Visual Lab Assistant
Using Augmented Reality To Aid Users In Laboratory Environments

Ricardo Danza Madera
Abstract

Visual Lab Assistant

Ricardo Danza Madera

This thesis was inspired by the desire to make working in cluttered spaces easier. Laboratories are packed full of instruments and tools that scientists use to carry out experiments; this inevitably leads to an increased risk for human error as well as an often uncomfortable experience for the user. Protocols are used to carry out experiments and other processes where missteps will most likely spoil the entire experiment. How could one improve the overall experience and effectiveness in such environments? That’s when the idea of using Augmented Reality (AR) came to mind.

The challenge was to be able to follow a protocol using AR. The application would require objects to be tracked in space while working and recognize which state of the process the user was in. Using OpenCV for the Computer Vision aspect of the application and writing the software in C++, it was possible to create a successful proof-of-concept. The final result was an AR application that could track all the objects being used for the example protocol and successfully detect, and warn, when the user had made a mistake while creating a series of bacteria cultures. There is no doubt therefore, that with more time and development, a more polished product is possible. The question that is left to answer nonetheless, is whether such an application can pass a UX evaluation to determine its usability-value for users in a professional environment.
# Contents

1 Abstract 3

2 Introduction 7

2.1 Prior Work 8

2.2 My Contributions 8

2.3 Important Terms 9

3 Requirements 13

4 Design 15

4.1 Theory 15

4.2 Required Technologies 17

4.2.1 OpenCV 17

4.2.2 ArUco 18

4.2.3 Marker Tracking 18

4.2.4 States and State Machine 18

4.3 Program Design 19

4.3.1 Necessary Modules 19
2 Introduction

Many have seen science-fiction movies and looked in wonder at the transparent screens with floating text being moved and manipulated with hand gestures. Technology and its possibilities continue to intrigue humans who have not stopped trying to push the boundary of what has been considered possible. Yet in truth, many technologies previously limited to the realm of science fiction are much more plausible today than they have ever been; and they are here to make our lives better.

Most enterprises today have to balance between productivity and safety. Prioritize productivity above all else and the risk for accidents increases, have too many safety regulations and profitability would suffer. The obvious solution is to find the perfect balance, to get as much productivity as possible without compromising safety. Since the field of computer science continuously tries to use hardware and software to improve all aspects of life, it is only natural to attempt to solve this problem.

Many work environments are crowded with a wide variety of instruments, many of which if used wrongly, can be hazardous to either the user or the product. To ensure safety, a company often employs safety protocols that dictate adherence to a strict set of actions to reduce the risk of an accident. The more complicated the procedure however, the higher the probability of errors.

A biology laboratory is an example of a place where mistakes can easily spoil an experiment or get a user seriously injured; it is also easy to make mistakes. When working with bacteria and sterile environments and instruments, it is often not possible to “fix” an error, having to instead restart the experiment. For example, if a batch of bacteria cultures gets contaminated, there is no other solution than to discard all the Petri dishes used and start again. The classical solution, as mentioned before, is to use protocols, sometimes written on a whiteboard for people to see in the laboratory, or having them on a laptop nearby.

What if one could use Augmented Reality to help with following protocols and warn for potential errors? Such a program would be able to track instruments and objects in a work space and based on their actions, maintain different states which match the steps in a protocol. It would also warn the user if one attempts an illegal state transition as well as if one has done something that could spoil the experiment. An application that deterministically controls the execution of a process will not be prone to mistakes because of having worked five hours and being eager to eat lunch. The fact that an AR interface is used means that, it is easier to convey information to the user regarding the state of the program and what might have gone wrong in case of a mistake.
It seems like with the existing technology, it could be possible to create an AR application that can track objects and follow through on a protocol, changing states as the user progresses through the task and warning if an error/violation occurs. The application should also warn if there is an illegal use of an instrument, whether a protocol is active or not, for example, when a non-sterilized inoculation loop is about to get dipped into a vial with a bacterial solution.

2.1 Prior Work

While searching for similar applications, a good example was found developed by the company GE[1]. They used a Kinect[2] sensor that could identify several toolboxes in front of it and could warn if the user reached for the an incorrect box during the execution of a task. Their set-up used a projection on to the workbench that showed the user a short animation of the next step the user had to take in the process.

Kinect uses a combination of intensity images and infra-red depth images to determine the position and orientation of the object it tracks as well as the individual parts of the object. For example, Kinect can track a human face looking in one direction and the palm of the hand facing a different one.

The goal of the GE team was to increase productivity in areas of work they called the Four D’s; “dirty, dangerous, difficult and dull jobs”. Another article by the Harvard Business Review focuses on a similar topic, it showed that Augmented Reality increased productivity of a worker installing wiring, the increase was of 34% [3]. From the examples just mentioned, it seems like there is a theme when using Augmented Reality in workplaces and that is to increase safety and productivity. This thesis will therefore focus on a similar topic, taking a biology laboratory as the target environment.

2.2 My Contributions

The application developed for this thesis-work will only use “visual” images, i.e. a matrix with gray-scale and RGB-scale intensity values, from which the program will extract all necessary information. Depth and distance estimation will be done using the Pinhole Camera Model and markers will be used to identify physical objects. In other words, this project will use two-dimensional projections of the real world and based on values previously acquired after calibrating the camera, it should be possible to reconstruct the three-dimensional (3D) position and orientation of an object. Knowing the accurate positions of the objects is a necessary first-step on which other parts of the application will build upon.
After solving the distance estimation problem, this application will be able to follow through the steps of a laboratory protocol. The software will use the location of the different markers to determine the positions of instruments; with this information, the software will then be able to keep track of the "state" of the entire process.

And since this is an AR application, it will be able to give the user visual feedback, through the display screen, of the program’s state. This application will then communicate to the user what state the user is in, it will show what instruments are being tracked, and it should warn when a mistake has been made.

2.3 Important Terms

Augmented Reality
An important building block of this thesis is the use of Augmented Reality, a dictionary definition found on the internet reads: “Augmented reality, commonly abbreviated ‘AR’, is computer-generated content overlaid on a real world environment” [4]. Specifically for this project, instead of overlaying information onto the real world, the medium used will be computer screen, which shows a representation of the real world captured by a camera [5].

Figure 1: Additional computer generated content has been overlaid on the screen.
The use of the AR concept of overlaying information on a computer screen is desired since it is an effective way for the user to interact with the application and to see what the application is “seeing”. The user can see the work environment through the camera as well as additional information regarding the protocol he/she is working with. In Figure 1, one can see the state in the lower left corner of the screen.

**Pose**
Pose in computer vision refers to the position and orientation of an object in space, relative to some coordinate system. The position refers to the coordinates of some center-point of the object, meanwhile the orientation is usually described by a rotation matrix, and indicates the direction in which the object is facing.

**Streaking**
When creating a bacterial culture, the act of drawing a line on the Petri dish with the inoculation loop is referred to as Streaking. This is done several times, and the inoculation loop is sterilized in between each stroke. The goal is to try and isolate single bacteria, as one can see in Figure 2 on the upper left corner where the “4” is, the dots will be bacterial colonies originating from one single cell. To achieve this it is important to sterilize, or “to flame”, the inoculation loop in between each streak.

![Diagram of Quadrant Streaking for isolation into pure culture](image)

Figure 2: Example of streaking. Note that the first streak is made after the inoculation loop has been dipped in the bacterial culture but subsequent streaks are done with a sterile loop and each streak overlaps the preceding one.
Frustum
A geometrical shape that describes the space in front of the camera that is visible to it[7]. Furthermore, for the purposes of this thesis, the frustum will specifically refer to the volume a camera can “see” where the Near and Far ends of the frustum will be delimited by the distances in which a marker can be effectively detected by the program. Taking Figure 3 as an example, the sides of the frustum are determined by the physics of the camera. The Near end however, will be the distance at which a marker is too close to be detected, conversely, the Far end will be determined by the distance at which a marker is too far away to be detected effectively.

Figure 3: Visual representation of a frustum [7].
3 Requirements

For the implementation of this thesis work to be considered successful, certain requirements need to be fulfilled by the application. The main goal is to follow through a protocol and both confirm the correct steps, as well as warn in case of any misstep. The protocol is a set of steps that need to be carried out in order to do something, such as creating a bacterial culture. The steps below describe a simplified sample protocol for creating one or several bacterial cultures; developing an application for the protocol below will be the basis for this thesis’ implementation.

1. Dip the inoculation loop in the vial containing the bacterial solution.
2. Pick up a Petri-dish and Streak.
3. Sterilize the loop using the Bunsen burner.
4. Wave loop in the air until it cools down. (Loop twice, starting from step 2.)
5. Stow away the Petri-dish and either go back to step 1 if one is to use several Petri dishes for more than 1 culture or end the protocol.

A successful implementation should therefore be able to recognize the different states of the process and be able to switch states based on the user’s actions. After every action in the list above, the protocols enters (or re-enters) a new state.

Further requirements are summarized in the list below.

- Recognize Aruco markers and be able to track the marker continuously in space.
- Correctly identify in which state the user is in; where a state corresponds to the steps outlined above.
- Notify in case the user skips a step.
- Warn in case the user has made a serious mistake; one which would require a restart of the work. An example could be using an unsterilized instrument.
- Display information relevant to the instrument or relevant to the process on-screen.
4 Design

No foundation has been laid without a blueprint first saying how. Before writing a program, there has to be a goal, broken up into the necessary components to achieve it. This section covers most of the research done for while designing the application that was to be developed.

4.1 Theory

Neither humans nor cameras truly see the world in three dimensions but rather “perceive” it as being so. This perception is created by interpreting a two-dimensional projection of a three-dimensional light source. Biology takes care of judging the distance and pose of every object we observe. With a computer however, it is up to a programmer to write the software capable of interpreting and processing visual information.

The Pinhole Camera Model (PCM) provides us with the mathematics needed to interpret 2D-projections of the real world. Equation 1 forms the basis of the PCM; this equation is used to calculate the pose and distance between real life objects that are in the camera’s field of view.

\[
\mathbf{s}m' = A \begin{bmatrix} R | t' \end{bmatrix} M'
\]

Equation 2 shows the same equation as in 1 but expanded.

\[
s \begin{bmatrix} x \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}
\]

\((X, Y, Z)\) are the 3D coordinates of a point in world space.
\((u, v)\) are the coordinates of the projection point in pixels.
\(A\) is the camera matrix, or a matrix of intrinsic parameters.
\((cx, cy)\) is a principal point that is usually at the image center.
\(f_x, f_y\) are the focal lengths expressed in pixel units.
Matrix $A$ is obtained after performing a camera calibration and is constant as long as one is using the same camera that was used while calibrating. The second matrix, $[R|t']$, called a matrix of extrinsic parameters, is unique for each point in 3D space. This matrix is used for translating an object’s 3D pose the camera’s 2D image plane. Having each object’s “real world” position mapped on to the camera’s coordinate system is what enables us to more accurately determine distances between objects, since there is a common coordinate system used for all objects. Figure 4 provides a more visually intuitive understanding of the pinhole camera model.

![Pinhole camera model representation](image)

In Figure 4, a fixed coordinate system with respect to the camera is shown. The plane in the middle is known as the projection plane, and it is determined by the focal length of the camera. (One can see on the lower left corner of the plane the equation $z = f$) Since the camera matrix and distance coefficients are intrinsic, i.e., constant for each individual camera, the frustum-space will vary in dependence to the specific camera being used.

**Projecting 3D points to the image plane**

The model in Figure 4 can be used to estimate the Euclidean distance between objects. Given the points $(u, v)$ of an object on the projection plane, one can solve Equation 2, assuming one already has the extrinsic parameters for each object and the intrinsic parameters of the camera, and get the coordinates $(X, Y, Z)$ of the object on the camera’s fixed coordinate system. After calculating the $(X, Y, Z)$ coordinates of two objects, say $(X_1, Y_1, Z_1)$ and $(X_2, Y_2, Z_2)$, one can then find the euclidean distance between them which, assuming all parameters are correct, will be the accurate distance between these two objects in the “real world.”
Any algorithm implemented to calculate distances between two objects using a camera will involve the following high-level steps:

- Calibrate the camera so as to create the camera matrix and get the distance coefficients.
- Get the coordinates of the object on the image plane along with the extrinsic parameters for that object’s point.
- Calculate the 3D position of the object, using the model from Equation 2.
- Use the 3D coordinates obtained from two objects and calculate the Euclidean distance between them.

4.2 Required Technologies

4.2.1 OpenCV

The main software element used for this thesis work is the OpenCV library [10]. It is an open-source library of functions, algorithms and data structures used for Computer Vision and Image Processing. These range from fundamental functions for filtering operations to very advanced algorithms for face recognition and deep neural networks.

OpenCV was not the obvious first candidate for the project but after the initial weeks of research and testing, it became evident that this library would be the most suited. (See Section 5 for a further discussion.) The main reason why OpenCV was chosen can be summarized as follows:

- It is mainly based on C++, which makes for fast-running code, necessary when one wants to process image matrices at a rate that is smooth for the eye, i.e. around 20 frames per second or more.
- It is open source, which means more resources can be found online which are not off-limits monetarily or prohibited by usage rights constraints.
- It contains “sub-libraries” that are immediately necessary for the thesis (Aruco) as well as others that could potentially be used to further develop the application.
- It was compatible with the available hardware and software; a desktop computer, a web camera and Microsoft Visual Studio 2017.
- OpenCV contains the Aruco library of markers. (Section 4.2.2 discusses this library.)
4.2.2 ArUco

The Aruco library [11, 12] is a library included as part of the OpenCV contributions library. It contains the necessary functionality to generate and detect the planar markers called Aruco markers. The advantage of Aruco markers is that they can have small dimensions and, thanks to the work made by the people developing that library, they can be efficiently detected in an image and their pose determined. Very importantly, OpenCV is able to track these markers even in movement.

![Aruco marker example](image)

Figure 5: Example of an Aruco marker [13].

4.2.3 Marker Tracking

There needs to be a way to keep track of different instruments’ positions while an operator is using them. The solution for this thesis work is to attach Aruco markers on the instruments and using the built-in OpenCV function for detecting markers. This process in turn calculates the corresponding extrinsic parameters with which one is able to estimate a marker’s pose. Calculating the distance between the markers can then be performed following the steps mentioned in Section 4.1.

4.2.4 States and State Machine

As described in the Requirements section, it is necessary for the application to keep track of each “stage” of the protocol. Some software is necessary to create states that the program can recognize and follow. Using a State Machine (SM) is an effective way to implement a process that follows a set of rules and disallows certain actions. The SM will give confirmation to the user of the action taken and at what stage of the process the user is in, as well as triggering a warning if the user executes an illegal action. Using an already developed SM library is preferred over spending time to develop a custom SM for the application [14]. The cited library, called tinyFSM, uses C++ templates so that the user can create custom states, transition functions and events.
**How States change**

The simplest method of detecting actions is by simply triggering an event when the distance between two points goes under a certain threshold. For example, once the inoculation loop is less than a centimeter from a Bunsen burner, then the *sterilization Step* can be considered completed, and a transition to the next state can be triggered.

4.3 **Program Design**

4.3.1 **Necessary Modules**

The program contains a `main.cpp` file which is the main node for all other code files in the project. These classes are:

- Instrument.h
- Calibration.h
- Protocol.h

See Figure 6.
Protocol.h This class inherits from a Finite State Machine template in tinyfsm.hpp:

class Elevator: public tinyfsm::Fsm<Elevator> {}.

Custom States are then defined as well as transition conditions. After implementation, a Protocol object will keep track of the States that correspond to the protocol steps mentioned in the Requirements section, Section 3. In a more developed application, there would be several Protocol classes for the different lab-protocols one can expect to work with.
Instrument.h Instances of this class will contain information of the object’s associated Aruco marker ID, the object’s position and type (e.g. LOOP or PETRI), as well as several flags that might be necessary in the program. Instances of Instrument are mainly used as a bridge between the physical instruments and the software. Instrument objects send information of different events to the Protocol instance which in turn decides the course of action to take.

Calibration.h A module heavily inspired by George Fleckes’ video series on OpenCV. This is used for determining the camera matrix and distortion coefficients of the camera that is being used. Without these, it is impossible to effectively estimate the distances between objects in 3D space [15].

vLabAss.cpp The main file containing the main() function that initializes and drives the application as well as containing several important function definitions.

4.3.2 Program Flow

Like any good software, a good program flow is necessary for efficiency and for easier understanding by the programmer. The diagrams below illustrate the preliminary software design for the application.

If one were running the program for the first time, it would be necessary to perform a camera calibration, only then would one be able run the application correctly. The reason being that without having performed a camera calibration, there would not be any intrinsic parameters to load (Camera Matrix and Distance Coefficients). The intrinsic parameters are stored on a text file in the same directory as the program files and are read by the loadCalibrationInfo() function.
A Mat object (along with other variables) is initialized when `startWebcamMonitoring()` is called. For every cycle of the loop in Figure 7, the data from the captured image is copied onto the Mat object. This object is then used as a basis for other functions in the program. `Draw()` creates several UX elements such as floating text, labels, visual coordinate axes on the markers, etc and puts them on the outputFrame thus creating an AR effect. The outputFrame is returned and it is the image that is displayed to the user. The same Mat object is reused throughout the application’s running time.
The protocol method uses the Protocol class and is responsible for the operation of a State Machine based on the information gathered about the instrument’s positions. This can be the triggering of events and state transitions.
Figure 9: The conceptual State Machine with the states in the squares and the triggering events on the arrows.

Figure 9 shows the rules and transitions that need to be implemented in the Protocol class in order to have a working application in accordance with the requirements that have been previously set up.
5 Implementation

The initial phase of this thesis work was dedicated to testing and evaluating different tools and frameworks and choosing the best candidate. Although it was initially intended to develop for the Android environment using the ARCore library, it was not possible for reasons that are hereby discussed; it was necessary therefore to find an alternative.

ARCore was considered in the first place since it was described as being an easy to use framework for developing AR applications on Android. It was also described as being able to carry out motion tracking. Motion tracking in ARCore did not however refer to the ability to track a real world object in motion. It instead referred to the ability of the program to track the phone’s own position in space [16].

Parallel with the need to understand ARCore was the need to have basic knowledge of Android development. The main tool used to this end was Android Studio by JetBrains; it was necessary to allocate time to the understanding of Android development.

Android Studio provided the advantage of being able to create, with relative ease, a user interface to interact with. Using the built in preview tool for example, there was no need to compile the app to view it on the phone after every modification, instead, one could see the effect of the change on the preview. Predefined classes for Android applications result in less time spent creating a user interface and more time focused on the core of the problem; such as writing the modules for calibrating the camera, detecting distances and the State Machine.

A requirement for the application is to be able to track physical objects moving in space, preferably using some sort of marker, and estimate the distance between them. After learning more about ARCore and building some simple introductory applications, it became apparent that it was not possible to track markers in ARCore. Information regarding marker tracking was not easy to come by, and there seemed to be some applications on Stack Overflow where programmers managed to use ARCore to track a moving object but little insight or explanations was offered on how this was accomplished. Moreover, on ARCore's developer’s page, it is stated that tracking moving images was simply not possible and the images it could identify had to be quite large. The only thing that ARCore was capable of detecting were picture frames, which are not practical for use on smaller instruments. Therefore, tracking a marker like a QR-code was simply not possible. Note: At the time of access this was the case, the new version of ARCore (1.9) is able to track moving images such as advertisements on the streets. Still not useful for this application [17].
At this point, given the fact that there was no built in system for tracking moving physical objects and developing such a system would be time consuming, it became clear that ARCore had to be changed for another library or tool. It was initially unclear which tool/library could be used instead of ARCore so further research was necessary. After scouring the web for potential solutions, several candidates that showed promise appeared, at least, judging from the basic descriptions in their documentations. Examples of candidates were: Wikitude, Hololens, ARKit, OpenCV and AR.js; perhaps other which were not considered at all.

The most important first step was to determine which of the previously mentioned candidates was able to track a relatively small marker in real-time. It was preferable if it could be used in Android development as well. Out of the aforementioned candidates, the most sensible seemed to be OpenCV, more specifically its Java library. (The specifics of why OpenCV is a good candidate have been discussed in Section 4.2.1.) The most interesting aspect of OpenCV being its dedicated marker library called Aruco, which is extremely useful for this thesis work.

The process of creating an application that used OpenCV in Android was difficult nonetheless. The first challenge was trying to link a native library to the application being written in Java on Android Studio, a process which was unfamiliar and with too many “moving parts” to learn. A conceptual example of this is as follows: Assume one has to learn/un-derstand $B$ in order to implement $A$, but $B$ requires understanding of $C$ which in turn requires understanding of $D$. This chain of “knowledge dependencies” was deemed too lengthy and containing too many topics which would have prolonged the work considerably. The difficulty was compounded by the lack of supporting material that could be found.

An unexpected turn was the fact that the Aruco library, which was the most important library of OpenCV for this thesis work, was not available in the OpenCV-core library but in a separate library called OpenCV-contrib. So far, it was the core library that had been successfully integrated with the Android Studio project, it did not however contain the Aruco suite; finding and integrating an OpenCV-contrib library would require further troubleshooting and time. Initial attempts to link both libraries were not successful so it had not yet been possible to track markers. It was not until an alternative method to integrate OpenCV to an Android project was found that it was possible to use a Java version of the OpenCV contributions library.
After integrating the OpenCV libraries with the Android Studio project by linking to Quickbirdstudios’ Github repository through Gradle, it was possible to use the Aruco suite in Android \[18\]. This meant that a prototype app could be made to detect Aruco markers with an Android device. However, when trying to estimate distances between two markers it became apparent that one needed to first perform a camera calibration in order to get the intrinsic parameters of the camera. (The PCM shown in Section 4.1). The problem was that the calibration process was rather complex and most resources focused on how to calibrate using the C++ version of the library. There were much less resources when it came to Java and Android, so the only option was to move forward through trial and error. Furthermore, a lot of time had already been spent troubleshooting the problems that had arisen thus far in the development process. The extra overhead caused by the fact that all of this was being written as part of an Android App created another cascade effect that would require too much time to solve. It was then decided to abandon the attempt at implementing this application in Android.

An alternative solution had to be found to develop an application that fulfilled the design and requirements that have been outlined in previous sections. Hololens seemed like a good alternative. The drawback with Hololens was the fact that one would need a very expensive piece of hardware, the Hololens itself, with a cost of around $ 3000 \[19\]. It also required the use of C# in combination with the Unity graphical engine, this would require extra learning time to familiarize oneself with the tools and libraries. The IT institution in Uppsala had a Hololens which could be lent out but one of the responsible people stated that: “It is not guaranteed that we can.” Hololens would nonetheless, be the only feasible option if nothing else worked.

After discussion with the supervisor, it was decided to still use OpenCV but instead of attempting to develop an application for a mobile device, the application would simply be developed on a desktop computer using a web camera and making sure that the requirements were fulfilled. The programming language used would be C++, which is the main language OpenCV is developed in. Using OpenCV’s C++ library meant that one took away the complexity of trying to develop an Android app and could instead focus on the core of the problem. One would not need to deal with the overheads of understanding and debugging Android Studio or making sure that the program would work within the framework of a mobile application; focusing instead on the Computer Vision aspect.

As a result, the code and layout became much cleaner and there were many more resources and much more help to be found when working with OpenCV’s default implementation language, C++. This also meant that one had access to the latest version of OpenCV with potentially less bugs and more optimized as well.
5.1 Implementation of the software

The final implementation follows a similar pattern to that in Figure 7 with some changes made while fixing errors and optimizing.

5.1.1 Program Flow

![Updated Program Flow Diagram]

**Figure 10:** Updated Program Flow. The “built-in” refers to functions included in the Aruco/OpenCV library, while “implemented” refers to functions implemented for this thesis work.
The code runs sequentially and uses a frame (which is a Mat object) as a central component. Every loop iteration begins with the program capturing an image from the camera, writing this information to the Mat frame, and passing to different functions a handle to the frame object. The first two functions in the loop in Figure 10 extract information from the frame. The second two functions process the information gathered and communicate with other modules such as the State Machine and Instrument objects. Finally, in the fifth step, drawDetectedMarkers() and renderTextOnFrame(), information is overlaid onto the frame and returned to the display window for the user to see. The loop then begins again, and the frame object is overwritten with information from a new image captured by the camera.

**detectMarkers()** looks for Aruco markers that might be present in the captured frame; one needs to choose a dictionary of Aruco markers to use for this function, markers outside of the chosen dictionary will not be detected. This function also stores the 2D-coordinates of each of the marker’s corners, i.e. the pixel values on the display window after the 3D-coordinates have been projected onto the screen.

**estimatePoseSingleMarkers()** calculates the pose of each of the detected markers and produces the extrinsic parameters associated with each marker.

**storeMarkersMap()** is a function that cycles through the integer vector containing the detected Aruco markers’ IDs, if the marker is present in the instrument list already, then its rotation and translation vectors are updated. If a new marker has been detected, then a new object is created and added to the list of detected instruments.
void storeMarkersMap(map<int, instrumentData*>, markerMap, Mat& frame, vector<int> markerIds, vector<int> acceptableInstruments, vector<Vec3d> rotationVectors, vector<Vec3d> translationVectors, Mat cameraMatrix, Mat distanceCoefficients)
{
    for (int i = 0; i < markerIds.size(); i++)
    {
        int key = markerIds[i];

        if (find(acceptableInstruments.begin(), acceptableInstruments.end(), markerIds[i]) != acceptableInstruments.end())
        {
            if (markerMap->count(markerIds[i]) > 0)
            {
                (*markerMap)[key]->rvec = rotationVectors[i];
                (*markerMap)[key]->tvec = translationVectors[i];
                (*markerMap)[key]->instrument->threeDimCoordinates = translationVectors[i];
                (*markerMap)[key]->instrument->rotationVec = rotationVectors[i];
            }
            else
            {
                instrumentData* newInst = new instrumentData{
                    new Instrument(markerIds[i], translationVectors[i], cameraMatrix, distanceCoefficients),
                    rotationVectors[i], translationVectors[i]};
                markerMap->insert(map<int, instrumentData*>::value_type(key, newInst));
                newInst->instrument->threeDimCoordinates = translationVectors[i];
            }
        }
    }
}

Listing 1: New markers are stored and already existing markers are updated.
checkProximity() does several things. It firstly uses the Loop instrument as a starting point and iterates through the list of instruments that exist in the program instance. (The variable loop in line 3 of Listing 2 is a pointer to the Instrument object with ID 0.) At every iteration, the method instrument->madeContact(iterInstrument) will be called, and after True is returned five times in a row, the function will then call the Loop’s react(Instrument* target) method, which takes in as an argument a pointer to the target Instrument. This method will create an event, depending on the instrument it reacts with, that is sent to the Protocol object. For most instruments, the coordinates used to check proximity are the same as the center of the marker, except for the Loop which has a virtual point that would correspond to the actual location of the physical tip of the loop.

Note also that there is a pointer stowP. This is an Instrument object with corresponds to a marker placed statically on the table. The purpose of a “Stow Point” is to be a trigger that, upon contact with a Petri dish, will tell the State Machine that the Petri dish has been put aside.

```c++
void checkProximity(map<int, instrumentData*> instrumentsMap, Protocol& protocol)
{
    instrumentData* loop;
    instrumentData* stowP;
    instrumentData* target;

    loop = (*instrumentsMap)[0];
    stowP = (*instrumentsMap)[11];
    loop->instrument->createPointOfLoop();

    for (auto const& inst : (*instrumentsMap))
    {
        if (inst.first != 0 && inst.first != 11)
            target = inst.second;

        if (loop->instrument->madeContact(target->instrument))
            {*(target->counter) += 1;
             if ({(target->counter) > 6 && target->instrument->hasDisengaged ==
                 true})
                 loop->instrument->react(target->instrument, protocol);

            }
        else if (stowP->instrument->stowContact(target->instrument))
            { stowP->instrument->stowPointReact(target->instrument, protocol);
        }
        else
            {*(target->counter) = 0;
        }
    }
}
```

Listing 2: Check Proximity function.
The application will start in the Clean State by default. As is visible in Figure 1.

5.1.2 Program Modules

Apart from the main file which contains the main() function and the definitions to the functions inside. There are two more classes that play a central role in the running of the process: the Instrument class and the Protocol class.

Instrument

```cpp
class Instrument {
public:
  int arucoId;
  cv::Point coordinates; // where on the screen the marker is located
  cv::Point3d threeDimCoordinates;
  cv::Point3d loopTip;
  cv::Point3d flameTip;
  cv::Vec3d rotationVec;
  cv::Vec3d translationVec;
  bool hasDisengaged = true;

  enum instrType {
    LOOP = 0,
    EPENDORPH = 1,
    BURNER = 2,
    PETRI = 3,
    STOW = 4
  };
  instrType iType;

  void react(Instrument* target, Protocol protocol);
  void stowPointReact(Instrument* target, Protocol protocol);
  bool madeContact(Instrument* instA);
  bool stowContact(Instrument* instA);
  void assignType(int id);
  void createPointOfLoop();

  Instrument(int id, cv::Vec3d markerCenterCoord, cv::Mat camMat, cv::Mat distCoeff);
  ~Instrument();

private:
  cv::Mat cameraMatrix;
  cv::Mat distanceCoefficients;
};
```

Listing 3: Instrument class definition.
Instead of having several sub-classes for the different instruments, only one Instrument class was used and the physical instrument it represents would be denoted by a member of type an `enum` as defined in line 13 of Listing 3. The Instrument class contains all the methods that could be used by any of the different Instrument types.

Even though the camera matrix and distance coefficients are static, having them stored as members in each instrument makes it easier to access these values which are, after all, constant for the duration of the process.

The tip of the loop is calculated by using the rotation vector of the instrument. The rotation vector is multiplied with a scalar; the length from the center of the marker to the tip of the Inoculation Loop. And the resulting vector is added to the 3D coordinates of the marker’s center point in space; thus creating a virtual point that correctly corresponds to the location of the physical tip of the loop.

Create Point of Loop

```cpp
void Instrument::createPointOfLoop()
{
    Point3d centerPoint3d = this->threeDimCoordinates;
    Point3d tipOfLoop = centerPoint3d;

    // Projection of the point where the tip ought to be.
    Mat rotMat;
    Rodrigues(this->rotationVec, rotMat);
    Mat rotMatTpose = rotMat.t();
    double* tmp = rotMatTpose.ptr<double>(0);
    Point3d prolongPoint(tmp[0] * 0.108, tmp[1] * 0.108, tmp[2] * 0.108);
    tipOfLoop += prolongPoint;

    // End of point projection code
    this->loopTip = tipOfLoop;
}
```

Listing 4: Method that creates a coordinate for the Inoculation Loop.

The result can be visually displayed as in Figure 11.
Figure 11: The yellow line is mainly for the visual effect of having something connecting the dot at the tip to some sort of base. The virtual point is in the center of the red circle, far on the left side of the picture.
The code below shows the `React(...)` method that is called when the loop has made contact with an instrument.

```cpp
void Instrument::react(Instrument* target, Protocol protocol)
{
    if (target->hasDisengaged == true)
    {
        if (target->iType == EPENDORPH)
        {
            cout << "Reacted with Ependorph tube" << endl;
            protocol.dispatch(LoopDippedInVial());
            target->hasDisengaged = false;
        }
        else if (target->iType == BURNER)
        {
            cout << "Reacted with Bunsen Burner" << endl;
            protocol.dispatch(LoopSterilize());
            target->hasDisengaged = false;
        }
        else if (target->iType == PETRI)
        {
            cout << "Reacted with Petri dish" << endl;
            protocol.dispatch(Streak());
            target->hasDisengaged = false;
        }
    }
}
```

Listing 5: React method.

The method `stowPointReact(...)` on line 24 in Listing 3 works in a similar way to the previously shown `react()` method with he main difference being that this method only sends one type of event; the special case that occurs when a Petri dish has been placed on the “Stow Point.”
Protocol The Protocol class as mentioned before inherits from the `tinyfsm.hpp` module, more specifically, it has to inherit from the FSM class..

```cpp
class Protocol : public tinyfsm::Fsm<Protocol> {

public:
    void react(tinyfsm::Event const &) { std::cout << "Stay here" << std::endl; }

    virtual void react(LoopDippedInVial const &);
    virtual void react(Streak const &);
    virtual void react(LoopSterilize const &);
    virtual void react(Stow const &);
    void react(Soil const &); // This last one is shared by all states, so it's not virtual.
    virtual string myState();

    Protocol();

    virtual void entry(void) {}
    void exit(void) {};
};
```

Listing 6: The Protocol class.

Note that there are several overloads to the `react(...)` method, depending on the type of event that is sent as an argument. Below is the definition of the Event type.

```cpp
struct LabEvent : tinyfsm::Event {
    int id;
};
```

Listing 7: Event struct.

Just like for the State Machine, the Event class needs to be inherited from `tinyfsm.hpp`. Different predefined `LabEvents` are created depending on the actions done by the user. These events are dispatched from the `react()` method in the Instrument object to the Protocol object, where the state is changed depending the current state of the SM and the event received.
In the file Protocol.cpp we find the definitions for different State classes. An example of such a state-definition is below, for the *Clean* state:

```cpp
class Clean : public Protocol {

    void react(LoopDippedInVial const & e) override {
        transit<PreStreak>();
    }

    void react(Streak const & e) override {
        transit<Spoiled>();
    }

    string myState() {
        return "Clean";
    }
};
```

Listing 8: Definition of the SM-state “Clean” and the state-transitions it allows.

5.1.3 Errors that arose during development and their solutions

Below are some of the most prominent problems that arose during the development process.

**Inaccurate Z-axis:** In the first trial, when drawing a set of coordinate axes on the markers, the z-axis line was drawn starting in the center of the marker and reaching for the upper left corner of the screen. The problem seemed to have been the calibration, according to the OpenCV documentation, one needs to feed a good amount of “z-axis information” to the program in order to get a good calibration result; this means that one needs to provide the calibration algorithm with many images at varying angles so that it can produce a more accurate camera matrix and distance coefficients [9].

After doing so the first time, the z-axis vector pointed in the general correct direction but it was not precise and it seemed to “wobble” with every frame. After several calibration attempts, the precision of the axis increased making for better pose estimations; the final result looked like in Figure 1.

**Detecting contact:** To determine when contact has occurred between two instruments, a simple distance threshold will be used; when instrument A is within a distance of less than 1cm from B, then the objects are considered to have come into contact with one another.
To estimate the distance one simply calculate the magnitude of the vector between the two points. One first needs the 3D coordinates for each of the instruments’ points involved. These coordinates are obtained from the translation vector associated with the marker which is generated by the `estimatePoseSingleMarkers(...)` function. A translation vector contains three elements corresponding to the \( x \), \( y \) and \( z \) axis, therefore, using the translation vectors of the two instruments one can find the euclidean distance between them.

While testing the contact detection it became apparent that the distortions experienced in each frame could cause problems. Since for every frame, the detected points are not in exactly the same place and sometimes, though rarely, they might be significantly distant from the actual position of the marker. This could sometimes trigger an erroneous contact event despite the two objects being too far away from each other. To solve this, a counter was implemented which meant that the contact needs to be registered several times, say five times, before a trigger is activated; activating this trigger also resets the counter. Not getting a consecutive contact will also reset the counter. This method produced much more predictable behaviour from the program. There might have been a problem of lagging the process, but given a rate of 20 frames per second and requiring five frames to register an event, this would translate into a latency of around 0.2 seconds which is simply “barely noticeable.”

**State Machine:** There was the option of implementing an own State Machine, not being very familiar with the design principals that make a good state machine however, it was decided after consulting with the supervisor, to use a general library that already contains a state machine implementation. This would reduce the time requirement of testing that the SM implementation was correct.

The library used is called tinyFSM [14]. It is a simple State Machine that uses class templates [20] allowing for the user to implement and use custom classes. This library provides the essentials needed, that is, enforcing transition conditions and providing transition functions.

A preliminary function was created, `void simulatingStateMachine()` where the functionality of the SM was tested using keyboard inputs to generate events that would later be generated by the live application.

**Description of scanning for instruments:** To increase the speed of the scanning process it was thought to have each element in the list calculate the distance between itself and the remaining instruments ahead of itself in the vector, and *not* those that have already been previously scanned.
This later gave way to simply scanning the distance between the loop and all other instruments in the list, since it is only the loop that really comes into contact with other instruments. There is no practical way an Ependorph tube would get close to a Bunsen burner, if someone were to do such a thing then they shouldn’t be in the laboratory in the first place. The complexity of scanning instruments is therefore $O(n)$.

**Dealing with phantom objects:** The function `detectMarkers()` searches for Aruco markers in the current frame. Sometimes the algorithm might confuse certain distortions in the image as being a marker, like having a keyboard nearby whose letters might resemble a marker for example. This then triggers the creation of an instrument object that gets stored in the Map containing the other instruments.

A solution for handling this is to create a vector of `int`'s that represent the IDs of the instruments that are allowed to exist in a program instance. After a marker is detected, the program iterates through `std::vector<int> allowedMarkerIds` and if the ID is not found in the vector then it is not created. This proved to be an effective solution that eliminated the problem of having the Map of instruments get populated with “ghost” objects.

**An action spans several frames:** One can have a simple idea: “The first time the inoculation loop comes into contact with the Petri dish, it is an accepted action. If the loop comes into contact with the Petri dish again, before having been sterilized, then one will have spoiled the protocol.” This seems like a simple idea conceptually, but the reality of implementing this on a computer is different.

Say that on frame $x$, the tip of the inoculation loop has made contact with the Petri dish, by getting below the distance threshold that triggers an event, thus changing the State of the protocol. So far so good, but keep in mind that conceptually, if one touches the the Petri dish *again* without first sterilizing, then the process is spoiled. In the next frame $x + 1$, which is captured approximately $1/20th$ of a second later, the inoculation loop tip is still within event-triggering range of the Petri dish, thus provoking the creation of a new event. The program therefore interprets this as the loop making contact with the Petri dish *again* and since the SM is in a new state from when it was in frame $x$, this new event will cause the SM to switch states to *Spoiled*.

The solution to this was to create a new condition before the program calls the `react()` function, this was a Bool variable called `hasDisengaged`. This variable starts out as `true` and is toggled upon contact with another instrument. While `hasDisengaged` is `false`, a new event will not be sent even if the loop is still making contact with the dish. `hasDisengaged` is toggled back to `true` once the distance between the inoculation loop and the Petri dish is greater than 10cm, thus resulting in a correct and intuitive behaviour for the program since one action, such as streaking, can span over several frames.
Using a Hash Map Instead Of Vectors

The first implementation used vectors to store marker IDs, Instrument pointers, translation vectors, and rotation vectors. To get the instrument with Aruco marker ID of 10, one would need to find the index where this ID is stored at in the `markerId` vector. Once the value `i` for the index is obtained, then one would use the same `i` to find the associated Instrument pointer, translation vector, and rotation vector. In the code snippet below, the index `i` refers to the same instrument and its properties.

```cpp
markerId[i]
instruments[i]
tVectors[i]
rVectors[i]
```

The problem with this solution was that after implementing `allowedMarkerIds`, discussed in the previous section, there would sometimes be more markers than Instrument objects in their respective vectors, causing the information stored at the same index of different vectors to be mismatched. Trying to create a solution that would keep these vectors synchronized proved to be very difficult and error-prone. It was therefore preferable to store all data pertaining to one instrument in one `struct`. These structs, called `codeinstrumentData`, would be stored in a Hash Map (`std::map`) where the key for each key-value pair would be the marker ID.

```cpp
struct instrumentData
{
    Instrument* instrument;
    Vec3d rvec, tvec;
    int* counter = new int(0);
};
```

Listing 9: Instrument Data struct.

This eliminated all errors associated with index mismatch and null-pointer exceptions as well inaccuracies of distance estimations; which occurred when an instrument had a wrong translation and rotation vector associated to itself thus producing wrong coordinate values.
6 Results

The requirements listed in Section 3 outline the goals of this thesis work and the conditions that should be met in order to consider this a successful implementation. And regarding this aspect, it does. The application is able to track a protocol through its steps, warn the user when an action is not allowed. And since it is able to track the steps in a protocol then this implies that the program is able to track several markers in real time and estimate distances between them and react when some of the markers come into contact.

There are however, several important aspects to be discussed, since the answer to the question, “Can a program fulfill the requirements set out?” is a simple “yes”, the answer to “How well does the program fulfill these requirements?” is not as simple. This discussion of the results will look at its qualitative and quantitative aspects.

Firstly, at the beginning of this thesis, it was not clear whether it would be possible in any measure. Recall that this work did not begin with the prior knowledge that OpenCV was the best tool for this task, therefore a part of this thesis work was to find a suitable tool to create an Augmented Reality application with, one that was capable of fulfilling the requirements laid out in the Requirements section. The frameworks that were considered initially turned out to be unsuitable for developing this type of application. This was because they could not track moving objects or identify relatively small markers and attempting to develop a module that could do so “from scratch” would mean that even a simple prototype would have been outside the scope of this work. The first two to three weeks were therefore dedicated to finding a solution to the question “What language, what hardware, and what tools can be used?” It was during this time that it became apparent that OpenCV’s C++ library would fulfill all the requirements needed.

Therefore, given the premise that it was not certain that this application was possible with current technology, it can be considered a success the fact that it was indeed possible. Thus leaving the questions of “How good?”, “How usable?” and “How effective?” to be solved in the future. Nonetheless, below is a discussion of the qualitative and quantitative results of this thesis work.

6.1 Qualitative Results

A video from a camera does not show a continuous flow of an image but a discrete sequence of frames in rapid succession that create the illusion of movement. Therefore, the rate at which frames are shown (known as fps; standing for frames per second) should be such so that it seems fluid to the human eye. A desirable fps, even though there may be different opinions on this, is around 20; more fps will create a smoother view, but having a much slower rate will make the video “stutter”.
As mentioned before, the goals set out in the requirements were fulfilled and during the running of the program the fps is maintained at around 20, creating a smooth visual experience during the operation of the program. This has the effect that the user does not need to look at the real instruments and at the screen at the same time and can instead solely look at the video being streamed while working.

On the other side, the camera has to be rather close to the workbench to be able to accurately read the instruments (Further discussion in 6.2) What this results in is that the user is not in a completely “relaxed” and “natural” position while working; there’s an imaginary box inside of which all actions need to happen, the frustum, otherwise one will be either outside the frame or too far away from the camera for the markers to be detected.

The size of the frustum may be reduced by a variety of factors. One of such example is lighting; too much light will make the dark areas of the marker reflect more light, thus reducing the contrast needed for the program to detect the markers. Too little light and the camera will increase its ISO making the images more distorted. The closeness required for the camera to work effectively meant that instruments were spaced closer together than they usually would in a real laboratory environment. Figure 12 shows an example using a strong source of coming from one point in the room, e.g. a large lamp in (a), and having several smaller light sources with more evenly distributed light in (b).

![Images showing different light sources](image)

(a) Strong light source  
(b) Even light source

Figure 12: Images show the effects of different light sources on the marker.
6.2 Quantitative Results

Note that all the tests were carried out on a computer with the following specification and the following camera, results most likely will vary on different hardware.

- Processor: Intel(R) Core(TM) i5-4690K 3.50GHz
- RAM: 8.00 GB
- Graphics Card: NVIDIA GeForce GTX 1080
- OS: Windows 10 Home
- Camera: HP 4310 Webcam

6.2.1 Lighting

It is hard to say quantitatively how much lighting is too much or too little without a luxmeter. There are nonetheless several observations made while testing.

1. Having a gleaming surface distorts the detection, therefore, a matte surface is better. Also, having a single light source in the room can create a bright spot on the workbench that worsens performance, as is shown in Figure 12.

2. Being next to a window with sunlight in will also harm detection, since the excessive brightness will reduce contrast between the dark and light parts of the marker.

6.2.2 FPS Performance With Multiple Markers

How many instruments can be tracked before the fps dips too low? The program was able to track 8 markers without going below 21.5 fps. This is a satisfactory result in the context of this thesis-work since there is never a moment where more than 8 markers are present on the screen at the same time. Not all Petri dishes for example need to be visible only the one that the user is working with at the moment.
6.2.3 Effective Range

From Figure 13, one can see that after 60 cm the effectiveness of marker detection starts to suffer. The percentage is calculated based on the number of frames recorded per second and in how many of these frames the marker was detected. For example, if the marker was detected in 19 frames out of 20 in a second then the detection rate is 95 % in that second.

![Distance(cm) vs. Detection Rate](image)

Figure 13: Graph showing how the detection rate of a marker deteriorates when the distance is increased.

6.2.4 Area Covered

Given the maximum effective distance established in section 6.2.3, around 60 cm, and the field-of-view angle derived from the Camera Matrix values in Equation 2, the dimensions and volume of the frustum can be calculated using trigonometry. The volume of the frustum would therefore be approximately 26800 cm$^3$. The maximum area that can be covered by the camera when it is perpendicular to a surface would be approximately 1344 cm$^2$, more or less a square with sides of 36 cm.
6.2.5 Attempted Improvements

Thresholding did not seem to increase effectiveness for detection, at least not using the methods tried; binarizing the image at different thresholds. It turns out that the built-in detection algorithm `detectMarkers()` does thresholding on the input image. Preliminary testing showed no improvement in marker detection range when supplying the detection function with an already binarized image. However, `detectMarkers()` contains an input-parameter called `detectorParameter` which has a default value [13]. One can nonetheless change the values of the members for this class thereby modifying the thresholding that `detectMarkers()` does on the input frame. This possibility was discovered too late in the development process meaning that this thesis work was developed using the default values.
7 Conclusions

At the time of writing this essay, very similar examples of applications could not be found. There were however some examples of applications that used AR in the workplace such as the that by GE that was taken up in the introduction. The difference in this thesis work is that unique markers enable the application to correctly identify an instrument regardless of its position on the work area. This application does not require depth sensors to make distance and pose estimations of different instruments, instead relying only on images from a camera.

Given that distances between points in 3D space could be calculated, it was then possible to connect this to a State Machine programmed to transition based on the proximity of instruments. (Recall that events are triggered when instruments come into contact with each other, something that can be easily determined by calculating when the distance between them has gone below a certain threshold.)

Since the starting point of this application is the images captured by the camera, the same image can be displayed to the user with additional information overlaid onto it. Examples are the notification that an error has been made, drawing a square around the markers which tells the user that the marker has been identified, and text that tells the user the current state of the Protocol.

One can conclude that using Augmented Reality in work spaces shows promise, since the technology to develop AR applications like the one described in this thesis exists today, technology is therefore not a barrier. The way I see it, the success or failure of an AR lab assistants rests on the willingness of other companies to invest time and funds in its development, as well as on the result of a more in depth UX-testing which was beyond the scope of this thesis.

For the immediate future development of this application, apart from refining the software side and increasing its functionality, it is necessary to carry out UX-testing with these types of applications to know if they are user friendly enough to the people who would use them. It would need to be established if the users experience an increase in comfort while using AR in their daily work and if it objectively improves the efficiency of the work being done.

This application can be expanded by adding other hardware elements apart from the camera. For example, for a rack of Ependorph tubes, one could create a grid using laser trip sensors that will trigger when the tip of an instrument, such as a pipette or inoculation loop, is dipped into a tube in the rack. To make the program more proactive, one could implement algorithms that measure the speed and direction of an instrument’s movement and given the location of other instruments in the work space, it would be able to estimate...
whether two instruments, that shouldn’t come into contact, are on a collision course. If the two instruments were on a collision course, the application could proactively warn the user of the impending error, instead of simply warning after the error has been made. An interesting development would also be the ability to react to gestures; in the case of streaking bacteria for example, there is a specific hand movement that is carried out, and the program could react only when this gesture is detected.

One could use image recognition to identify certain instruments without the need of having a marker on them, especially those instruments that are not unique. An example could be Petri dishes, since one is working with one dish at a time and usually puts it away after the streaking is complete. The important thing is to not make a mistake with the Petri dish one is working with at the moment, the ones that have been stowed away are usually safe. I leave this to the judgment of biologists and chemists however, since they are the Domain Experts while I do not have extensive experience of working in a laboratory.

These are some of the improvements that can be suggested for this thesis work, there is however, always room for improvement in science, so there are potentially a hundred ways in which this prototype can be made better. The important part has been showing that it is possible and feasible to fulfill the fundamental requirements that are needed to create an AR application, the way is then open to endless developments.
8 Appendix

Since this thesis has been about AR, it is in order to show the reader how the program looks like when working. This content has been added in the appendix since it is hard to guarantee that these links will still exist in five or fifty years.

In the link below, a “trivial mistake” has been committed, the user has dipped the inoculation Loop in the bacterial solution, then sterilized the Inoculation Loop and dipped it again in the solution. This sequence of events does not compromise the experiment and has no effect on the outcome. One therefore simply goes from the PreStreak-state back to the Clean-state.

https://youtu.be/Xt8R97K2oZQ

The following example shows a full correct trial:
https://youtu.be/22bH06x-wQE

And this video shows that the application has detected an error while following the protocol:
https://youtu.be/11btttSolB4
References


[9] *Camera Calibration and 3D Reconstruction*. URL: https://docs.opencv.org/4.1.0/d9/d0c/group__calib3d.html


