markers.

```
# Generating the marker
markerImage = numpy.zeros((200, 200), dtype=numpy.uint8)
markerImage = cv2.aruco.drawMarker(dictionary, 20, 200, markerImage, 1);
```

3.4 Unity

To make a scene for the ArUco marker to be detected as closely as possible to real life, a program called Unity was used. Unity is a platform to develop and create games. It is used to gain a competitive edge in a rapidly evolving industry landscape in real-time 3D, production for animation and film projects, and is used to create experiences in 3D for real-world applications [19].

To start using Unity, the software needed to be downloaded [20]. Then a 3D project was created. When the project window of Unity was open, Unity Editor interface was used. In this window, there are five main areas. These areas are; scene view and game view, hierarchy window, project window, inspector window and toolbar, which are used to create and monitor the scene. The scene view and game view are found in the center of the Unity Editor window, and is the interactive window to the created scene. In this window, the scene view can be used to view objects from various angles and to manipulate the objects. In the hierarchy window, found on the left side, all the objects in the scene are organized and listed. On the bottom of the Unity Editor window, the project window is found. Here, all of the available assets to use in the project are listed. The assets from the project window can be dragged directly into the scene view to be added to the scene. In the inspector window, to the right, the detailed information about each object is configured. When an object is selected from the scene view, the components that describe the properties and the behaviors are available in the inspector window. The last main area is the toolbar, found on the top of the Unity Editor window. The toolbar is used to select the different objects in the scene and to adjust them. The point of view could also be changed from here [21]. The Unity Editor window is shown in Figure 3.1.



Directional Light Static
Tag Untagged Layer Default

Transform
Position X 0 Y 3 Z 0
Rotation X 50 Y -30 Z 0
Scale X 1 Y 1 Z 1

Light
Type Directional
Color
Mode Mixed
Intensity 1
Indirect Multiplier 1
Shadow Type Soft Shadows
Baked Shadow Angle 0
Realtime Shadows
Strength 1
Resolution Use Quality Settings
Bias 0.05
Normal Bias 0.4
Near Plane 0.2
Cookie None (Texture)
Cookie Size 10
Draw Halo
Flare None (Flare)
Render Mode Auto
Culling Mask Everything

Figure 3.1: The Unity Editor window [21].

3.4.1 Creating the scene

To start the project, the first task was to make a scene in Unity where the images of the ArUco marker could be taken. At first, a new project was created with a 3D core. Then the empty 3D project was opened. A plane was used as the ground and a cube was created to attach the ArUco marker onto, so the camera could detect the marker. To make the plane and the cube, the hierarchy window was used. In this window the camera is also listed as the main camera. By right clicking in this window and choosing "3D objects" and then plane, a plane was created. The same procedure went for the cube. When the plane and the cube was implemented in the workspace, they could be resized to the desired size. Then the camera was chosen in the hierarchy window and the properties and behaviors were changed in the inspector window. The camera was set to "physical camera" and the focal length, field of view and sensor type were set to the desired size and range. The focal length was set to 50, field of view was placed at 10.6832 and the sensor type chosen was 35mm 2-perf. When this was completed, an ArUco marker was generated in Python and implemented into Unity by dragging it to the project window. The ArUco marker was placed on the cube in front of the camera view. When the marker was in position above the ground-plane, a new plane was created as the shadow-plane.

To make it easier and more efficient to generate data, some adjustments were made in Unity before making the script. At first, the view direction was changed to be in the y-direction. Then the ArUco marker was seen from above with the ground-plane in the background. The shadow-plane was then placed above the camera and had the same size as the cube. The size of the cube was set to 10x10x10 units. Since the plane is 10x bigger than the cube in unity, since the plane is 10 units per scale, the plane was then set to scale 1x1x1 to match the size of the cube. The ground-plane was set to scale 10x10x10, so that is was 10 times bigger than the cube. When the scale of the different objects was right, the light was adjusted. By choosing Directional Light in the hierarchy window, the light was set to directional [22]. The sun was rotated by 90 degrees in x-direction to get the sun right above the cube and the shadow-plane. Since the sun is right above the cube with the marker and the shadow-plane, the plane does not have to be bigger than the cube to cover the cube completely. Figure 3.2 below shows the scene view in Unity. The scene view is zoomed in on the ground-plane and the cube with ArUco markers on in Figure 3.3, and on the camera and shadow-plane in Figure 3.4.

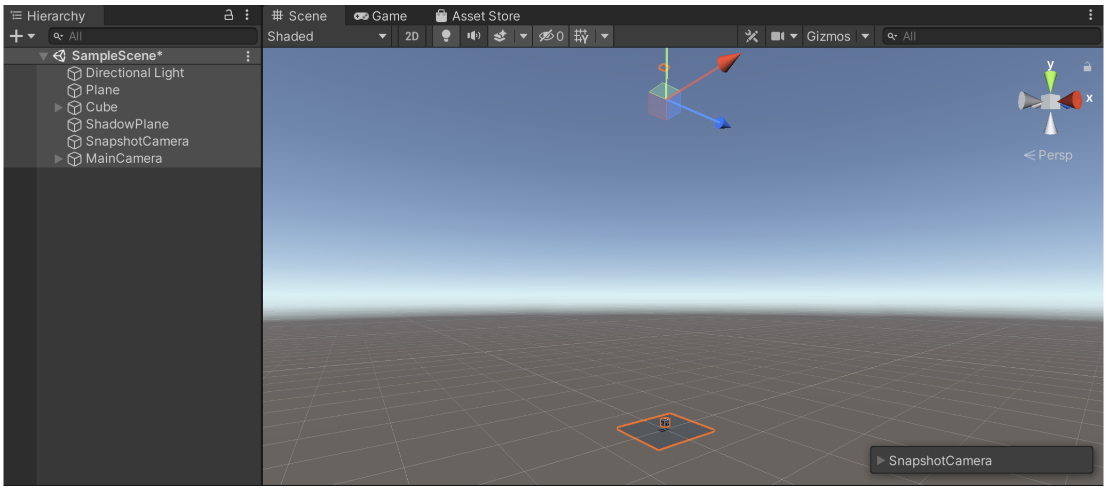


Figure 3.2: The scene view in Unity.

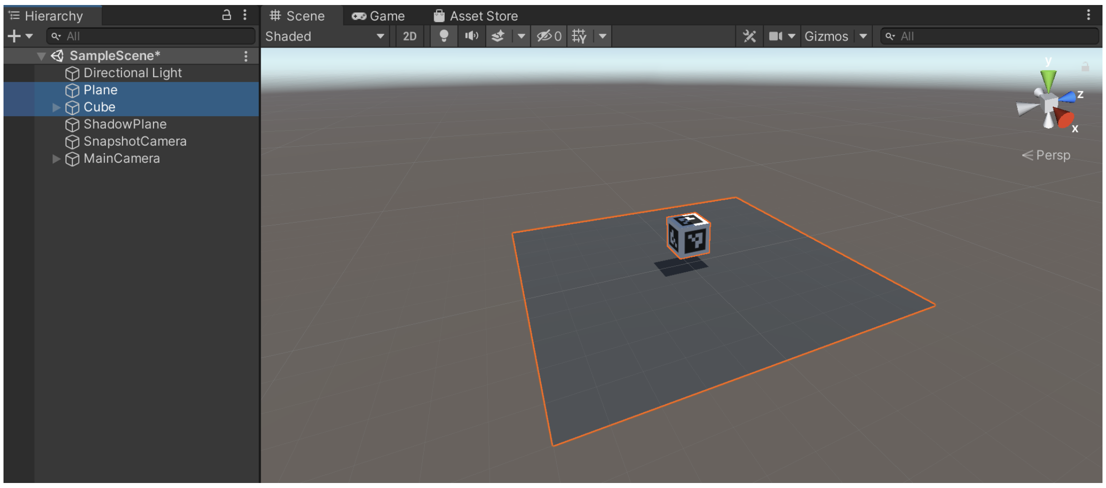


Figure 3.3: The scene view in Unity zoomed in on the ground-plane and the cube with ArUco markers.

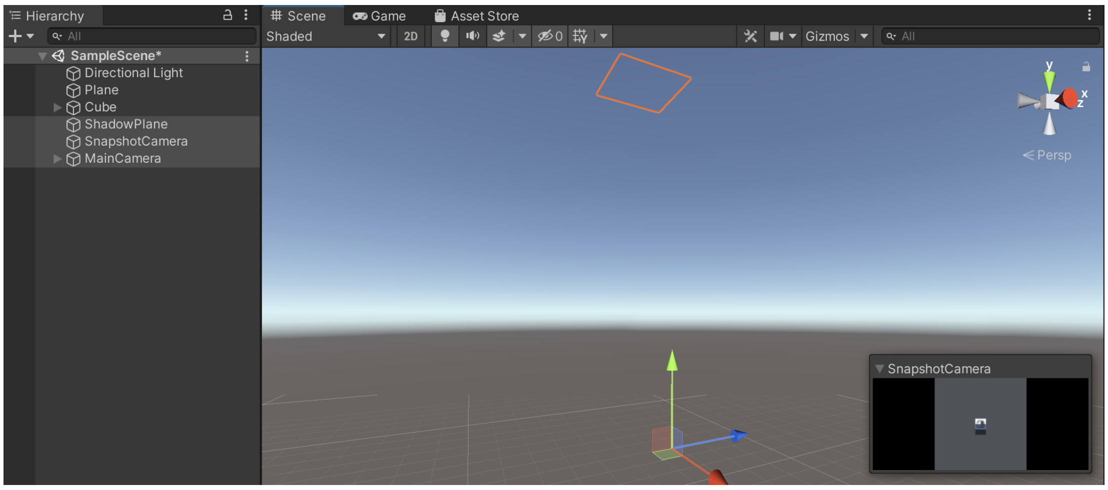


Figure 3.4: The scene view in Unity zoomed in on the cameras and shadow-plane.

After fixing all the objects and lighting, the shadow preferences were changed. This is since the further away the camera was from the cube, the more the shadow disappeared. Shadow rendering was necessary to fix this problem. To change the shadow, the bias to the directional light was changed and set to 0.05 [23]. To see a greater difference between the white ground-plane and the cube with the white border around the ArUco marker, the color intensity was adjusted and blue-colored material was added to the ground-plane [24], [25].

3.5 Parameters

To make the detection of the ArUco marker robust and better, different parameters were varied. By varying the parameters, the model can be trained to detect the marker with different difficulties or interruptions in the scene. The different parameters that can vary in this scene are camera position and orientation, ArUco markers position and orientation, lighting, shadow, motion blur, etc. This results in a large amount of different combinations of variations. Therefore, some limitation must be set. The parameter variation used in this study is described in section 3.5.1.

3.5.1 Parameter variation

Before scripting the movements of each object, some parameters needed to be set. The parameters which were fixed were the ground-plane, the camera and the directional light. The cube with the marker and the shadow-plane were both going to be rotated and changing position. The parameter variations can be seen in Table 3.1 below. Ground-plane was set to (0,0,0) for the position and (0,0,0) for the rotation. The camera was set to (0,500,0) for the position, which means it is moved 500 units in the y-direction. The rotation of the camera was set to (90,0,0) to rotate the camera 90 degrees in x-direction to look down on the marker. The position of the directional light was set to (0,0,0) and rotation to (90,0,0) to rotate the sun 90 degrees and get it exactly above the marker, camera and shadow-plane.

Table 3.1: Variation of the parameters in the Unity scene.

Parameter	**Variation (x,y,z)**
Camera position	Fixed
Camera orientation	Fixed
Ground-plane position	Fixed
Ground-plane orientation	Fixed
Directional light position	Fixed
Directional light orientation	Fixed
ArUco marker position	(0,10,0) to (0,450,0)
ArUco marker orientation	(0,0,0) to (89,359,89)
Shadow-plane position	(-10,550,-10) to (10,550,10)
Shadow-plane orientation	(0,0,0) to (89,359,89)

As seen in Table 3.1, the y-coordinates of the position of the ArUco marker are the only ones changing. This is because data augmentation can be used to move the marker in different places of the image. The same image is used, but with different perspectives. Therefore, the only position that needs to change is the distance from the camera. However, the orientation of the ArUco marker is changed in every direction. The marker is rotated between 0-89°in the x-direction, 0-359°in the y-direction and 0-89°in the z-direction. The reason why x- and z-direction of rotation is set to 89°is because if it is rotated 90°it will be the same as 0°, just in the opposite direction. The same reason is why 359°is chosen in the y-direction. The position of the shadow-plane is, as seen, only changed in x- and z-direction. The y-direction stays the same since it is always above the camera so it will not block the camera view. The x-direction and z-direction is set to move between -10 and 10 because the size of the marker is the cube size, 10x10x10. When the shadow-plane is moved in either x-direction or z-direction by -10 or 10, it is outside of the marker and will not make any shadow on the marker. As for the rotation, the same argument as in the position of the marker is valid for the chosen parameters here.

By choosing these parameters to variate, it will give many possible combinations of the parameters. To reduce the number of possibilities and to be able to calculate the number of possible combinations, integer is used. By using integer, it gives enough possible combinations needed for this project. It would be possible to use double or float if there were more combinations needed. For the marker, the total combinations of different parameters with integer is

$$441 * 90 * 360 * 90 = 1285956000 \tag{3.1}$$

The reason why it is not 440*89*360*89 is because zero also needs to be counted.

The total of possible parameter combinations of the shadow-plane is

$$21 * 21 * 90 * 360 * 90 = 1285956000 \tag{3.2}$$

Which is the same amount as combinations of the marker when using numbers of integer. This means that the total combinations of parameters within all the objects are

$$1285956000 + 1285956000 = 2571912000 \tag{3.3}$$

In total there are 2 571 912 000 possible variations of parameters. This number indicates how many possible combinations of parameters that can be used, just with numbers of integer. Depending on how many subnumbers that are used, the number can become much larger.

3.5.2 Scripts and activity diagram

When all of the objects were made, implemented and had the right coordinates to start with, the scripts attached to the different objects were made. A new script called "CameraCapture.cs" was created. This script contained a class with the different parameters, a list over which parameters to variate, the variations of the parameters and also the creation of a json file to each image taken [26]. The script also changes the parameters to random numbers between the given parameter range before a new script called "SnapshotCamera.cs" implements the parameter list and takes the images. A new camera was made as an object in Unity, with the same parameters as Main Camera to take the images. Two additional scrips were made, "MoveShadow.cs" and "MoveAruco.cs". These two scripts are used to move the shadow and the ArUco marker to the given parameters. Both of the scripts get the parameter list from "CameraCapture.cs", and "MoveShadow.cs" changes the position and rotation of the shadow, and "MoveAruco.cs" changes the position and rotation of the Aruco Marker. Unity uses C# as the programming language. Therefore, all of the scripts used in Unity was coded in C#. The scripts can be seen in Appendix B, C, D and E. The activity diagram seen in Figure 3.5 shows step by step how the codes are built and each task they do.

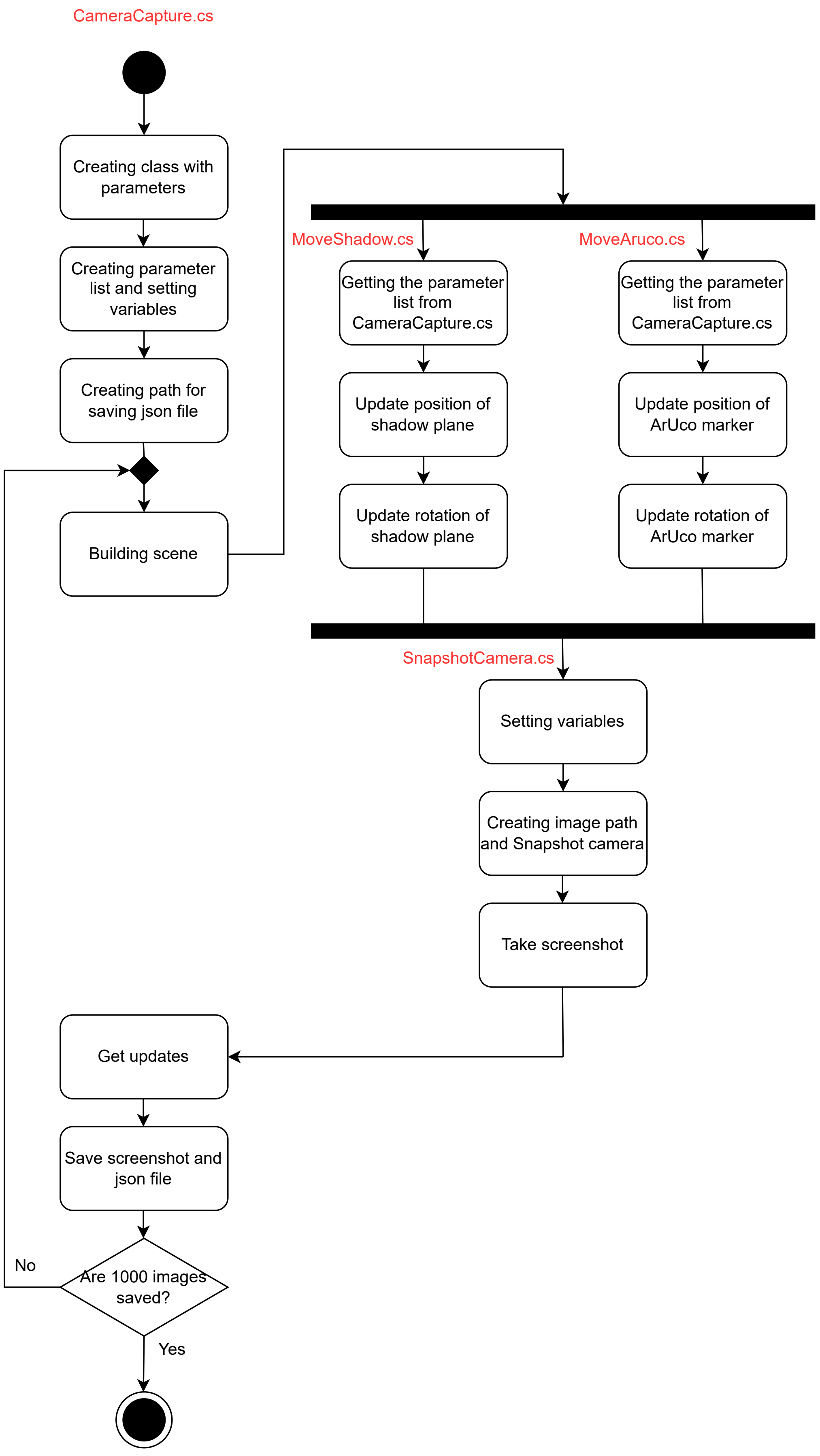

CameraCapture.cs
Creating class with parameters
Creating parameter list and setting variables
Creating path for saving json file
Building scene
MoveShadow.cs
Getting the parameter list from CameraCapture.cs
Update position of shadow plane
Update rotation of shadow plane
MoveAruco.cs
Getting the parameter list from CameraCapture.cs
Update position of ArUco marker
Update rotation of ArUco marker
SnapshotCamera.cs
Setting variables
Creating image path and Snapshot camera
Take screenshot
Get updates
Save screenshot and json file
Are 1000 images saved?
No
Yes

Figure 3.5: Activity diagram of what the different scripts are responsible for.

As seen in the activity diagram, the script "CameraCapture.cs" is the main script, which contains the class with parameters and the list with them. This script also creates the path for the json files and builds the scene. The other scripts "MoveShadow.cs", "MoveAruco.cs" use the list of parameters from "CameraCapture.cs" and change the position and rotation of the shadow-plane and the ArUco marker. The code "SnapshotCamera.cs" is used to create the path for the images and to take the images. Then "CameraCapture.cs" updates the parameters the desired number of times, and saves the desired number of images.

Further in the activity diagram, the script "CameraCapture.cs" starts to run first and then "MoveShadow.cs" and "MoveAruco.cs" starts to update the positions and rotations after the scene is built. When the parameters are updated, "SnapshotCamera.cs" creates the path for the images and takes the screenshots. When this is done, the "CameraCapture.cs" script gets the updates and saves the new parameters to the json file and also runs the desired number of screenshots. Unity has its own framework, which is why the scripts run like this, without having to add this to the code.

Due to the conversion from quarterion to angles, the editor in Unity displayed strange numbers instead of the parameters given when the ArUco marker and the shadow-plane was rotated. To fix this problem, a function needed to be changed in the "MoveAruco.cs" and "MoveShadow.cs" scripts. The function for rotation, which used each of the given rotation parameters, had to be changed. The function was changed from "transform.rotate" to "TransformUtils.GetInspectorRotation", so that the editor window in Unity showed the right rotation parameters [27].

The lines of codes for changing the position and rotation of the ArUco marker and the shadow-plane can be seen in Listing 3.2 and Listing 3.3 below. The whole script can be seen in Appendix C and D.

Listing 3.2: The two lines of code for updating the position and rotation of the ArUco marker.

```
//Changing the position of the ArUco marker:
transform.position = new Vector3(frameParms.posx_aruco, ...
   frameParms.posy_aruco, frameParms.posz_aruco);
//Changing the rotation of the ArUco marker:
framep = new Vector3(frameParms.orix_aruco, frameParms.oriy_aruco, ...
   frameParms.oriz_aruco);
UnityEditor.TransformUtils.SetInspectorRotation(transform, framep);
```

Listing 3.3: The two lines of code for updating the position and rotation of the shadow-plane.

```
//Changing the position of the shadow-plane:
transform.position = new Vector3(frameParms.posx_shadow, ...
   frameParms.posy_shadow, frameParms.posz_shadow);
//Changing the rotation of the shadow-plane:
framep = new Vector3(frameParms.orix_shadow, frameParms.oriy_shadow, ...
   frameParms.oriz_shadow);
UnityEditor.TransformUtils.SetInspectorRotation(transform, framep);
```

There was a large difference between the image with and without the shadow covering the marker. This can be seen in Figure 3.6, which shows the image saved of the ArUco marker in the camera view without the shadow. Figure 3.7 shows the image of the ArUco marker with the shadow from the shadow-plane.

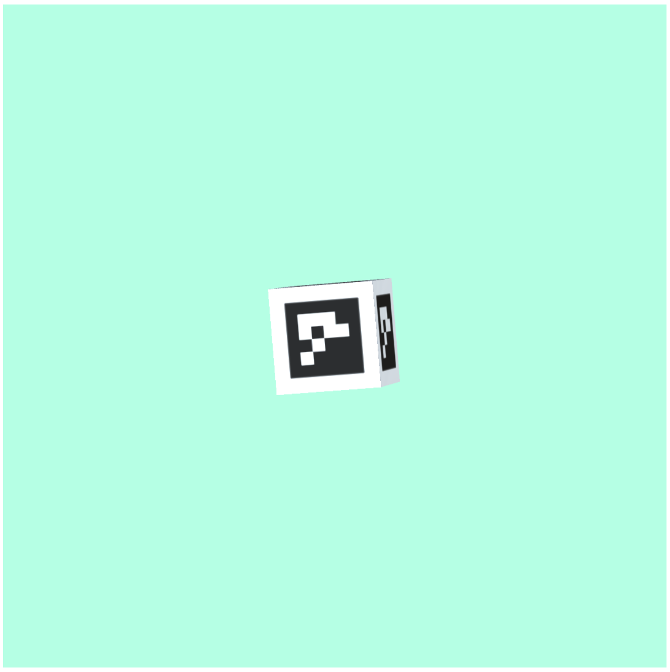


Figure 3.6: The image saved with the ArUco marker in front of the camera view.

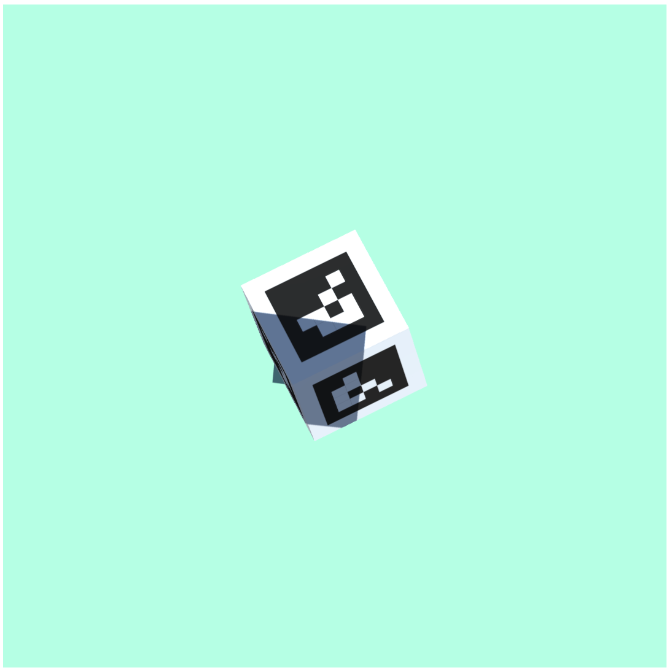


Figure 3.7: The image of the ArUco marker with shadow on it.

When the scripts were working on the Unity environment and could create different positions and rotations of the ArUco marker and the shadow, it was time to start generating data. The data set will consist of different images which each has its own json file attached to it.

3.6 Generating data

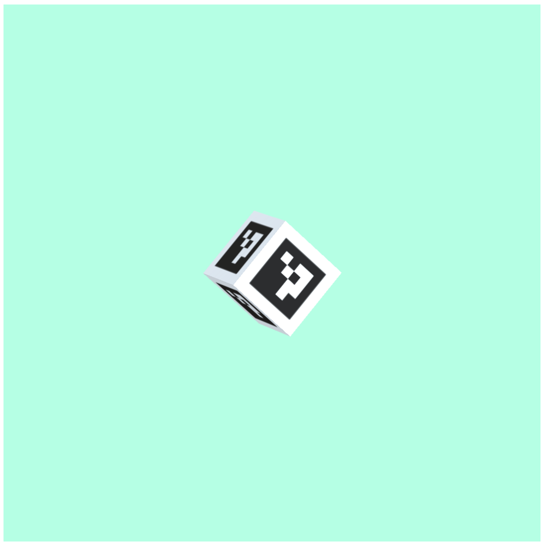
Data was generated for each step and different parameter that it were desired to test the chosen algorithm on. Each subsection in this section 3.6 are described below. The codes used to generate training data from Unity can be seen in Git[1].

3.6.1 Data generation in Unity with different positions, rotations and shadow

To generate the data, all the information about the parameters in the images needed to be known, changed and saved. The generated dataset consisted of 1000 images. Each image has different parameters for the position and the rotation for both the ArUco marker and the shadow-plane. When generating data, all of the scripts were working together so that each parameter was updated before taking the image and creating the json file to that specific image. This was done a thousand times. Some examples of the differences in position and rotation of the ArUco marker and the shadow-plane can be seen in the figures below. Figure 3.8a shows the ArUco marker when the position is close to the camera and when it is rotated. Figure 3.8b shows the marker when it is far from the camera and when it is rotated. Figure 3.8c shows when the markers position is close to the camera, rotated and when the shadow covers most of the marker. Figure 3.8d shows when the ArUco marker is far from the camera, rotated and is covered only a little in the corner by the shadow since the shadow also is rotated.

[1]https://gitlab.com/siljeheb/deep-aruco

(a) The ArUco markers position is close to the camera view and it is rotated.


(b) The ArUco markers position is far from the camera view and it is rotated.

(c) The ArUco markers position is close to the camera view, it is rotated and a large part is covered by the shadow.

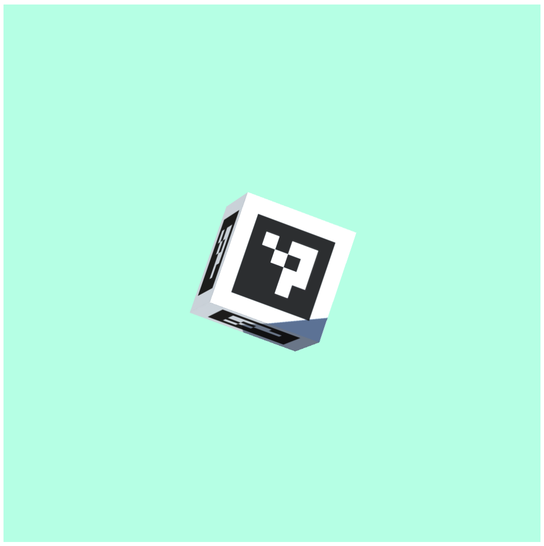
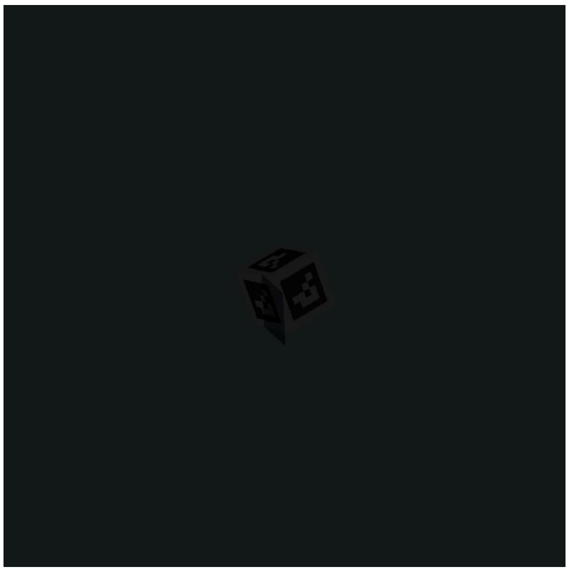

(d) The ArUco markers position is far from the camera view, it is rotated and covered only a little by the shadow.

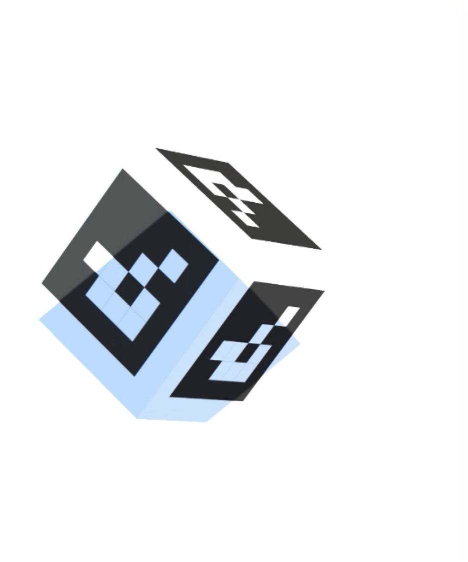
Figure 3.8: Figure a), b), c) and d) shows the marker in different positions and rotations, with and without shadow.

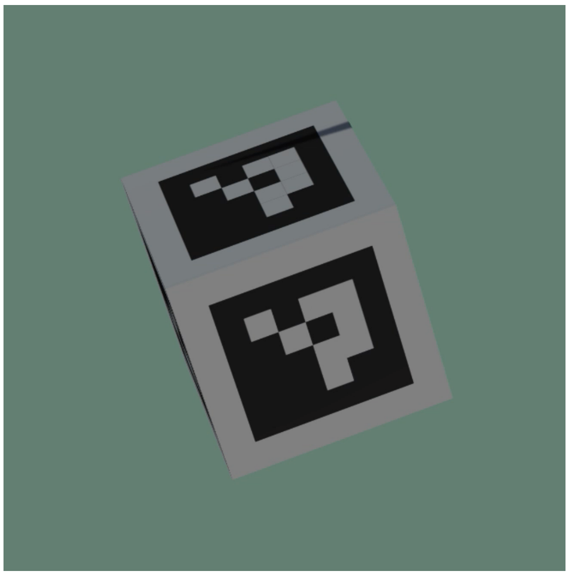
As seen in all of these figures, both the ArUco marker and the shadow has had its position and rotation changed. This shows that the scripts for varying the parameters are working and are updating the new positions and rotations for each image. Since it is confirmed that all the scripts work as they should, the training of the model can start.

3.6.2 Data generation in Unity with different brightness

Another parameter that was varied was the brightness in the scene. To reduce or increase the brightness, a package from Unity Assets Store was downloaded to save time and resources [28]. This package is called "SimpleLUT Adjustment" and includes a script that, when added to the main camera, can adjust different parameters such as the brightness. This makes it easy to slide the brightness bar to each side to either reduce or increase the brightness in the camera view. The figures below show how the camera view is when the brightness is reduced and increased. Figure 3.9a shows the camera view when the brightness is reduced by 90 percent. Figure 3.9b shows the camera view when the brightness is increased by 90 percent. Figure 3.9a shows when the brightness is reduced by 50 percent, and Figure 3.9d shows when the brightness is increased by 50 percent.


(a) Scene with 90 percent reduced brightness.


(b) Scene with 90 percent increased brightness.


(c) Scene with 50 percent reduced brightness.

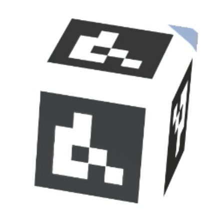
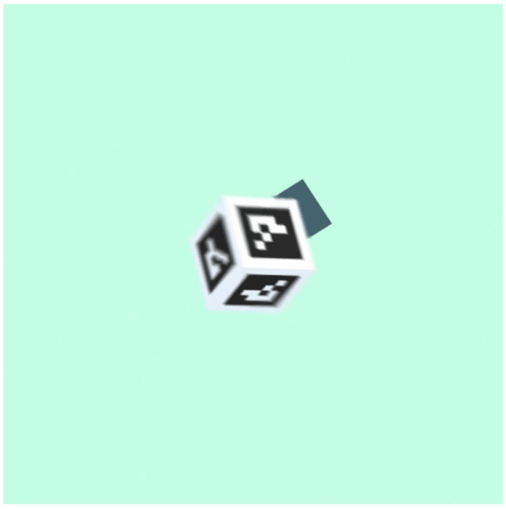
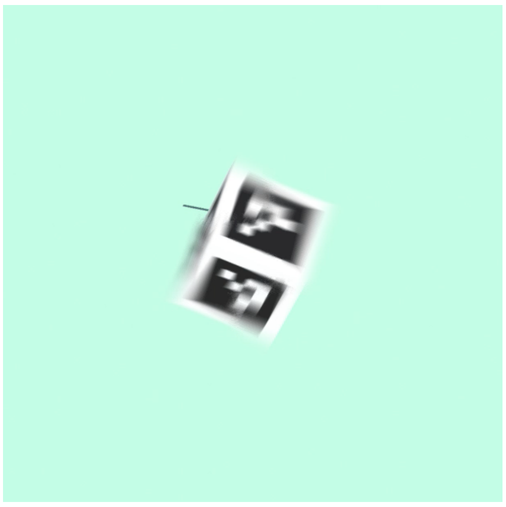

(d) Scene with 50 percent increased brightness.

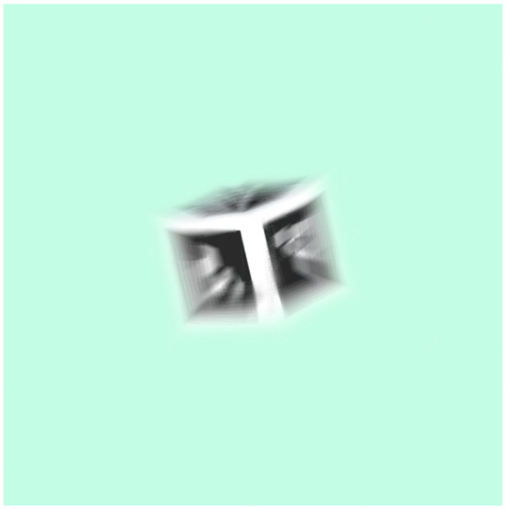
Figure 3.9: Figure a), b), c) and d) shows a selection of variations in the brightness.

3.6.3 Data generation in Unity with motion blur

To get motion blur when the marker was moving in Unity, the first step was to go to package manager and install the package "Post Processing". When the post processing package was successfully installed, a processing profile was created in the assets by right clicking and choosing create, then processing profile. Next step was to create an empty object under the cube, which was called "Effect". A component was added to the new object in the inspector window. This component was the "Post Processing Volume". Here the post processing profile that was created is added to the profile of this component. A box collider was then created as a new component and scaled to the desired scale where the effect was desired to be. This was scaled to (100,100,100), 10 times larger then the cube and as large as the background in the scene. The blend distance in the post processing volume component was then set to 5. When this was done, the effect was added. Motion blur was chosen and the shutter angle was set to 80°and the sample count to 25. A post processing layer was added to the main camera. Then a layer was assigned to the post processing layer and to the effect, the layer used was "TransparentFX". After this, the mode was set to "Fast Approximate Anti-aliasing (FXAA)" and fast mode was chosen [30]. The scene was now ready, and motion blur was added. The figures below shows an excerpt of the variation in motion blur from the dataset of 100 images. Figure 3.10a shows screenshot 3, where there is a low amount of motion blur. The next figure, Figure 3.10b, shows image 5 with a small increase in motion blur. In Figure 3.10c, there is more motion blur and Figure 3.10d has the largest amount of motion blur.


(a) Motion blur in screenshot 3.


(b) Motion blur in screenshot 5.


(c) Motion blur in screenshot 59.

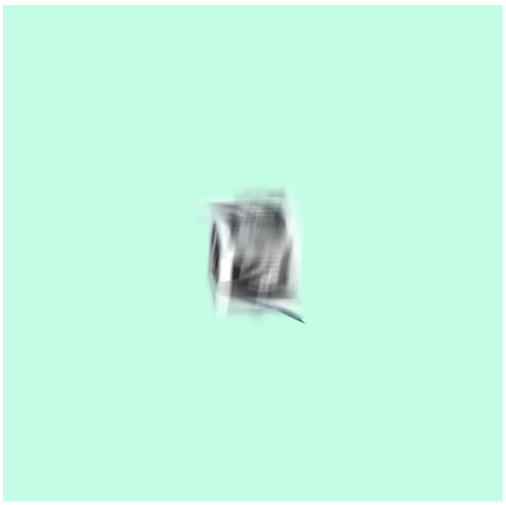

(d) Motion blur in screenshot 97.

Figure 3.10: Figure a), b), c) and d) shows how the motion blur is varied.

3.6.4 Data generation in Unity with interruption on the camera

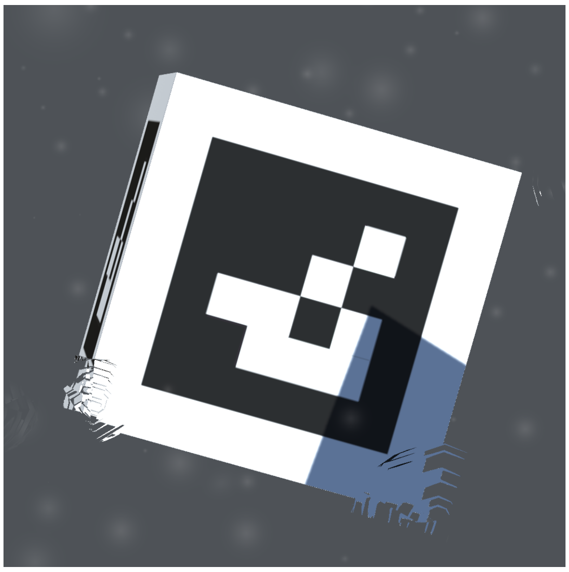
In this section, there are three subsections where interruptions are added to the scene to generate data with more difficulty to test the detection algorithm on. The added interruptions are rain drops, contrast changing and fog.

Rain Drops

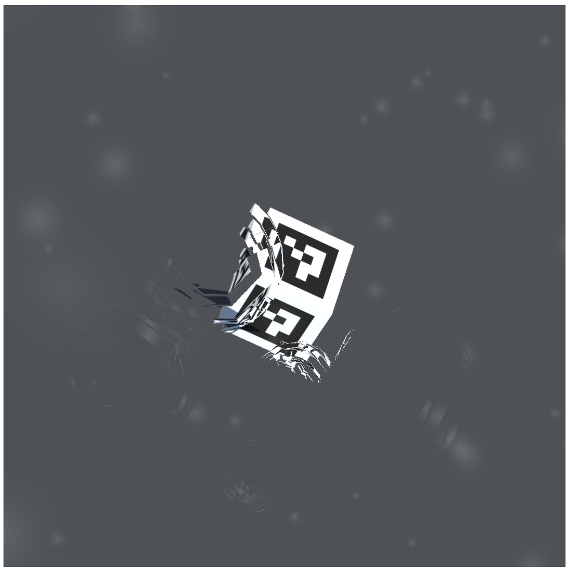
To make the scene more realistic, rain drops were added as an interruption. This was to test if the detection algorithm was able to detect the marker with rain drops on the camera lens. To make the rain drops, a package called "Standard Assets" was downloaded from asset store and imported to the project [32]. When the package was imported, a new layer was created, called "Camera_Effect". Into this layer, both of the textures from glass refraction were applied and the texture type was set to "normal map". Then a new material was created. This material was called "Rain_Camera_Material_Small_Medium" and the shader was set to "FX", "Glass" and "Stained BumpDistored". After this, the desired texture for small and medium rain drops was added and the distortion was set to 40. The same was done for another new material created called "Rain_Camera_Material_Big", where the desired texture for big rain drops were added and the distortion was set to 30. When this was set up and ready, an effect was created in the hierarchy window. The effect was chosen to be a partial system to get the desired effect of the rain drops. In the inspector window, all of the different preferences can be set to the effect. By scrolling all the way down to "renderer" and setting sorting layer to the "Camera_Effect layer", the rain drop effect is added. To make it more realistic, the rest of the parameters are set to the desired numbers. Start speed is set to 0, 3D start size is chosen where the scale of the rain drops are decided, start rotation and simulation speed are changed. Then under emission, the "Rate over Time" is changed to set the frequency of the rain drops and under shape, box is chosen and the scale for the rain drops is added. Size over lifetime is added to give it a realistic view since the drop hits and spreads out, which is also why noise is added to some of these effects. The same procedure was done for all of the four effects; "RainDrop_Camera_Small", "RainDrop_Camera_Medium", "RainDrop_Camera_Big" and "RainDrop_Camera_Trail". Except for that "trail" was also added to the last effect to get the rain drops to flow out on the camera lens [33]. To be able to see the rain drop as best as possible, the background was changed to a darker shade. A dataset consisting of 100 images was created. Some images from the virtual environment in Unity are shown in Figure 3.11 below.

(a) Rain drops in screenshot 3.

(b) Rain drops in screenshot 41.



(c) Rain drops in screenshot 51.

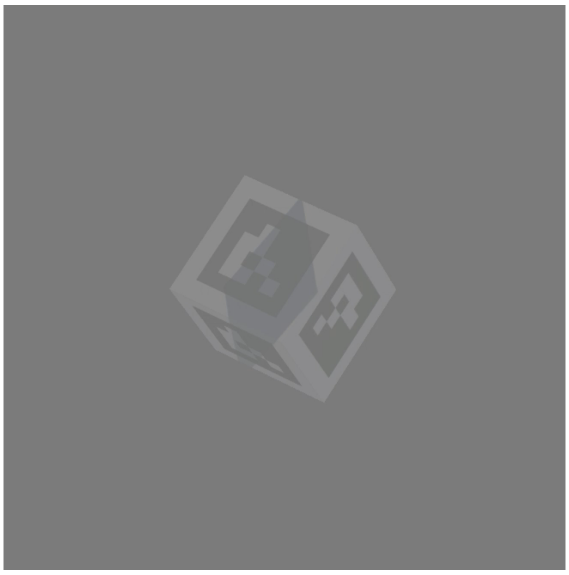


(d) Rain drops in screenshot 91.

Figure 3.11: Figure a), b), c) and d) shows how the rain drops varies in the images.

Changing contrast

Another parameter changed to add more difficulty to the detection was the contrast. A dataset with 100 images was created with the contrast decreased by -90%, and another dataset containing 100 images where the contrast was increased by 90%. To change the contrast, the package that was downloaded when the brightness was adjusted could be used. This was the package "SimpleLUT". Just under the sliding bar to adjust the brightness, there is a new sliding bar where the contrast can be adjusted. The contrast was chosen to be decreased to -90% and increased to 90% to see the biggest difference and use the generated data to test the detection algorithm.


(a) Decreasing contrast by -90% in screenshot 1.

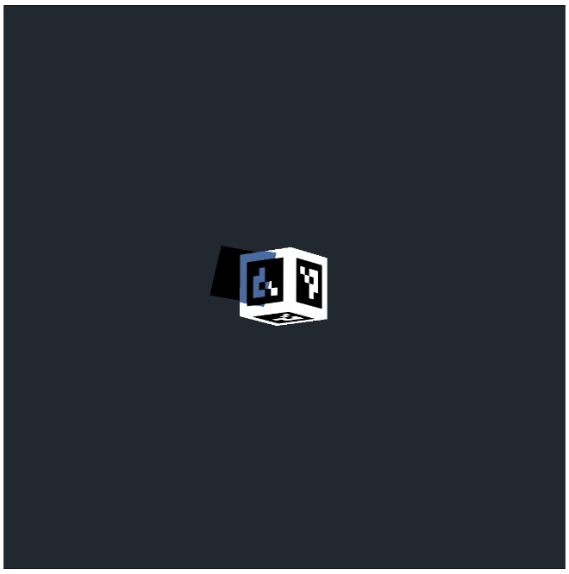
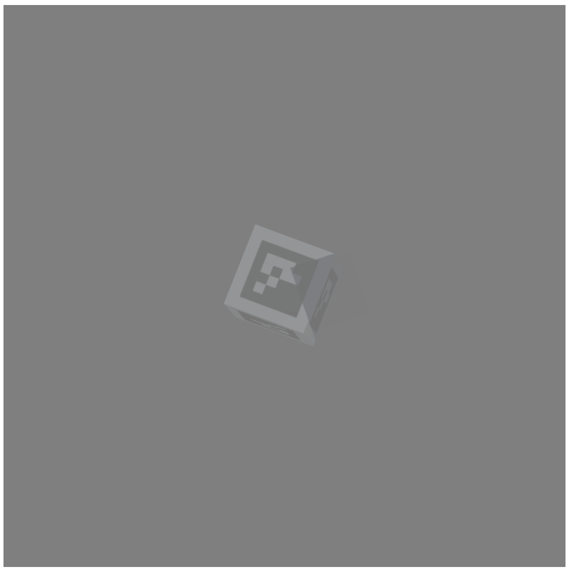
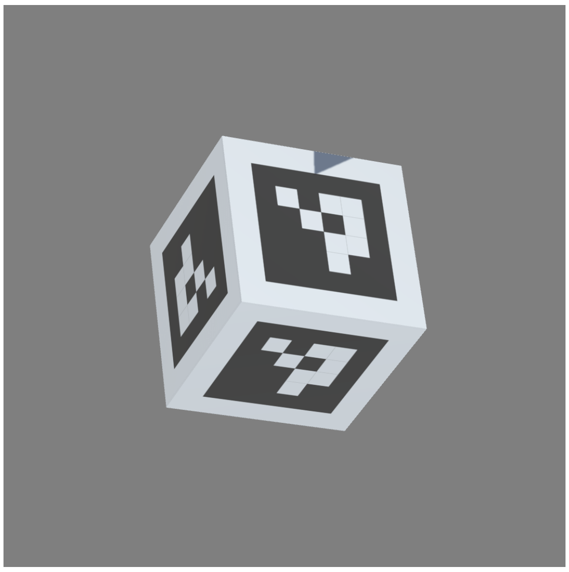

(b) Increasing contrast by 90% in screenshot 1.

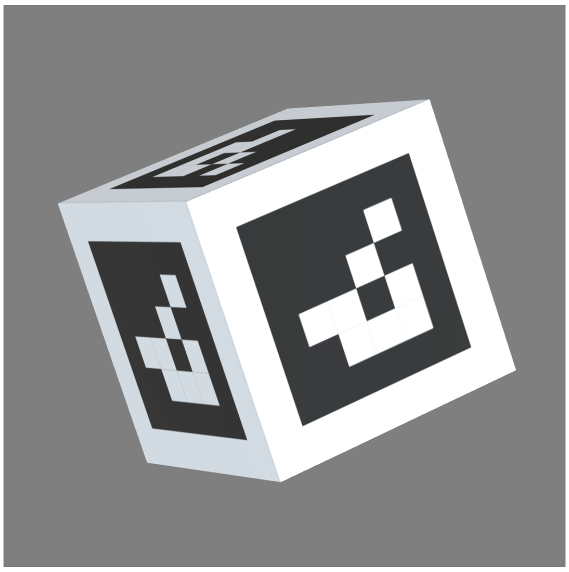
Figure 3.12: Figure a) and b) shows how the contrast varies in the images from -90% to +90%.

Adding fog in the virtual environment

To create a dataset with more interruptions, fog was added to the scene. This was done by clicking on the "window" tab in the upper left corner, and then choose "Rendering" and select "Lighting". Then a lighting panel opens beside the inspector window. In this panel, "Environment" was chosen and fog was added. Here the desired density, color and other adjustments for the fog could be added [34]. The color was set to a light gray color to attempt to accurately match the real world. The density was tested with different adjustment, but 0.004 was chosen for the dataset as the marker could be seen the entire time then. A dataset consisting of 100 images was made with this interruption in the scene as with to have a variation to test the detection accuracy. Figure 3.13 shows a random selection of some images with fog in the environment. As seen in the images, the fog is denser on the markers that are further away and clearer on the markers that are closer to the camera, just as in the real world.


(a) Fog drops in screenshot 3.


(b) Fog drops in screenshot 41.


(c) Fog drops in screenshot 51.

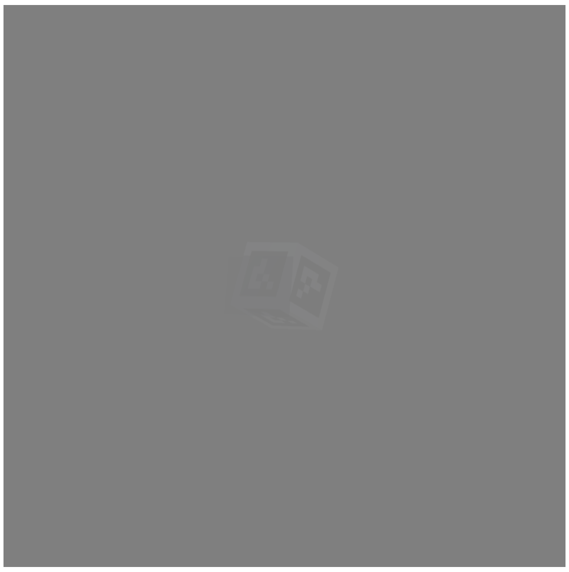

(d) Fog in screenshot 91.

Figure 3.13: Figure a), b), c) and d) shows how the fog varies on the images.

3.7 Detection algorithm

3.7.1 Chosen algorithm: DeepTag

The algorithm chosen for training the model to recognize the ArUco marker in different lighting and motion blur is DeepTag. DeepTag is chosen since it uses a two-stage framework that is robust and accurate, and can be used for different marker families since it detects Region of Interest (ROI), keypoints and symbols to decide the markers' ID [10]. This algorithm was also tested to give excellent results on motion blur, which was taken into the consideration for choosing algorithm since motion blur was one of the desired investigated disturbances in this project. For the detection, DeepTag is given an input image to start with. When the input image is implemented in the algorithm, groups of keypoints are detected. The different groups DeepTag can detect each belong to a specific marker. After this, DeepTag uses a two-stage detection scheme to achieve accuracy and robustness. This scheme can be seen in Figure 3.14. Figure 3.15 shows the network structure in details. The DeepTag code can be seen in Git[2].

[2]https://github.com/herohuyongtao/deeptag-pytorch

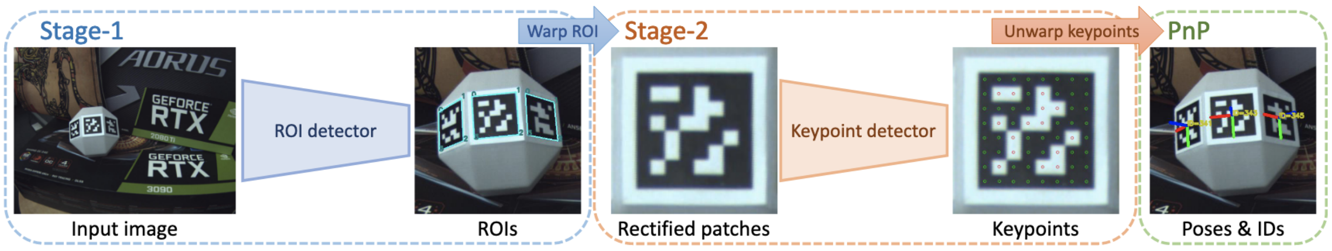


Figure 3.14: This figure shows the overview of the system. It uses a two-stage scheme to detect the ROIs of the marker and then estimates the group of keypoints. From this, the 6-DoF poses for the marker can be detected and the ID is determined. This figure is borrowed from the DeepTag report [10].

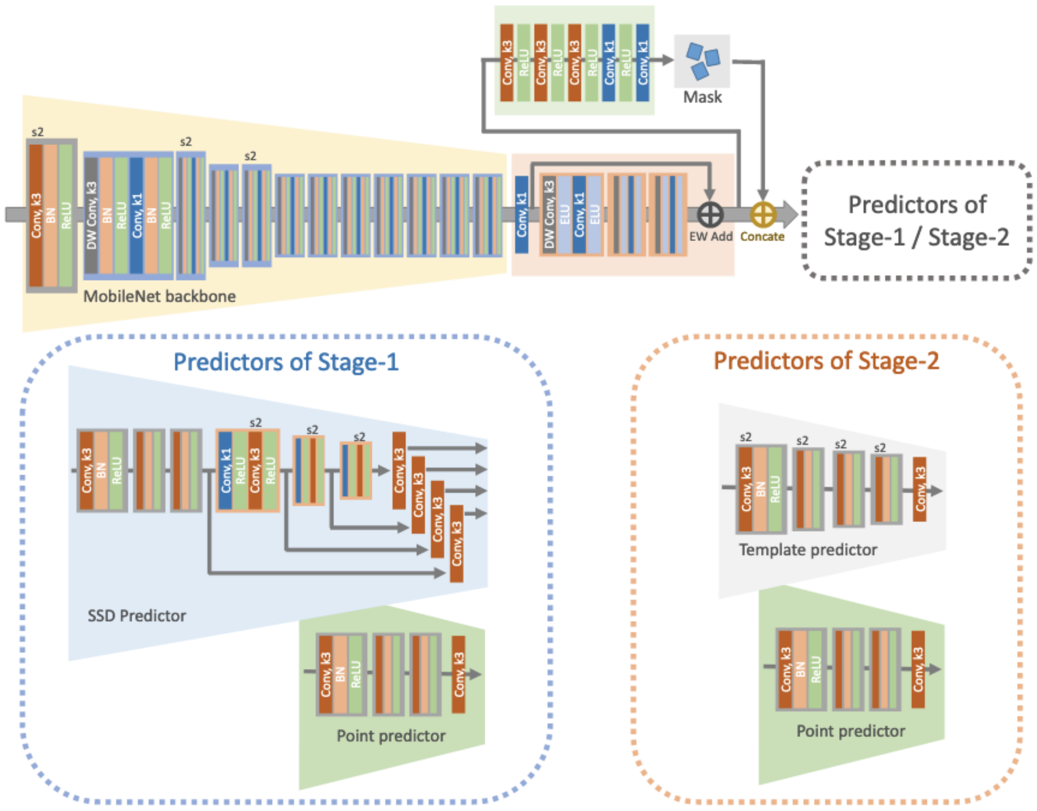


Figure 3.15: This figure shows the structure of the Network used for detection ROI and the keypoints. "Conv" for convolutional layer, "DW Conv" for depthwise convolutional layer, "k1" for size 1 of spatial kernel, "k3" for size 3 of spatial kernel, "s2" for convolution which has a stride of 2, and for the element-wise addition "EW add" is used. This figure is borrowed from the DeepTag report [10].

The way DeepTag works is that ROI, which stands for Region of Interest, is estimated for every marker that is a potential first. In clockwise order, each ROI has at least 4 non-collinear points. After this, it generates a rectified patch which contains a homography matrix. This matrix maps the Region of Interest to points in the patch that are predefined. The detected symbols and keypoints are sorted and estimated in this patch such that the ID of the marker can be found by checking the library for the different markers for the decoded symbols. By applying the homography matrix inverse, the positions of the keypoints in the original image can be obtained. Since the marker size is now known, the Perspective-n-Point (PnP) algorithm is then used to estimate the 6-DoF pose of the marker [10].

3.7.2 Using DeepTag on ArUco dataset

The algorithm and scripts from DeepTag were available in git from the DeepTag article [29]. After cloning the git repository, the setup for DeepTag could be used to detect the ArUco markers in the images from Unity. To use the DeepTag algorithm on the dataset of ArUco markers, a json file with the necessary parameters for the DeepTag algorithm had to be created. A file called "config_aruco.json" which contained information about the input image was made. The necessary information from the input image was whether the input was a video or an image, filepath to the image, marker family, hamming dist, which is checking the marker library, codebook of the marker, camera intrinsic matrix, camera distortion coefficients, and the size of the marker. After the json file was made, the code for the DeepTag algorithm could be used with the new json file.

The DeepTag code first imports arguments, then loads the config and the json file with all of the parameters. After this is done, the models and the marker library are loaded. Then the code tries to detect the markers in the image and decides the family, the number of markers, and where in the image they are.

Chapter 4

Results

The results of this project are divided into sections to get a better overview over each step. The first section shows the results from generating the ArUco markers. The next section shows the results from the virtual environment in Unity, and then the results from the detection of ArUco markers are shown.

4.1 ArUco marker dataset

The ArUco markers were generated in Python. The marker generation result can be seen in the figures below. Figure 4.1 shows ArUco marker number 0, which is the one used in this project. To demonstrate some examples of other markers, marker 20 and 33 are also generated and shown below in Figure 4.2 and Figure 4.3.

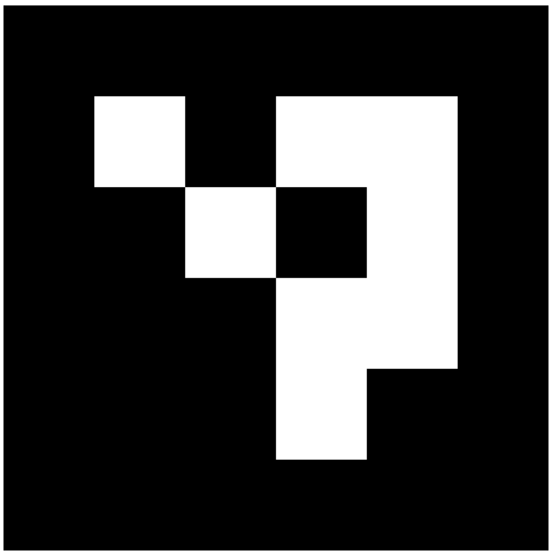


Figure 4.1: ArUco marker 0.

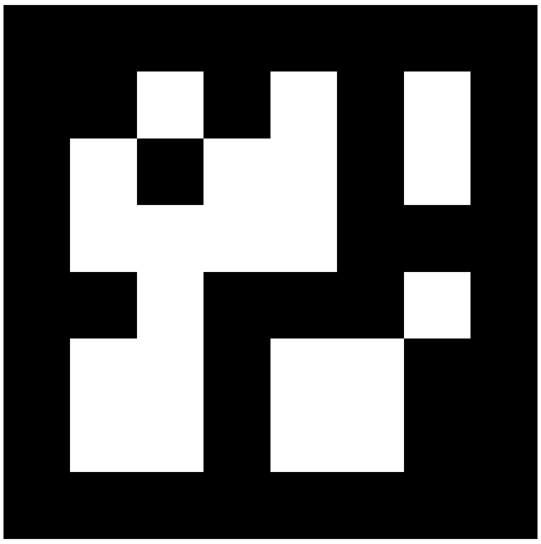


Figure 4.2: ArUco marker 20.

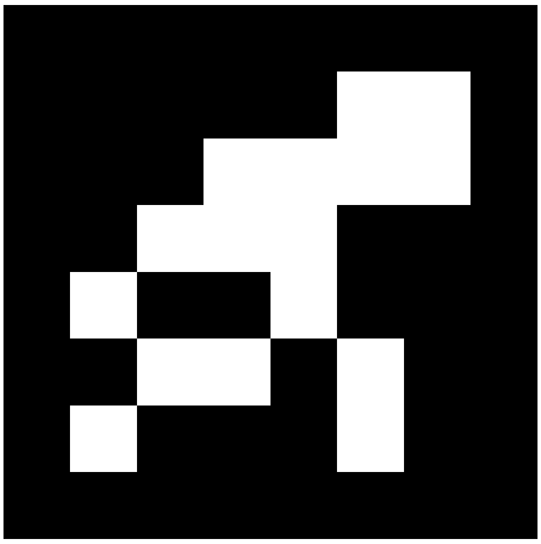


Figure 4.3: ArUco marker 33.

4.2 Virtual environment in Unity

The virtual environment for the markers was made in Unity. The environment was created and set up in the desired way and the results are shown in the figures below. A small selection of the dataset is shown in the figures. The dataset consists of 1000 images (0-999) and 1000 json files (0-999). Each image has its own json file which contains the necessary information about the objects in the image. Figure 4.4 shows an image of the ArUco marker and Listing 4.1 shows the json file with the necessary information belonging to the image.

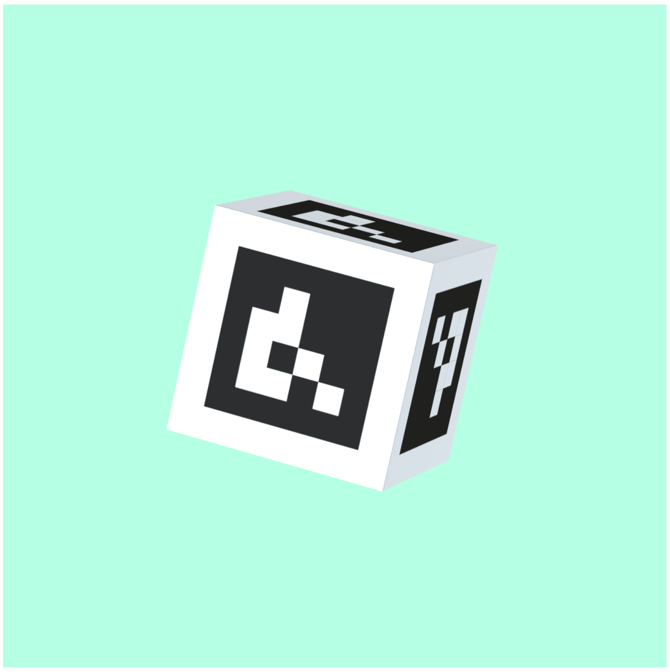


Figure 4.4: ArUco marker in the virtual environment in Unity, screenshot 802.

Listing 4.1: Data from the json file belonging to screenshot 802.

```
{"posx_aruco":0.0,"posy_aruco":334.0,"posz_aruco":0.0,
"orix_aruco":71.0,"oriy_aruco":72.0,"oriz_aruco":58.0,
"posx_camera":0.0,"posy_camera":500.0,"posz_camera":0.0,
"orix_camera":90.0,"oriy_camera":0.0,"oriz_camera":0.0,
"posx_shadow":9.0,"posy_shadow":550.0,"posz_shadow":-8.0,
"orix_shadow":85.0,"oriy_shadow":256.0,"oriz_shadow":1.0,"brightness":0.0}
```

As seen in Listing 4.1 the position and rotation of the different objects in Figure 4.4 are displayed. The position of the ArUco marker in x-, y- and z-direction is (0, 334,0). The rotation of the marker is (71,72,58). The position of the camera is fixed to (0,500,0), and the rotation to (90,0,0). The position of the shadow-plane in screenshot 802 is (9,550,-8) and the rotation is (85,256,1). The y-coordinate to the position of the shadow-plane is always fixed so that the shadow is higher than the camera, and not in front to cover the view.

Some other results with different parameters for the position and rotation of the marker and shadow-plane is shown in the figures below. In Figure 4.5, the marker is further away from the camera and the shadow-plane covers a larger part of the ArUco marker. Listing 4.1 contains the different parameters for the position and rotation of the marker, camera and shadow. The camera position is fixed in every scenario, but the position and rotation of the marker and the shadow-plane changes for each image. In Figure 4.5 the parameters for the position of the ArUco marker in x-, y- and z-direction are (0,24,0). The parameters for the rotation of the marker are (85,312,61). The camera is still fixed to (0,500,0) for the position and (90,0,0) for the rotation. The position of the shadow-plane is (2,550,2) and the rotation is (10,134,56).

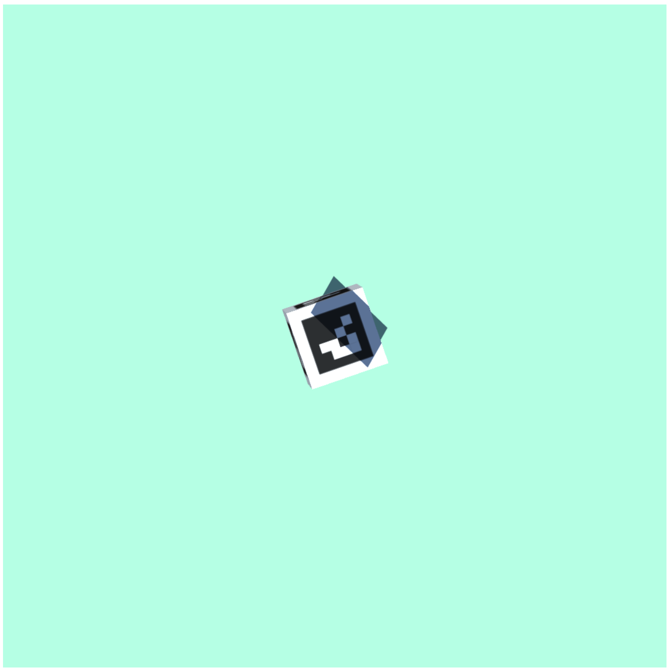


Figure 4.5: ArUco marker in the virtual environment in Unity, screenshot 287.

Listing 4.2: Data from the json file belonging to screenshot 287.

```
{"posx_aruco":0.0,"posy_aruco":24.0,"posz_aruco":0.0,
"orix_aruco":85.0,"oriy_aruco":312.0,"oriz_aruco":61.0,
"posx_camera":0.0,"posy_camera":500.0,"posz_camera":0.0,
"orix_camera":90.0,"oriy_camera":0.0,"oriz_camera":0.0,
"posx_shadow":2.0,"posy_shadow":550.0,"posz_shadow":2.0,
"orix_shadow":10.0,"oriy_shadow":134.0,"oriz_shadow":56.0,"brightness":0.0}
```

Figure 4.6 shows a different scenario where the marker is much closer to the camera. In this case the parameters for the position of the marker are (0,396,0) and the rotation is (47,202,58). Fixed camera at position (0,500,0) and rotation (90,0,0). For the shadow-plane, the position is (-1,550,4) and the rotation is (23,298,6). All of the parameters for Figure 4.6 can be seen in Listing 4.3.

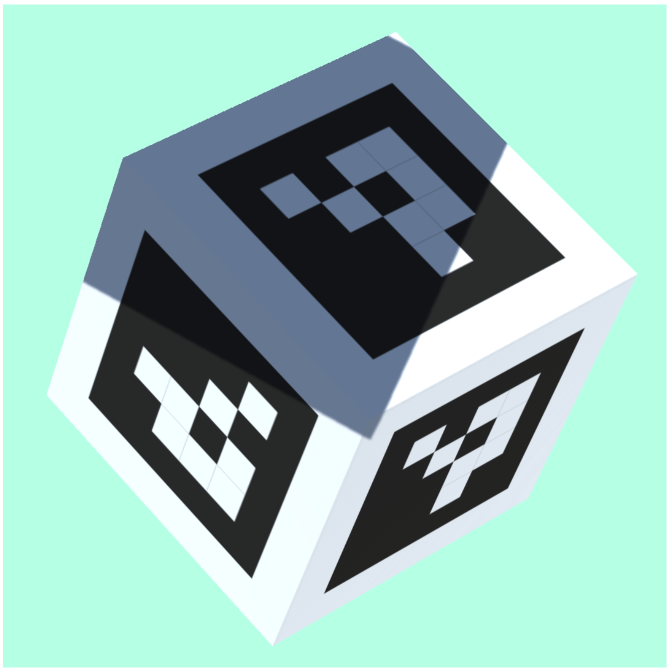
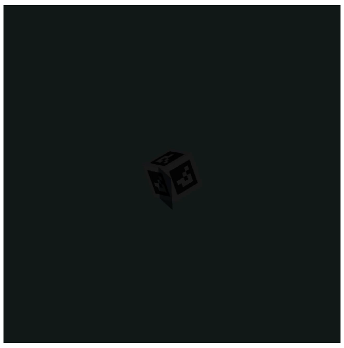


Figure 4.6: ArUco marker in the virtual environment in Unity, screenshot 34.

Listing 4.3: Data from the json file belonging to screenshot 234.

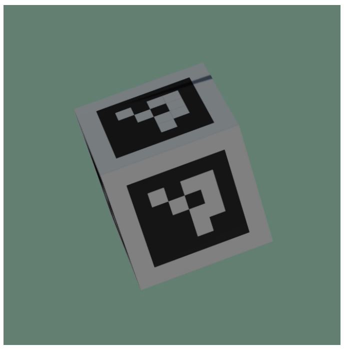
```
{"posx_aruco":0.0,"posy_aruco":396.0,"posz_aruco":0.0,
"orix_aruco":47.0,"oriy_aruco":202.0,"oriz_aruco":58.0,
"posx_camera":0.0,"posy_camera":500.0,"posz_camera":0.0,
"orix_camera":90.0,"oriy_camera":0.0,"oriz_camera":0.0,
"posx_shadow":-1.0,"posy_shadow":550.0,"posz_shadow":4.0,
"orix_shadow":23.0,"oriy_shadow":298.0,"oriz_shadow":6.0,"brightness":0.0}
```

The same procedure was repeated for the generation of the images with different brightness, motion blur and different interruption on the camera lens. A smaller dataset containing 100 images of each alternative was created. All of the screenshots in this dataset also have json files containing the parameters for each image. The different parameters for the position, rotation and the shadow were still changed, but one more parameter was added. In the first step, the adjustment of brightness was added to the scene. An excerpt of the images is shown below. Figure 4.7 shows 12 images where the brightness is adjusted to -90%, -50%, 50% and 90%. After the detection with adjustment to the brightness had been performed, motion blur was added. Figure 4.8 shows some of the results from the dataset with motion blur in the scene. When markers with motion blur were detected, other interruptions were added to the scene. The first interruption was rain drops added to the camera lens, some images of this is shown in Figure 4.9. After the rain drops were added, the contrast was changed, which is shown in Figure 4.10. The last interruption added was fog. This was added to the scene to check how the detection would work if there was fog in the environment. Some images from the dataset with fog is shown in Figure 4.11.

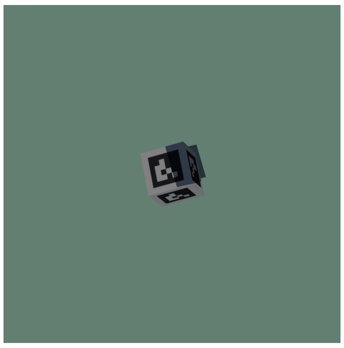

(a) Brightness at -90% on screenshot 1.

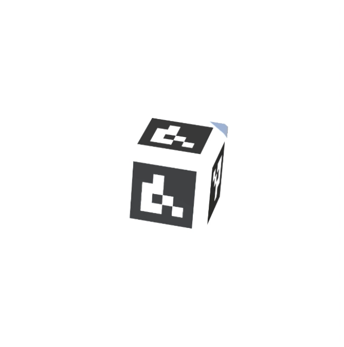
(b) Brightness at -90% on screenshot 35.

(c) Brightness at -90% on screenshot 41.


(d) Brightness at -50% on screenshot 4.

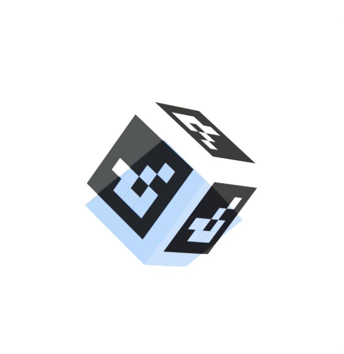
(e) Brightness at -50% on screenshot 17.

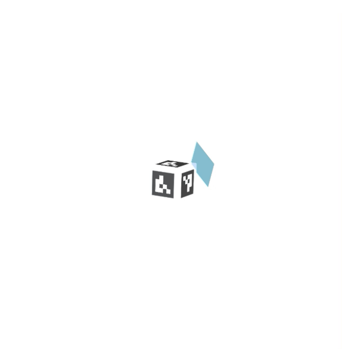

(f) Brightness at -50% on screenshot 59.


(g) Brightness at 50% on screenshot 8.

(h) Brightness at 50% on screenshot 21.

(i) Brightness at 50% on screenshot 94.


(j) Brightness at 90% on screenshot 6.


(k) Brightness at 90% on screenshot 15.

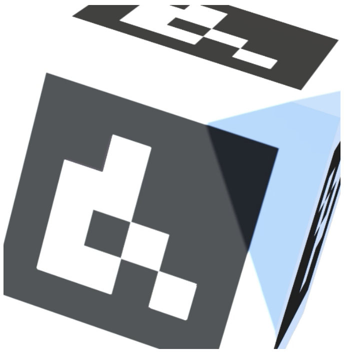
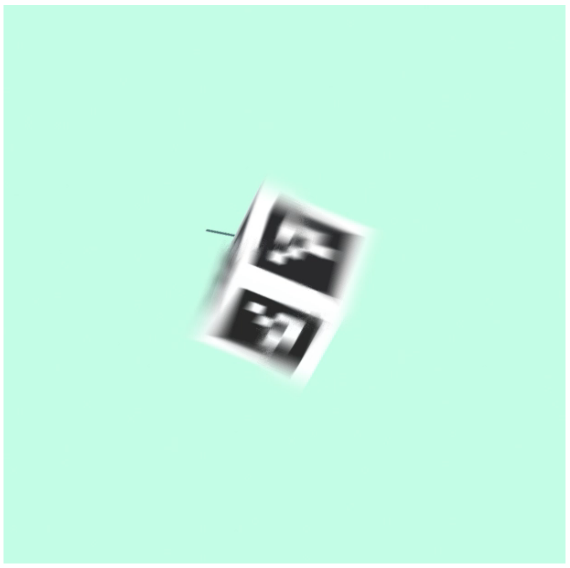

(l) Brightness at 90% on screenshot 21.

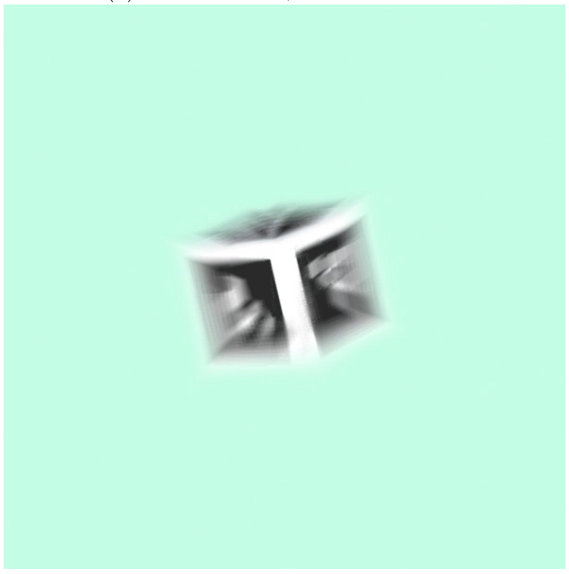
Figure 4.7: Figure a), b) and c) shows the images with brightness at -90%. Figure d), e) and f) shows the images with brightness at -50%. Figure g), h) and i) shows the images with brightness at 50%. Figure j), k) and l) shows the images with brightness at 90%.

(a) Motion blur, screenshot 3.


(b) Motion blur, screenshot 5.

(c) Motion blur, screenshot 13.

(d) Motion blur, screenshot 29.


(e) Motion blur, screenshot 59.

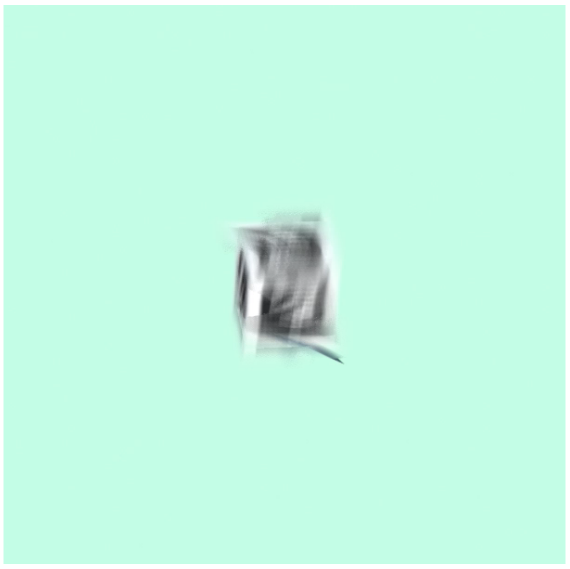

(f) Motion blur, screenshot 97.

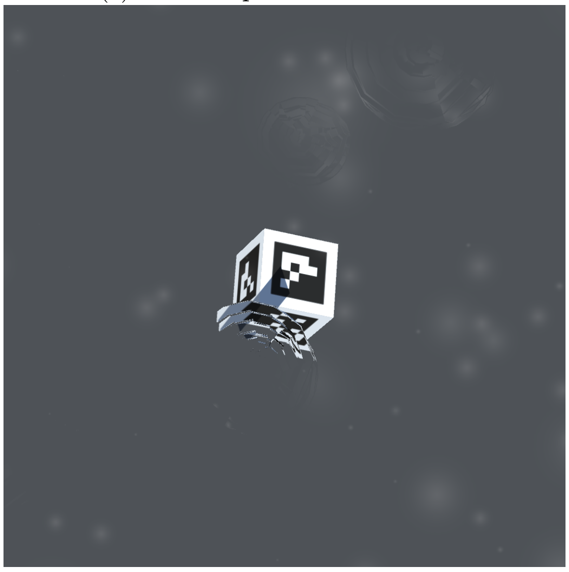
Figure 4.8: Figure a), b), c), d), e) and f) shows the images with motion blur added.

(a) Rain drops in screenshot 2.

(b) Rain drops in screenshot 11.

(c) Rain drops in screenshot 31.

(d) Rain drops in screenshot 45.


(e) Rain drops in screenshot 80.

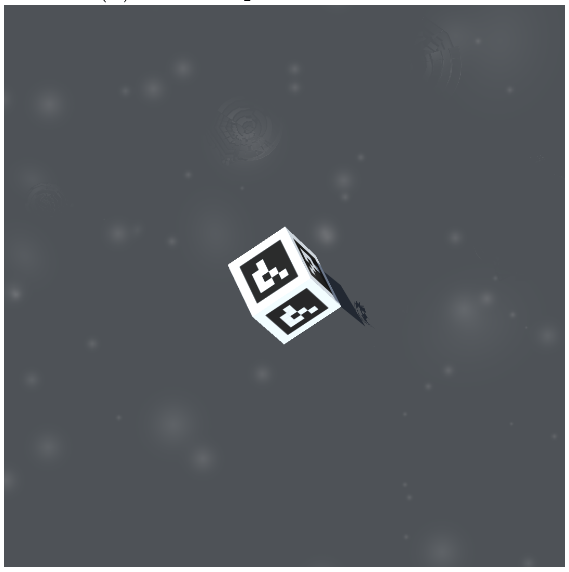
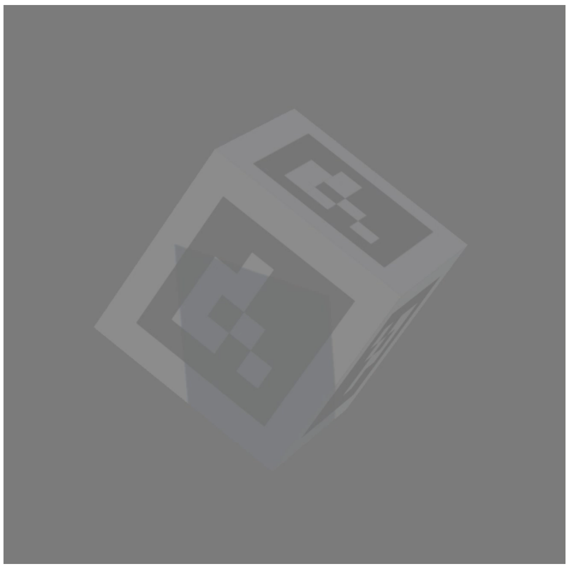
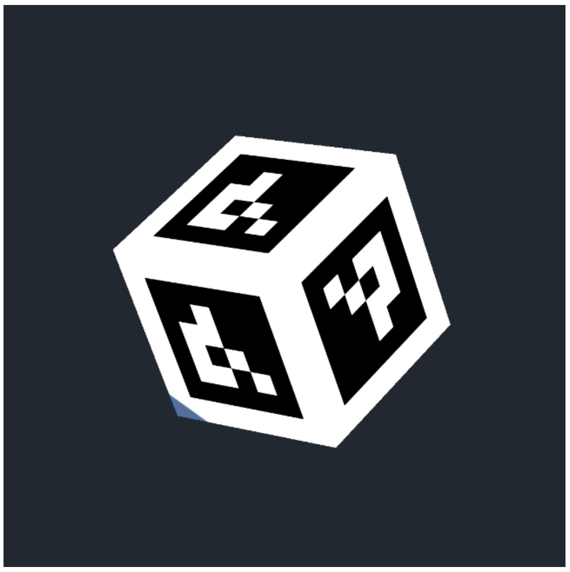
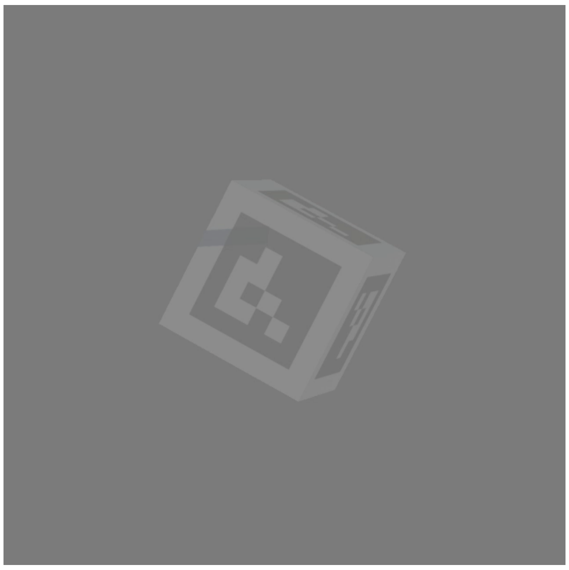
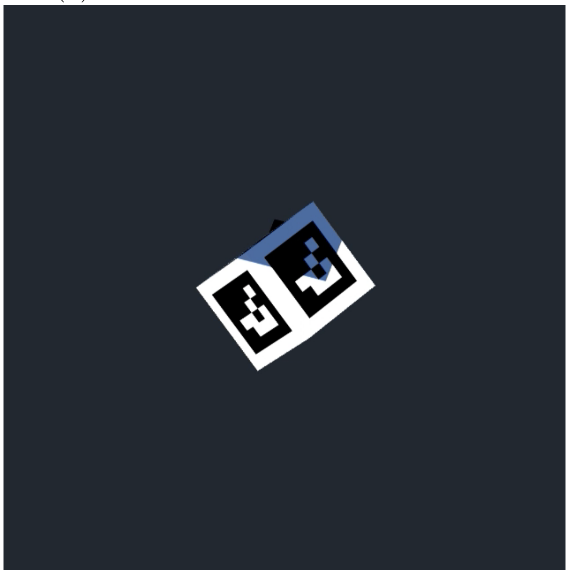

(f) Rain drops in screenshot 96.

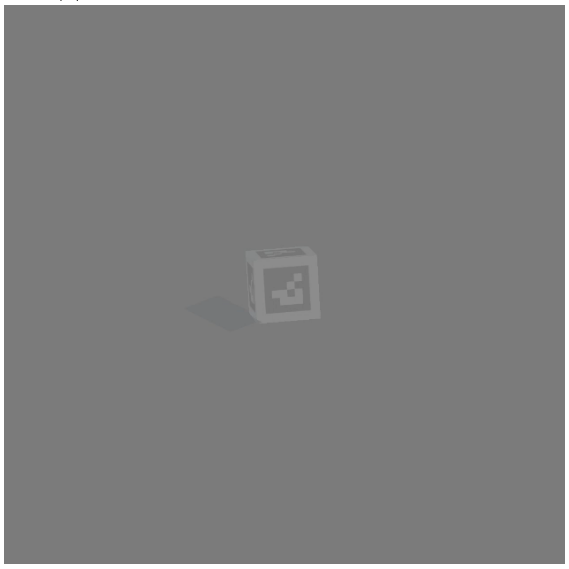
Figure 4.9: Figure a), b), c), d), e) and f) shows a variation of the images with rain drops on the camera lens.


(a) Decreased contrast in screenshot 8.


(b) Increased contrast in screenshot 10.


(c) Decreased contrast in screenshot 17.


(d) Increased contrast in screenshot 15.


(e) Decreased contrast in screenshot 42.

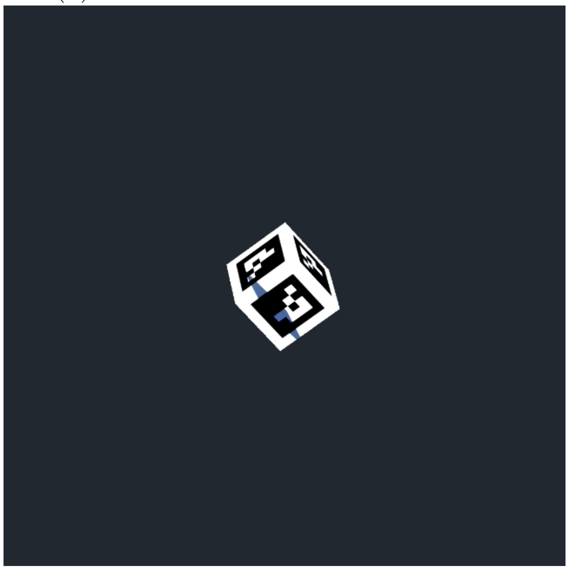
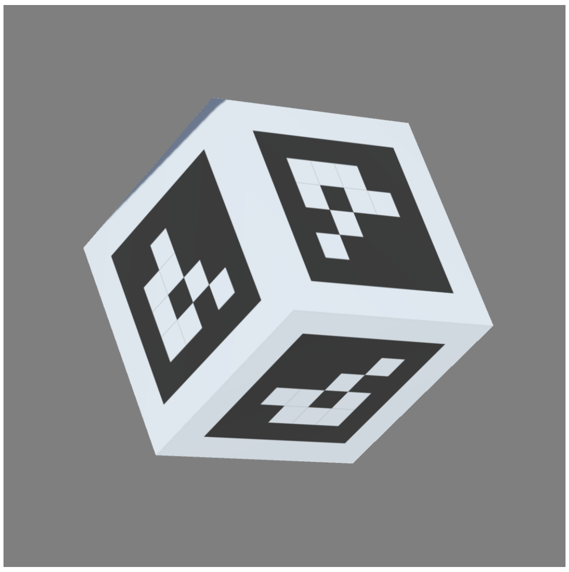
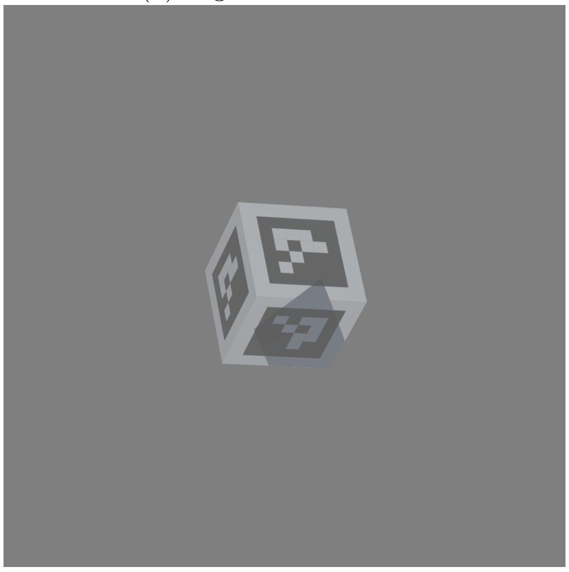

(f) Increased contrast in screenshot 71.

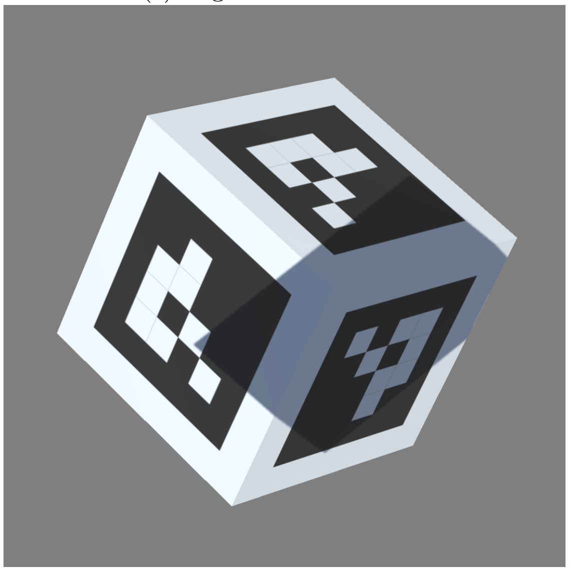
Figure 4.10: Figure a), b), c), d), e) and f) shows the different contrast between decreased contrast to -90% and increased contrast to 90%.

(a) Fog in screenshot 6.


(b) Fog in screenshot 17.

(c) Fog in screenshot 23.


(d) Fog in screenshot 31.


(e) Fog in screenshot 53.

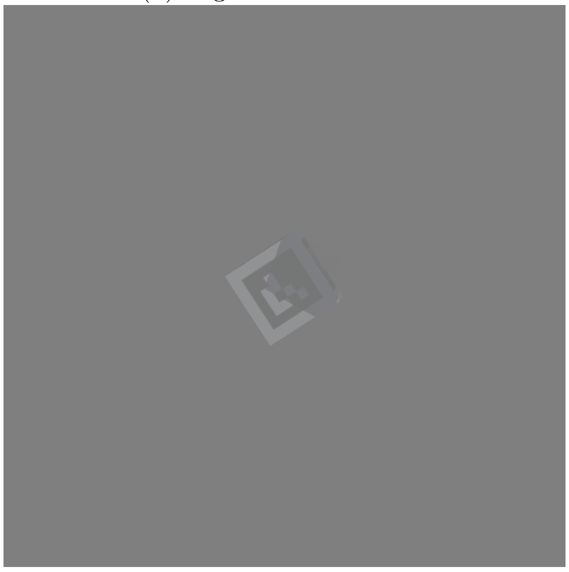

(f) Fog in screenshot 60.

Figure 4.11: Figure a), b), c), d), e) and f) shows the fog with the different density depending on how close the marker is.

4.3 Detection of ArUco markers

To detect the ArUco markers, the existing detection algorithm "DeepTag" was used. The results from marker detection with different positions, rotations, shadow, brightness, motion blur and interruptions on the camera view can be seen in the next sections.

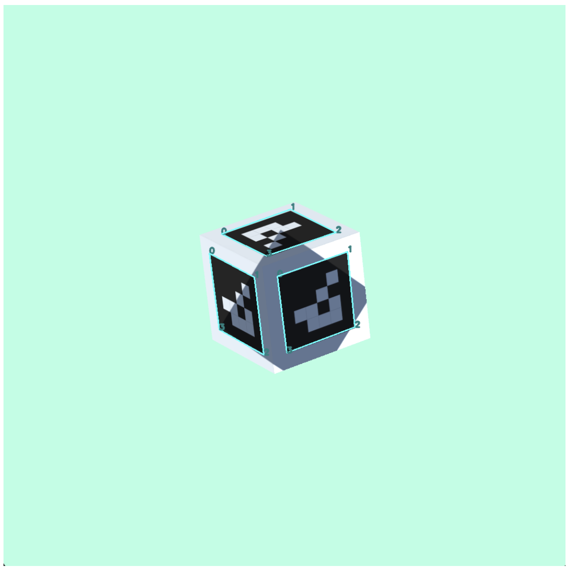
4.3.1 Detection of ArUco markers with shadow and different positions and rotations

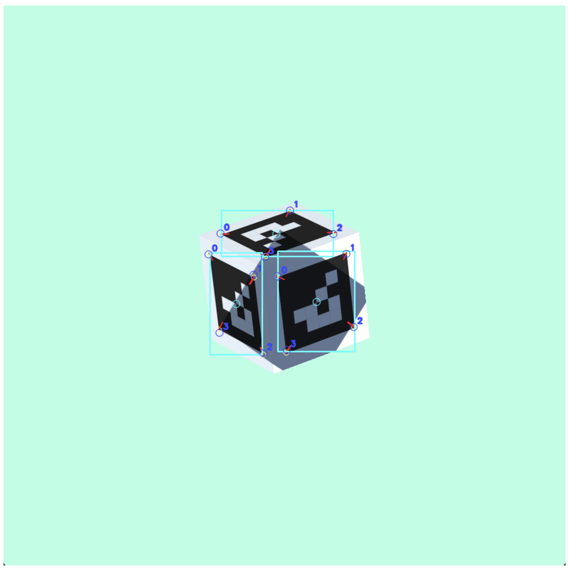
Out of the dataset with a 1000 images, approximately 1/3 of the images were used with the DeepTag detection algorithm to test the detection quality of the algorithm on this dataset. The first 300 images of the dataset were used. Of 300 images, the DeepTag algorithm was able to detect almost all markers. There were 17 images where the DeepTag algorithm was not able to detect the markers since these markers were too close to the camera and the whole marker was not visible. This means that the DeepTag algorithm was able to detect 94,4% of the markers in the dataset with and without shadow, and also in different positions and rotations. If the images too close to the camera for detection are disregarded, the DeepTag algorithm was able to detect the ArUco marker in different positions and with different rotations, and with shadow covering half or all of it, with 100% accuracy. The figures below show some of the different scenarios with shadow, without shadow, marker close to camera, marker far away and when the markers is too close to the camera and can not be detected. Table 4.1 shows the results from the detection of ArUco markers with different position, rotation and shadow.

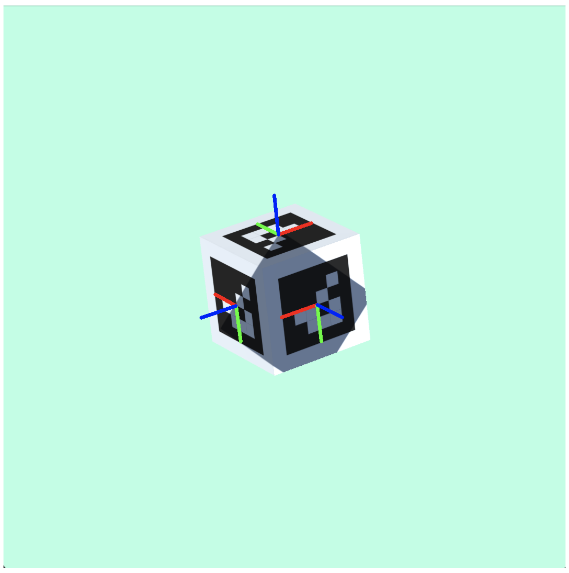
Table 4.1: Results from the detection with shadow and different positions and rotations.

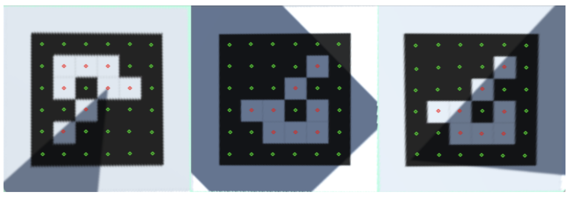
Position, rotation and shadow	**Number of images**	**Detection rate (%)**
Markers detected	283/300	94,4
Marker too close to camera lens	17/300	0

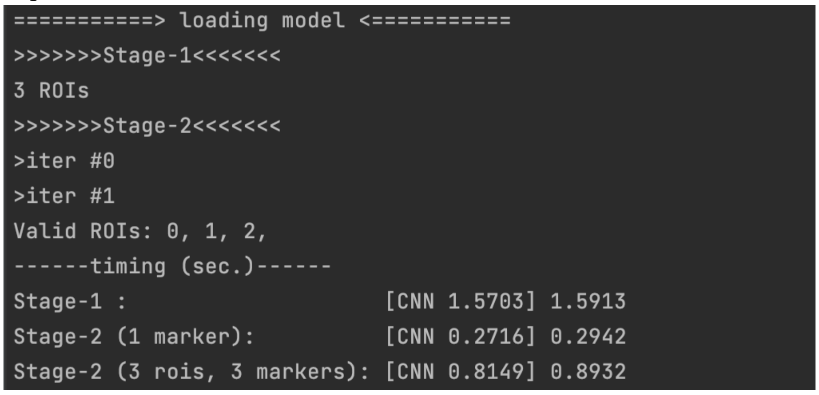
The figures below are from screenshot 45, from the dataset of the 1000 images. As seen in the screenshot, this image has three ArUco markers, and is largely covered by the shadow-plane. The figures shows each step for the detection of the ArUco markers in the image. Figure 4.12a shows the ROIs detected, and is a part of the first step of the detection model. The figure shows 3 ROIs detected. Figure 4.12b is also a part of the first step, and shows the detection of boxes and corners of the markers in the image. In Figure 4.12c, the poses of the markers are shown. Seen here are 3 poses and with the directions. Figure 4.12d is a part of the second stage, and shows the markers detected in the image. This stage also shows the right orientation of the marker. As seen in the keypoint detection, three markers are shown with different orientations. In Figure 4.12e, the terminal from Python can be seen. The terminal shows the loading of the model, including each step. It also shows how many ROIs and how many markers that are detected. Here there are 3 ROIs and 3 markers detected, which matches with the image. The result is that the DeepTag algorithm is able to detect the right amount of ArUco markers with shadow in the image.


(a) Detection of ROIs (Stage-1) in screenshot 45.


(b) Detection of boxes and corners (Stage-1) in screenshot 45.


(c) Image with pose in screenshot 45.


(d) Keypoints detected (Stage-2), number of markers on screenshot 45.



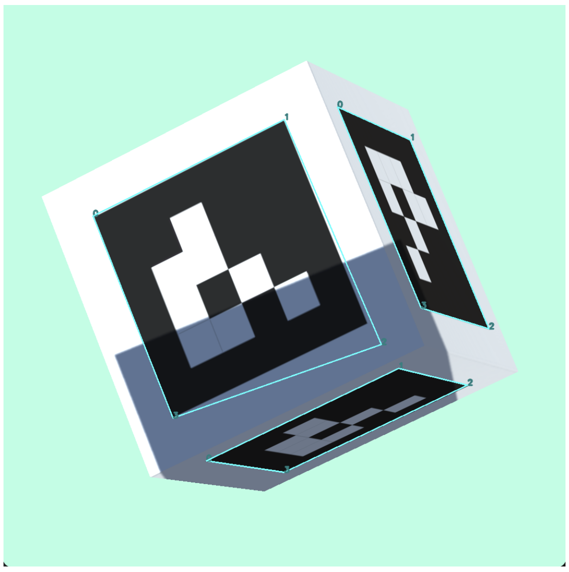
```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
3 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1, 2,
------timing (sec.)------
Stage-1 :                     [CNN 1.5703] 1.5913
Stage-2 (1 marker):           [CNN 0.2716] 0.2942
Stage-2 (3 rois, 3 markers): [CNN 0.8149] 0.8932
```

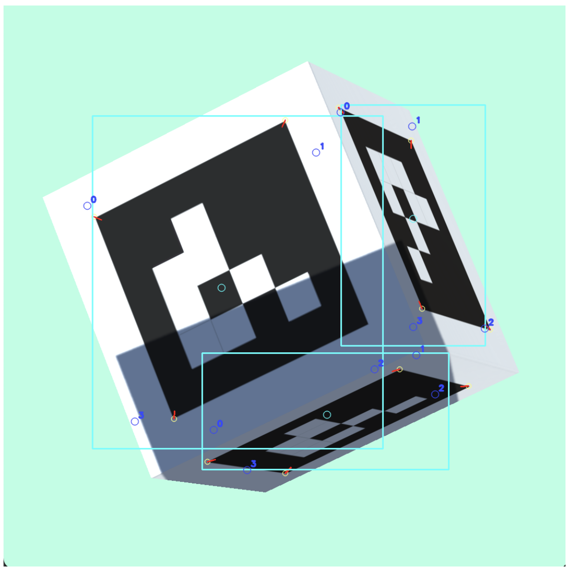
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 45.

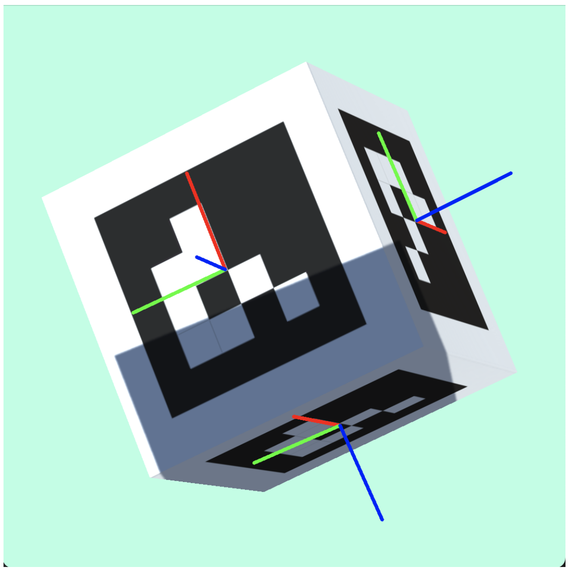
Figure 4.12: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model.

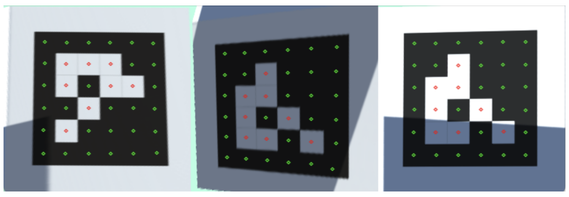
The figures below show the same results as the figures above, the markers are detected. The difference in the image used below, screenshot 50, is that the cube with the markers is much closer to the camera view and only half of it is covered by shadow. The detection algorithm is able to detect all of the three markers in the image. Figure 4.13a shows the ROIs detected. Figure 4.13b shows the detection of boxes and corners. Both of these images are a part of the first stage. Figure 4.13c shows the poses and the direction of the ArUco markers. The keypoints detected are shown in Figure 4.13d, and is a part of the second stage. The last figure that belongs to screenshot 50 is of the terminal, shown in Figure 4.13e. The terminal for this image shows the loading model, stage 1, stage 2 and that 3 ROIs and 3 markers are detected. This results matches with the keypoints detected and the number of markers in the image.

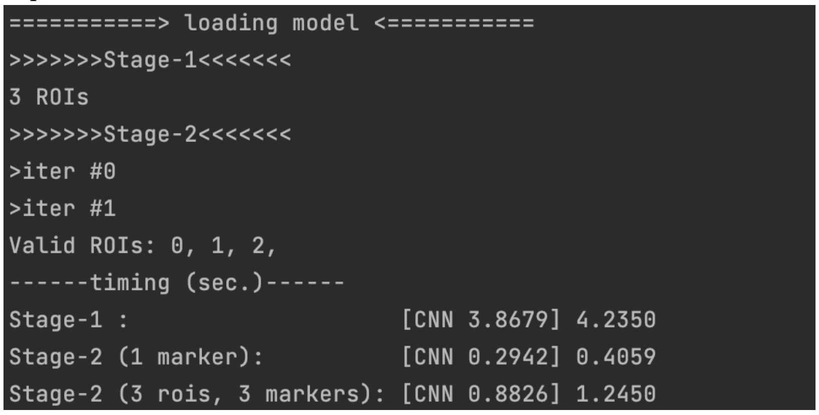
Under the image of screenshot 50, another image of screenshot 86 is shown. From screenshot 86, another rotation of the marker is shown. In this image, only two sides of the cube with the ArUco markers are visible from the camera view. Therefore, only two markers can be seen. The markers are not covered by shadow and the detection works properly as well. Figure 4.14a shows that two ROIs are detected. Figure 4.14b shows that two sides of the box and corners are detected, which completes stage one. Figure 4.14c shows the poses and the direction of the markers. Stage two is completed and shown in Figure 4.14d, and shows the keypoints detected and that two markers are detected. This can be confirmed in the terminal in Figure 4.14e where the loaded model is shown. This shows that 2 ROIs and 2 markers are detected.


(a) Detection of ROIs (Stage-1) in screenshot 50.


(b) Detection of boxes and corners (Stage-1) in screenshot 50.


(c) Image with pose in screenshot 50.

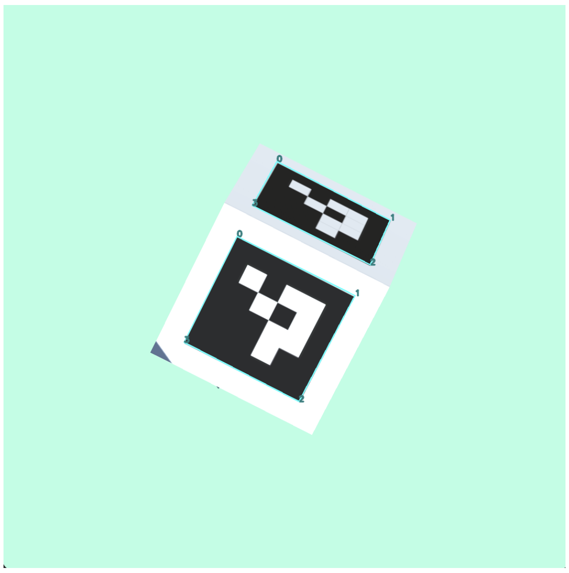

(d) Keypoints detected (Stage-2), number of markers on screenshot 50.

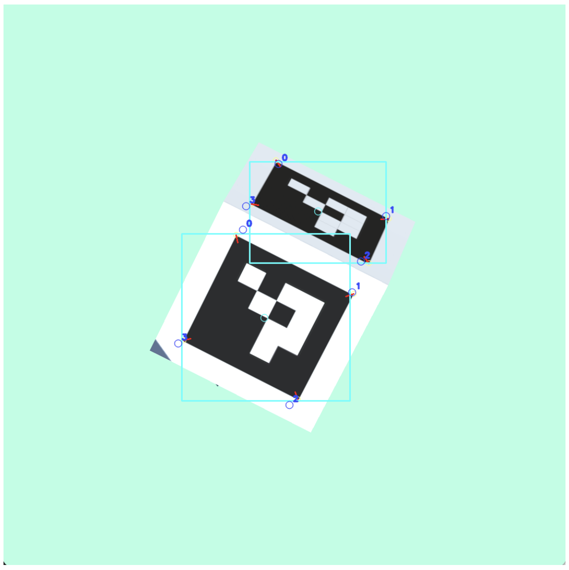


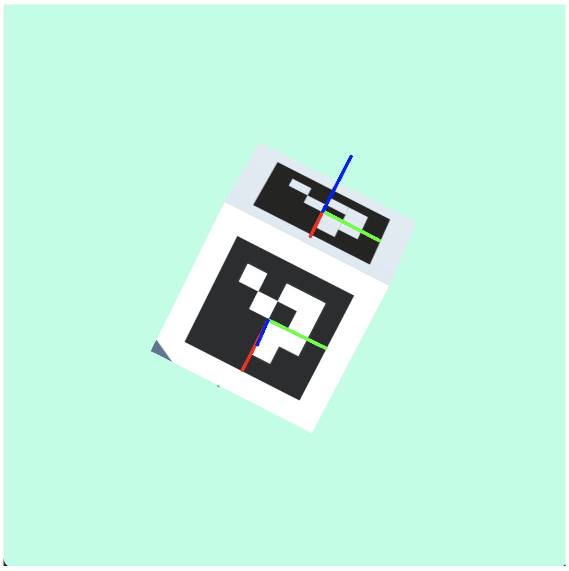
```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
3 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1, 2,
------timing (sec.)------
Stage-1 :                    [CNN 3.8679] 4.2350
Stage-2 (1 marker):          [CNN 0.2942] 0.4059
Stage-2 (3 rois, 3 markers): [CNN 0.8826] 1.2450
```

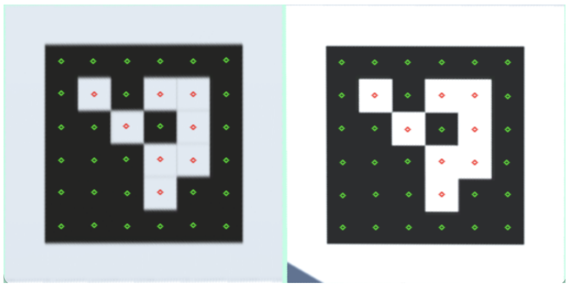
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 50.

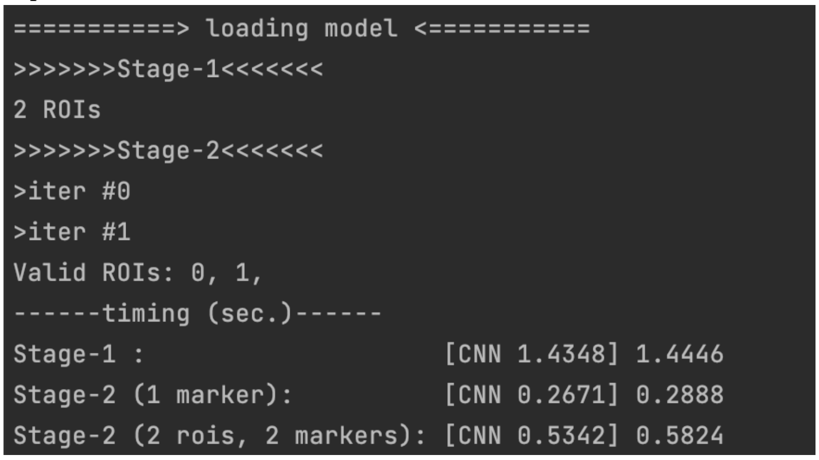
Figure 4.13: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model.


(a) Detection of ROIs (Stage-1) in screenshot 86.


(b) Detection of boxes and corners (Stage-1) in screenshot 86.


(c) Image with pose in screenshot 86.


(d) Keypoints detected (Stage-2), number of markers on screenshot 86.



```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
2 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1,
------timing (sec.)------
Stage-1 :                    [CNN 1.4348] 1.4446
Stage-2 (1 marker):          [CNN 0.2671] 0.2888
Stage-2 (2 rois, 2 markers): [CNN 0.5342] 0.5824
```

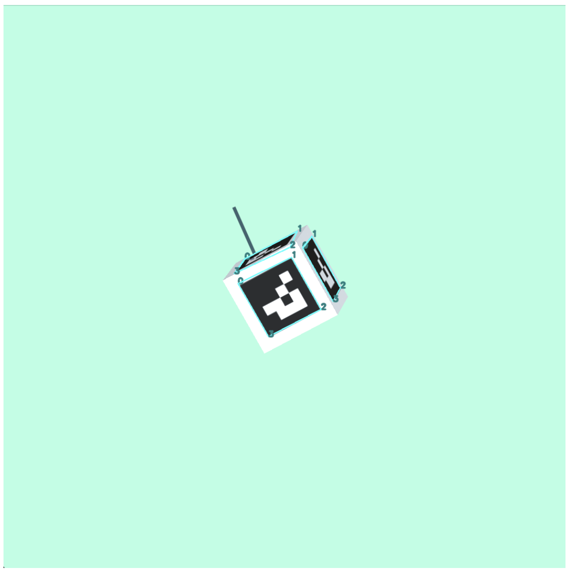
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 86.

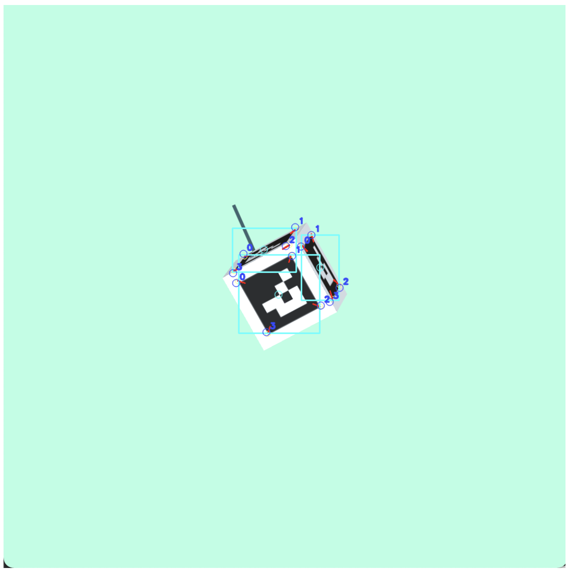
Figure 4.14: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model.

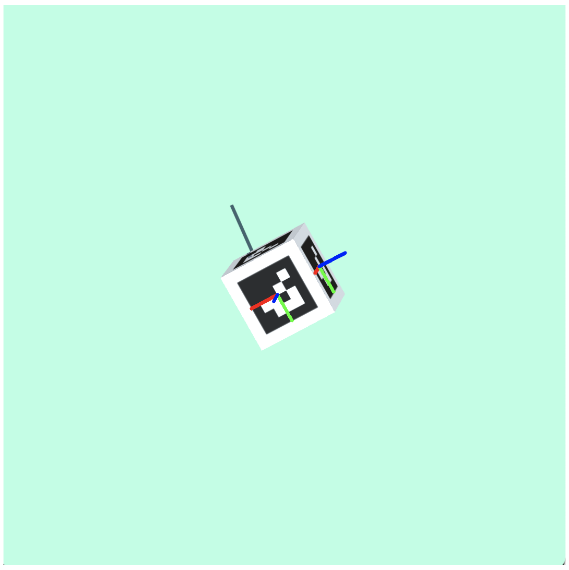
In some of the images, the cube with the ArUco markers on each side was rotated so much that one of the sides was barely visible. When this happened, the DeepTag algorithm was not able to detect every marker on every side of the cube, but it was always able to detect the marker in the front when the whole marker was in the image, no matter the position, rotation or with shadow. When this is used in a real scenario, there would not be a cube with markers on each side tilted as much as the sides of this cubes, and it would therefore not be a problem.

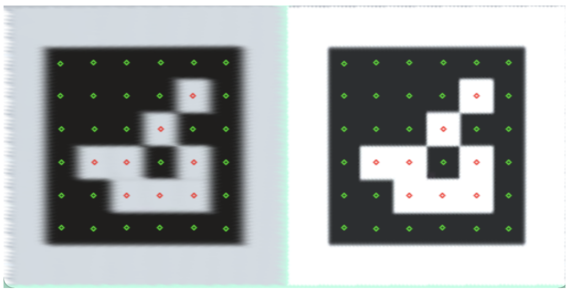
The figures below show an image where this occurs. Screenshot 138 shows two markers clearly and one top marker is barely visible. The results from the detection of this marker is shown in the images below. Figure 4.15a shows that three ROIs are detected. Figure 4.15b shows that two boxes and corners are detected. Figure 4.15c shows the poses of the two markers detected, and the keypoints with the two markers are shown in Figure 4.15d. The terminal shows the results with 3 ROIs and 2 markers detected. This is, however, acceptable as sufficient detection quality.

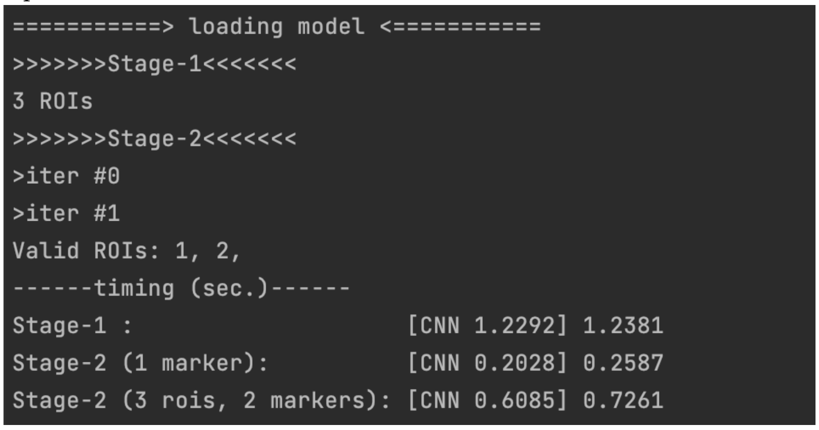
The figures from screenshot 53, which are below the images from screenshot 138, show when the DeepTag algorithm is unable to detect the markers. As seen here, the marker is too close to the camera for the whole marker to be visible. When this happens, the detection model is unable to detect the boxes and corners of the markers and can not set a pose or direction of the marker. Nor can it detect the ArUco marker. As seen from Figure 4.16a, the detection algorithm is able to detect one ROI from parts of the marker directly in front of the camera view. Figure 4.16b shows that there are no corners to detect, and therefore, Figure 4.16c can not detect any markers or poses for them. Thus in stage two, there are no keypoints detected. This is illustrated where the Figure 4.16d is just black without any markers. The terminal also confirms this results as 1 ROI is detected and 0 markers, seen in Figure 4.16e. This happens with 17 of the 300 screenshots from the dataset used for this part of the detection.


(a) Detection of ROIs (Stage-1) in screenshot 138.


(b) Detection of boxes and corners (Stage-1) in screenshot 138.


(c) Image with pose in screenshot 138.

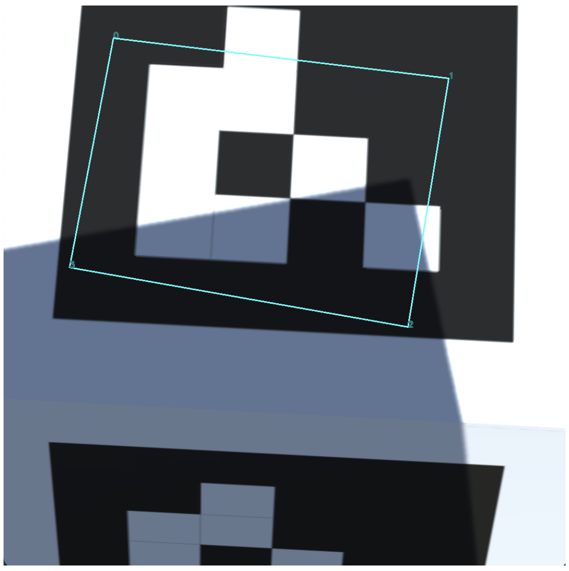

(d) Keypoints detected (Stage-2), number of markers on screenshot 138.

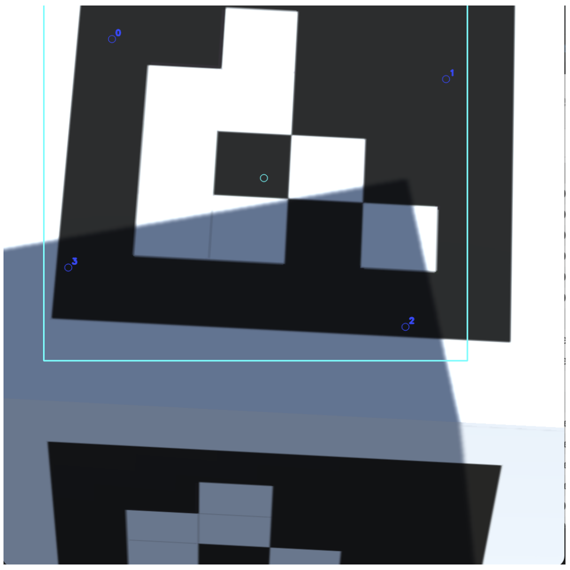


```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
3 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 1, 2,
------timing (sec.)------
Stage-1 :                    [CNN 1.2292] 1.2381
Stage-2 (1 marker):          [CNN 0.2028] 0.2587
Stage-2 (3 rois, 2 markers): [CNN 0.6085] 0.7261
```

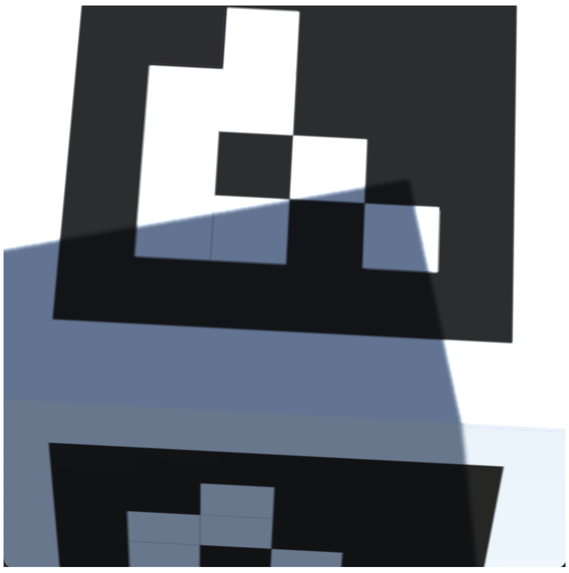
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 138.

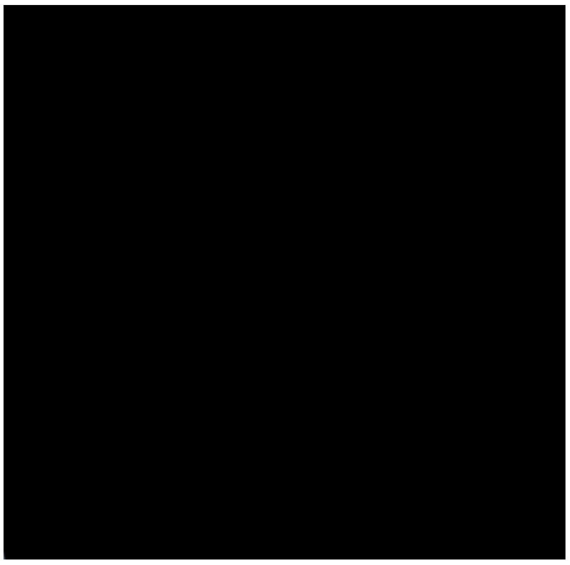
Figure 4.15: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model.


(a) Detection of ROIs (Stage-1) in screenshot 53.



(b) Detection of boxes and corners (Stage-1) in screenshot 53.


(c) Image with pose in screenshot 53.


(d) Keypoints detected (Stage-2), number of markers on screenshot 53.

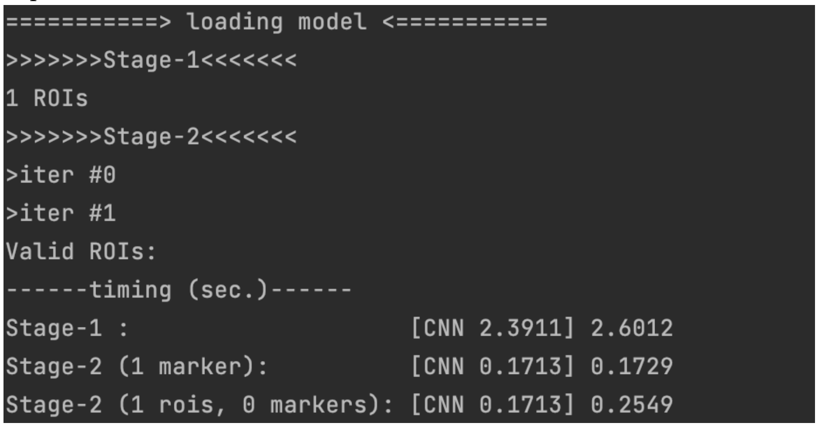

```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
1 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs:
------timing (sec.)------
Stage-1 :                     [CNN 2.3911] 2.6012
Stage-2 (1 marker):           [CNN 0.1713] 0.1729
Stage-2 (1 rois, 0 markers): [CNN 0.1713] 0.2549
```
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 53.

Figure 4.16: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model.

4.3.2 Detection of ArUco markers with different brightness in the scene

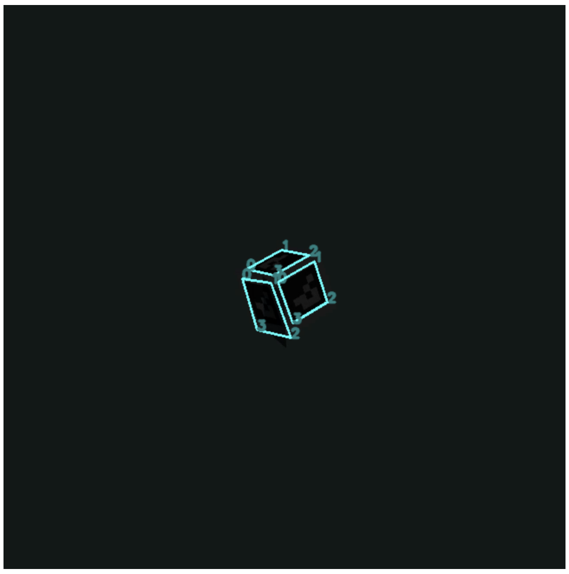
When the ArUco markers with different parameters for position, rotation and shadow were detected, more parameters were added to the scene. The next step was to adjust the brightness and the results are shown below. Four different stages of brightness were tested. First with -90% brightness, then -50% brightness to see if the detection worked with less light. After this, the brightness was increased to 50%, and then 90% to see if the detection also works with more light.

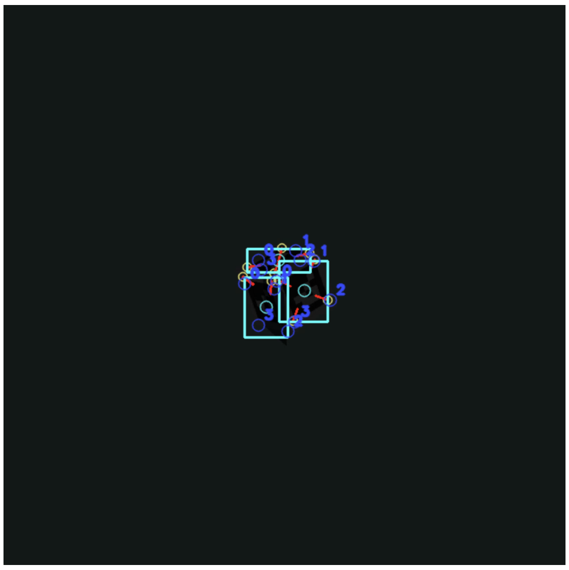
The results from the detection with different brightness had the same complications when marker was too close to the camera as in the first detection. When the whole marker could not be seen, the DeepTag algorithm was not able to detect the ArUco marker. On the dataset with brightness -90%, this occurred 6 times out of a dataset with 100 images, while it occurred 2 out of 100 times in the dataset with brightness -50%. This constitutes 6% and 2% of the datasets respectively. In the datasets with increased brightness it occurred 5 out of 100 times when the brightness was +50% and 3 out of 100 times when the brightness was +90%. This constitutes 5% and 3% of the datasets respectively.

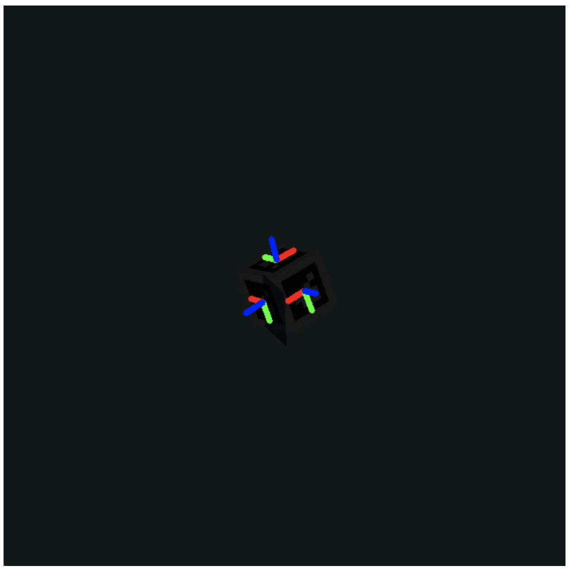
Figure 4.17 shows the result from the detection of screenshot 1 from the dataset with brightness decreased to -90%. Figure 4.17a shows the ROIs detected in the image. The boxes and corners in the image are shown in Figure 4.17b. These two steps are from the first stage of the detection. The next figure, Figure 4.17c, shows the image with pose and direction of the markers. Figure 4.17d shows the keypoints from stage two of the detection. The last image in the figure is from the terminal, Figure 4.17e. As seen from these results, the DeepTag algorithm is able to detect all of the 3 markers in this image.

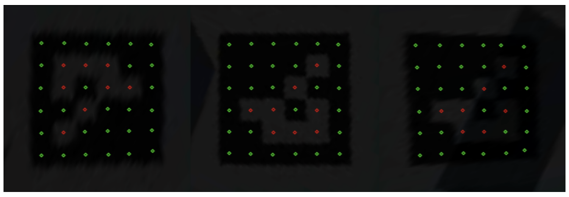
Results from the detection of the markers in the dataset with -90% brightness showed that there were issues with marker detection when there was too much shadow covering the marker. The figure below, Figure 4.18, shows the results from the detection of the markers in screenshot 41 of the dataset with reduced brightness to -90%. Each of the subfigures in the figure shows the different stages of the detection. From these results, it is shown that the DeepTag algorithm has problems with detecting the marker when the shadow covers a large part of the marker. In this case, the shadow is covering approximately half of the marker, and the algorithm is unable to detect any marker because the dark shadow becomes too similar to the background when the brightness is reduced. As shown in the last subfigure, Figure 4.18e, 0 ROIs and 0 markers are detected in the image.

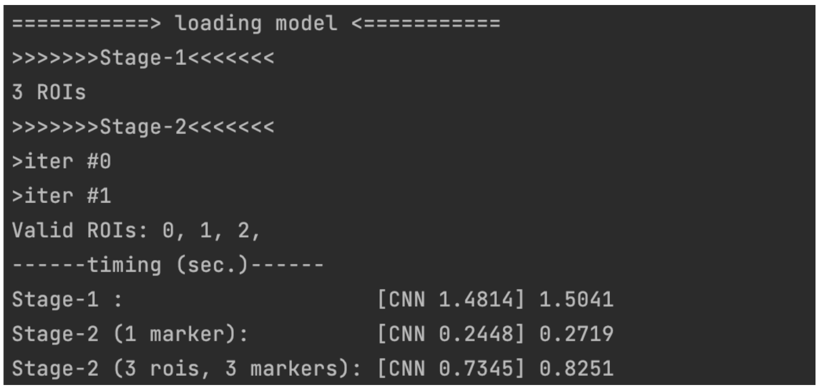
In 10 of 100 images, or 10%, the shadow covered a large enough part of the marker such that it prevented detection. This was only a problem when the brightness was reduced to -90%. Since two of the images did not have the whole marker in the image, and therefore were not detectable, the results of the detection is that the DeepTag algorithm was able to detect 86% of the dataset with -90% brightness. The results is displayed in Table 4.2.


(a) Detection of ROIs (Stage-1) in screenshot 1 with brightness -90%.


(b) Detection of boxes and corners (Stage-1) in screenshot 1 with brightness -90%.


(c) Image with pose in screenshot 1 with brightness -90%.

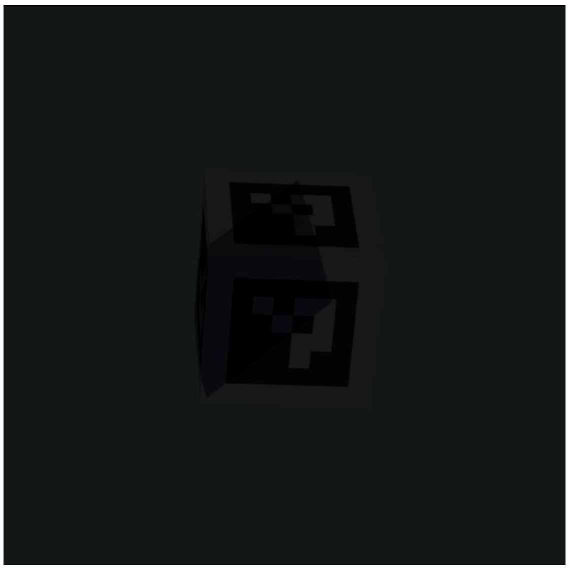

(d) Keypoints detected (Stage-2), number of markers on screenshot 1 with brightness -90%.



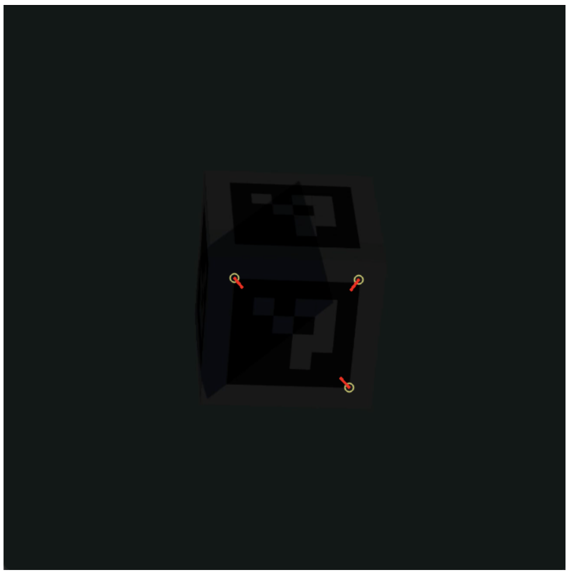
```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
3 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1, 2,
------timing (sec.)------
Stage-1 :                    [CNN 1.4814] 1.5041
Stage-2 (1 marker):          [CNN 0.2448] 0.2719
Stage-2 (3 rois, 3 markers): [CNN 0.7345] 0.8251
```

(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 1 with brightness -90%.

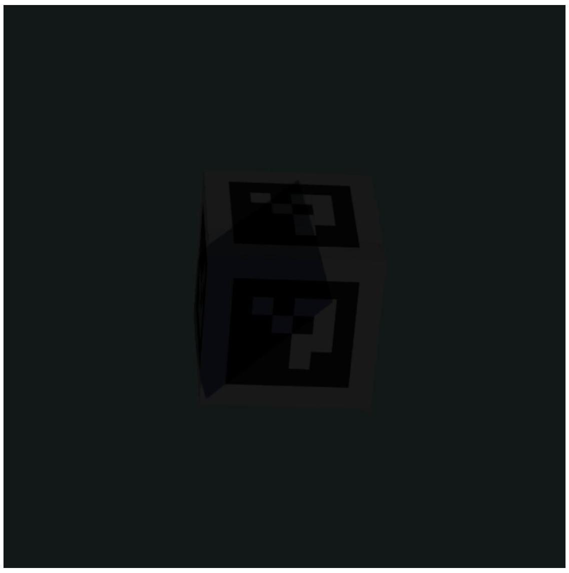
Figure 4.17: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with brightness -90% in the image.



(a) Detection of ROIs (Stage-1) in screenshot 41 with brightness -90%.

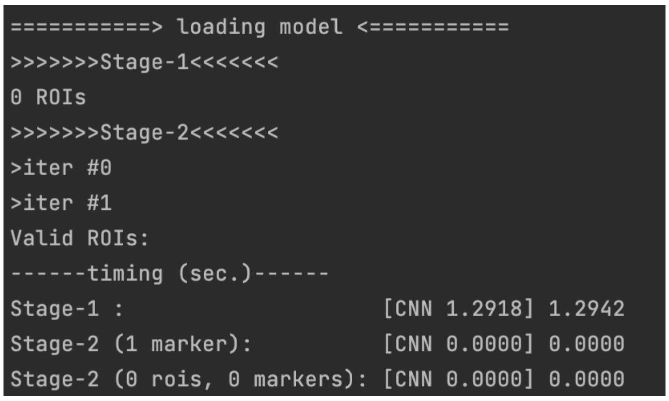


(b) Detection of boxes and corners (Stage-1) in screenshot 41 with brightness -90%.



(c) Image with pose in screenshot 41 with brightness -90%.

(d) Keypoints detected (Stage-2), number of markers on screenshot 41 with brightness -90%.



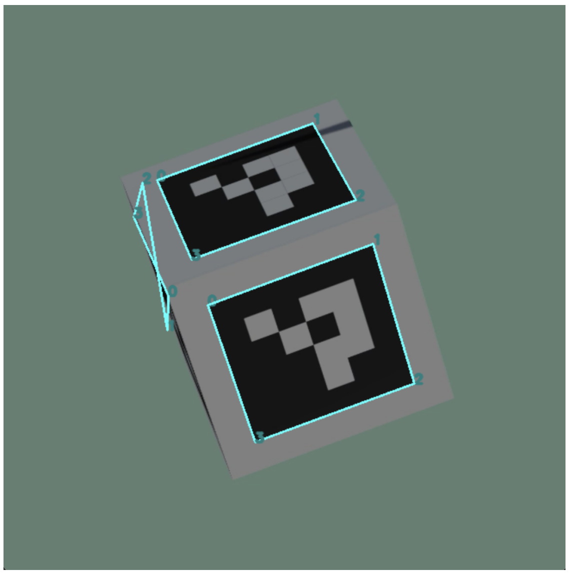
```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
0 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs:
------timing (sec.)------
Stage-1 :                    [CNN 1.2918] 1.2942
Stage-2 (1 marker):          [CNN 0.0000] 0.0000
Stage-2 (0 rois, 0 markers): [CNN 0.0000] 0.0000
```

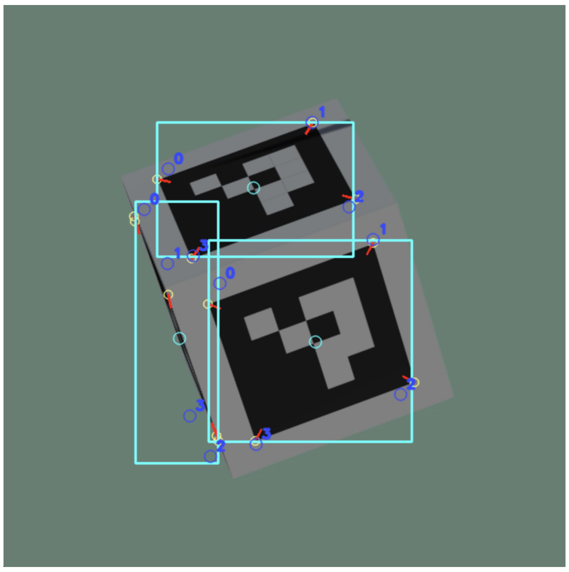
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 41 with brightness -90%.

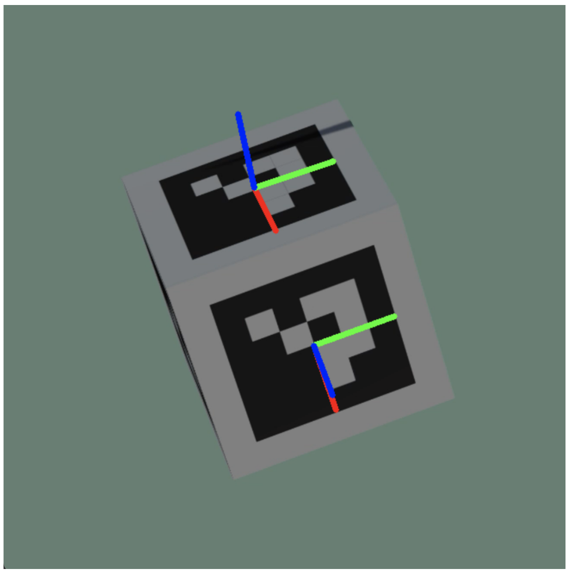
Figure 4.18: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with brightness -90% in the image.

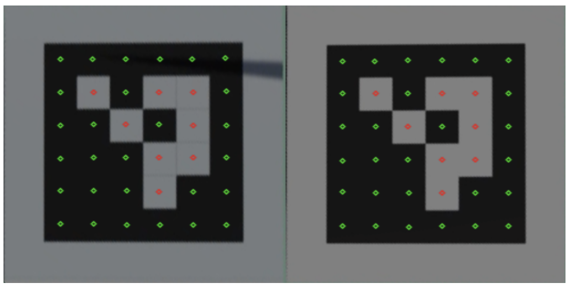
The next figures goes through a small excerpt of the dataset with decreased brightness to -50%. This is shown in Figure 4.19 and Figure 4.20. Results from running the DeepTag algorithm on the datasets with the brightness decreased to -50% show that the algorithm detects most of the ArUco markers with or without shadow. In these images, there is a larger difference in the colors due to more light than in the images with brightness reduced to -90%.

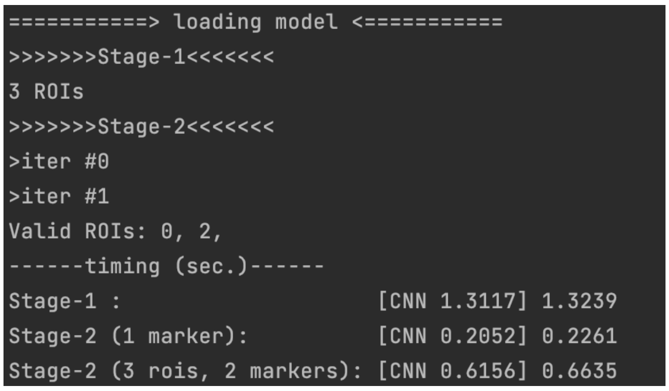
From the results of the marker detection from the dataset with reduced brightness to -50%, the detection is very good. The only markers the DeepTag algorithm had problems detecting were the ones with the corner of the markers outside the camera view. This constituted 6% of the dataset, which means it was able to detect 94% of the dataset with 100 images. Figure 4.19 shows the results of screenshot 4, where 2 markers are detected. Figure 4.20 shows the results of screenshot 59 and there are 3 markers detected in the image. The results are displayed in Table 4.2.


(a) Detection of ROIs (Stage-1) in screenshot 4 with brightness -50%.


(b) Detection of boxes and corners (Stage-1) in screenshot 4 with brightness -50%.


(c) Image with pose in screenshot 4 with brightness -50%.

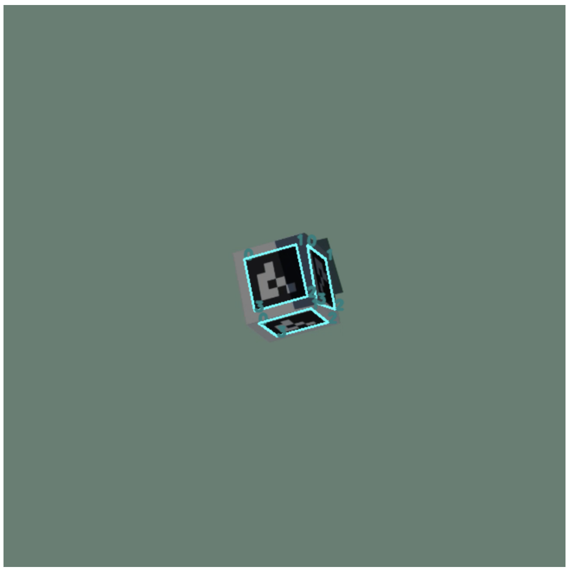

(d) Keypoints detected (Stage-2), number of markers on screenshot 4 with brightness -50%.

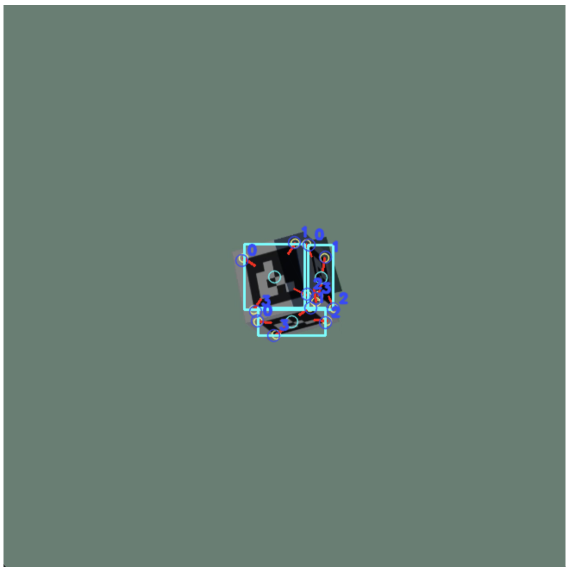


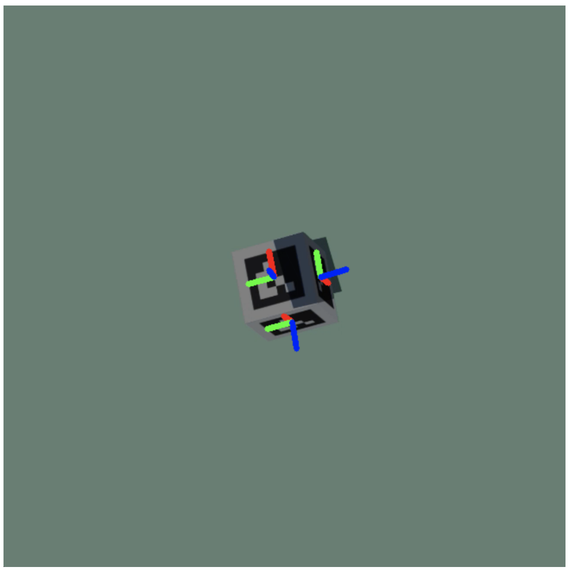
```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
3 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 2,
------timing (sec.)------
Stage-1 :                  [CNN 1.3117] 1.3239
Stage-2 (1 marker):        [CNN 0.2052] 0.2261
Stage-2 (3 rois, 2 markers): [CNN 0.6156] 0.6635
```

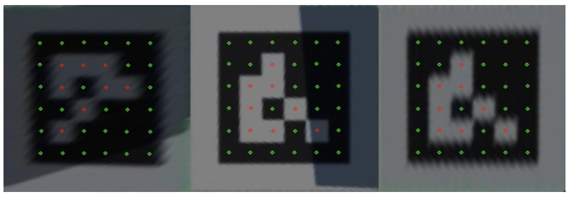
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 4 with brightness -50%.

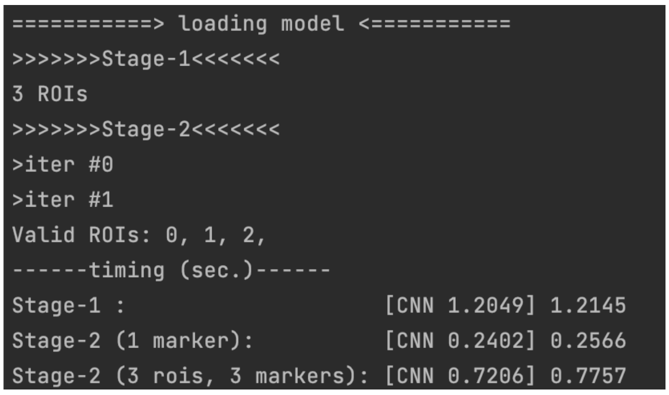
Figure 4.19: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with brightness -50% in the image.


(a) Detection of ROIs (Stage-1) in screenshot 59 with brightness -50%.


(b) Detection of boxes and corners (Stage-1) in screenshot 59 with brightness -50%.


(c) Image with pose in screenshot 59 with brightness -50%.


(d) Keypoints detected (Stage-2), number of markers on screenshot 59 with brightness -50%.



```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
3 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1, 2,
------timing (sec.)------
Stage-1 :                       [CNN 1.2049] 1.2145
Stage-2 (1 marker):             [CNN 0.2402] 0.2566
Stage-2 (3 rois, 3 markers): [CNN 0.7206] 0.7757
```

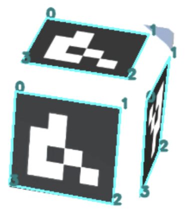
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 59 with brightness -50%.

Figure 4.20: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with brightness -50% in the image.

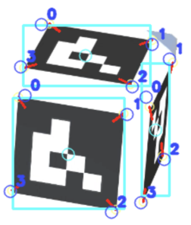
When the brightness was increased by 50% and by 90%, the images were almost the same. They became very light, and the background was not visible anymore. One difference that separated the datasets with 50% and 90%, was the shadow. The shadow was clearer and more visible with the brightness increased by 50%, then it was with the brightness increased by 90% where the shadow is so light that it was almost invisible on the marker. From the dataset with brightness increased by 50%, Figure 4.21 shows the results from screenshot 8. Three markers are detected, which matches the image. Figure 4.22 shows the results of the detection in screenshot 94, where one marker is detected. From the dataset with brightness increased by 90%, Figure 4.23 shows the results from screenshot 6. In this image, three markers are detected, which is correct. The last figure from the brightness increased by 90%, Figure 4.24, shows the results from screenshot 15 where all the three markers in the image were detected as well. In the dataset with brightness increased by 50%, 95% of the images with markers were detectable. From the dataset with brightness increased by 90%, the algorithm was able to detect 97% of the images. The results are displayed in Table 4.2.

Table 4.2: Results from the detection with brightness adjusted.

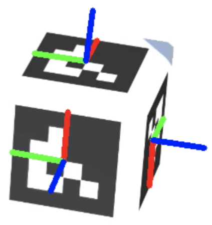
Brightness (%)	**Number of images**	**Detection rate (%)**
-90	86/100	86
Shadow covering ArUco marker at -90	10/100	0
Marker too close to camera lens at -90	6/100	0
-50	94/100	94
Marker too close to camera lens at -50	6/100	0
50	95/100	95
Marker too close to camera lens at 50	5/100	0
90	97/100	97
Marker too close to camera lens at 90	3/100	0



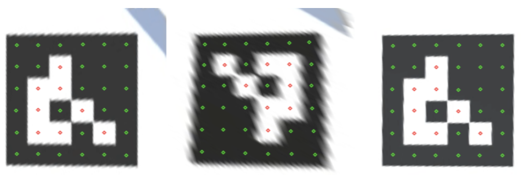
(a) Detection of ROIs (Stage-1) in screenshot 8 with brightness 50%.



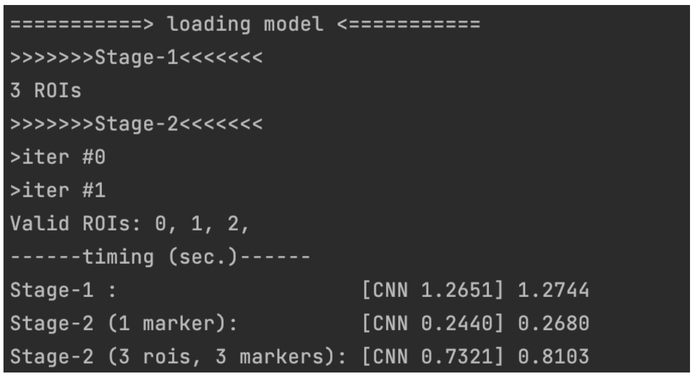
(b) Detection of boxes and corners (Stage-1) in screenshot 8 with brightness 50%.



(c) Image with pose in screenshot 8 with brightness 50%.



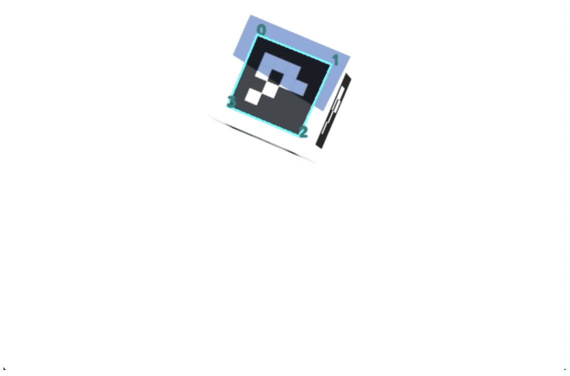
(d) Keypoints detected (Stage-2), number of markers on screenshot 8 with brightness 50%.



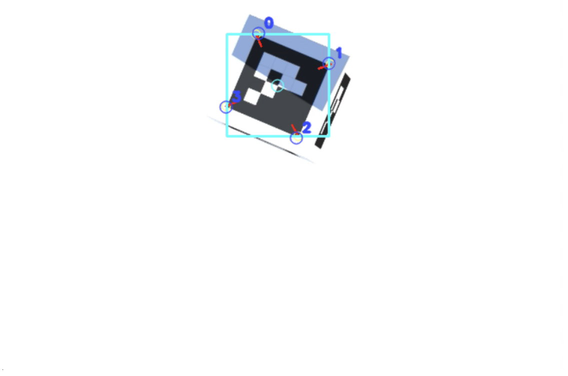
```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
3 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1, 2,
------timing (sec.)------
Stage-1 :                     [CNN 1.2651] 1.2744
Stage-2 (1 marker):           [CNN 0.2440] 0.2680
Stage-2 (3 rois, 3 markers): [CNN 0.7321] 0.8103
```

(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 8 with brightness 50%.

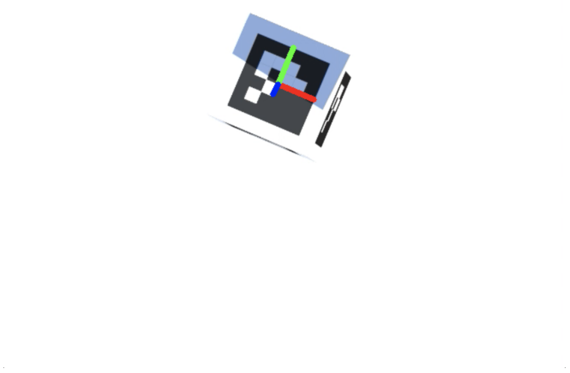
Figure 4.21: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with brightness 50%.



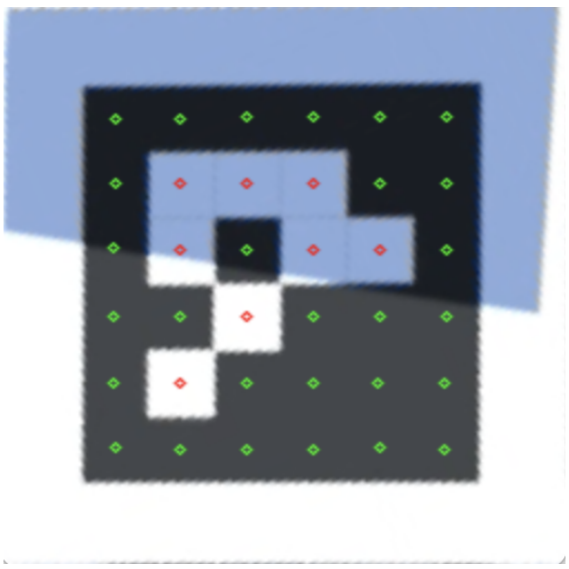
(a) Detection of ROIs (Stage-1) in screenshot 94 with brightness 50%.



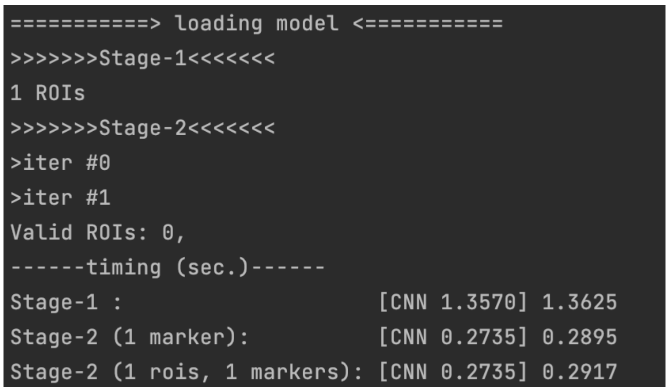
(b) Detection of boxes and corners (Stage-1) in screenshot 94 with brightness 50%.



(c) Image with pose in screenshot 94 with brightness 50%.



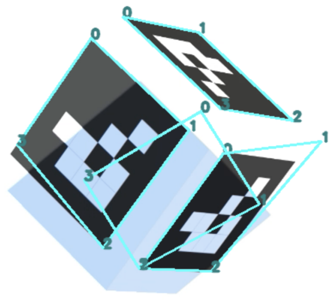
(d) Keypoints detected (Stage-2), number of markers on screenshot 94 with brightness 50%.

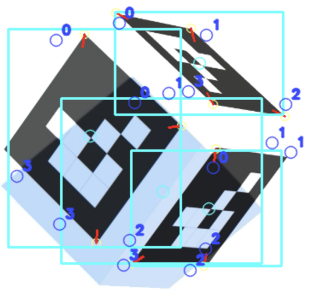


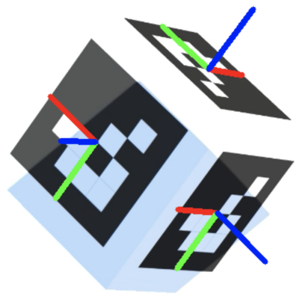
```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
1 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0,
------timing (sec.)------
Stage-1 :                   [CNN 1.3570] 1.3625
Stage-2 (1 marker):         [CNN 0.2735] 0.2895
Stage-2 (1 rois, 1 markers): [CNN 0.2735] 0.2917
```

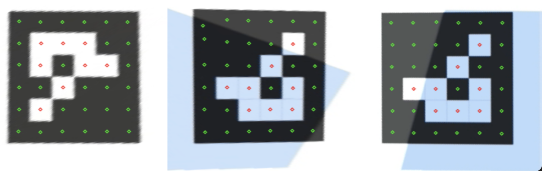
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 94 with brightness 50%.

Figure 4.22: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with brightness 50%.


(a) Detection of ROIs (Stage-1) in screenshot 6 with brightness 90%.


(b) Detection of boxes and corners (Stage-1) in screenshot 6 with brightness 90%.


(c) Image with pose in screenshot 6 with brightness 90%.


(d) Keypoints detected (Stage-2), number of markers on screenshot 6 with brightness 90%.

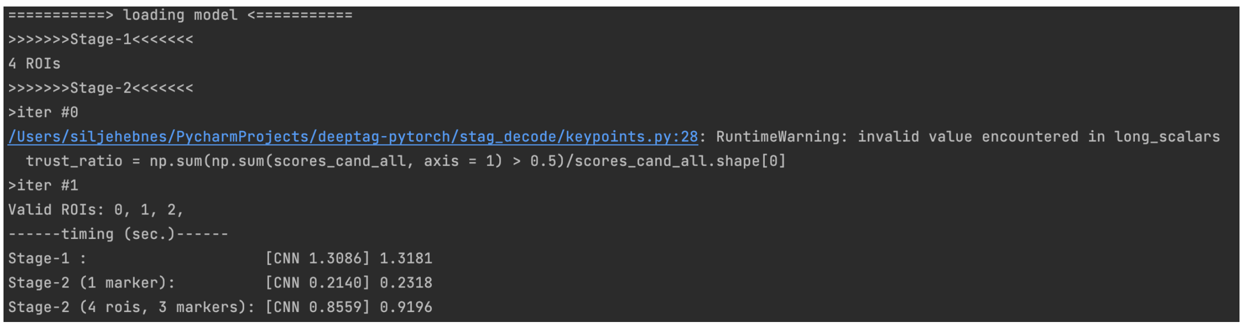
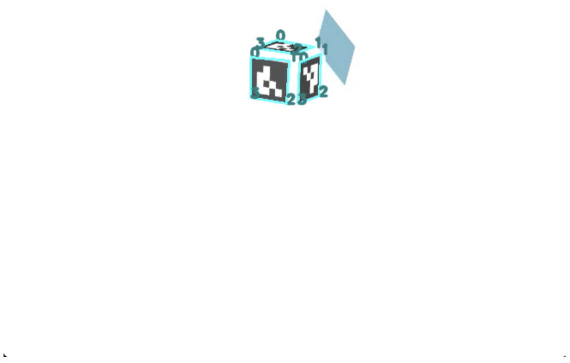
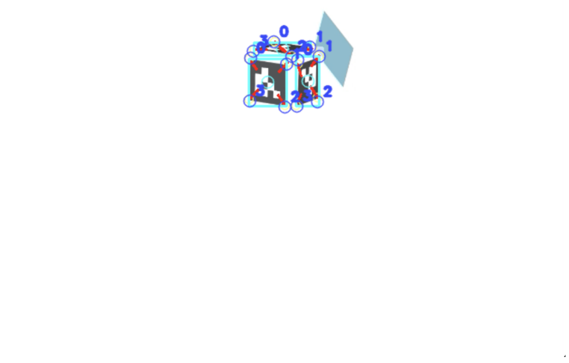

```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
4 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
/Users/siljehebnes/PycharmProjects/deeptag-pytorch/stag_decode/keypoints.py:28: RuntimeWarning: invalid value encountered in long_scalars
  trust_ratio = np.sum(np.sum(scores_cand_all, axis = 1) > 0.5)/scores_cand_all.shape[0]
>iter #1
Valid ROIs: 0, 1, 2,
------timing (sec.)------
Stage-1 :                  [CNN 1.3086] 1.3181
Stage-2 (1 marker):        [CNN 0.2140] 0.2318
Stage-2 (4 rois, 3 markers): [CNN 0.8559] 0.9196
```
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 6 with brightness 90%.

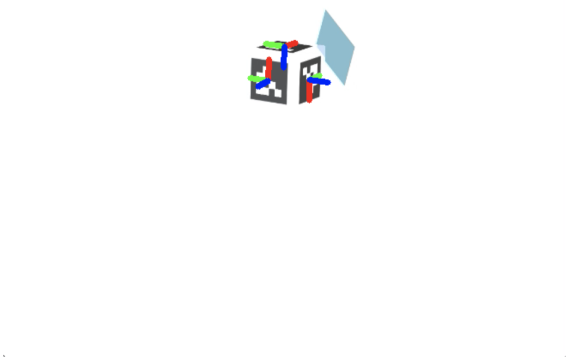
Figure 4.23: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with brightness 90%.



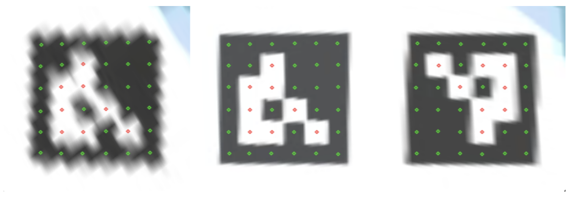
(a) Detection of ROIs (Stage-1) in screenshot 15 with brightness 90%.



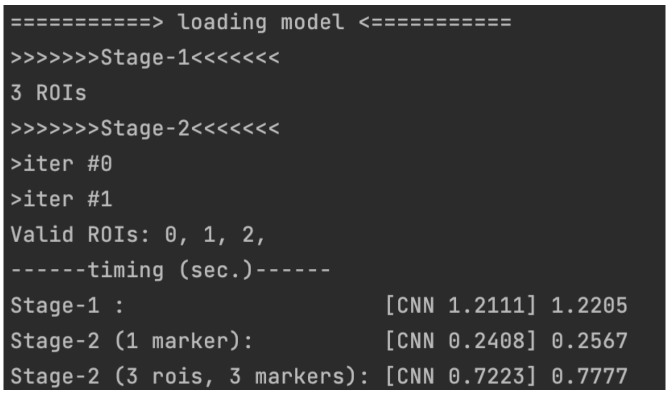
(b) Detection of boxes and corners (Stage-1) in screenshot 15 with brightness 90%.



(c) Image with pose in screenshot 15 with brightness 90%.



(d) Keypoints detected (Stage-2), number of markers on screenshot 15 with brightness 90%.



```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
3 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1, 2,
------timing (sec.)------
Stage-1 :                     [CNN 1.2111] 1.2205
Stage-2 (1 marker):           [CNN 0.2408] 0.2567
Stage-2 (3 rois, 3 markers): [CNN 0.7223] 0.7777
```

(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 15 with brightness 90%.

Figure 4.24: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with brightness 90%.

4.3.3 Detection of ArUco markers with motion blur

The detection of ArUco markers with the DeepTag algorithm was also tested with motion blur added to the scene. When testing the detection on the dataset with a 100 images with random parameters, the results came back less favourable than in the previous detections. The algorithm had difficulty with detection of the most blurry images, and was not able to detect the markers in these images. Another problem was the same as in all of the other detections, if the marker was too close and the whole marker was not visible in the camera view, the marker could not be detected. Based on experience with the previous datasets, this was expected. Out of 100 images, 25 of them were too blurry for the algorithm to detect the marker. This makes up 25% of the dataset. Only 2% of the images had the marker too close to the camera view for detection. This generates a result of 73% detection success on the dataset with motion blur added to the images. The figures below show an excerpt of the results from the detection. The results are displayed in Table 4.3.

Table 4.3: Results from the detection with motion blur.

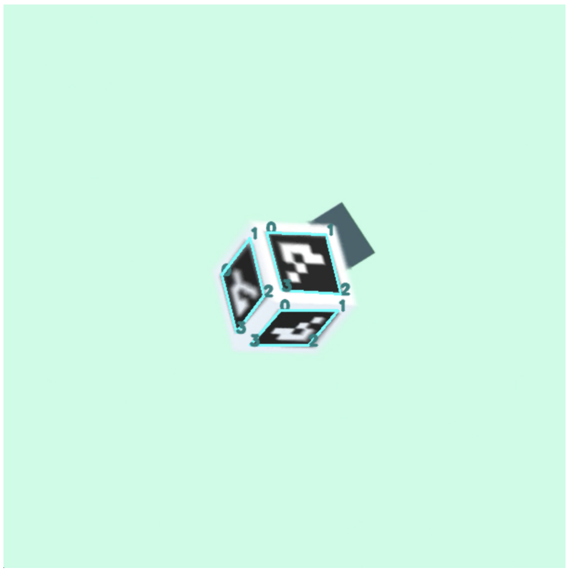
Motion blur	**Number of images**	**Detection rate (%)**
ArUco marker detected	73/100	73
Too much motion blur	25/100	0
Marker too close to camera lens	2/100	0

The first figure below, Figure 4.25, shows the results from the detection of markers on screenshot 3. As shown here, three markers are detected, which is equivalent to the amount of markers in the image. There is only a small amount of motion blur in this image, and the algorithm is able to detect all of the markers in this scenario.

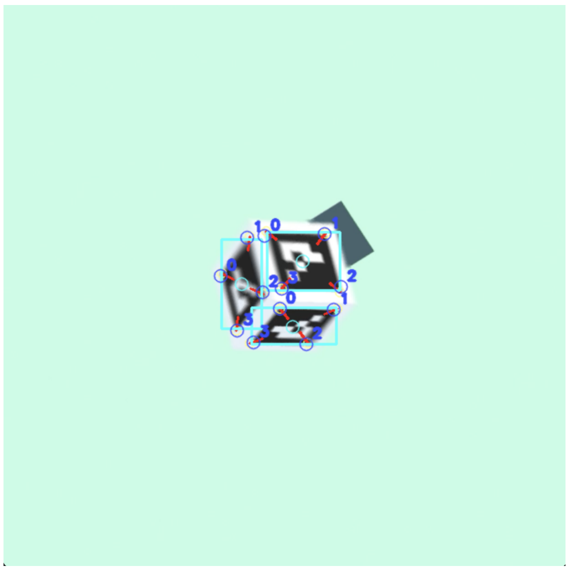
Figure 4.26 shows the results where there is an increased amount of motion blur in the image. This is from screenshot 5 in the dataset, and the algorithm is able to detect two markers, as expected. Even with this amount of motion blur, the model is able to detect the black and white squares in the marker correctly.

Figure 4.27 shows the results from screenshot 29. In the image, the marker is closer, but is more blurred. However, the algorithm is able to detect the three ArUco markers in the image perfectly, even with the shadow covering parts of them. All of the ROIs, boxes and corners, image with pose and keypoints are shown, together with the terminal showing the stages of the detection and the result.

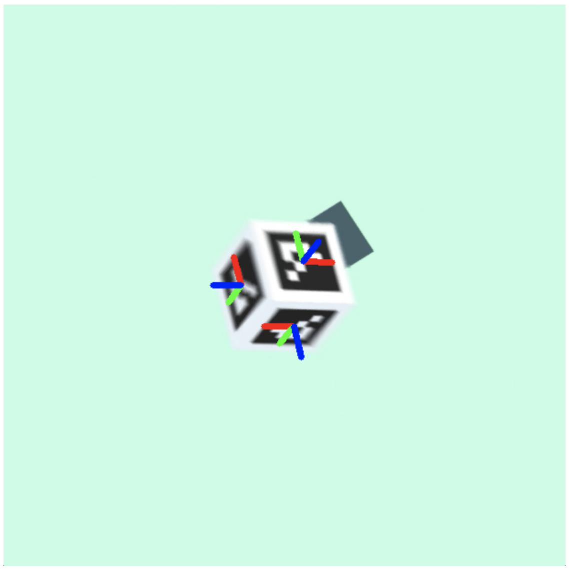
Figure 4.28, which is the last image shown from the dataset with motion blur, shows the results of screenshot 59. In this image, the cube with the markers on is significantly more blurred than the images above. This happened to several of the images in the dataset, and the DeepTag algorithm was not able to detect the markers. As seen here, the markers are too faded to find the pose and corners. Therefore, no keypoints are found and the algorithm is not able to show the markers.



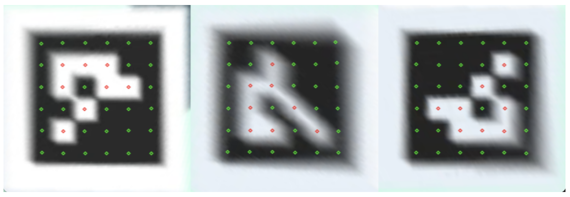
(a) Detection of ROIs (Stage-1) in screenshot 3 with motion blur.



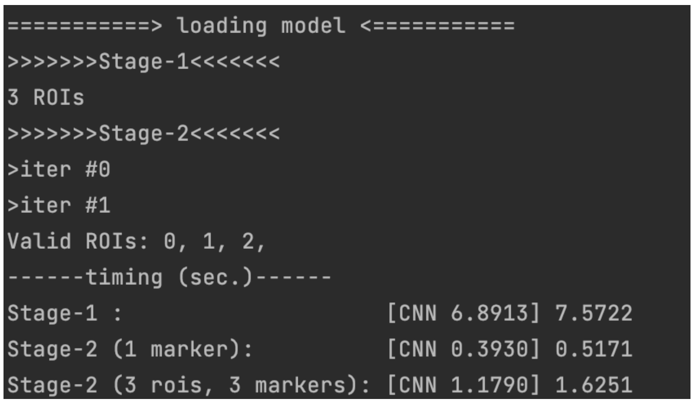
(b) Detection of boxes and corners (Stage-1) in screenshot 3 with motion blur.



(c) Image with pose in screenshot 3 with motion blur.



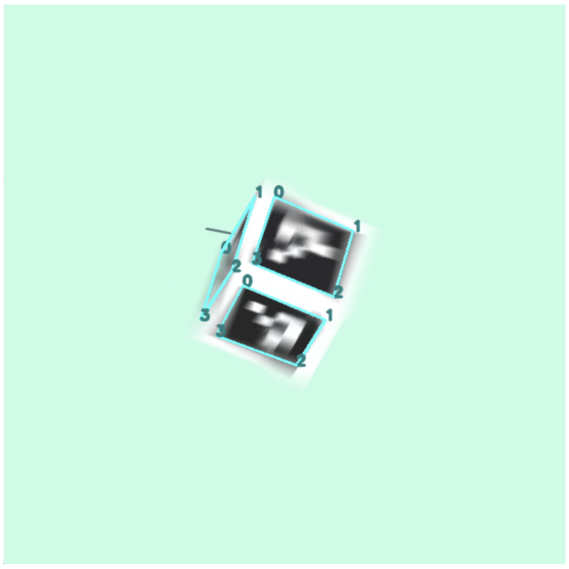
(d) Keypoints detected (Stage-2), number of markers on screenshot 3 with motion blur.



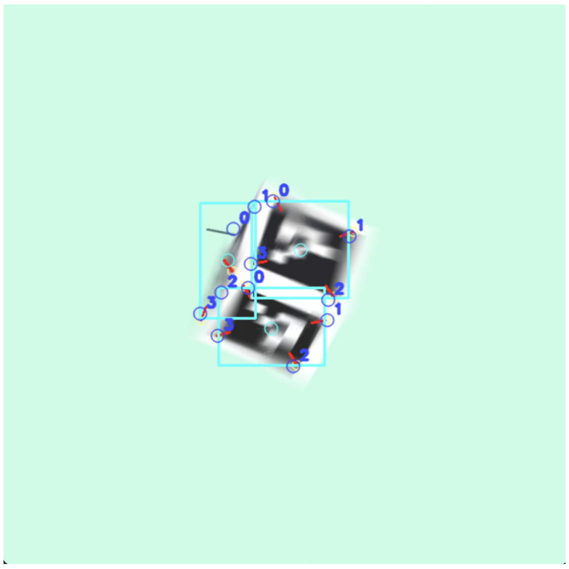
```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
3 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1, 2,
------timing (sec.)------
Stage-1 :                      [CNN 6.8913] 7.5722
Stage-2 (1 marker):            [CNN 0.3930] 0.5171
Stage-2 (3 rois, 3 markers): [CNN 1.1790] 1.6251
```

(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 3 with motion blur.

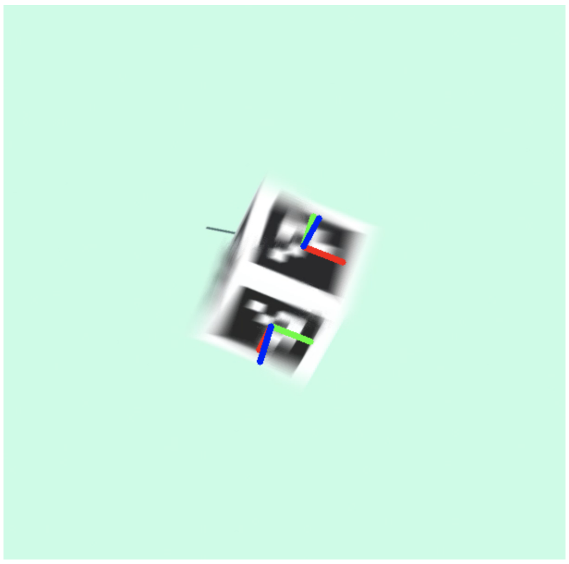
Figure 4.25: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with motion blur.



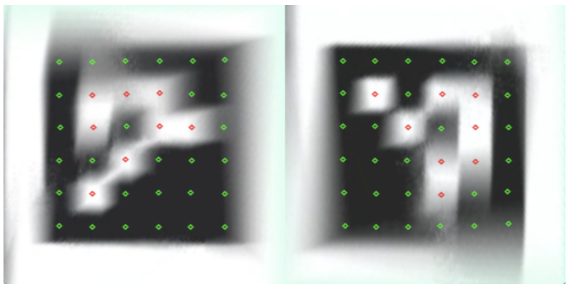
(a) Detection of ROIs (Stage-1) in screenshot 5 with motion blur.



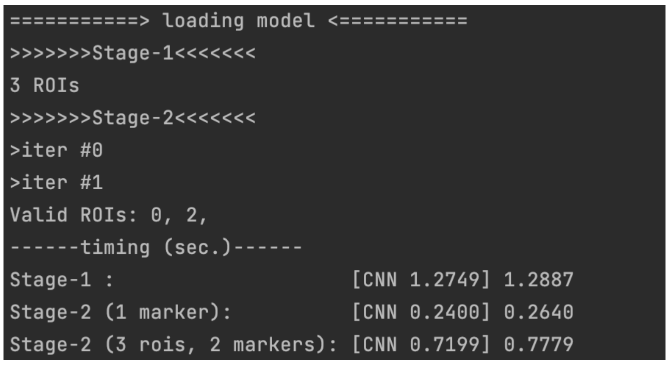
(b) Detection of boxes and corners (Stage-1) in screenshot 5 with motion blur.



(c) Image with pose in screenshot 5 with motion blur.



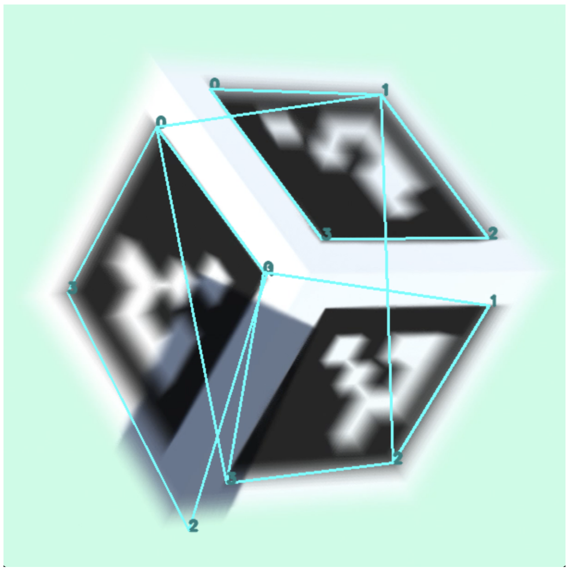
(d) Keypoints detected (Stage-2), number of markers on screenshot 5 with motion blur.



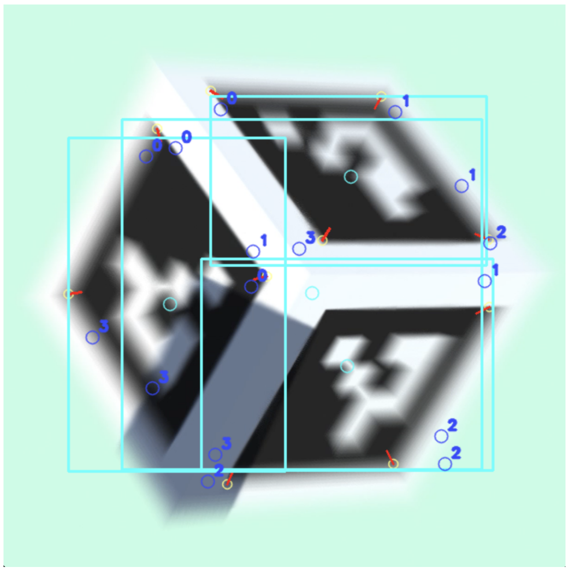
```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
3 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 2,
------timing (sec.)------
Stage-1 :                    [CNN 1.2749] 1.2887
Stage-2 (1 marker):          [CNN 0.2400] 0.2640
Stage-2 (3 rois, 2 markers): [CNN 0.7199] 0.7779
```

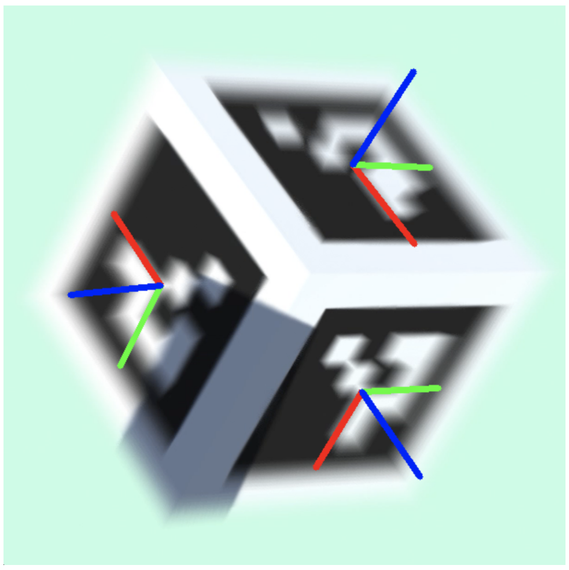
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 5 with motion blur.

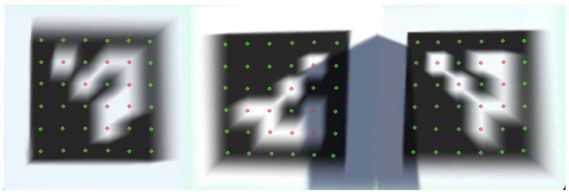
Figure 4.26: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with motion blur.



(a) Detection of ROIs (Stage-1) in screenshot 29 with motion blur.


(b) Detection of boxes and corners (Stage-1) in screenshot 29 with motion blur.


(c) Image with pose in screenshot 29 with motion blur.


(d) Keypoints detected (Stage-2), number of markers on screenshot 29 with motion blur.

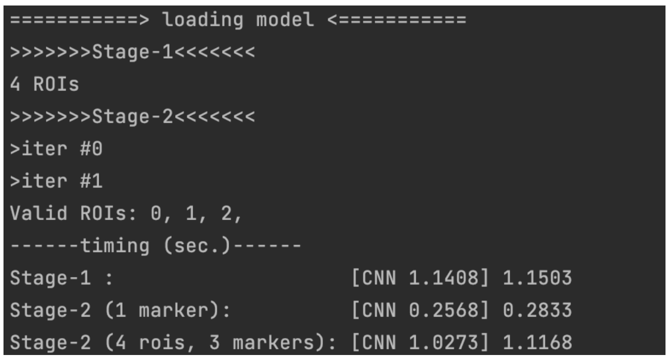
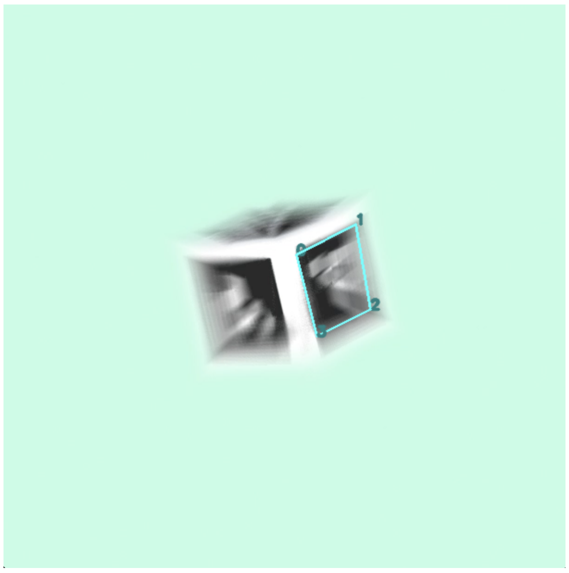
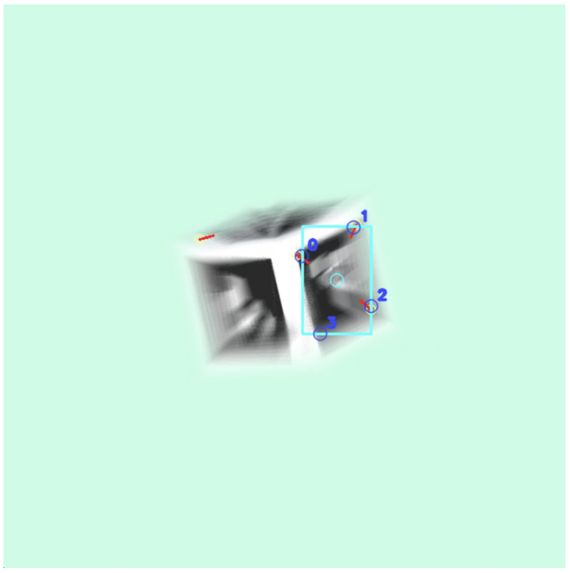

```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
4 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1, 2,
------timing (sec.)------
Stage-1 :                    [CNN 1.1408] 1.1503
Stage-2 (1 marker):          [CNN 0.2568] 0.2833
Stage-2 (4 rois, 3 markers): [CNN 1.0273] 1.1168
```
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 29 with motion blur.

Figure 4.27: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with motion blur.

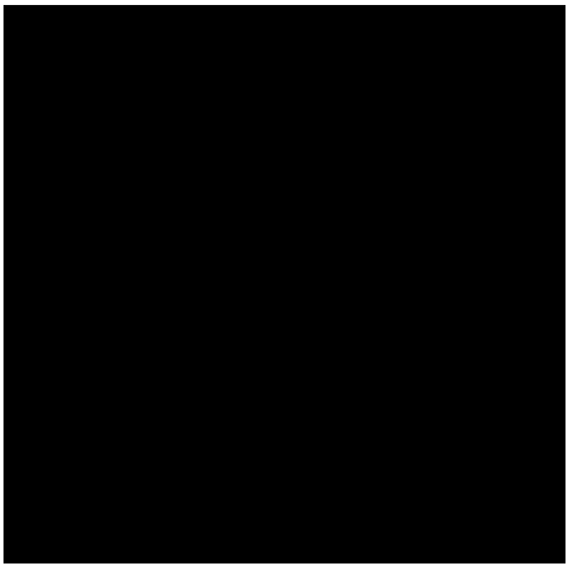


(a) Detection of ROIs (Stage-1) in screenshot 59 with motion blur.

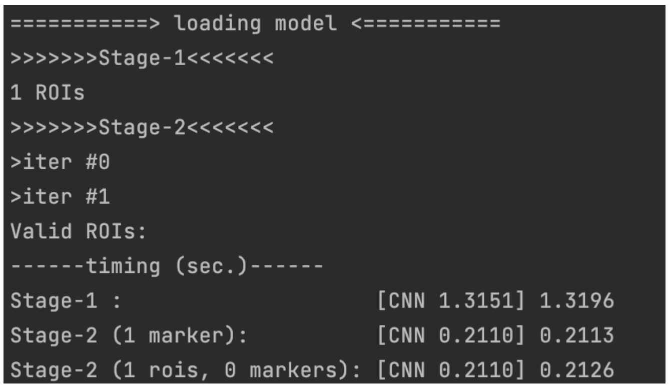


(b) Detection of boxes and corners (Stage-1) in screenshot 59 with motion blur.

(c) Image with pose in screenshot 59 with motion blur.



(d) Keypoints detected (Stage-2), number of markers on screenshot 59 with motion blur.



```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
1 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs:
------timing (sec.)------
Stage-1 :                    [CNN 1.3151] 1.3196
Stage-2 (1 marker):          [CNN 0.2110] 0.2113
Stage-2 (1 rois, 0 markers): [CNN 0.2110] 0.2126
```

(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 59 with motion blur.

Figure 4.28: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with motion blur.

4.3.4 Detection of ArUco markers with interruption on the camera

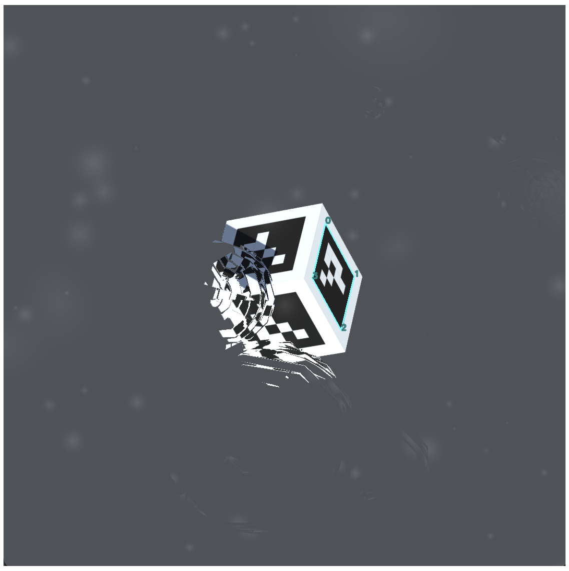
Detection with Rain Drops

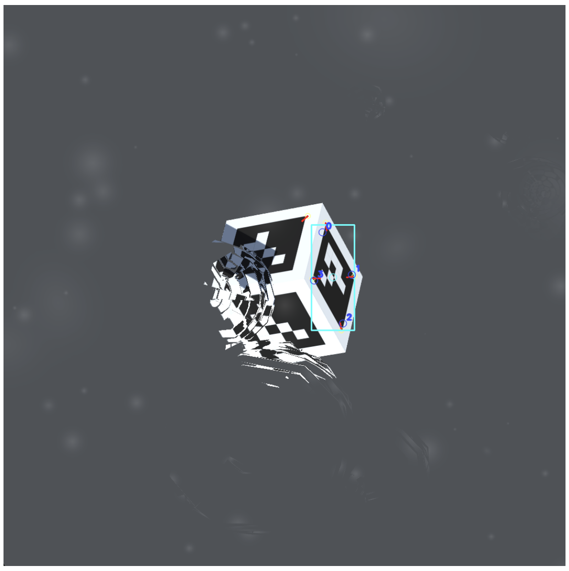
To test the DeepTag algorithm further in a real-world environment, rain drops were added. This was to check if the detection would be sufficient if rain drops would hit the camera. As seen from the results of the detection of the ArUco marker with rain drops in the image, the algorithm has difficulties detecting the marker when there is a rain drop on it. In the cases where there is a rain drop on only one side of the cube with markers, it is able to detect the rest of the markers. When the drop is right on the cube and all of the boxes and corners of the marker are not visible, the DeepTag algorithm is not able to detect the marker. This happened 4 out of 100 times, which constitutes 4% where no markers can be detected because of rain drops. As expected, the markers that were too close to the camera, where the whole marker was not visible, were not detected in this case either. This happened only twice, which constitutes 2% of the dataset. However, there were many other cases where a rain drop hit one marker on the side of the cube, but the rest of the markers were still detected. In this cases, there was always one or more marker detected. This happened in 22 of the 100 cases. In total, the DeepTag algorithm was able to detect markers in 94% of the dataset. Table 4.4 shows the results from the detection with rain drops in the scene.

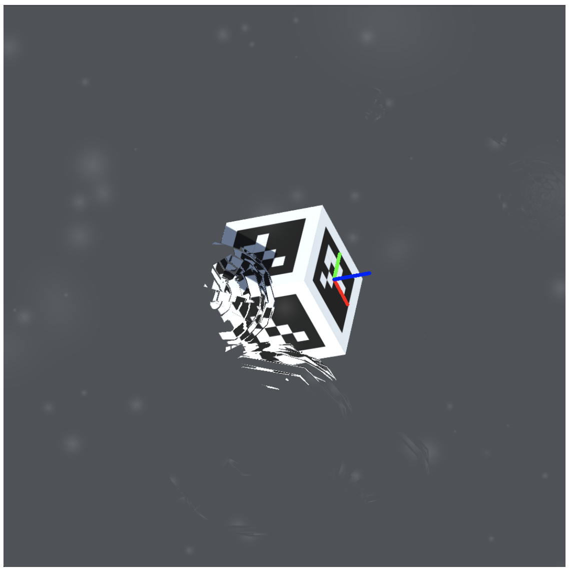
Table 4.4: Results from the detection with rain drops in the scene.

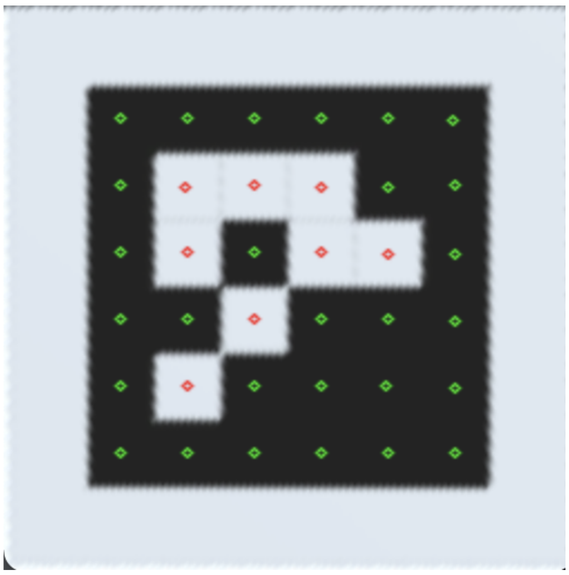
Rain drops	**Number of images**	**Detection rate (%)**
ArUco marker detected	94/100	94
Too blurry due to rain drops	4/100	0
Marker too close to camera lens	2/100	0

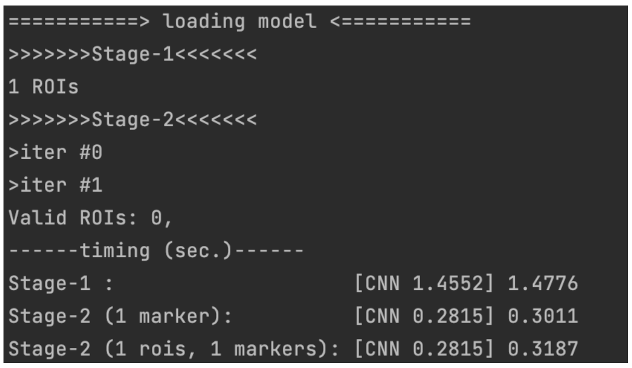
The figures below shows a random selection of the detected cases. Figure 4.29 shows the results from the detection of markers in screenshot 3. As seen in the figures, the detection algorithm is able to detect one of the three markers on the cube. Since two of the markers have a rain drop on them, it is not possible to see all the boxes and corners needed for the detection to work. Figure 4.30 shows the results from the detection of markers in screenshot 41. These figures shows that one marker on the side of the cube has a rain drop on it and can therefore not be detected. The rest of the markers on the cube are detected and displayed. The results of the detection of markers in screenshot 51 is shown in Figure 4.31. As seen here, the marker is not covered by rain drops, and the algorithm is able to detect the visible marker. Figure 4.32 displays the results from screenshot 91 where the marker is covered by rain drops. In this image, there is not one whole marker to be seen. The algorithm is not able to detect the boxes and corners of the markers properly, so no marker is detected.


(a) Detection of ROIs (Stage-1) in screenshot 3 with rain drops in the image.


(b) Detection of boxes and corners (Stage-1) in screenshot 3 with rain drops in the image.


(c) Image with pose in screenshot 3 with rain drops in the image.

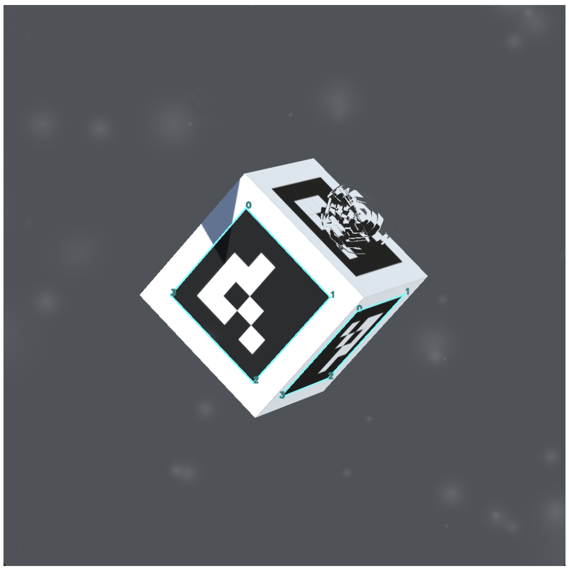

(d) Keypoints detected (Stage-2), number of markers on screenshot 3 with rain drops in the image.

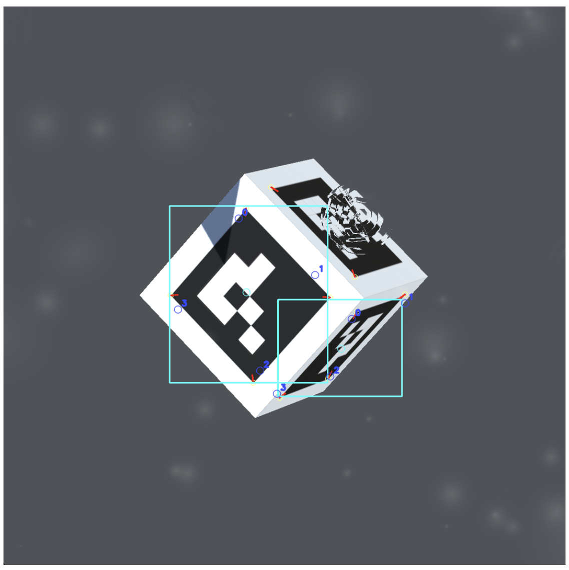


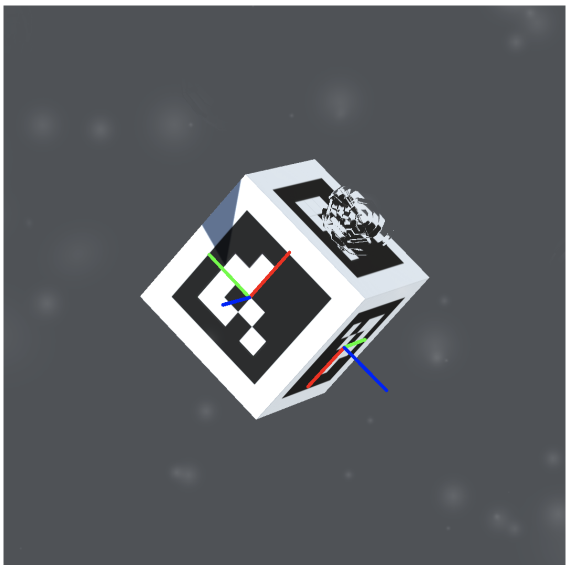
```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
1 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0,
------timing (sec.)------
Stage-1 :                      [CNN 1.4552] 1.4776
Stage-2 (1 marker):            [CNN 0.2815] 0.3011
Stage-2 (1 rois, 1 markers): [CNN 0.2815] 0.3187
```

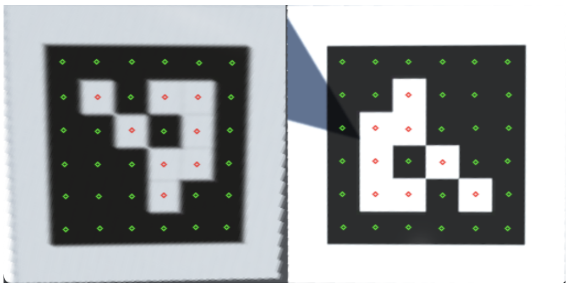
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 3 with rain drops in the image.

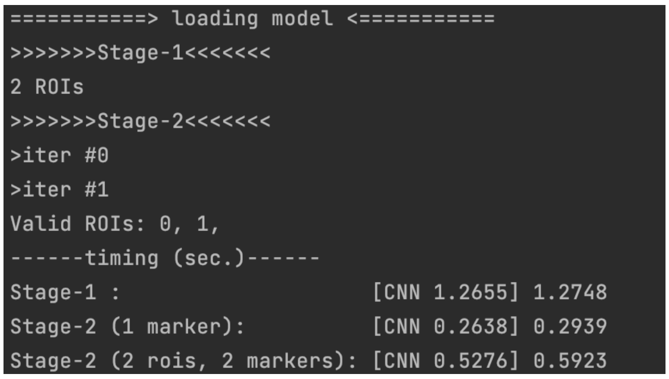
Figure 4.29: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with rain drops in the image.


(a) Detection of ROIs (Stage-1) in screenshot 41 with rain drops in the image.


(b) Detection of boxes and corners (Stage-1) in screenshot 41 with rain drops in the image.

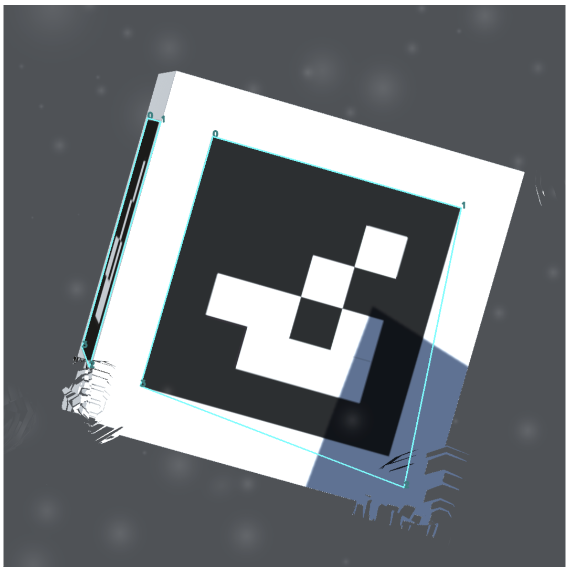

(c) Image with pose in screenshot 41 with rain drops in the image.

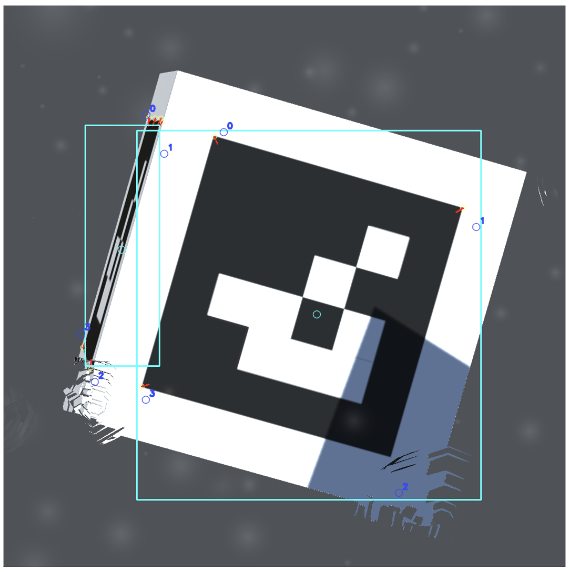

(d) Keypoints detected (Stage-2), number of markers on screenshot 41 with rain drops in the image.

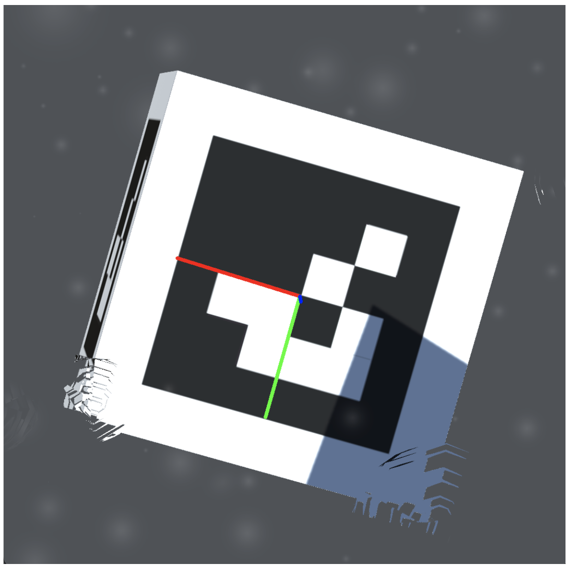

```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
2 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1,
------timing (sec.)------
Stage-1 :                    [CNN 1.2655] 1.2748
Stage-2 (1 marker):          [CNN 0.2638] 0.2939
Stage-2 (2 rois, 2 markers): [CNN 0.5276] 0.5923
```

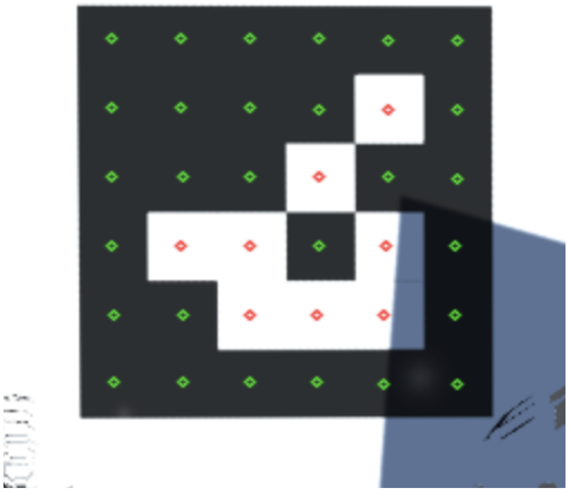
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 41 with rain drops in the image.

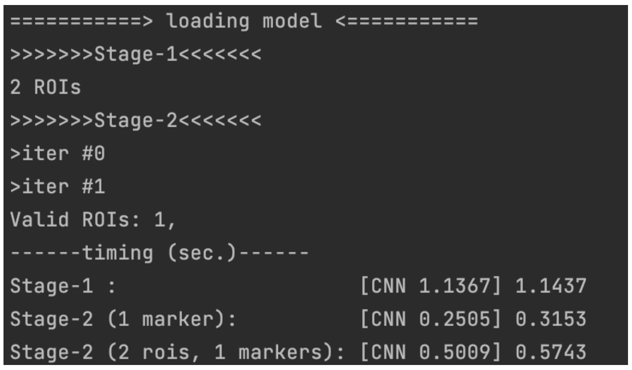
Figure 4.30: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with rain drops in the image.


(a) Detection of ROIs (Stage-1) in screenshot 51 with rain drops in the image.


(b) Detection of boxes and corners (Stage-1) in screenshot 51 with rain drops in the image.


(c) Image with pose in screenshot 51 with rain drops in the image.

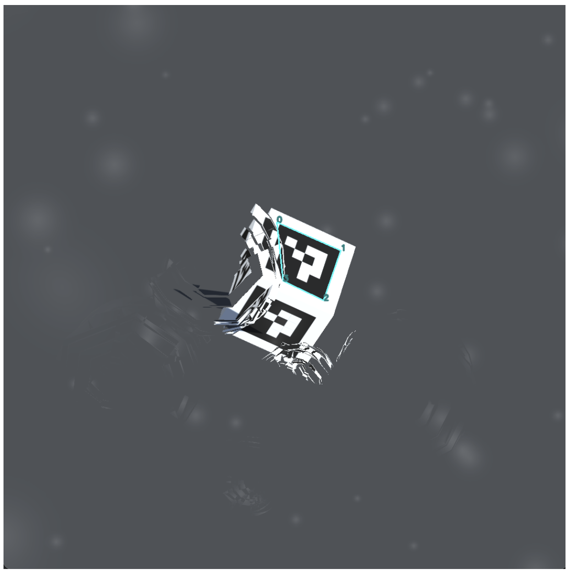

(d) Keypoints detected (Stage-2), number of markers on screenshot 51 with rain drops in the image.



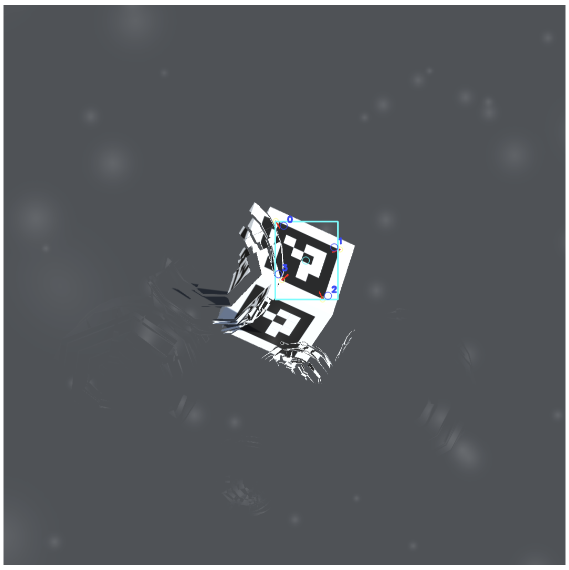
```
============> loading model <============
>>>>>>>Stage-1<<<<<<<
2 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 1,
------timing (sec.)------
Stage-1 :                     [CNN 1.1367] 1.1437
Stage-2 (1 marker):           [CNN 0.2505] 0.3153
Stage-2 (2 rois, 1 markers): [CNN 0.5009] 0.5743
```

(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 51 with rain drops in the image.

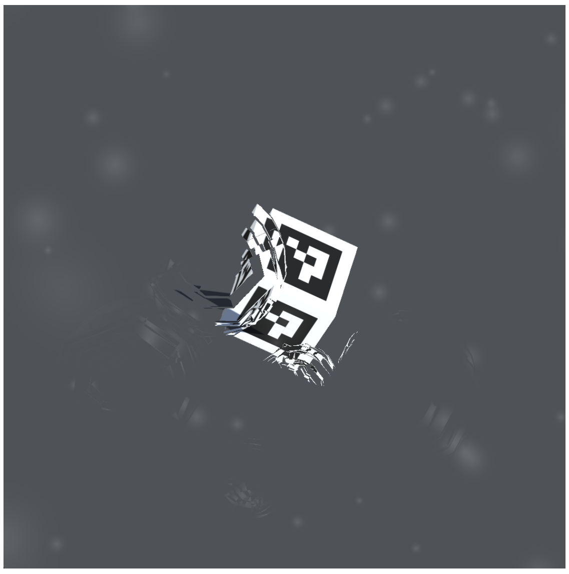
Figure 4.31: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with rain drops in the image.



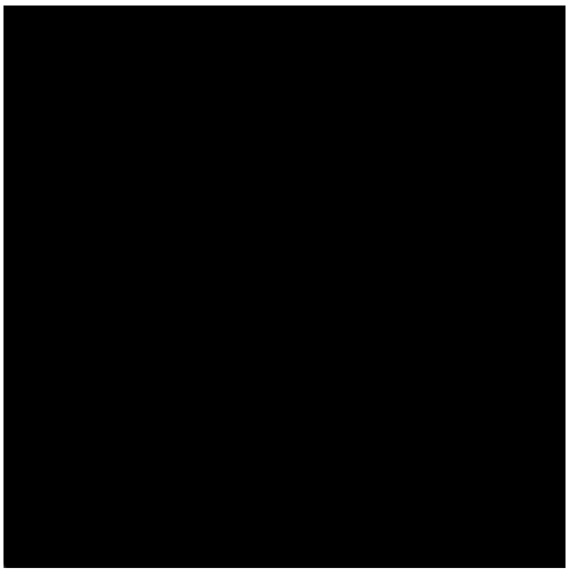
(a) Detection of ROIs (Stage-1) in screenshot 91 with rain drops in the image.



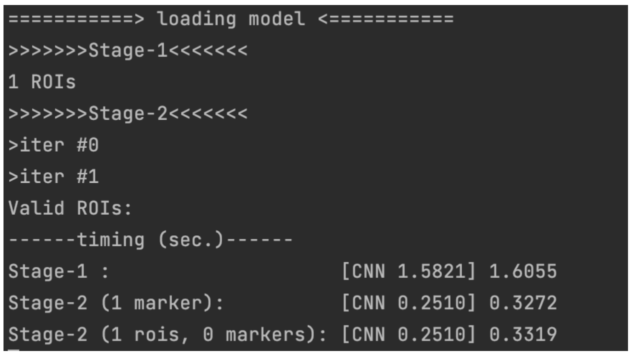
(b) Detection of boxes and corners (Stage-1) in screenshot 91 with rain drops in the image.



(c) Image with pose in screenshot 91 with rain drops in the image.



(d) Keypoints detected (Stage-2), number of markers on screenshot 91 with rain drops in the image.



```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
1 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs:
------timing (sec.)------
Stage-1 :                     [CNN 1.5821] 1.6055
Stage-2 (1 marker):           [CNN 0.2510] 0.3272
Stage-2 (1 rois, 0 markers): [CNN 0.2510] 0.3319
```

(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 91 with rain drops in the image.

Figure 4.32: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with rain drops in the image.

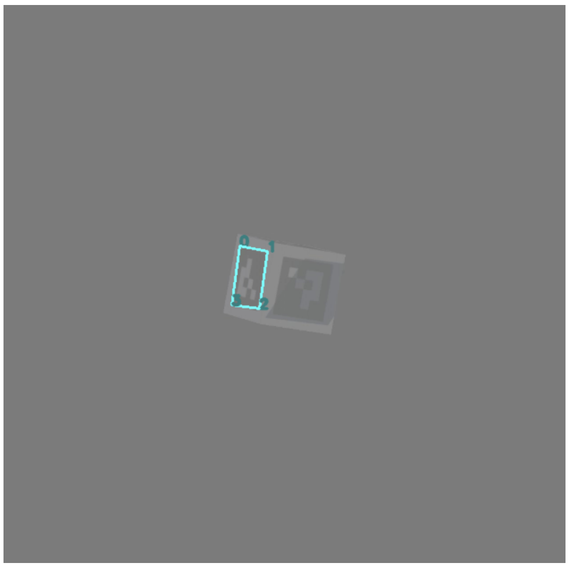
Detection with different contrasts

The contrast was changed in the scene to test if the DeepTag detection algorithm was able detect well with decreased contrast and increased contrast. The contrast was decreased to -90% in one of the datasets and increased to 90% in the other dataset, both containing 100 images. The results from the detection are shown in the figures below. As seen in Figure 4.33, the DeepTag algorithm has trouble detecting the marker with decreased contrast to -90% when the shadow covers larger parts of it. The marker then becomes too similar to the background and the algorithm is unable to detect the marker with shadow on it. However, it is still able to detect the rest of the markers in the image that are not covered by shadow. Figure 4.34 shows that the detection algorithm easily detects the marker with the contrast decreased to -90% when there is no shadow covering it. Out of 100 images with decreased contrast to -90%, there was 8 images where the markers could not be detected due to the shadow covering the markers and 5 images where the marker was too close to the camera. Calculating this, the DeepTag algorithm is able to detect markers in 87% of the images.

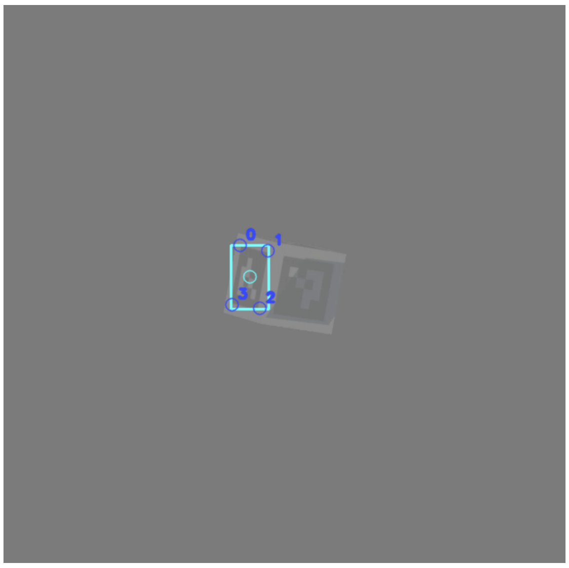
The detection was also tested with the contrast increased to 90%. This gave a much better result. When the increase in contrast is this large, the detection was good regardless if there was a shadow or not in the image. Figure 4.35 shows the result from the detection with shadow covering the marker. The algorithm is still able to detect all of the three markers with both the right position and rotation. Figure 4.36 shows excellent results from the detection of increased contrast, with and without shadow in the image. Two markers are detected, which is the same amount of markers as in the image. In this case, where the contrast was increased, the DeepTag algorithm was able to detect all the markers in the image except when the marker was too close to the camera and did not show all the corners in the camera view. The markers too close to the camera constituted 5% of the dataset. Apart from this, the detection with increased contrast was excellent and was able to detect markers in 95% of the images. If the markers outside of the camera view were excluded, the algorithm would have detected 100% of the dataset. The results are displayed in Table 4.5.

Table 4.5: Results from the detection with different contrasts.

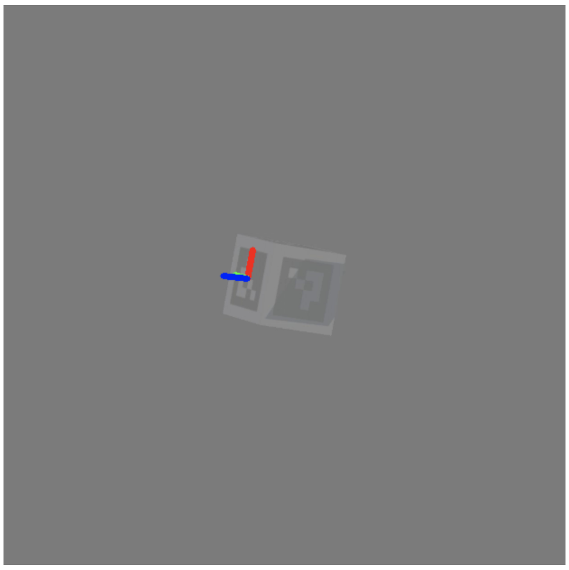
Contrast (%)	**Number of images**	**Detection rate (%)**
-90	87/100	87
Shadow covering ArUco marker at -90	8/100	0
Marker too close to camera lens at -90	5/100	0
90	95/100	95
Marker too close to camera lens at 90	5/100	0



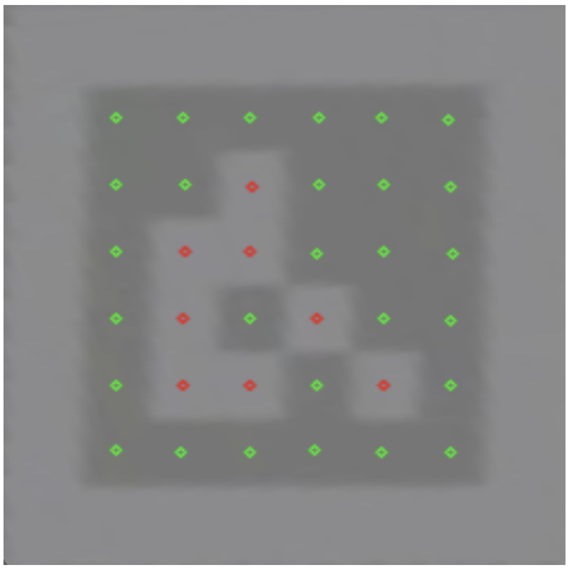
(a) Detection of ROIs (Stage-1) in screenshot 25 with contrast decreased to -90%.



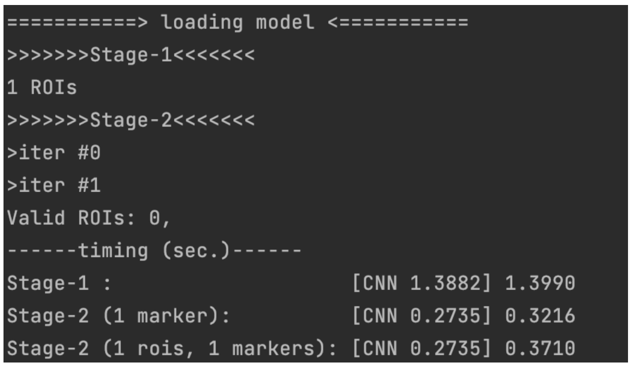
(b) Detection of boxes and corners (Stage-1) in screenshot 25 with contrast decreased to -90%.



(c) Image with pose in screenshot 25 with contrast decreased to -90%.



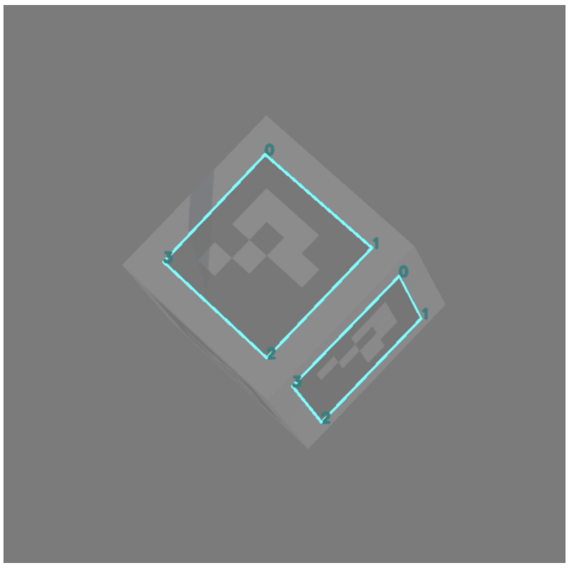
(d) Keypoints detected (Stage-2), number of markers on screenshot 25 with contrast decreased to -90%.



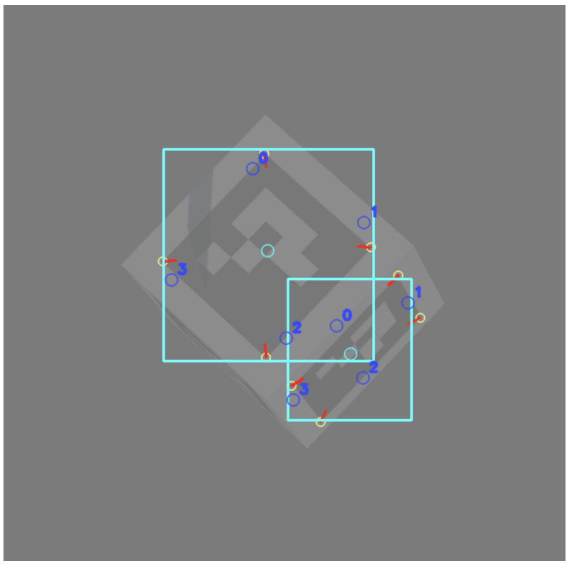
```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
1 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0,
------timing (sec.)------
Stage-1 :                     [CNN 1.3882] 1.3990
Stage-2 (1 marker):           [CNN 0.2735] 0.3216
Stage-2 (1 rois, 1 markers): [CNN 0.2735] 0.3710
```

(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 25 with contrast decreased to -90%.

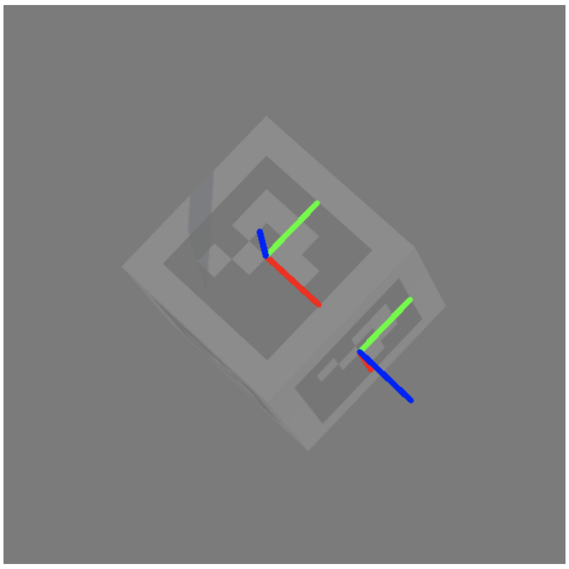
Figure 4.33: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with contrast decreased to -90%.



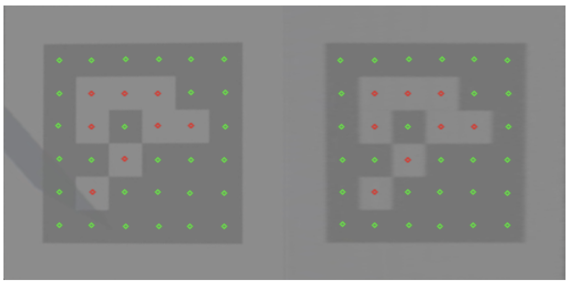
(a) Detection of ROIs (Stage-1) in screenshot 38 with contrast decreased to -90%.



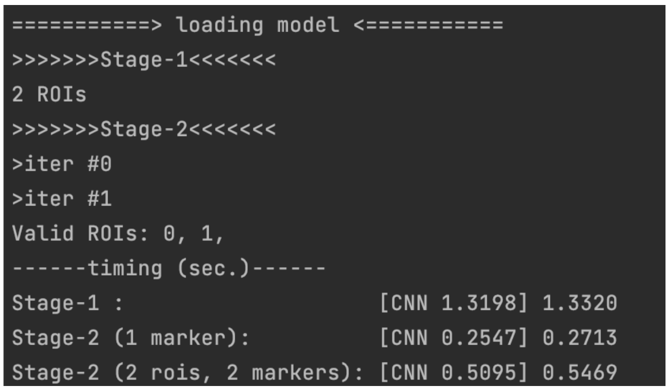
(b) Detection of boxes and corners (Stage-1) in screenshot 38 with contrast decreased to -90%.



(c) Image with pose in screenshot 38 with contrast decreased to -90%.



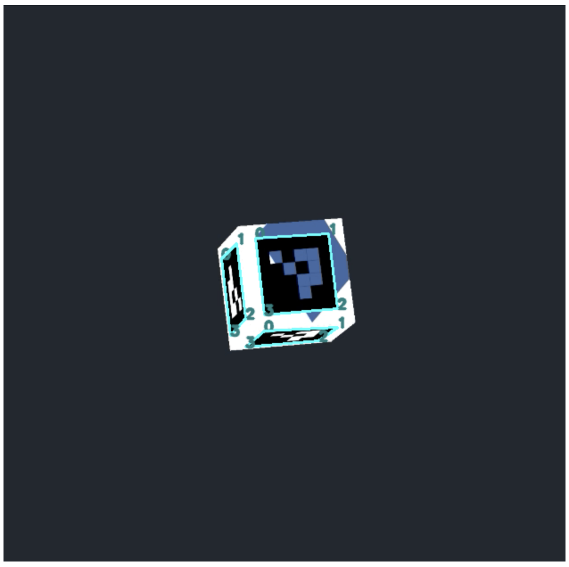
(d) Keypoints detected (Stage-2), number of markers on screenshot 38 with contrast decreased to -90%.



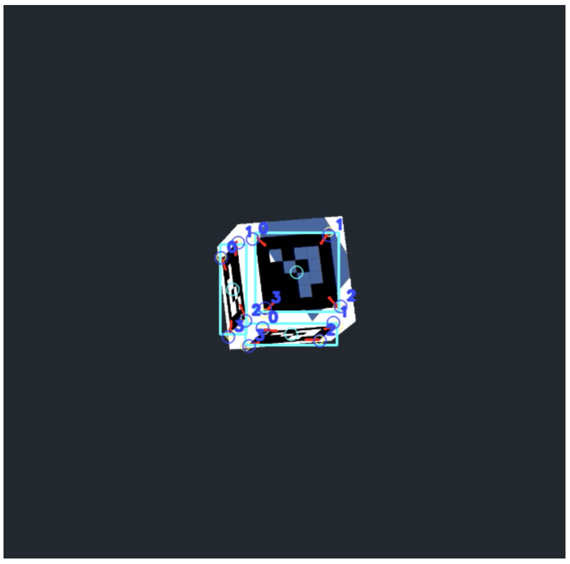
```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
2 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1,
------timing (sec.)------
Stage-1 :                  [CNN 1.3198] 1.3320
Stage-2 (1 marker):        [CNN 0.2547] 0.2713
Stage-2 (2 rois, 2 markers): [CNN 0.5095] 0.5469
```

(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 38 with contrast decreased to -90%.

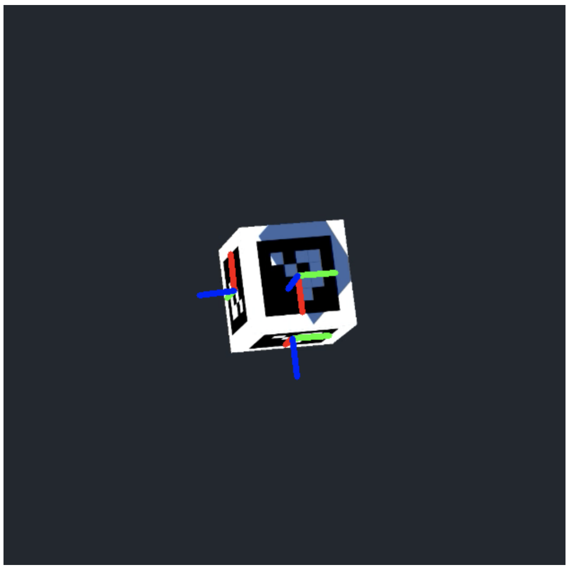
Figure 4.34: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with contrast decreased to -90%.



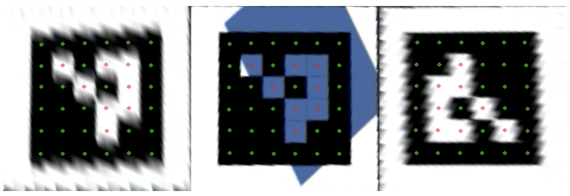
(a) Detection of ROIs (Stage-1) in screenshot 28 with contrast increased to 90%.



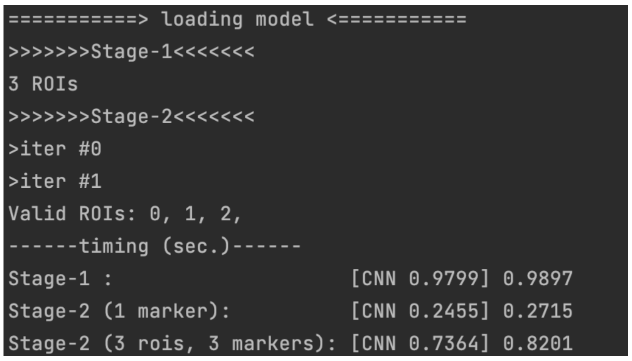
(b) Detection of boxes and corners (Stage-1) in screenshot 28 with contrast increased to 90%.



(c) Image with pose in screenshot 28 with contrast increased to 90%.



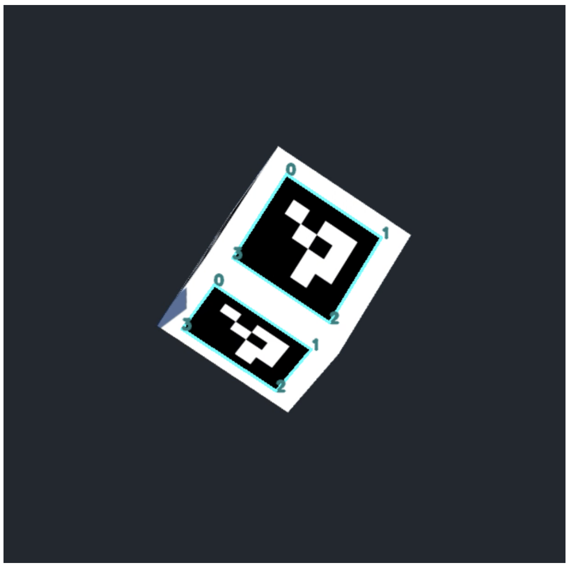
(d) Keypoints detected (Stage-2), number of markers on screenshot 28 with contrast increased to 90%.



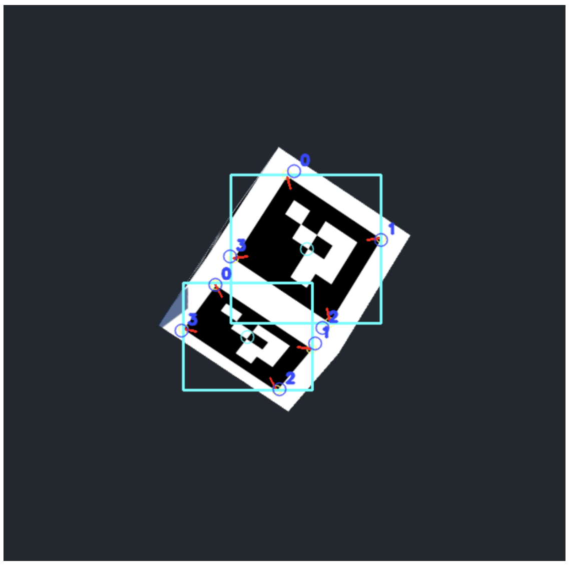
```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
3 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1, 2,
------timing (sec.)------
Stage-1 :                     [CNN 0.9799] 0.9897
Stage-2 (1 marker):           [CNN 0.2455] 0.2715
Stage-2 (3 rois, 3 markers): [CNN 0.7364] 0.8201
```

(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 28 with contrast increased to 90%.

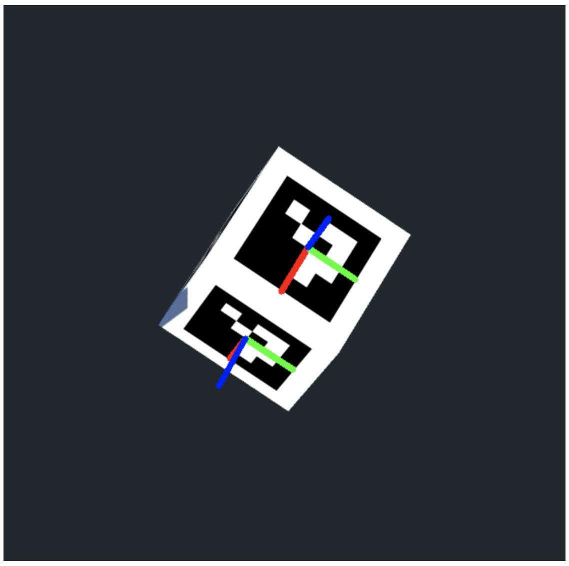
Figure 4.35: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with contrast increased to 90%.



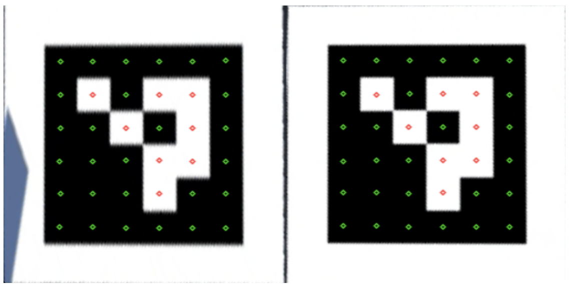
(a) Detection of ROIs (Stage-1) in screenshot 90 with contrast increased to 90%.



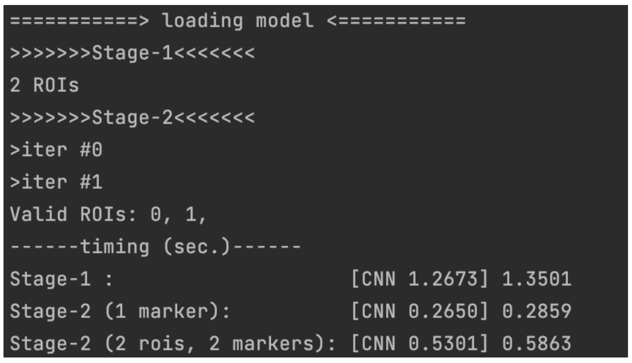
(b) Detection of boxes and corners (Stage-1) in screenshot 90 with contrast increased to 90%.



(c) Image with pose in screenshot 90 with contrast increased to 90%.



(d) Keypoints detected (Stage-2), number of markers on screenshot 90 with contrast increased to 90%.



```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
2 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1,
------timing (sec.)------
Stage-1 :                   [CNN 1.2673] 1.3501
Stage-2 (1 marker):         [CNN 0.2650] 0.2859
Stage-2 (2 rois, 2 markers): [CNN 0.5301] 0.5863
```

(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 90 with contrast increased to 90%.

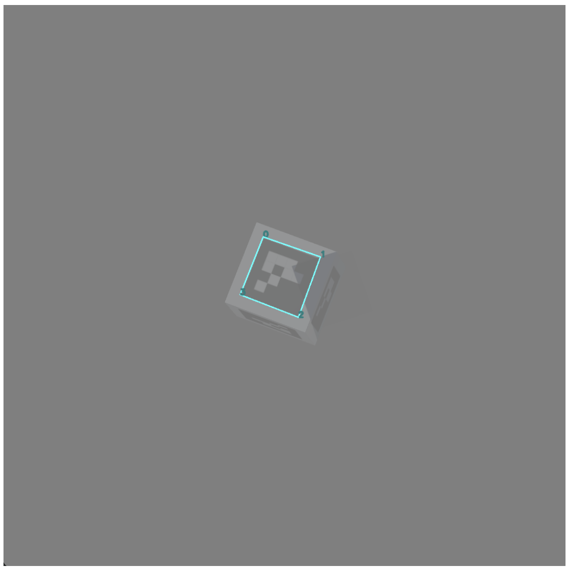
Figure 4.36: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with contrast increased to 90%.

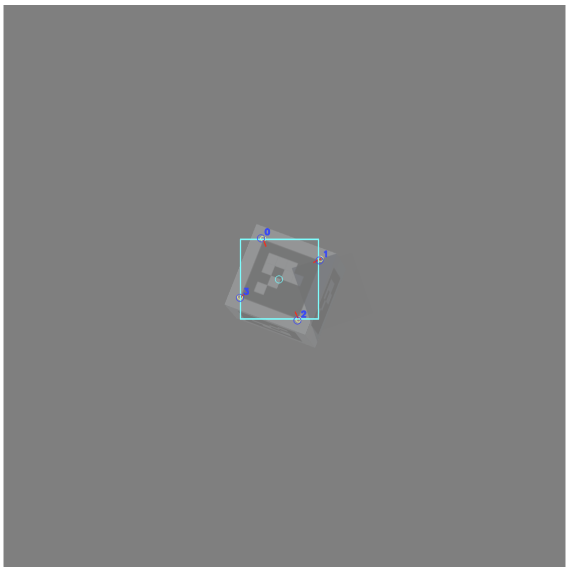
Detection with fog in the scene

A detection scenario with fog in the scene was also performed. This was to test the detection algorithm to other real-world scenarios that are likely to occur. The results from the dataset of 100 images is shown in a excerpt of figures below. Figure 4.37 shows the result from screenshot 3 with fog on. In this image, the marker is far away from the camera, but still visible, and the fog is dense. The DeepTag algorithm is able to detect the ArUco marker in the image even with a little shadow covering the marker on the side. In Figure 4.38, where the results from screenshot 7 is shown, the ArUco marker is closer to the camera and the fog is therefore less dense. The detection algorithm is then able to detect all of the three markers in the image. In the next figure, Figure 4.39, shows the results from screenshot 41. This figure is the figure displayed with the marker closest to the camera. The marker is therefore seen clearly and there is even more visibility then in the previous figure. The algorithm is also able to detect all of the three markers here. Figure 4.40, which is the last figure displayed below, shows the results from screenshot 87. Here the marker is the furthest away and barley visible. The cube is rotated so that two sides are shown. One of the sides of the cube with an ArUco marker is covered by shadow and the other side with another ArUco marker does not have any shadow covering it. However, the DeepTag algorithm is not able to detect any of the markers in this image. This is due to the dense fog making the marker invisible for the algorithm to detect it, which happened in 9% of the images. The images where the marker was too close to the camera constituted 4%, the total detection was then at 87%. The results are displayed in Table 4.6.

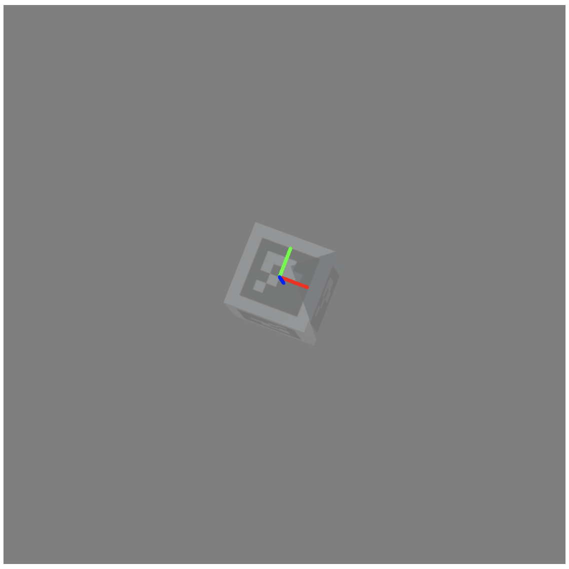
Table 4.6: Results from the detection with different contrast.

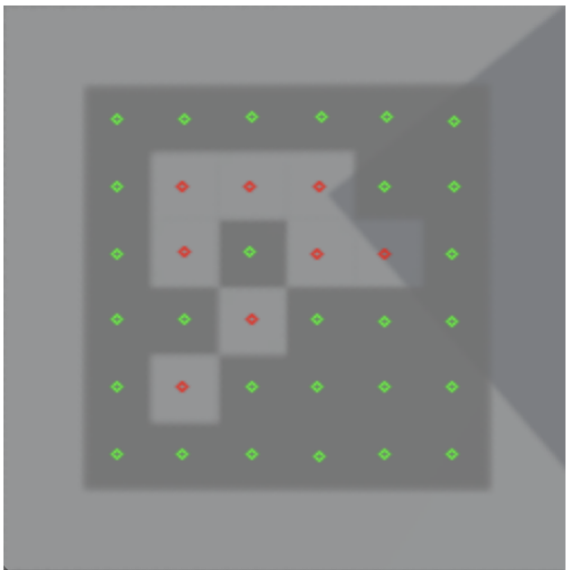
Fog	**Number of images**	**Detection rate (%)**
ArUco marker detected	87/100	87
Too much fog	9/100	0
Marker too close to camera lens	4/100	0


(a) Detection of ROIs (Stage-1) in screenshot 3 with fog in the scene.



(b) Detection of boxes and corners (Stage-1) in screenshot 3 with fog in the scene.


(c) Image with pose in screenshot 3 with fog in the scene.


(d) Keypoints detected (Stage-2), number of markers on screenshot 3 with fog in the scene.

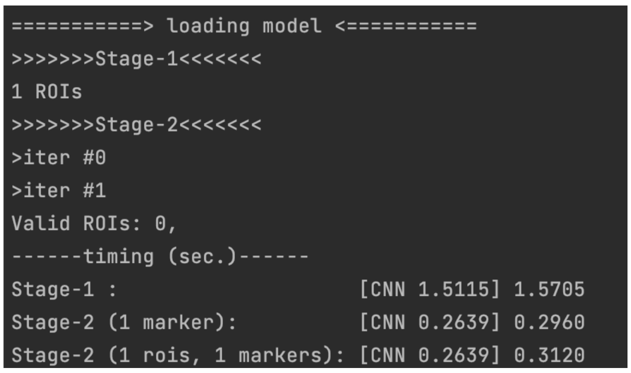
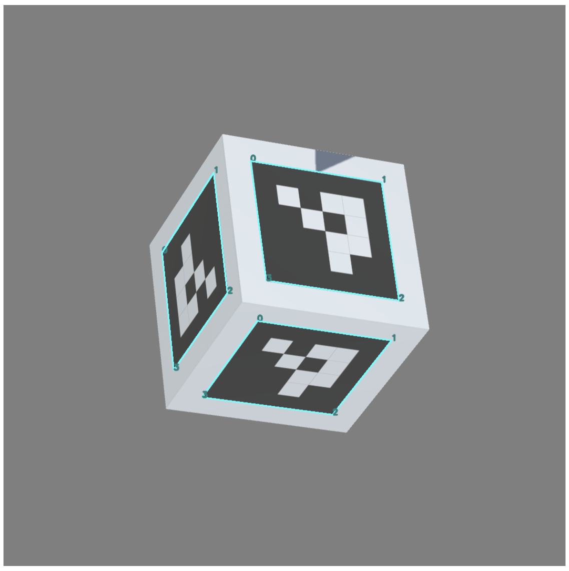
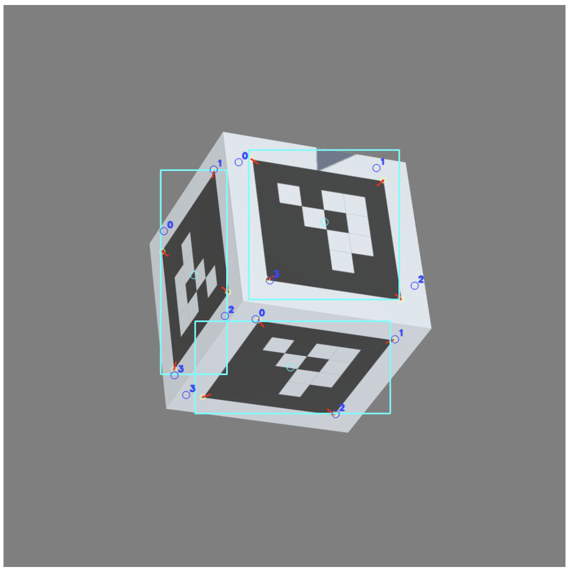
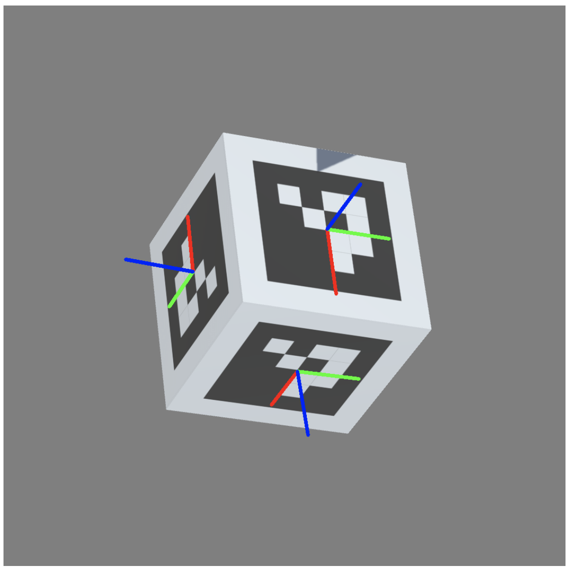
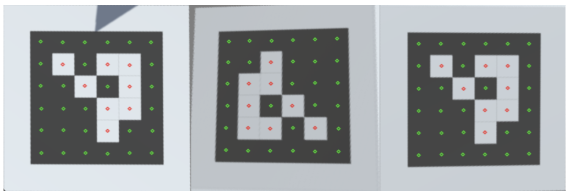

```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
1 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0,
------timing (sec.)------
Stage-1 :                    [CNN 1.5115] 1.5705
Stage-2 (1 marker):          [CNN 0.2639] 0.2960
Stage-2 (1 rois, 1 markers): [CNN 0.2639] 0.3120
```
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 3 with fog in the scene.

Figure 4.37: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with fog in the scene.


(a) Detection of ROIs (Stage-1) in screenshot 7 with fog in the scene.


(b) Detection of boxes and corners (Stage-1) in screenshot 7 with fog in the scene.


(c) Image with pose in screenshot 7 with fog in the scene.


(d) Keypoints detected (Stage-2), number of markers on screenshot 7 with fog in the scene.

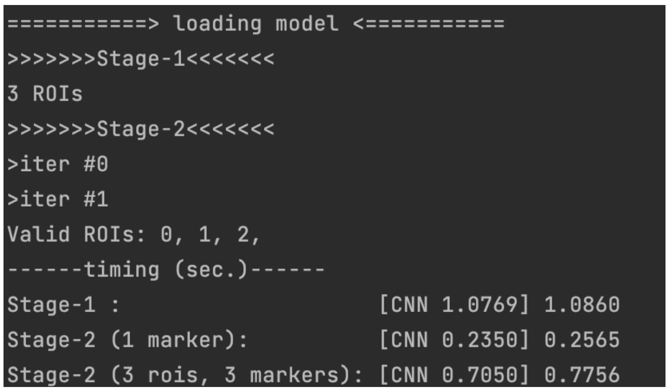
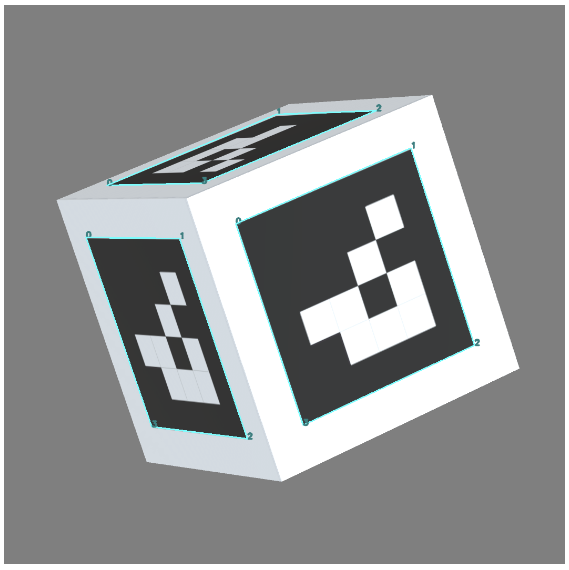
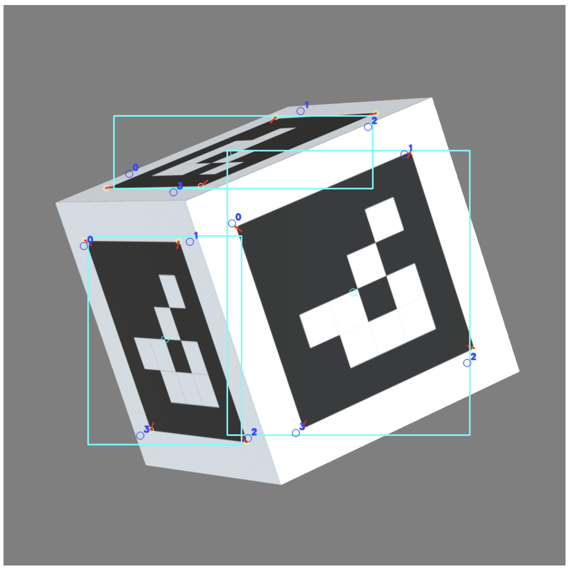
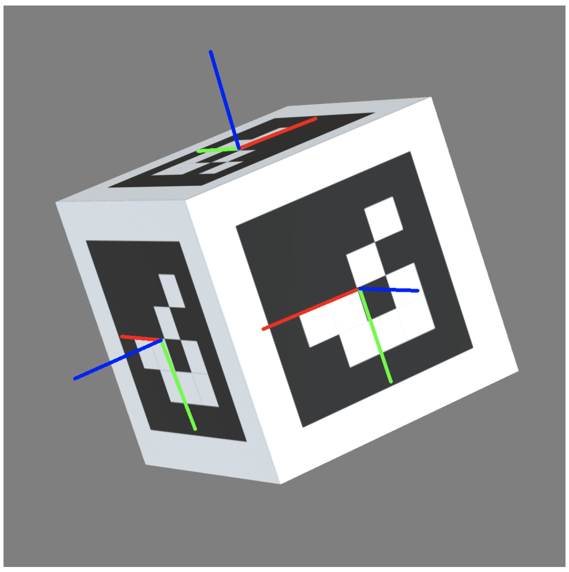
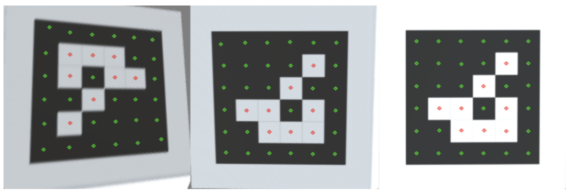

```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
3 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1, 2,
------timing (sec.)------
Stage-1 :                    [CNN 1.0769] 1.0860
Stage-2 (1 marker):          [CNN 0.2350] 0.2565
Stage-2 (3 rois, 3 markers): [CNN 0.7050] 0.7756
```
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 7 with fog in the scene.

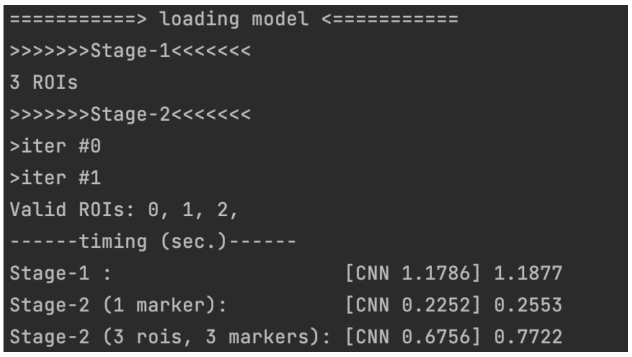
Figure 4.38: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with fog in the scene.


(a) Detection of ROIs (Stage-1) in screenshot 41 with fog in the scene.


(b) Detection of boxes and corners (Stage-1) in screenshot 41 with fog in the scene.


(c) Image with pose in screenshot 41 with fog in the scene.


(d) Keypoints detected (Stage-2), number of markers on screenshot 41 with fog in the scene.



```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
3 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs: 0, 1, 2,
------timing (sec.)------
Stage-1 :                    [CNN 1.1786] 1.1877
Stage-2 (1 marker):          [CNN 0.2252] 0.2553
Stage-2 (3 rois, 3 markers): [CNN 0.6756] 0.7722
```

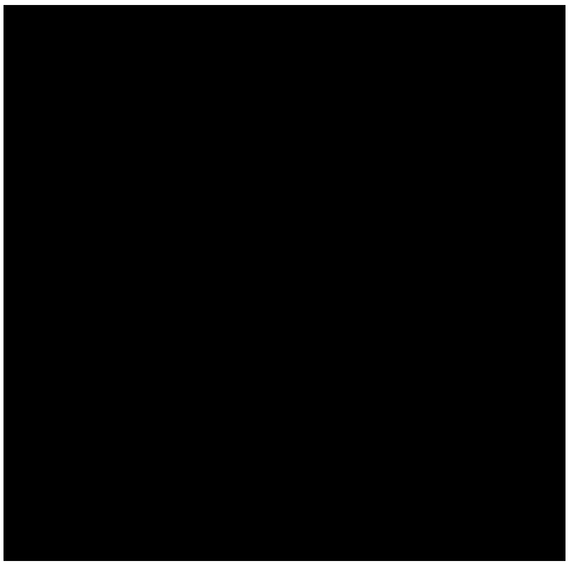
(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 41 with fog in the scene.

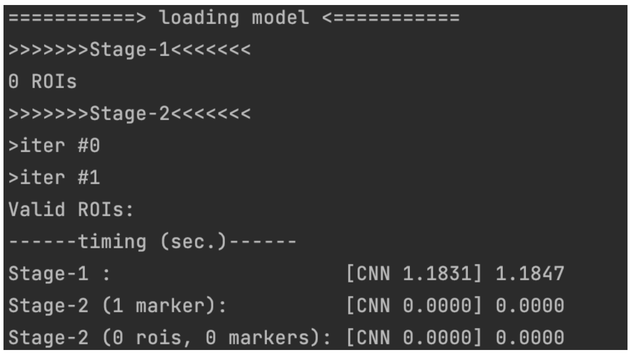
Figure 4.39: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with fog in the scene.

(a) Detection of ROIs (Stage-1) in screenshot 87 with fog in the scene.

(b) Detection of boxes and corners (Stage-1) in screenshot 87 with fog in the scene.

(c) Image with pose in screenshot 87 with fog in the scene.


(d) Keypoints detected (Stage-2), number of markers on screenshot 87 with fog in the scene.



```
===========> loading model <===========
>>>>>>>Stage-1<<<<<<<
0 ROIs
>>>>>>>Stage-2<<<<<<<
>iter #0
>iter #1
Valid ROIs:
------timing (sec.)------
Stage-1 :                    [CNN 1.1831] 1.1847
Stage-2 (1 marker):          [CNN 0.0000] 0.0000
Stage-2 (0 rois, 0 markers): [CNN 0.0000] 0.0000
```

(e) The terminal for loading the model with the stages and number of ROIs and markers detected in screenshot 87 with fog in the scene.

Figure 4.40: Figure a) and b) show the results from stage 1, detection of ROIs, boxes and corners. Figure c) shows the image with poses. Figure d) shows stage 2 with the detected keypoints and markers and e) shows the terminal and what is detected by the model with fog in the scene.

Chapter 5

Discussion

5.1 Implementation

The objective of this project was to generate training data and propose a solution to detect ArUco markers with different difficulties, since the existing algorithm for detection in OpenCV with interruptions in the scene is less than adequate [5]. To find an adequate method for ArUco markers, the state of the art of this technology was researched. The chosen algorithm was retrieved from the report "DeepTag: A General Framework for Fiducial Marker Design and Detection" [10]. When the detection algorithm was chosen, ArUco markers were generated. Python was used to generate different markers, but only one was used for testing purposes in this study. Only one kind of marker was necessary to test the detection because the purpose of the project was to detect the marker in different scenarios with different difficulties and interruptions. A script was created for the generation of ArUco markers so that it could easily be changed to a different marker if it is desirable to test and train the detection algorithm on different markers for further work.

In order to digitally generate training data and change parameters frequently, Unity was used. The project was able to proceed as planned if the country was shut down again due to covid-19. In Unity, the real-world environment was created with the ArUco marker in the scene. Before data generation could begin, all of the actions to each object was scripted in C#. As shown in the activity diagram in Figure 3.5, a main script was created to generate the parameter and put these in a list. Different scrips were created for each of the objects in the scene. With different scripts for the different tasks, it was easier to maintain an overview over when each of the different actions occurred. The scripts made it possible to move the objects in the desired positions so the dataset could be generated. By scripting the movements, changing parameters or addition of objects were made easier. The different objects created in the virtual environment were the ground plane, the cube with the ArUco markers attached, the shadow-plane, and the camera to take images. Once the scripts worked and run properly, the training data could be generated. The most difficult part of generating the training data was to get the rotation parameters correct. Due to the framework in Unity, the rotation parameters needed to be changed from quarterion to angles. Research identified a simple function that could convert these, so that the rotation parameters were displayed in the right form. When the model was working properly, it was time to generate the dataset to use for detection. The dataset was created by hitting the play button in Unity, and the scripts in C# that were attached to each objects ensured that the parameters were changing and that the screenshots with a json file containing the parameters were saved to the computer. Once the first dataset with changing position, rotation and shadow was created, more difficult parameters were added to the scene. The generation of datasets was run without difficulties. Datasets with more difficulties such as, different brightness, motion blur, rain drops, different contrast and fog were also created. These difficulties can help simulate a real-world environment and was added to test how the ArUco marker would be detected under these circumstances.

5.2 Detection comparison

When the dataset with training data was generated, the proposed detection algorithm, DeepTag, was tested to see if it could detect the ArUco markers. The initial results were promising, indicating a high accuracy potential in the application of the DeepTag algorithm to the problem. The different tables in Results 4 shows the detection of each part where interruptions are added to the scene. Table 5.1 below, shows an overview over all of the detection from each section put together in one table to give a better comparison.

Table 5.1: Detection overview.

Position, rotation and shadow	**Number of images**	**Detection rate (%)**
Markers detected	283/300	94,4
Marker too close to camera lens	17/300	0
Brightness (%)	**Number of images**	**Detection rate (%)**
-90	86/100	86
Shadow covering ArUco marker at -90	10/100	0
Marker too close to camera lens at -90	6/100	0
-50	94/100	94
Marker too close to camera lens at -50	6/100	0
50	95/100	95
Marker too close to camera lens at 50	5/100	0
90	97/100	97
Marker too close to camera lens 90	3/100	0
Motion blur	**Number of images**	**Detection rate (%)**
ArUco marker detected	73/100	73
Too much motion blur	25/100	0
Marker too close to camera lens	2/100	0
Rain drops	**Number of images**	**Detection rate (%)**
ArUco marker detected	94/100	94
Too blurry due to rain drops	4/100	0
Marker too close to camera lens	2/100	0
Contrast (%)	**Number of images**	**Detection rate (%)**
-90	87/100	87
Shadow covering ArUco marker at -90	8/100	0
Marker too close to camera lens at -90	5/100	0
90	95/100	95
Marker too close to camera lens at 90	5/100	0
Fog	**Number of images**	**Detection rate (%)**
ArUco marker detected	87/100	87
Too much fog	9/100	0
Marker too close to camera lens	4/100	0

From the detection overview, all of the results are displayed in one table. The first section in the table shows the results from the detection on the first dataset. The first dataset was where the position, rotation and the shadow were changed. The detection was done on the first 300 images of the dataset with 1000 images, since 300 images were assumed to give a sufficient indication of the accuracy of the detection algorithm with respect to images at large distance from the camera, close to the camera, and with or without shadow. From detected images, the results from the DeepTag algorithm was a 94,4% detection rate. The only images the algorithm was not able to detect were the images where the marker was too close to the camera. As seen in Table 5.1, this occurred 17 out of 300 times, which is 5.6%. However, the algorithm was not expected to detect the markers in the images where there was not a whole marker to be detected. Since the detection algorithm depends on finding the boxes and corners to be able to detect the frame of the marker and the white and black boxes inside of it, this was impossible when the whole marker was not visible. The DeepTag algorithm detected the markers with the shadow covering the marker or parts of it in this scene. When the cube with the ArUco markers was rotated at an angle where one of the markers on one of the sides of the cube was barely seen, the DeepTag algorithm could not detect it.This was expected as the markers were almost invisible and are shown as a black line with some white on the side of the cube. The DeepTag algorithm was always able to detect the remaining markers on the cube when at least one whole marker was in front of the camera, independent of the position, rotation or if there was shadow in the image. In a real-world scenario, it is likely that there would only be one marker in 2D placed in front of the camera view, and therefore this is of less importance as long as one of the markers in the image could be detected regardless of the angle.

The next created dataset was with adjusted brightness. The results from this detection of ArUco markers were interesting. The brightness was decreased by -90% and -50% to test the accuracy of the DeepTag algorithm in detecting markers in poor lighting. Brightness was also increased by 50% and 90% to test the accuracy of the algorithm in this scenario. Datasets, consisting of 100 images, for each of these lighting adjustments were created. As seen in Table 5.1, the results were dependent upon the lighting in the scene. When the brightness in the scene was reduced to -90%, some complications were discovered. When the shadow covered half or more of the marker in this poor lighting, the DeepTag detection algorithm was not able to detect any ArUco markers. This problems occurs since the marker becomes too similar to the background when the shadow covers it and the scene is dark. Since this occurred in 10% of the images from the dataset with brightness decreased to -90%, it would be preferable to train the algorithm better for detection in poor lighting. It is recommended that this is investigated further in later studies to be able to increase detectability of ArUco markers in dark scenarios. The DeepTag algorithm was able to detect the markers in the other scenarios with adjusted brightness. With the brightness decreased to -50%, the algorithm was able to detect the markers in all of the images where the whole marker could be seen, with or without shadow. The same was the case for when the brightness was increased to 50% and 90%. In these cases, the algorithm could easily separate the markers without any problem because the black and white boxes in the marker, and the frame with corners were clearly visible. As expected, and similar to the previous scene, when the marker was too close to the camera, the marker could not be detected. However, this situation was not common in the dataset, and the dataset was considered to have sufficient amount of markers in different positions to evaluate the impact of brightness on marker detection.

After the detection of the marker in different positions, rotations, with or without shadow and with different brightness in the scene, other interruptions were added. Motion blur was added to test how well the DeepTag algorithm could detect ArUco markers with this interruption in the scene. The dataset created with motion blur on the images consisted of 100 images. When testing the detection algorithm on this dataset, the results were poor compared to the other cases. As expected, the algorithm was not able to detect the markers that could not be fully seen in the camera view, but the algorithm also had difficulties detecting the images with a large amount of motion blur as shown in Table 5.1. When motion blur was decreased, the markers were easily detected. In 73% of the images, the DeepTag algorithm was able to detect the marker with motion blur in the scene. It became harder as the amount of motion blur increased and the algorithm was unable to detect 25% of the images in the dataset due to this. The marker became too blurry and the corners were faded and blended with the background. In some cases the whole marker was distorted due to the amount of increased motion blur. Shadow did not have a detection impact, only the motion blur had a significance. Thus, the result of the detection of markers with motion blur is disappointing and this algorithm needs further work with more training on motion blur to be used for this purpose in the real-world.

To test the DeepTag algorithm on more difficult tasks with interruptions that could simulate the real-world, rain drops were added to the scene. A dataset of 100 images was created to test how good the detection was when it was raining. The result indicated that the algorithm was able to detect the marker during rain, except from when the rain drop hit right on the marker. The amount of images where the marker was detected is shown in Table 5.1. When the rain drop hit the camera right where the marker was and blurred out, it was not possible for the DeepTag algorithm to detect the marker under the rain drop since it could not find the boxes and corners needed to detect it. However, the results seen in Table 5.1 from the detection with rain drops on the camera lens were at 94%, which is a good detection rate. The 94% represents situations where at least one marker in the image was detected. If all of the images where there was one or more markers that could not be detected due to rain drops hitting the camera lens right above the marker were counted as not detectable, it would have constituted 22%. Adding this with the 2% that could not be detected due to the marker being too close to the camera lens, means that only 76% of the dataset achieved 100% detection of all the markers in the image. Further work is recommended to train the detection algorithm upon detecting the markers even when a rain drop is hitting right where the marker is or when it flows on the camera lens.

After the algorithm was tested with the rain drops hitting the camera, different contrasts were added to test if it had any impact on the detection. As shown in Table 5.1, the results were excellent when the contrast was increased. This is because it makes it easier to see the different colors in the image with increased contrast. It was therefore easy for the detection algorithm to find the contrast between the black and white boxes in the marker, the shadow and the background. The marker was detected with 100% certainty, except for when the marker was too close to the camera, as expected. When the contrast was decreased, the detection rate became poorer. It was more difficult for the detection algorithm to see the difference between the white and black boxes in the marker, the shadow and the background. The detection algorithm was not able to separate the marker and the background when the shadow covered larger parts of the marker. The detection algorithm was only able to detect the markers in 87% of the images, with 5% of the markers being too close to the camera view and 8% constituted shadow covering the marker. The algorithm should therefore be trained better on decreased contrast. However, if the algorithm is trained better on different lighting and darker scenarios, it would probably be able to detect markers in images with decreased contrast as well.

To test even further with scenarios relevant for the real world, fog was added to the scene. The results from marker detection shown in Table 5.1 were interesting. When the marker was far away from the camera, the fog became very dense. The closer the marker was to the camera, the clearer and more visible the marker became. This is similar to the real world, where the fog behaves in the same way. The closer an object is in the fog, the clearer it is. On the markers where the fog were most dense and they were far away, the detection algorithm was not able to detect the markers. This happened 9 out of 100 times, which is not a high frequency, but it could be improved. The algorithm can be trained better to detect markers in the fog. If the algorithm is trained to better detect decreasing brightness, the algorithm would probably be better fitted to detect the markers in the dense fog as well. The markers that were close to the camera and only had a clear fog over, had a high detection rate. The algorithm could easily detect these markers. The marker detection accuracy with fog was 87% and would have been 91% if the markers too close to camera were removed. The results could be improved and it is recommended that the DeepTag detection algorithm is trained further in future studies.

5.3 Comparing results from this method and the method in DeepTag

The biggest differences between the DeepTag study and this study is that the DeepTag project detects multiple different markers including ArUco markers, while this study only uses ArUco markers. The dataset in the DeepTag report consisted of 10 000 images for each marker, where there are 10 different poses with 1000 images each. From the results in the report about DeepTag, 100% of the ArUco markers were detected. However, the reason for this higher rate of detection accuracy in the DeepTag study than in this study with only ArUco markers, is that this study uses increasingly difficult scenarios. When the dataset for DeepTag was created, the images were captured physically and therefore had a smaller variation and an lower degree of difficulty than the dataset in this report. The markers in the DeepTag study were presented with one marker in the image and with a cube with several markers. However, the cube was never rotated to the degree where one of the markers was barley seen, so the DeepTag algorithm was always able to detect all of the markers. Since the experiment in DeepTag was done physically, the marker was also never too close to the camera such that the whole marker could not be seen. If the images where this happened in this study about ArUco markers were removed, 100% detection accuracy would have been achieved here in almost all of the cases if the contrast and blur was not of a large amount. The cases where the DeepTag algorithm still would not be able to detect all of the markers in this project would be with the brightness reduced to -90%, where there was a large amount motion blur, where the rain drops made blur or where there was a small contrast. However, all of these cases have the same problem where there is either too little contrast and light or too much blur. This was not tested in the DeepTag study. The DeepTag study tested detection with motion blur, but that study used less motion blur than this study. In this study on ArUco markers, motion blur was included when random parameters were added and the cube was moving at the same time. In some of the images with the largest amount of motion blur it was barely recognized that there was a marker on it. If the parameters were not drawn randomly and motion blur was added to a still image, the results would probably be very different. If that was the case, the results from this report would probably be the same as in the DeepTag report where the detection rate was at 100% with motion blur added to the still image. Since the DeepTag report did not test the detection with different brightness, it is hard to compare the results from this. However, this is probably the main issue that the DeepTag algorithm could be trained further on, in addition to to motion blur in moving images.

5.4 Comparing results from this report to ChArUco report

Since the ChArUco report contributed to inspire this project, the results from that report and this report are also compared. The ChArUco project used both extreme lighting and motion blur with ArUco markers, since a ChArUco board is a chessboard full of ArUco markers in the white squares. The ChArUco project uses 2D markers where they apply different lighting, motion blur and other interruptions and make datasets consisting of videos. In that project, the results were compared to the OpenCV detection algorithm and the ChArUco detection worked at least twice as good as the detection in OpenCV in 11 scenarios when motion blur was added. The dataset in ChArUco consisted of 22 videos. The videos in the ChArUco project are still images from the video where motion blur is added. Compared to this project, the markers changes position and rotation which makes the motion blur even more severe in some cases. Under extreme blur, the ChArUco project is able to detect 78% of the videos. In this report, the DeepTag algorithm used for detection is able to detect 73% of the dataset with motion blur. However, the algorithm used in ChArUco would probably not be able to detect the markers in the images with the most motion blur seen in this project with ArUco markers, since the motion blur is so much higher than what the detection algorithms are trained for. Since the parameter change for positions and rotations happens at the same time as the motion blur, the markers are either distorted or faded out so the boxes and corners are not visible. With the different brightness, the ChArUco project tested videos where it was almost completely dark. The algorithm worked better on the ChArUco boards in the darkness than what the DeepTag algorithm did on the ArUco markers in this project. An interesting study could be to test the algorithm from ChArUco on this ArUco project with the dataset with increased brightness to -90% to see if it would give a better detection rate. Regardless, the DeepTag algorithm used in this project needs better training for poor lighting and could be trained more on extreme motion blur.

Chapter 6

Conclusion

The proposed method is proven to work well on ArUco markers. The generation of data was done in Unity where it was easier to change parameters and add more difficulties to the scene since it was done digitally. It was therefore more efficient to generate new datasets than it would have been to set up the environment and take the images physically. The DeepTag algorithm worked on all of the datasets, but was challenged with complications on different difficulties and interruptions. For the most part, the complications were the same. The marker in the image was either hard to separate from the background due to lack of light, contrast or fog, or the markers were not recognizable due to a large amount of motion blur or rain drops flowing over the camera lens and making blur. It is therefore concluded that the DeepTag algorithm needs more training upon darkness and lack of light, and on motion blur. This would probably help the algorithm overcome more of the other difficulties and interruptions easier. The algorithm in this project is shown to work better on images with higher contrast and lighting where the detection rate would have been 100% if the images where the marker was too close to the camera were excluded. The comparison between this report and the two other papers showed that the DeepTag algorithm worked well on what it was trained for, and perhaps as good as the algorithm used in ChArUco study.

6.1 Further work

Further work that can be done is to train the DeepTag algorithm better on especially dark scenarios where there is very little light, on motion blur and when a raindrop flows over the marker and makes blur. The algorithm can also be trained better when there are small contrasts in the image, but this will probably be improved if the algorithm is trained upon lack of light. The same problem was present with the fog. When the fog was dense, the contrast was low and it was difficult for the algorithm to detect the marker. Further work would therefore be to train the algorithm for these purposes.

Other things that are recommended to be done further on this topic can be to test with different ArUco markers, and then maybe also expand to other markers. This would help to make the model more robust and accurate. By training the model to detect different markers of either the same family or different families, the model will be better at detecting the right markers in the angles and position, but also in the right environments. To verify the overall real-world efficiency of the proposed algorithm, real-world data has to be used.

Bibliography

[1] Open CV, Open Source Computer Vision, "Detection of ArUco Markers". Website: https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html (12.01.22)

[2] Laurent Itti, "Demo ArUco: Simple demo of ArUco augmented reality markers detection and decoding.", http://jevois.org. Website: http://jevois.org/moddoc/DemoArUco/modinfo.html (12.01.22)

[3] Oleg Kalachev, "ArUco markers generator!". Website: https://chev.me/arucogen/ (17.01.2022)

[4] D. Jurado-Rodríguez, R. Muñoz-Salinas, S. Garrido-Jurado and R. Medina-Carnicer, "Design, Detection, and Tracking of Customized Fiducial Markers," in IEEE Access, vol. 9, pp. 140066-140078, 2021, doi: 10.1109/ACCESS.2021.3118049. (18.01.2022)

[5] Danying Hu, Daniel DeTone, Vikram Chauhan, Igor Spivak, and Tomasz Malisiewicz, "Deep ChArUco: Dark ChArUco Marker Pose Estimation", in arxiv, CVPR 2019, doi: https://doi.org/10.48550/arXiv.1812.03247 (07.01.2022)

[6] H. Sarmadi, R. Muñoz-Salinas, M. A. Olivares-Mendez and R. Medina-Carnicer, "Detection of Binary Square Fiducial Markers Using an Event Camera," in IEEE Access, vol. 9, pp. 27813-27826, 2021, doi: 10.1109/ACCESS.2021.3058423. (18.01.2022)

[7] Francisco J. Romero-Ramirez, Rafael Muñoz-Salinas, Rafael Medina-Carnicer, "Speeded Up Detection of Squared Fiducial Markers", in researchgate, Image and Vision Computing , 2018, doi: 10.1016/j.imavis.2018.05.004 (19.01.2022)

[8] Ehambram, A., Hemme, P. and Wagner, B., "An Approach to Marker Detection in IR- and RGB-images for an Augmented Reality Marker", in Proceedings of the 16th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2019), pages 190-197 ISBN: 978-989-758-380-3, doi: 10.5220/0007810301900197 (19.01.2022)

[9] Collins, T., Bartoli, A. Infinitesimal Plane-Based Pose Estimation. Int J Comput Vis 109, 252–286 (2014). doi: https://doi.org/10.1007/s11263-014-0725-5 (20.01.2022)

[10] Zhuming Zhang, Yongtao Hu, Guoxing Yu, Jingwen Dai, "DeepTag: A General Framework for Fiducial Marker Design and Detection", in arxiv, 2021, doi: https://doi.org/10.48550/arXiv.2105.13731 (19.01.2022)

[11] Girshick, Ross B.. “Fast R-CNN.” 2015 IEEE International Conference on Computer Vision (ICCV) (2015): 1440-1448. (03.03.2022)

[12] Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", in arxiv, 2016, doi: https://doi.org/10.48550/arXiv.1506.01497 (03.03.2022)

[13] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, Alexander C. Berg, "SSD: Single Shot MultiBox Detector", in arxiv, 2016, doi: https://doi.org/10.48550/arXiv.1512.02325 (03.03.2022)

[14] TechTarget Contributor, "Convolutional neural network", 2018. Website: https://www.techtarget.com/searchenterpriseai/definition/convolutional-neural-network (27.04.2022)

[15] Sumit Saha, "A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way", in Towards Data Science, 2018. Website: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (27.04.2022)

[16] Luciano Strika, "How to Train Convolutional Neural Networks in Python (TensorFlow Eager API)", in Towards Data Science, 2019. Website: https://towardsdatascience.com/convolutional-neural-networks-an-introduction-tensorflow-eager-api-7e99614a2879 (27.04.2022)

[17] JetBrains, "Download PyCharm", Version: 2022.1. Website: https://www.jetbrains.com/pycharm/download/#section=mac (31.01.2022)

[18] Sunita Nayak, "Augmented Reality using ArUco Markers in OpenCV (C++ / Python)", LearnOpenCV, 2020. Website: https://learnopencv.com/augmented-reality-using-aruco-markers-in-opencv-c-python/ (23.01.2022)

[19] Unity. Website: https://unity.com/ (27.01.2022)

[20] Unity, "Download Unity". Website: https://unity.com/download (07.01.2022)

[21] Unity Technologies, "Explore the Unity Editor", 2022. Website: https://learn.unity.com/tutorial/explore-the-unity-editor-1#6124fff2edbc2a5a1a77fc38 (18.01.2022)

[22] Unity Technologies, "Types of light". Website: https://docs.unity3d.com/Manual/Lighting.html (22.03.2022)

[23] Amit Patpatia, "Unity Shadows Not Connecting To Objects (Shadow bias Solution)", YouTube, 2021. Website: https://www.youtube.com/watch?v=Plp-LEm_G4A (22.03.2022)

[24] Unity, "Deep dive with post processing color grading | Unite Now 2020", YouTube, 2020. Website: https://www.youtube.com/watch?v=1w9p-CrZYCU (22.03.2022)

[25] SpeedTutor, "Changing MATERIAL Colors in Unity [Instances & More]", YouTube, 2020. Website: https://www.youtube.com/watch?v=VEAU95v5MO8 (22.03.2022)

[26] Microsoft, "File.WriteAllText Method". Website: https://docs.microsoft.com/en-us/dotnet/api/system.io.file.writealltext?view=netframework-4.8#system-io-file-writealltext(system-string-system-string-system-text-encoding) (05.03.2022)

[27] Unity Technologies, "TransformUtils.GetInspectorRotation", 2022. Website: https://docs.unity3d.com/ScriptReference/TransformUtils.GetInspectorRotation.html?fbclid=IwAR0CwDf2FZSeXphhEkeqlIIWW50o0i_zCnRkuaeAcNDZCLGEKui6XPwZf9M (01.04.2022)

[28] Jeff Johnson, "Simple LUT Adjuster", Unity Asset Store, 2017. Website: https://assetstore.unity.com/packages/vfx/shaders/fullscreen-camera-effects/simple-lut-adjuster-51762 (27.03.2022)

[29] Yongtao Hu, "deeptag-pytorch", GitHub, 2021. Website: https://github.com/herohuyongtao/deeptag-pytorch (19.01.2022)

[30] The Game Guy, "How to add MOTION BLUR to your UNITY game - Post Processing Effect", YouTube, 2020. Website: https://www.youtube.com/watch?v=CSfnAbvtbQ4 (14.05.2022)

[31] Nameer Hirschkind, Saruque Mollick, Jyo Pari, "Convolutional Neural Network", Brilliant, Retrieved 13:16, 2022. Website: https://brilliant.org/wiki/convolutional-neural-network/ (19.05.2022)

[32] Unity Technologies, "Standard Assets", Unity Assets Store, Unity, 2020. Website: https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-for-unity-2018-4-32351#description (21.05.2022)

[33] Dobbey, "[Unity] REALISTIC CAMERA RAINDROP WITHOUT SCRIPTING | Quick Tip 2", YouTube, 2018. Website: https://www.youtube.com/watch?v=OG7zyogrUTs&t=1s (02.05.2022)

[34] Unity Ace, "How to Add Fog in Unity", YouTube, 2022. Website: https://www.youtube.com/watch?v=RBI9hQWD_xY (22.05.2022)

[35] IBM Cloud Education, "Deep Learning", Artificial intelligence, IMB, 2020. Website: https://www.ibm.com/cloud/learn/deep-learning (23.05.2022)

Appendix A

generateArUco.py

Listing A.1: generateArUco.py script.

```python
# Written by Silje Wetrhus Hebnes on 31.01.2022

import cv2
import numpy

# Generating ArUco markers:

# Loading the predefined dictionary
dictionary = cv2.aruco.Dictionary_get(cv2.aruco.DICT_6X6_250)

# Generating the marker
markerImage = numpy.zeros((200, 200), dtype=numpy.uint8)
markerImage = cv2.aruco.drawMarker(dictionary, 20, 200, markerImage, 1);
cv2.imwrite("marker20.png", markerImage);
```

Appendix B

CameraCapture.cs

Listing B.1: CameraCapture.cs script.

```
// Written by Silje Wetrhus Hebnes on 26.01.2022

using System.Collections;
using System.IO;
using UnityEngine;
using System.Collections.Concurrent;
using System.Threading.Tasks;
using System.Collections.Generic;
using System.Text;
using System;
using Random = System.Random;

[System.Serializable]
public class Parameters         //Creating parameter class
{
    public float posx_aruco;    //Position x-direction for ArUco
    public float posy_aruco;    //Position y-direction for ArUco
    public float posz_aruco;    //Position z-direction for ArUco
    public float orix_aruco;    //Rotation x-direction for ArUco
    public float oriy_aruco;    //Rotation y-direction for ArUco
    public float oriz_aruco;    //Rotation z-direction for ArUco
    public float posx_camera;   //Position x-direction for Camera
    public float posy_camera;   //Position y-direction for Camera
    public float posz_camera;   //Position z-direction for Camera
    public float orix_camera;   //Rotation x-direction for Camera
    public float oriy_camera;   //Rotation y-direction for Camera
    public float oriz_camera;   //Rotation z-direction for Camera
    public float posx_shadow;   //Position x-direction for Shadow-plane
    public float posy_shadow;   //Position y-direction for Shadow-plane
    public float posz_shadow;   //Position z-direction for Shadow-plane
    public float orix_shadow;   //Rotation x-direction for Shadow-plane
    public float oriy_shadow;   //Rotation y-direction for Shadow-plane
    public float oriz_shadow;   //Rotation z-direction for Shadow-plane
    public float brightness;    //Brightness in Unity
}

public class CameraCapture : MonoBehaviour
{
    public List<Parameters> parameterList = new List<Parameters>();     ...
       //List of parameters
    public SnapshotCamera snapCam;

    //Range of variables for data generation:
    private int posy_aruco_min = 10;
    private int posy_aruco_max = 450;
    private int orix_aruco_min = 0;
    private int orix_aruco_max = 89;
    private int oriy_aruco_min = 0;
    private int oriy_aruco_max = 359;
    private int oriz_aruco_min = 0;
    private int oriz_aruco_max = 89;
    private int posx_shadow_min = -10;
    private int posx_shadow_max = 10;
    private int posz_shadow_min = -10;
    private int posz_shadow_max = 10;
    private int orix_shadow_min = 0;
    private int orix_shadow_max = 89;
    private int oriy_shadow_min = 0;
    private int oriy_shadow_max = 359;
    private int oriz_shadow_min = 0;
    private int oriz_shadow_max = 89;


    private int frame = 0;

    //Path for images:
    private string imageFolderName = "CameraImages";
    private string imagePath;

    //Path for json file:
    private string jsonFolderName = "jsonFolder";
    private string jsonPath;

    //Starting the script
    void Start()
    {
        Application.targetFrameRate = 2;
        Random r = new Random();
        jsonPath = Path.Combine(Environment.CurrentDirectory, jsonFolderName);
        Directory.CreateDirectory(jsonPath);

        Debug.Log("Starting screen capture script..");

        for (int i = 0; i < 1000; i++)      //Changing the parameters and ...
            creating the json file 1000 times
        {
            parameterList.Add(new Parameters()
            {
                posx_aruco = 0,
                posy_aruco = (r.Next(posy_aruco_min, posy_aruco_max)),
                posz_aruco = 0,
                orix_aruco = (r.Next(orix_aruco_min, orix_aruco_max)),
                oriy_aruco = (r.Next(oriy_aruco_min, oriy_aruco_max)),
                oriz_aruco = (r.Next(oriz_aruco_min, oriz_aruco_max)),
                posx_camera = 0,
                posy_camera = 500,
                posz_camera = 0,
                orix_camera = 90,
                oriy_camera = 0,
                oriz_camera = 0,
                posx_shadow = (r.Next(posx_shadow_min, posx_shadow_max)),
                posy_shadow = 550,
                posz_shadow = (r.Next(posz_shadow_min, posz_shadow_max)),
                orix_shadow = (r.Next(orix_shadow_min, orix_shadow_max)),
                oriy_shadow = (r.Next(oriy_shadow_min, oriy_shadow_max)),
                oriz_shadow = (r.Next(oriz_shadow_min, oriz_shadow_max)),
                brightness = 0
            });

            // save current object to a file
            string jsonString = JsonUtility.ToJson(parameterList[i]);
            string jsonFilePath = Path.Combine( jsonPath, "jsonFile_" + i ...
                + ".json");
            File.WriteAllText(jsonFilePath, jsonString, Encoding.UTF8);
        }
        Debug.Log("Parameters complete ");
    }

    private void Update()
    {
        //Gets the screenshots from "SnapshotCamera.cs" and updates them.
        snapCam.CallTakeSnapshot();
        Debug.Log("UPDATE frame no: " + frame);
        frame++;
    }

    void LateUpdate()
    {
        //Updating 1000 times.
        if (frame >= 1000)
        {
            UnityEditor.EditorApplication.ExitPlaymode();
        }
    }
}
```

Appendix C

MoveShadow.cs

Listing C.1: MoveShadow.cs script.

```
// Written by Silje Wetrhus Hebnes on 02.02.2022

using UnityEngine;

public class MoveShadow : MonoBehaviour
{
    public Vector3 framep;

    private Parameters frameParms = new Parameters();
    private int frame = 0;

    void Update()
    {
        Debug.Log("Shadow frame no: " + frame);
        //Getting parameters from list in "CameraCapture.cs":
        frameParms = ...
           GameObject.Find("MainCamera").GetComponent<CameraCapture>().
        parameterList[frame];
        //Changing the position of the shadow-plane:
        transform.position = new Vector3(frameParms.posx_shadow, ...
           frameParms.posy_shadow, frameParms.posz_shadow);
        //Changing the rotation of the shadow-plane:
        framep = new Vector3(frameParms.orix_shadow, ...
           frameParms.oriy_shadow, frameParms.oriz_shadow);
        UnityEditor.TransformUtils.SetInspectorRotation(transform, framep);
        frame++;
    }
}
```

Appendix D

MoveAruco.cs

Listing D.1: MoveAruco.cs script.

```
// Written by Silje Wetrhus Hebnes on 02.02.2022

using UnityEngine;

public class MoveAruco : MonoBehaviour
{
    public Vector3 framep;

    private Parameters frameParms = new Parameters();
    private int frame = 0;

    void Update()
    {
        Debug.Log("Shadow frame no: " + frame);
        //Getting parameters from list in "CameraCapture.cs":
        frameParms = ...
           GameObject.Find("MainCamera").GetComponent<CameraCapture>().
        parameterList[frame];
        //Changing the position of the ArUco marker:
        transform.position = new Vector3(frameParms.posx_aruco, ...
           frameParms.posy_aruco, frameParms.posz_aruco);
        //Changing the rotation of the ArUco marker:
        framep = new Vector3(frameParms.orix_aruco, frameParms.oriy_aruco, ...
           frameParms.oriz_aruco);
        UnityEditor.TransformUtils.SetInspectorRotation(transform, framep);
        frame++;
    }
}
```

Appendix E

SnapshotCamera.cs

Listing E.1: SnapshotCamera.cs script.

```
// Written by Silje Wetrhus Hebnes on 10.03.2022

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Collections;
using System.IO;
using UnityEngine;
using System.Collections.Concurrent;
using System.Threading.Tasks;
using System.Collections.Generic;
using System.Text;
using System;
using Random = System.Random;

[RequireComponent(typeof(Camera))]
public class SnapshotCamera : MonoBehaviour
{
    //Variables:
    private Camera snapCam;

    public int resWidth = 256;
    public int resHeight = 256;

    private string imageFolderName = "CameraImages";
    private string imagePath;
    private int frame = 0;

    private bool takeHiResShot = false;

    private void Awake()
    {
        //Creating image path:
        imagePath = Path.Combine(Environment.CurrentDirectory, ...
           imageFolderName);
        Directory.CreateDirectory(imagePath);

        snapCam = GetComponent<Camera>();
        if (snapCam.targetTexture == null)
        {
            snapCam.targetTexture = new RenderTexture(resWidth, resHeight, ...
               24);
        }
        else
        {
            resWidth = snapCam.targetTexture.width;
            resHeight = snapCam.targetTexture.height;
        }
        snapCam.gameObject.SetActive(false);
    }


    public void CallTakeSnapshot()
    {
        snapCam.gameObject.SetActive(true);
    }

    void LateUpdate() {
        Texture2D screenShot = new Texture2D(resWidth, resHeight, ...
           TextureFormat.RGB24, false);
        snapCam.Render();
        RenderTexture.active = snapCam.targetTexture; // JC: added to ...
           avoid errors
        screenShot.ReadPixels(new Rect(0, 0, resWidth, resHeight), 0, 0);
        byte[] bytes = screenShot.EncodeToPNG();
        string imageFilePath = Path.Combine(imagePath, "ScreenShot_" + ...
           frame + ".png");
        File.WriteAllBytes(imageFilePath, bytes);
        Debug.Log("Snapshot taken: " + frame);
        frame++;
        snapCam.gameObject.SetActive(false);
    }
}
```