
FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

# BACHELOR THESIS

Alexander Smirnov

## Hand Tracking for Mobile Virtual Reality

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Jakub Gemrot, Ph. D.

Study programme: Computer Science (IOIA)

Specialization: Computational linguistics

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date 22.07.2021 signature

Title: Hand Tracking for Mobile Virtual Reality

Author: Alexander Smirnov

Department / Institute: Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Jakub Gemrot, Ph. D.

Abstract: Mobile Virtual Reality inspired by Google Cardboard (where a smartphone is placed in a cheap headset with no electronics) has lost its popularity after standalone VR headsets appeared on the market. The devices like Oculus Quest support full tracking of the real-world position and rotation of the headset and a pair of controllers (or even just the user's hands), all without any external sensors or computers, while Cardboard-like applications only track the rotation of the user's head. We tried to create a framework that could provide a similar experience to the standalone headsets - a mobile VR application capable of tracking itself and the user's hands in 6 degrees of freedom, all while only requiring a smartphone with a few cheap additions. We used Unity AR Foundation to achieve headset tracking and a variety of OpenCV algorithms to solve hand tracking - ArUco markers, Color Thresholding, Camshift, and deep learning approaches like OpenPose and YOLOv3. Our focus was on testing the hand tracking algorithms, and we discovered that while they are far from perfect, the concept is feasible, and, with some improvements, the framework could become a real competitor in the space of standalone VR headsets.

Keywords: virtual reality, computer vision, mobile games, hand tracking

# Contents

# Introduction

Mobile Virtual Reality has been around for a while. It started back in 2014 with Google Cardboard - a cheap headset made almost entirely out of cardboard, designed to be used together with a smartphone. The users only needed to install an application of their choice, slide their device into the headset, and enjoy the VR experience.

This approach, however, only used the smartphone's gyroscope sensors, and therefore was only able to track the rotation of the user's head - the so-called 3 Degrees of Freedom tracking (3DoF).

Not long after the release of Google Cardboard, industry-leading VR companies like HTC, Oculus and Razer released a number of of their own headsets that were able to track not only the rotation, but also the real-world position of the headset and even a pair of controllers (6 Degrees of Freedom tracking, or 6DoF). This quickly made Google Cardboard with its 3DoF obsolete.

A few years later these headsets were upgraded in such a way that they no longer needed a separate computer to operate - all of the tracking in 6DoF was performed using only the onboard sensors and processors. Oculus Quest is an example of such a headset.

Seeing this made us think whether it was still possible to bring mobile VR back. Compared to the standalone headsets, modern smartphones have similar or even higher resolution, processing speed and sensors, so in combination with cheap Cardboard-like headsets they have the potential to be used as an alternative to standalone headsets.

This thesis presents a mobile VR application that is capable of tracking itself and the user's hands in 6DoF, describes the steps taken to build the application, and compares various hand tracking methods against each other.

# 1. Goals

The main goal of this thesis is to create a mobile VR application which will show that it is possible for mobile VR to provide a similar experience to the standalone commercial headsets, while being much cheaper. The application is not meant to be a finished product, but instead a proof of concept.

Additionally, we want to explore and compare various methods of hand tracking, benchmark their speed and accuracy, find their pros and cons and try to decide on the best one.

The following chapters include the steps we took to achieve these goals. The Analysis chapter outlines the thought process we went through when designing the app, the Theory chapter explains some of the advanced topics used in the thesis, the Software Setup chapter goes into detail of the implementation of the app, and the Experiments chapter describes the tests we ran and their outcomes. The following Discussion and Conclusion chapters sum up the gathered results.

## 2. Analysis

There were four main things we needed to replicate to achieve an experience similar to a standalone VR headset - integration with a Google Cardboard-like headset, full tracking of the position and rotation of the headset, hand tracking, and the virtual world for the user to interact with.

Intractable worlds are most often found in games, so we decided to use a game engine as the main tool for creating the target application. Out of many available game engines we ended up using Unity due to its simplicity and the number of integrations with various tools that could be used in the thesis. This solved the problem of building an intractable world.

The headsets like Google Cardboard all follow a similar concept - they have a place for the user's smartphone to slide in and two lenses placed between the user's eyes and the smartphone. A typical Cardboard application then displays the virtual world in a sort of a split screen - instead of using the full screen of the smartphone, the view of the world is split into two parts and displayed in two sides of the phone, providing a separate image for each eye. Capturing the two views from two different points offset like the human eyes provides depth to the virtual world. The Unity engine provided enough tools to implement this behavior.

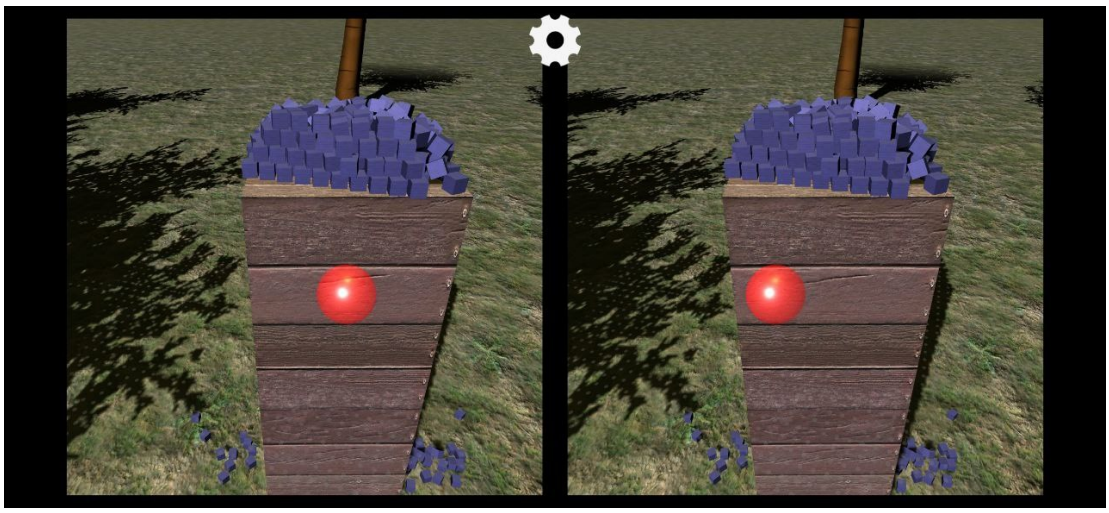

Figure 1. Example of a split screen view of the world.

The problem of headset tracking in 6 Degrees of Freedom has already been solved before, except for a different reason - Augmented Reality. In contrast with Virtual Reality, in Augmented Reality the camera feed from a handheld device is visible and enhanced with virtual objects. In order for these objects to appear fixed to the real world, the application needs to track the real-world position and rotation of the device and apply this information to the objects. This exact functionality is provided by a Unity framework called AR Foundation[1], and even though it is technically supposed to be used with Augmented Reality, it works with Virtual Reality just as well. This solved the problem of headset tracking.

Object tracking - calculating the coordinates of an object in an image - is a common problem in the area of Computer Vision. We decided that some of the object tracking algorithms would be applicable to the problem of hand tracking. In order not to code the algorithms from scratch, we used OpenCV[2] (Open Source Computer Vision Library) - a library that provides hundreds of useful tools for image processing and computer vision. For the purpose of this thesis, we chose four different methods for hand tracking:

- *ArUco*[3][4] - an algorithm for pose estimation of square markers. We chose it because it is fast, is able to detect the full pose (position and rotation) of a marker and is commonly used in computer vision for object tracking.
- *Color Thresholding* - an algorithm based on tracking regions of an image whose pixel values are within a specific color range. We chose it because it is one of the simplest and commonly known methods for tracking objects, and as such it could provide a baseline for other methods.
- *Camshift* - an algorithm that uses color probability distributions and histograms for object tracking. We chose it because it was suggested by another paper studying hand tracking - *A hidden Markov model based dynamic hand gesture recognition system using OpenCV*[7].
- *Deep Learning algorithms* - a set of algorithms based on neural networks. Deep learning has shown some great results in object detection, and we thought a detector trained to locate hands could be used for the purpose of hand tracking. For the purpose of this thesis we chose OpenPose, YOLOv3 and YOLOv3Tiny models, mostly due to their integrations with Unity and previous experience with them.

The algorithms we chose are based on different object detection concepts, and so would require different actions from the user. The ArUco detector works with markers, so, in order for it to work, the user would need to attach the markers to their hands. The Color Thresholding and Camshift algorithms operate on colors and perform best if the tracked object's color is not present in the background. Since the average hand color is quite commonly present in the surrounding environment, we thought the performance of these algorithms could be improved if the user were to wear bright gloves. The simplest choice would be to use blue surgical gloves, as they are easy to get and not expensive, therefore still achieving the goal of keeping the mobile VR setup cheap. Finally, the deep learning approaches are designed to work with the images of hands themselves, so they require no additional work from the user.

To sum up, the various chosen algorithms differ in terms of user experience - some of them requiring more setup than the others - but they all preserve the low cost of using the application. The following chapters will go over the details of these algorithms, the implementation of the application, the experiments we ran on the hand tracking methods and the discussion where we tried to find the best method in terms of speed, accuracy and user experience.

# 3. Theory

This chapter gives a brief introduction into the algorithms used for hand tracking by this thesis, as well as some concepts related to them.

## 3.1. ArUco

ArUco[3][4] is an open-source library for camera pose estimation using squared markers, as well as pose estimation of the markers themselves.

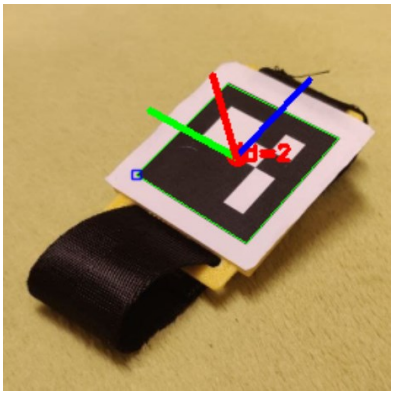

Figure 1. Estimated pose of an ArUco marker.

The library is able to detect specific square markers in an image and estimate their pose, providing their position and orientation in physical space. One of the hand tracking methods in this thesis uses these markers attached to the user's hands.

## 3.2. HSV and L$\alpha\beta$ Color Spaces

HSV (for hue, saturation, value) is color space designed to more closely align with the way human vision perceives color-making attributes. As stated in Wikipedia[5], "in this model, colors of each hue are arranged in a radial slice, around a central axis of neutral colors which ranges from black at the bottom to white at the top. The HSV representation models how colors appear under light".

L$\alpha\beta$[6] is a color space designed to be perceptually uniform - a given numerical change corresponds to a similar perceived change in color. It expresses color as three values: L for perceptual lightness, and $\alpha$ and $\beta$ for the four unique colors of human vision: red, green, blue and yellow.

These color spaces are used by the *Color Thresholding* and *Camshift* algorithms.

## 3.3. Color Thresholding

One of the hand tracking methods in this thesis uses Color Thresholding to locate hands in a given image. The concept is simple - given a color range, the algorithm finds all pixels in the image that are within this range.

Since colors in the RGB color space are coded using the three channels, it is more difficult to segment an object in the image based on its color, so this method works best with other color spaces like HSV or L$\alpha\beta$. HSV is most commonly used with Color Thresholding, while L$\alpha\beta$ was suggested in *A hidden Markov model based dynamic hand gesture recognition system using OpenCV*[7] since "the average skin mainly comprises a ratio of red and yellow, and in case of the L$\alpha\beta$ color space, the $\alpha$ component represents the pixel components position between red and green while the $\beta$ component represents the position between yellow and blue, making it less vulnerable to noise".

In this thesis the algorithm is used to find and track user's hands in the camera feed based on the sampled average hand color range.

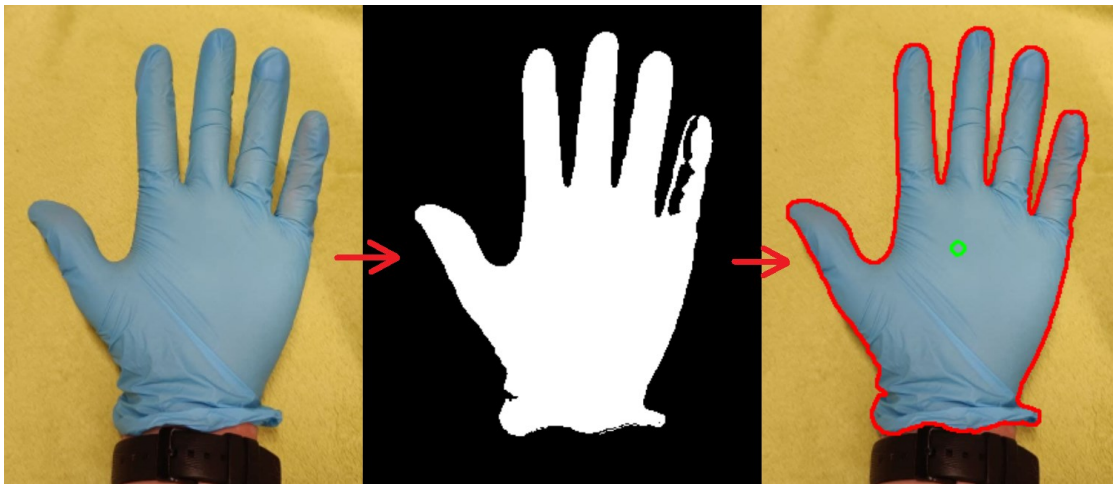

Figure 2. Color Thresholding example. Left - original image. Center - a mask showing only the colors in the range corresponding to the glove color. Right - contour of the masked pixels with its center point.

## 3.4. Camshift algorithm

As described in *Computer vision face tracking for use in a perceptual user interface*[8], the Camshift algorithm "is based on a robust nonparametric technique for climbing density gradients to find the mode (peak) of probability distributions called the mean shift algorithm[9]. The mean shift algorithm is modified to deal with dynamically changing color probability distributions derived from video frame sequences. The modified algorithm is called the Continuously Adaptive Mean Shift (CAMSHIFT) algorithm".

*Fast and Robust CAMShift Tracking*[10] sums the algorithm up as "CAMShift essentially climbs a gradient of a back-projected probability distribution computed from re-scaled color histograms to find the nearest peak with an axis-aligned search window. With this, the mean location of a target object is found by computing zeroth, first and second order image moments. The position and dimensions of the search window are updated iteratively until convergence".

This algorithm is used in the thesis for the purpose of hand tracking based on the initial hand position.

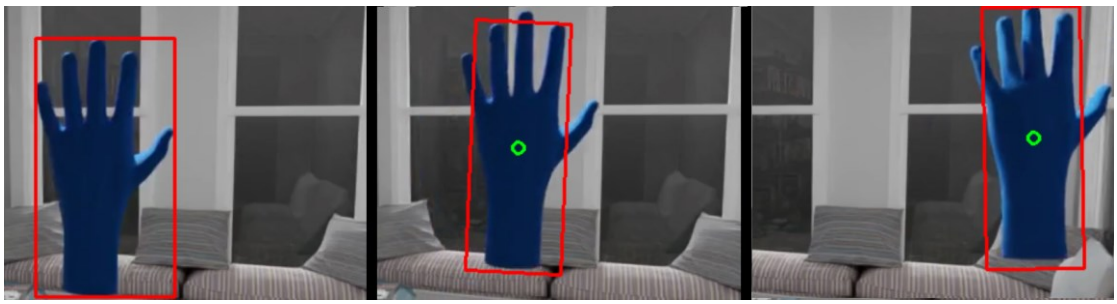

Figure 3. Camshift example. Left - initial hand bounding box. Center and Right - detected bounding boxes after a few frames.

## 3.5. OpenPose Neural Network

OpenPose[11][12][13][14] is an open-source neural network designed for 2D pose detection, including body, food, and facial key points.

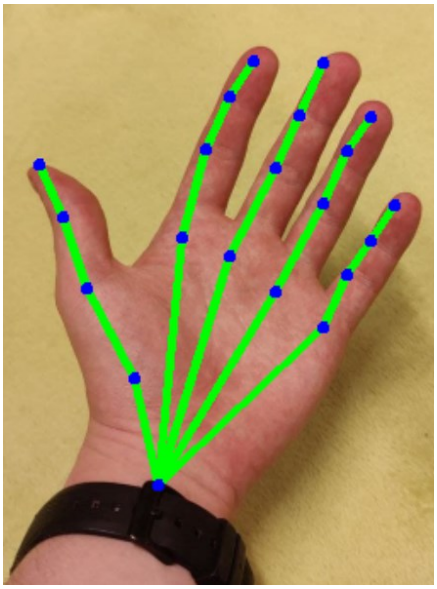

Figure 4. Example of a prediction made by OpenPose.

It is capable of detecting the hand pose in an image, providing the positions of 21 key points of the hand. This information can be used to match a full virtual hand skeleton to a detected physical hand.

## 3.6. YOLOv3 Neural Network

You Only Look Once[15] (YOLO) is a real-time neural network designed for object detection. YOLO is capable of finding the bounding boxes of specific objects in an image. It is one of the fastest object detection networks, outperforming similar models like Faster R-CNN[18], SSD MobileNet[19] or RetinaNet[20].

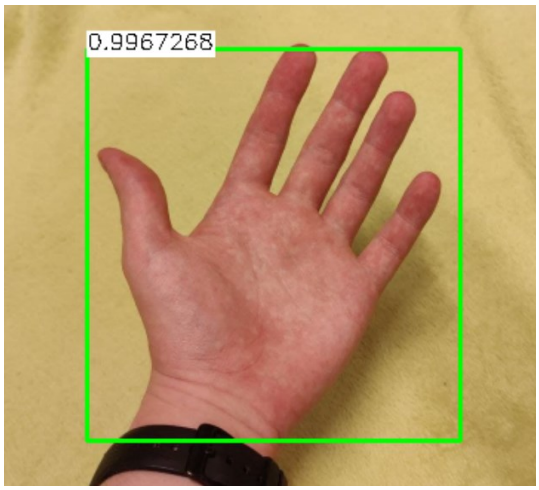


Figure 5. Example of a prediction made by YOLOv3.

YOLOv3 is an improved version of the older YOLO networks. YOLOv3Tiny is a smaller version of YOLOv3 that is much smaller and faster, but less accurate. The network allows for real-time hand detection without requiring the user to wear any additional markers.

# 4. Software Setup

This chapter describes the software and tools used in the development of the VR application, as well as code structure and implementations of used algorithms.

## 4.1. Development Environment, Libraries and Packages

The application was developed on Windows using Unity 2020.3.2f1. We also used several packages from the Unity Package Manager to aid with the development. Most of them are minor packages with textures or in-game objects used to create the demo world, and they will be listed in the Appendix. The two important packages related to the headset and hand tracking are:

- *AR Foundation*[1] *(version 4.0.12)* - used for headset tracking.
- *OpenCVForUnity*[17] *(version 2.4.3)* - provided an interface that allowed us to use OpenCV inside the application. Primarily used for hand tracking.

## 4.2. Cross-platform Development

While the application is designed to only work with mobile platforms, being able to test it on desktop while developing is beneficial. We use a few approaches to enable that behavior - the application runs as expected on mobile, and on desktop it removes several mobile-specific features (like headset tracking) and replaces them with similar desktop-specific features instead (like a keyboard and mouse movement controller).

The main tool we use in order to achieve this is a custom script - *PlatformVisibilityManager.cs*. When an object with this script attached starts its lifetime, the script checks the platform the application is running on and disables the object if the platform is different from the one set in the script configuration.

## 4.3. Application Structure

All of the code responsible for making the application work is attached to Unity GameObjects inside several Scenes. The project has four scenes:

- *StartMenu* is the first scene that is visible when the application is started. It allows the user to select the hand tracker they want to use and choose which of the following two scenes (*Main* or *MainEmpty*) to load next.
- *Main* is the main scene of the application. It contains all of the objects responsible for headset tracking, hand tracking and the virtual world. The core of this project is located in this scene.
- *MainEmpty* is an exact copy of the *Main* scene, with one difference - the virtual world is replaced with an empty world. This scene is primarily used for benchmarking the speed of the hand tracking algorithms.
- *MainExperiment* is a scene used for testing the accuracy of the hand trackers. It is similar to the *Main* and *MainEmpty* scenes, except it does not have a virtual world and any movement controllers, and runs the hand trackers on a prerecorded video instead of the camera feed. More information about this scene can be found in the Experiments and Appendix chapters.

The *StartMenu* scene mostly contains objects related to the User Interface (UI) - a dropdown field for the user to select the hand tracker they want to use, two buttons for loading the main scenes, and some text explaining how to use the application. There is, however, one important object that is related to the core of the application - *GameManager*. It is a container for the *GameManager.cs* script, and it is the only object that is persistent between scenes - that is, it is not removed when an active scene changes. The Game Manager is a source of truth containing the information about which hand tracker should be used, whether the debug information should be displayed, whether the application should use split screen, etc. It also contains the state of the application which takes one of the four values:

- *Menu* - when active, the starting menu is visible.
- *ColorPicker* - when active, the user can tap their screen to sample the range of colors at that point, required by some of the hand tracking algorithms.
- *WorldInstantiation* - when active, tapping on a detected AR plane will spawn the virtual world at that position.

- *Main* - when active, the virtual world is active and interactive.

As mentioned before, the core of the application is located in the *Main* scene (and, similarly, in the *MainEmpty* and *MainExperiment* scenes). The following objects are contained in the scene:

- *UICanvas* is an element that overlays the screen and contains all elements related to the UI - FPS (Frames Per Second) monitor, the Settings screen (described later), and the controls for moving forward from the Color Picker Stage.
- *AR* is a container element for everything related to Augmented Reality and headset tracking. It contains all of the standard elements provided by AR Foundation like an AR Camera (an element that follows the real-world position and rotation of the user's device) or an AR Plane Manager (responsible for finding surfaces in the real world based on the camera feed), as well as some custom objects like the Raycast World Instantiator (used for spawning a virtual world anchored to a point in the real world. Described in detail later). This element is only active on mobile platforms.
- *DesktopDebug* is a container element for objects used during the development and debugging of the application, and is only active on desktop platforms. It contains a Camera used to capture and display the virtual world, as well as a simple keyboard and mouse movement controller that allows moving the camera around in the virtual world.
- *CameraMatProviders* is a container for objects that are responsible for acquiring and providing an OpenCV Matrix containing the latest image captured by the device's camera. It contains two objects with the same functionality but for different platforms - one for mobile and one for desktop. The implementation of the providers is described in the next subchapter.
- *HandPositionEstimator* is an object containing one of the core scripts of the game - *HandPositionEstimator.cs*. It is responsible for running the hand tracker selected by the user on the latest camera frame and applying the detected hand position to the in-game objects used to represent hands. This script and the hand trackers themselves are explained in detail in the next subchapter. Additionally, this element contains the canvas used to display the camera feed with the output of the hand trackers, and a Color Picker element

used to preview the sampled color range required by some of the trackers (more information on the Color Picker follows in the next subchapter).

- *Splitscreen* is a container for the elements used to create the split screen view of the virtual world in order for the application to be usable in a Cardboard-like headset. Each of the cameras in the scene (the *AR Camera* and the *DesktopDebug Camera*) have two offset cameras attached to them, each representing an eye. The output from these cameras is then sent to the Image elements in this container, one for each eye as well. These images are aligned to the sides of the screen, in line with the lenses of a Cardboard-like device. Since mobile and desktop platforms use different cameras, this object also contains two separate image pairs only active in the respective platforms.
- *EvenSystem* is a standard Unity object responsible for listening to input events like touching the screen.
- *HandObjects* is a container for the in-game physics objects used to represent hands. *HandPositionEstimator* uses the hand positions detected by the hand trackers to apply them to these objects. These objects can physically interact with the virtual world.
- *World* is an object containing the virtual world, the actual place the user will see. It is only active on desktop, but an instance of the same world is created on mobile using the *RaycastWorldInstantiator*.

## 4.4. Code Structure

All of the custom code for this project is located in the *Assets/Scripts* directory. It contains three folders:

- *Experiments* is a folder containing scripts used for running the experiments. They are described in the Experiments chapter.
- *World* is a folder containing scripts used by the virtual world. It is not related to the core of the project and only contains scripts used in the demo world.
- *Internal* is a folder containing the core scripts used to make the application function - hand trackers, UI managers and various utils.

As mentioned above, the main scripts of the application are located in the *Internal* folder. Its structure is as follows:

- *UI* is a folder containing scripts for managing the User Interface. The scripts are simple and mostly used to forward the UI events to the Game Manager (for example, calling the *SetIsDebug* method on the *GameManager* when the debug toggle is changed in the settings screen).
- *Debug* is a folder containing scripts required to run the application on desktop. It is the parent folder for the mouse and keyboard movement controller script used by the desktop camera.
- *Utils* is a folder containing various custom utility functions. They are described in detail later.
- *CameraMatProviders* is a folder containing the scripts used to acquire the latest frame captured by the device's camera. They are described in detail later.
- *HandTrackers* is a folder containing the scripts for all of the hand tracking algorithms. Once again, a more detailed description is presented later.
- *GameManager.cs* is a file containing the *GameManager* class described in the previous chapter.
- *HandPositionEstimator.cs* is a file containing the *HandPositionEstimator* class. As described before, this class is responsible for using the data gathered by the hand trackers and applying it to the in-game objects representing hands. More detailed description follows.

### 4.4.1. *Utils*

This folder contains various utility functions that are used by the other scripts:

- *ColorUtils.cs* contains functions used by the Color Picker for getting a color range from a circle centered on a point in a camera frame, as well as the *ColorRange* structure to hold these values.
- *FpsMonitor.cs* is a script that calculates the current Frames Per Second that the application is running at and the average Frames Per Second over the first minute of running the app. It is also responsible for displaying this information on the screen. This script is used in the Experiments chapter when benchmarking the speed of the hand tracking algorithms.

- *HandTransform.cs* contains a structure used to hold the position, rotation and scale of a hand in the coordinate space of the virtual world. The hand trackers use this structure when returning the detected hand positions.
- *PlatformVisibilityManager.cs* is the script that is responsible for enabling and disabling objects based on the platform the application is running at, as described before.
- *RaycastWorldInstantiator.cs* is a script that allows the user to spawn the virtual world anchored to a point in the real world. During the *WorldInstantiation* stage this script listens for touches and, when one is detected, it sends a raycast from the touched point in the direction the camera is facing. If the raycast hits one of the planes detected by AR Foundation, the virtual world is spawned at that position.
- *ScreenUtils.cs* contains utils for transforming the coordinates of a point between the coordinate system of the camera frame and the Unity coordinate system of the screen.

### 4.4.2. *CameraMatProviders*

As mentioned before, we wanted to be able to run the application both on mobile and on desktop platforms. That raised a problem - the mobile application was using the AR Foundation, while the desktop one was not. The issue was that AR Foundation was using the smartphone's camera to track its position in space, so it was not possible to access the same camera from another script to get a frame to feed to a hand tracker.

Fortunately, AR Foundation provides a method for getting the latest camera frame indirectly, and that is what *MobileCameraMatProvider* is doing. After getting the frame it converts it into the correct format that OpenCV can understand and returns it when requested by the *HandPositionEstimator*.

When running on a desktop, however, everything related to AR is inactive, so a new method for getting the latest camera frame is needed. Since the smartphone's camera is no longer locked by AR Foundation, it was possible to acquire the latest camera frame directly using OpenCV. That is what *DesktopCameraMatProvider* is responsible for.

Both classes inherit from the *CameraMatProvider* class, and both have the *PlatformVisibilityManager* script attached to them. This way, the *HandPositionEstimator* can have a single instance of the *CameraMatProvider* class, and the correct child class will be used based on the platform the application is running on, making it possible to always get the latest camera frame regardless of whether the application is running on desktop or on mobile.

### 4.4.3. *HandPositionEstimator*

As mentioned above, this class is responsible for connecting the information gathered by the hand trackers to the virtual world. When initialized, it creates a private instance of a hand tracker determined by the hand tracker type stored in the Game Manager and then, on every update tick, it requests a camera frame from the *CameraMatProvider*, forwards it to the hand tracker instance and applies the detected positions to the in-game objects representing hands.

It is also responsible for managing the color picker. During the *ColorPicker* stage of the *GameManager* it listens for touches, and, if one was detected, calculates the color range of a circle around the selected point and passes that color range to the hand tracker instance. All variants of the hand trackers receive this information, but only some of them use it if the algorithm requires it. The script is responsible for updating the UI element of the color picker too - aligning it with the touched point and updating its color to represent the average of the colors within it.

Finally, it is responsible for updating the preview canvas image with the latest camera frame, potentially, if debug information is enabled by the user or the current stage is *ColorPicker* or *WorldInstantiation*), enhanced with the debug information returned by the hand tracker (hand outlines, ArUco marker borders and ids, etc.).

The Algorithm 1 attachment shows the basic pseudo code of this script.

```
public class HandPositionEstimator : MonoBehaviour
{
   void Start()  // Executed once when the class is instantiated
   {
      handTrackerType = gameManager.GetHandTrackerType()

      create an instance of the correct hand tracker class based on handTrackerType
   }

   void Update()  // Executed every in-game tick
   {
      frameMat = cameraMatProvider.GetMat()

      if (gameManager.GetStage() == Stage.ColorPicker)
      {
         sample the range from the touched point and forward it to the hand tracker instance
      }

      handPositions = handTrackerInstance.GetHandPositions(frameMat);

      apply detected hand positions to the in-game objects representing hands

      update the camera preview image with the information provided by
      the hand tracker (hand outlines, marker borders, etc.)
   }
}
```

Algorithm 1. *HandPositionEstimator* pseudo code.

#### 4.4.4. *HandTrackers*

This folder holds all of the classes containing implementations of the hand tracking algorithms described above. Each of the classes inherits from the *HandTracker* interface with the following methods defined:

- *Initialize* - called by the *HandPositionEstimator* on the first game tick when a camera frame was available. Receives information such as the size of the camera frame and the active camera in the virtual world, and is responsible for setting up private variables used by the hand tracker.
- *IsInitialized* - returns *true* if *Initialize* was successfully executed before. Is used by the *HandPositionEstimator* to prevent multiple initializations.
- *Dispose* - called at the end of the *HandPositionEstimator*'s life cycle. Is responsible for cleaning up the variables and memory used by the hand tracker.
- *GetHandPositions* - main method used by the *HandPositionEstimator* to get

the detected positions of the hands. Receives an OpenCV Mat representing the current camera frame and returns a List of *HandTransforms* - structures containing the position, rotation and scale of the hands in the coordinate system of the virtual world.

- *SetThresholdColors* - a method called by the *HandPositionEstimator* after the user used the Color Picker and sampled a new color range. The hand tracker instance saves these colors and uses them later if the algorithm requires them.

The following sections describe each of the hand trackers in detail.

### 4.4.5. *ArUcoTracker*

This tracker uses ArUco markers attached to the user's hands in order to determine their positions. Out of all the trackers implemented in the project, this tracker is the only one capable of detecting the full 3D position of the hand, including the distance from the camera.

OpenCV provides many tools for computer vision, and it is able to work with ArUco markers too - the ArUco contrib module[21] provides all necessary functions required to estimate the pose of a marker based on a camera frame.

As seen in the Algorithm 2, when detecting the markers the tracker does four things. First, it loads a predefined marker dictionary that contains the configurations of the markers that should be tracked (a map from marker ids to the structure of the marker - its size and the black or white values for each cell). Then, for each captured camera frame it finds the ids and 2D corner points of the markers it detected in the frame using the *Aruco.detectMarkers* function of OpenCV. Then, it calculates the 3D positions and rotations of the markers relative to the camera based on the detected corners. Finally, it converts these values into positions and rotations in the coordinate space of the virtual world and returns them in a list of *HandTransforms*.

The rotation data is converted using a utility function provided by OpenCVForUnity - *ARUtils.ConvertRvecToRot.* The position is converted by finding the center points of each marker in the screen space, converting them to the

coordinate space of the virtual world and moving them in the direction the virtual camera is facing by the Z value (distance from camera) of the position estimated by OpenCV. Doing this instead of using the estimated position directly is beneficial since this ensures the virtual hand object is always aligned with the marker in the preview camera feed projected on top of the virtual world, regardless of the dimensions of the user's screen and aspect ratio of the camera.

```
public class ArUcoTracker : HandTracker
{
   public void Initialize()
   {
      // Creates a dictionary of markers that the tracker will detect.
      // Defaults to the ArUco default 4x4 dictionary.
      markerDictionary = Aruco.getPredefinedDictionary()
   }

   public void GetHandPositions(Mat frameMat, out List<HandTransform> hands)
   {
      // Get the corner points and the ids of the markers detected in the frame.
      corners, ids = Aruco.detectMarkers(frameMat, markerDictionary);

      // Estimate positions and rotations of the markers relative to the camera
      transforms, rotations = Aruco.estimatePoseSingleMarkers(corners);

      convert the relative transforms and rotations into the coordinate system of the virtual
      world, create an instance of HandTransform for each marker and push it to
      the hands list.
   }
}
```

Algorithm 2. *ArUcoTracker* pseudo code.

#### 4.4.6 *ThresholdTracker*

This tracker uses the Color Thresholding algorithm, and it is designed to work with two color spaces - HSV and L$\alpha\beta$. This tracker requires the user to select a color range corresponding to their hand color, and, as mentioned before, *HandPositionEstimator* takes care of that.

As seen in the Algorithm 3, the tracker works by applying a sequence of OpenCV operations to the image in order to extract the hand positions from the camera frame. It starts by converting the input frame into the specified color space - HSV or L$\alpha\beta$. Then, it applies a small blur to the image in order to get rid of some of the noise.

```
public class ThresholdTracker : HandTracker
{
   public ThresholdTracker(bool useLab)
   {
      this.useLab = useLab;
   }

   public void SetThresholdColors(Color lower, Color upper)
   {
      this.lowerColor = lower;
      this.upperColor = upper;
   }

   public void GetHandPositions(Mat frameMat, out List<HandTransform> hands)
   {
      convert frameMat image into either HSV or Laß, based on the value
      provided to the constructor

      // Blur the converted image to remove noise
      Imgproc.blur(frameMat)

      // Create a mask of the camera frame corresponding to pixels in the color range
      handMask = Core.inRange(frameMat, this.lowerColor, this.upperColor)

      // Dilate the mask to remove noise
      Imgproc.dilate(handMask)

      // Find contours in the mask
      contours = Imgproc.findContours(handMask)

      find the contour with maximum area

      remove all contours with area smaller than a certain fraction of the maximum area
      and smaller than a certain threshold, ideally keeping only the contours of the hands

      for each contour, get its center point, convert it into the coordinate system of the virtual
      world and project it a specific distance in the direction the virtual camera is facing

      create a HandTransform for each projected position and push it to the hands List
   }
}
```

Algorithm 3. *Threshold Tracker* pseudo code.

It then applies a function that creates a hand mask of the camera frame - an image where all pixels within the color range are white, and the pixels outside the color range are black. The mask is then slightly dilated in order to remove more noise. Finally, a function is applied to the mask in order to find the contours in it - curves outlining the edges between black and white of the mask.

These contours are intended to represent outlines of the user's hands, but they are still vulnerable to noise. In order to remove most of it, the contours are filtered. First, the contour with maximum area is found. Then, all contours with area below a certain fraction of the maximum area are removed. Since hands are of similar size and, being right in front of the headset, occupy a sizable portion of the camera frame, this step is meant to remove the contours of similarly colored but smaller objects in the background. Finally, all contours with an area smaller than a predefined constant are removed for the same reasons.

The remaining contours are processed in order to find their position in the coordinate system of the virtual world. For each contour, its center point is found using its moments (moments are calculated using OpenCV). The center points are then converted to the coordinate system of the virtual world using a custom utility function, and, finally, they are translated a fixed predefined distance in the direction the virtual camera is facing. The translated point positions are then used to create instances of *HandTransform* and returned to the *HandPositionEstimator*. It is worth noting that since this method projects the hand points a fixed distance away from the camera, the movement of the hands away and toward the camera is not detected.

#### 4.4.7. *CamshiftTracker*

This tracker uses the Camshift algorithm in order to track the user's hands. OpenCV already provides an implementation of the algorithm, however, there were a few steps needed to be done in order to run it.

Since the Camshift algorithm operates on bounding boxes (or regions of interest), the first step is to find such a bounding box for the object we want to track (in our case, a hand). In this thesis this is done in a way that is very similar to how the *ThresholdTracker* works - the user uses a color picker to provide a color range, the algorithm masks out all the colors in that range and finds the contours. In this case, however, instead of finding and returning the center points of the contours, the algorithm finds the bounding rectangle of the contour with the maximum area and sets it as the initial region of interest for the Camshift algorithm.

```
public class CamshiftTracker : HandTracker
{
   public CamshiftTracker(bool useLab)
   {
      this.useLab = useLab;
   }

   public void SetThresholdColors(Color lower, Color upper)
   {
      this.lowerColor = lower;
      this.upperColor = upper;

      follow the same steps as in the ThresholdTracker to get the hand contours

      find the contour with the maximum area

      // Get the bounding rect of the maximum area contour
      initRoiRect = Imgproc.boundingRect(maxAreaContour)

      // Calculate the histogram of the bounding rect
      roiHist = Imgproc.calcHist(new Mat(latestFrameMat, initRoiRect))
   }

   public void GetHandPositions(Mat frameMat, out List<HandTransform> hands)
   {
      convert frameMat image into either HSV or Laβ, based on the value
      provided to the constructor

      camshiftDst = Imgproc.calcBackProject(frameMat, roiHist)

      rotatedRect = Video.CamShift(camshiftDst, initRoiRect)

      find the center point of the detected rotated rect, convert it into the coordinate
      system of the virtual world, create an instance of HandTransform out of it
      and push it into the List of hands.
   }
}
```

Algorithm 4. *CamshiftTracker* pseudo code.

After the initial region of interest is found, the *CamshiftTracker* uses OpenCV to find its histogram in the color space - HSV or L*αβ* - that was selected in the constructor, and later uses it to run the Camshift algorithm. The initial region of interest and its histogram are recalculated every time the user uses the color picker and selects a new color range.

Once the initial region and its histogram are available and *HandPositionEstimator* requests the hand positions, the algorithm proceeds with four steps. First, it converts

the received camera frame into the color space selected in the constructor. Then, it runs an OpenCV function *Imgproc.calcBackProject* on the frame with the histogram of the initial region of interest. It calculates the back projection of the camera frame against the provided histogram, which essentially shows how well the pixels of that frame fit the distribution of pixels given by the histogram. It, similarly to Color Thresholding, finds pixels with colors similar to the colors of the initial region of interest. After the back projection is calculated, the algorithm calls the OpenCV implementation of Camshift on it in order to receive the rotated rect bounding the tracked object in the new frame.

Once the rotated rect is calculated, the algorithm finds its center point and follows the same steps as in the *ThresholdTracker* - convert the coordinates of the center point into the coordinate space of the virtual world, offset it by a fixed distance in the direction the virtual camera is facing, and return it to the *HandPositionEstimator*. The Algorithm 4 attachment provides the pseudo code for this tracker.

It is worth mentioning that this tracker, similarly to the *ThresholdTracker*, is only able to track the user's hand in 2D, and the distance from the hand to the virtual camera remains constant. Additionally, this implementation of Camshift is only able to track one hand.

#### 4.4.8. *OpenPoseTracker*

This tracker uses the OpenPose neural network in order to get the full skeletal mesh of the user's hand. It was intended and designed in a way to check if it was viable to not only track the position of the hand, but also the exact gestures and positions of the individual fingers, similar to how it works in the commercial standalone VR headsets like Oculus Quest.

As shown by the Algorithm 5 attachment, the tracker starts by creating an instance of an OpenCV Dnn Module network provided with the OpenPose weights. Then, whenever a new camera frame is received, the tracker feeds the frame through the network and processes the prediction, getting the positions of each of the hand landmarks. The code for processing the output is provided by OpenCVForUnity.

```
public class OpenPoseTracker : HandTracker
{
   public Initialize()
   {
      create a dictionary from key point names to their ids used by the network, and a
      list of pairs used to identify connected pairs

      // Load the model from the weights and configs located in the
      // StreamingAssets/dnn folder using the OpenCV Dnn module
      openPoseNet = Dnn.readNet(caffemodelFilepath, prototxtFilepath)
   }

   public void GetHandPositions(Mat frameMat, out List<HandTransform> hands)
   {
      // Load the current frame into the OpenPose network and run predictions
      openPoseNet.setInput(Dnn.blobFromImage(frameMat))
      output = openPoseNet.forward()

      convert the output from the network into a list of 2D points using the code
      provided by the OpenCVForUnity asset

      for each pair of connected key points, draw a line between the predicted positions
      of the two key points
   }
}
```

Algorithm 5. *OpenPoseTracker* pseudo code.

Ideally, the positions detected by OpenPose should then be applied to a virtual rigged hand. However, as discussed in the Experiments chapter, this method appeared to be incredibly slow and not viable to use in a real application. Moreover, the OpenPose network only provided 2D positions of the hand key points, so it would require some extensions in order to be used for full 3D hand tracking. Because of these issues we decided not to extend the algorithm to function as an actual tracker and leave it as a proof of concept instead.

#### 4.4.9. *YoloTracker*

This tracker comes in two variants - one using the YOLOv3 network, and one using the smaller YOLOv3Tiny network. In contrast with OpenPose, YOLO only provides a bounding box for the whole hand, making it a smaller network that should perform better as well.

```
public class YoloTracker : HandTracker
{
   public YoloTracker(bool tiny)
   {
      this.tiny = tiny
   }

   public Initialize()
   {
      // Load the model from the weights and configs corresponding to the
      // network version specified in the constructor and located in the
      // StreamingAssets/dnn folder using the OpenCV Dnn module
      yoloNet = Dnn.readNet(modelFilepath, configFilepath)
   }

   public void GetHandPositions(Mat frameMat, out List<HandTransform> hands)
   {
      // Load the current frame into the OpenPose network and run predictions
      yoloNet.setInput(Dnn.blobFromImage(frameMat))
      output = yoloNet.forward()

      convert the output to a list of bounding boxes and their confidences using
      code provided by the OpenCVForUnity asset

      apply Non-Maximum Suppression to the bounding boxes using code provided
      by the OpenCVForUnity asset

      filter out boxes with confidence below a fixed threshold

      for each point, convert it to the coordinate space of the virtual world, translate it
      a fixed distance in the direction the virtual camera is facing, create an instance
      of HandTransform with the translated point and push it to the hands List
   }
}
```

Algorithm 6. *YoloTracker* pseudo code.

Similarly to the *OpenPoseTracker*, this tracker creates an OpenCV Dnn model when initialized. The weights for the YOLO models were not custom trained, but instead provided by *Florian Bruggisser on Github*[16]. Creating custom weights would be possible, but it would require building a custom annotated hand dataset and putting a lot of time into training the model, so we decided to use pre-trained weights for the purpose of this thesis.

As shown in the Algorithm 6 attachment, once this tracker receives a camera frame, it passes it through the network created during the initialization. Then, it

processes the output and obtains the coordinates and confidences of the bounding boxes outlining the detected hands. It then filters out the boxes with low confidences and applies Non-Maximum Suppression to the remaining boxes. This helps with the cases where a single object was detected multiple times with slightly different bounding boxes. An example of it can be seen in Figure 6. The code for both processing the network output and Non-Maximum Suppression was provided by the OpenCVForUnity asset.

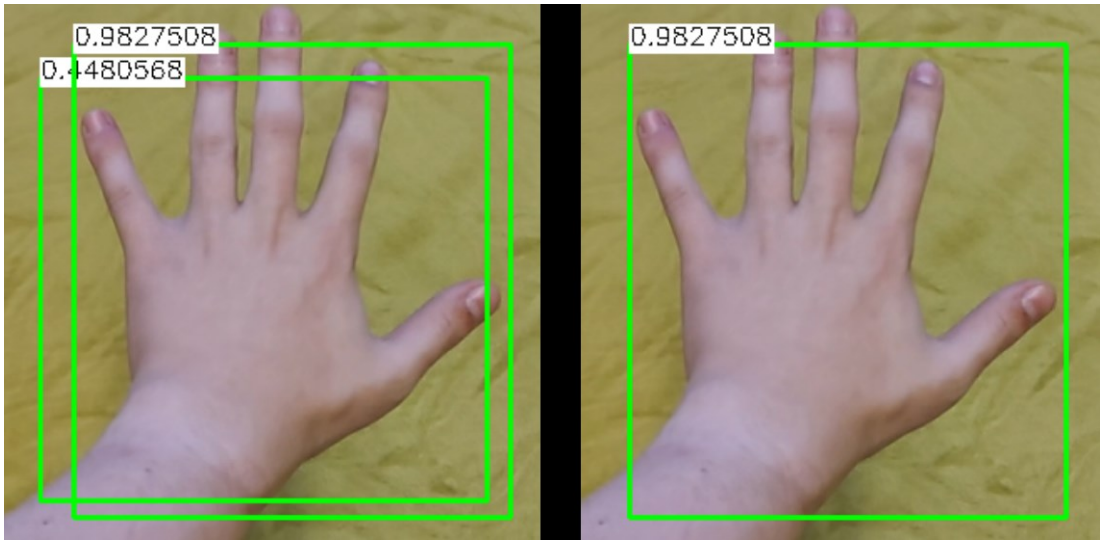


Figure 6. Left - unfiltered YOLOv3 predictions. Right - Non-Maximum Suppression applied to the same predictions.

The algorithm then finds the center points of each of the bounding boxes, converts them to the coordinate system of the virtual world and translates them a fixed distance in the direction the virtual camera is facing. Therefore, similar to the *ThresholdTracker* and the *CamshiftTracker*, this algorithm is also only capable of tracking the hands in two dimensions.

## 4.5. Screen Flow

The final application has five screens. The first screen is the starting menu (Figure 7) that provides some basic information on how to use the application, allows the user to select the hand tracker they want to use and proceed to the next screen.


Figure 7. Starting screen of the application.

The second screen is only visible if the selected algorithm requires a color range to operate. It is the color picker screen (Figure 8), and it allows the user to use the color picker. It shows a preview of the data detected by the algorithm and has a button to accept the color range.

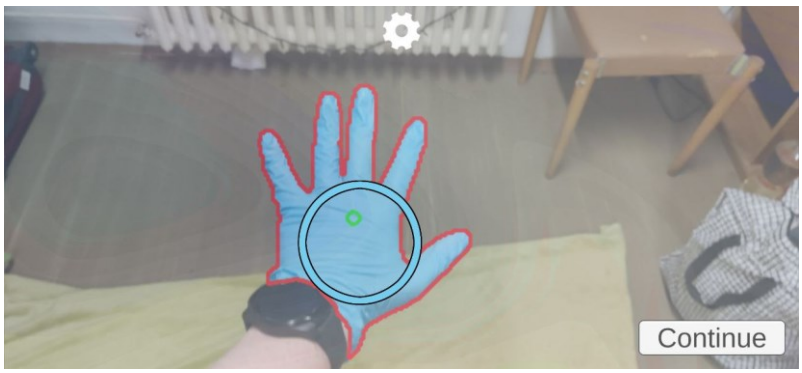


Figure 8. Color Picker example.

The third screen (Figure 9) allows the user to spawn the virtual world. It shows dots on the planes detected by AR Foundation, and tapping on any of them creates the virtual world at that point and moves the application to the next screen.

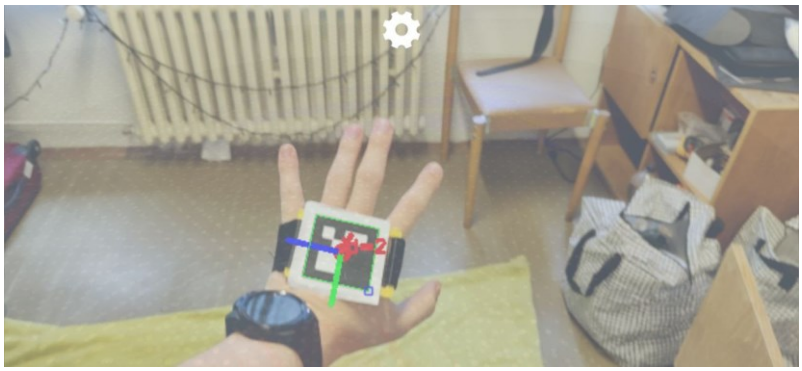

Figure 9. World Instantiation screen example.

The fourth screen (Figures 10 and 11) is the virtual world itself, where the user can move around and use their hands to interact with it.


Figure 10. Virtual world with the debug preview camera overlay.

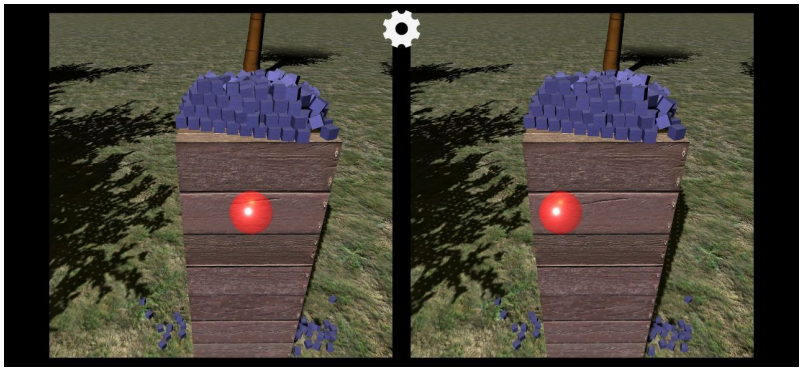

Figure 11. Virtual world without the debug preview camera overlay.

The fifth screen (Figure 12) is the in-game settings menu that can be activated from any screen except the starting menu. It allows the user to toggle debug information and the split screen.

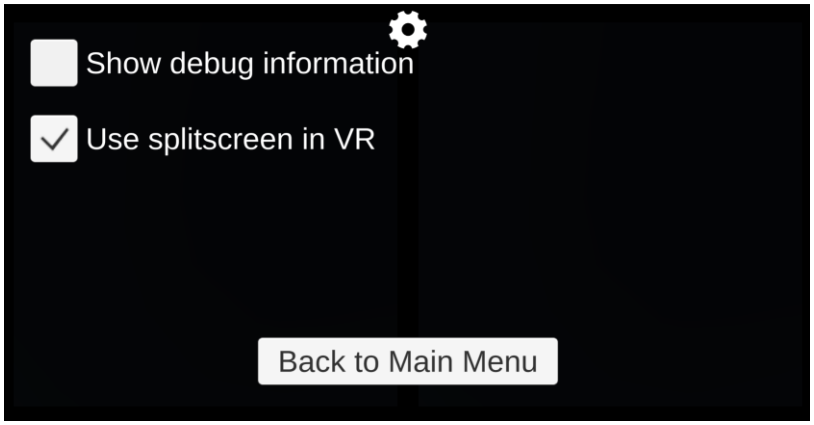


Figure 12. In-game settings screen.

# 5. Experiments

Since the application presented in the thesis uses only one approach for tracking the position and orientation of the headset (AR Foundation), the experiments are focused on benchmarking and comparing the various hand tracking algorithms. The goal of the experiments is to find the strengths and weaknesses of each method.

## 5.1. Experimental setup

The two important metrics measured for each algorithm are speed and accuracy. Speed is very important for VR applications since they need to be able to trick the user's brain into believing into the virtual world. If the application is not running smoothly, the user will not feel like their real-world actions (like looking around) are correctly relayed to the virtual world. Accuracy is important for similar reasons - if the hands are not tracked correctly, the user will feel more disconnected from the virtual experience.

### 5.1.1. Speed

The speed of a method is measured in Frames Per Second (FPS). The application contains an FPS monitor that calculates the average FPS over one minute, and two game scenes for the algorithms to be executed in. The first scene (*Empty – MainEmpty* scene in Unity) is almost empty, designed to give the full processing power of the device to the algorithm. The second scene (*World – Main* scene in Unity) contains elements that one can usually find in a game - terrain, vegetation, light sources and physics objects. It is designed to provide a more realistic benchmark, since the algorithms are most likely to be used in non-empty applications.

The FPS experiment is executed in the built application running on the OnePlus 8 Pro smartphone.

### 5.1.2. Accuracy

The accuracy is measured by running an algorithm on a prerecorded video footage and calculating 3 values:

- *Correct frames* - frames where the algorithm correctly located the hand (the output position is located within the visual boundary of the real hand).
- *Incorrect frames* - frames where the algorithm either detected a hand in a place where no hands are positioned or detected two separate hands in the visual boundary of one real hand.
- *Missed frames* - frames where the algorithm did not detect anything, but a real hand was present in the frame.

To compare the algorithms consistently and fairly, their inputs need to be as similar as possible. A good solution would be to test them on the same video recording, however, it is not possible since different algorithms require the hands to be presented in different ways - either just the hand, or the hand in blue gloves, or the hand with an ArUco marker.

Recording separate videos for each hand representation would be subject to inconsistencies in the hand position and movement, making the experiment unfair. The solution used in the thesis is to create the input videos using a 3D animation software, preserving the exact movement of a hand, while allowing us to change its appearance. Since most algorithms rely on colors to detect the hands, a realistic 3D animation should provide results similar to the results on real-world footage.

The animations for the experiments in this thesis were created using Blender. A sample result contains a simple room designed to represent an average background for the algorithm, and an animated hand moving in a predefined way. For each of the three hand representations (skin, blue glove, ArUco marker), the animation was rendered in three different lighting conditions. For each of the conditions there is a single 1000W light source located outside the room and shining through the windows, and a lightbulb in the room with varying power for the conditions - 50W, 20W and 0W. The path the hand takes was designed in such a way that the hand

moves at various speeds and is visible both in a shadow and in light. Motion blur and artificial shutter were used in the renders to provide a more realistic result.

Using artificial footage allowed us to test the algorithms on the same hand motions with different hand representations and different lighting conditions.

However, after running the experiments we discovered that while all the color-based algorithms were performing very similarly to how they performed on real-world footage, the neural network algorithms were performing visibly worse. So, as an exception, we also tested YOLOv3 and YOLOv3Tiny on a real-world video, aiming to more accurately compare them against each other while still preserving the consistency of the experiment.

These videos were then used as inputs in the *MainExperiment* scene in the Unity project. The scene is a stripped-down version of the main scene. It does not have any of the VR elements and its *HandPositionEstimator* was redesigned to accept inputs from the videos instead of the camera feed. Each of the algorithms was then tested on the videos, where we manually checked the detection results for each frame and marked it as either *correct*, *incorrect* or *missed*. The videos are located in the *StreamingAssets* folder of the project.

Figure 13. A frame from one of the artificially generated videos.

## 5.2. Results

The testing methods described above were applied to each algorithm. The overview of the results can be found in Diagrams 1, 2, 3, 4 and 5. The accuracy diagrams represent the amount of *correct, incorrect* and *missed* frames the algorithm produced at the three lighting conditions. The speed diagrams represent the FPS of each method in the *Empty* and *World* scenes. A more detailed explanation of the results follows in the next subchapters, and the specific numbers obtained during the experiments can be found in Tables 2, 3, and 4.

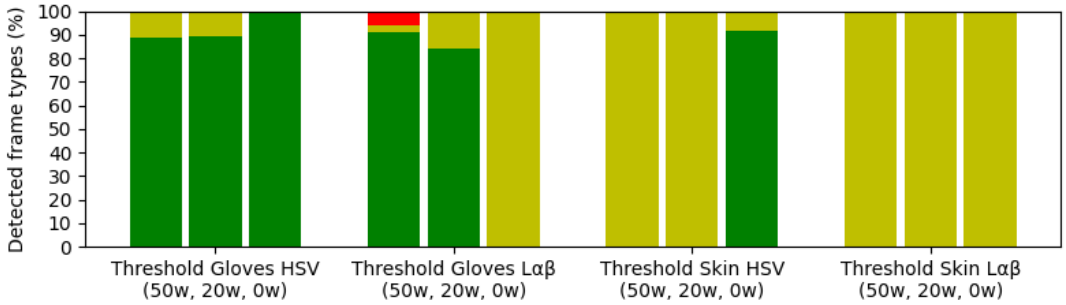


Diagram 1. Accuracy of the Color Thresholding algorithms.

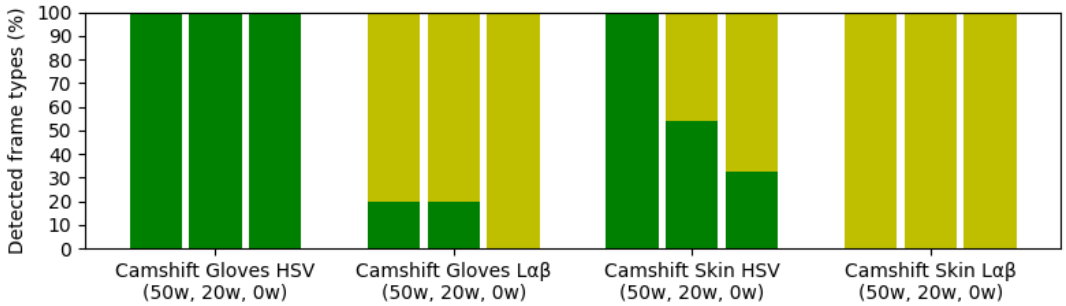


Diagram 2. Accuracy of the Camshift algorithms.

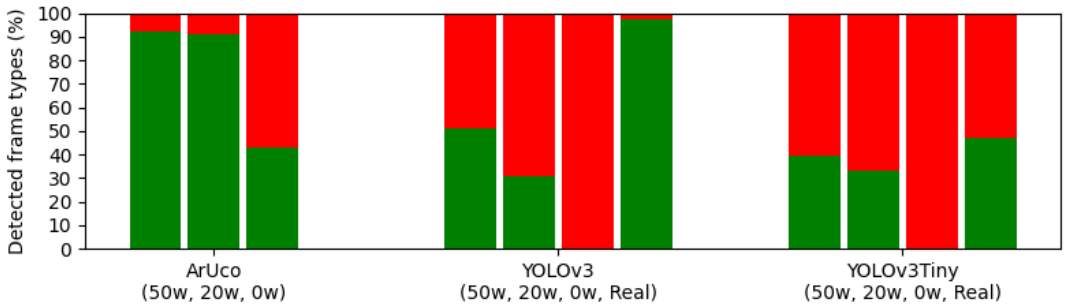


Diagram 3. Accuracy of the ArUco and YOLO algorithms.

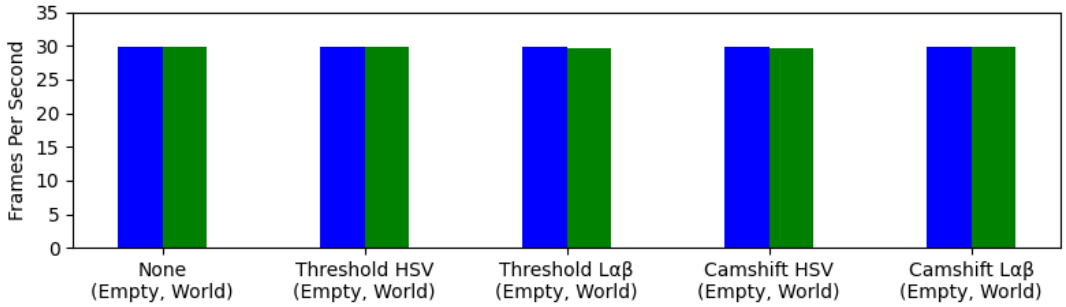


Diagram 4. FPS of the scenes with Threshold, Camshift and no algorithms.

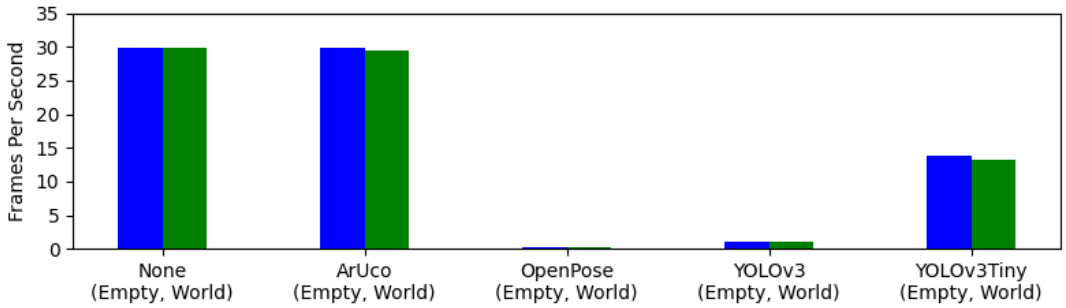


Diagram 5. FPS of the scenes with ArUco, Neural Network and no algorithms.

### 5.2.1 ArUco

The ArUco marker detection appeared to be very fast, reaching 29.86 average FPS in the *Empty* scene, and 29.48 FPS in the *World* scene. The difference from the FPS in a scene with no detectors is minor, making it a viable algorithm for use in a real application.

Looking at the accuracy, the algorithm managed to correctly detect 92% of the frames with the 50W light, 91% with the 20W light and 43% with the 0W light. The algorithm never produced an *incorrect* frame, so all of the frames that were not *correct* were *missed*. Upon closer inspection it appeared that the algorithm struggles with detecting a marker that is moving fast, most likely due to motion blur affecting the sharp edges of the marker.

It is also visible that the accuracy of the method decreases with lower light, which is most likely due to decreasing contrast between the white and black parts of the markers.

### 5.2.2. Color Thresholding

The scene with the Color Thresholding algorithm achieved similar FPS to both ArUco and the empty scene, averaging at 29.81 FPS for the *Empty* scene and 29.78 FPS for the *World* scene with the HSV color space, and 29.84 FPS for the *Empty* scene and 29.65 FPS for the *World* scene with the L$\alpha\beta$ color space. These speeds are similar enough to the base application with no detectors, making this algorithm a viable solution for use in a real application.

When measuring accuracy, the algorithm was tested in two main ways - either with hands with bright-blue gloves, or with simple skin-colored hands. Then, each of these ways was tested on the HSV and the L$\alpha\beta$ variants of the algorithm. Finally, each of the combinations was tested on three different lighting conditions as mentioned above.

The algorithm with gloves and HSV performed the best compared to the other three variants. It managed to correctly detect the hand in 89% of the frames with the 50W light, 90% with the 20W light, and a full 100% with the 0W light. It missed only a small number of frames, and the majority of errors in detection were the *incorrect* frames - the frames where a hand was found in a place where no real hand was located. This is expected, since the background contained some objects with colors similar to the blue glove.

Upon visual inspection of the results, it appeared that the algorithm mostly struggles with the frames where the tracked object changes its color. In the input videos this is represented by the hand moving in and out of direct light, changing the brightness of its color. This explains why lower values of light performed better in the Gloves and HSV experiment - with less light the color difference between an object in shadow and in light is also smaller.

The algorithm with gloves and L$\alpha\beta$ performed slightly worse than the HSV variant, achieving 91% *correct* frames at 50W, 84% at 20W, and 0% at 0W. As with the previous variant, the majority of errors are incorrect frames where the algorithm found regions in the background that are colored similarly to the user's hand. Surprisingly, while the algorithm performed reasonably well at 50W and 20W, it completely misclassified every frame at 0W. One reason for that is that in L$\alpha\beta$ at 0W the video contained a region that is much bigger than the glove while being similarly colored, making the algorithm classify that region as a hand instead.

Both the HSV and L$\alpha\beta$ versions of the algorithm performed worse on the input with a hand with no bright gloves. The HSV version correctly detected the hand in 0% of the frames at 50W, 0% at 20W, and 92% at 0W. The L$\alpha\beta$ version did not correctly detect a hand in a single frame, making incorrect detections for 100% of the frames in all lighting conditions. The only reasonable results were achieved by the HSV version of the algorithms, due to reasons similar to before - with less light, there's less color variations when the hand moves through the scene.

Overall, it appears that using bright gloves and the HSV color space provides the best results. The bright color of the gloves does not have many similarly colored objects in the background, making it easier to find the region corresponding to the hand using color thresholding, and the HSV color space outperformed L$\alpha\beta$ most likely because HSV is designed to represent colors under light, and that is exactly the setup used in the experiment videos.

In contrast with ArUco, the Color Thresholding algorithm performed *better* with lower values of light. As mentioned above, low levels of light mean less color

variation between shadows and light, making the color range of the hand consistent throughout the whole path that it takes.

### 5.2.3. Camshift Algorithm

Similar to the algorithms above, the HSV version of the Camshift Algorithm achieved an average of 29.75 FPS for the *Empty* scene and 29.73 FPS for the *World* scene, while the L*αβ* version achieved 29.88 FPS for the *Empty* scene and 29.81 FPS for the *World* scene. Once again, these framerates are very close to the framerate of a scene with no detectors, so the Camshift algorithm is also a viable approach that could be used in a real application.

The accuracy of the algorithm was measured in a similar way to the Color Thresholding algorithm - with gloves and without gloves, HSV and L*αβ*, and three lighting conditions. The results turned out to be similar as well - the HSV version with gloves performed the best compared to the other three, correctly detecting the hand in 100% of the frames in all lighting conditions. The histogram of the region of bright colored gloves used by the algorithm was quite different from the other regions of the frames, making it easier for the algorithm to locate the hand in a frame.

The HSV version of the algorithm even reasonably performed on the videos without gloves, correctly detecting 100% at 50W, 54% at 20W and 32% at 0W. At lower light levels, the algorithm seemed to lose tracking - the detected hand region would detach from the hand and start tracking another object (with somewhat similar colors).

A similar issue was present in the L*αβ* version of the algorithm running on the video with gloves. It correctly detected the hand in 20% of the frames at 50W and 20W, however, it did not detect it in any of the frames at 0W. In the case of the low light level, the problem appeared to be in the initial setup of the algorithm - the initial region of the hand was not calculated correctly, so the whole algorithm was tracking the wrong thing. The two higher levels of light, however, had an issue similar to the HSV variant of the algorithm - the hand was detected correctly in the first few

frames, but then the tracking window anchored itself to some other object and tracked it instead.

Another issue was found after running the L$\alpha\beta$ version of the algorithm on the videos without gloves. In all three lighting conditions, the algorithm did not correctly detect a hand in any of the frames, and instead found a hand in the wrong place for every frame. This happened mostly due to the same issue that happened with the gloves at 0W - the initial tracking window was not calculated correctly, so the algorithm was tracking the wrong thing.

Overall, the HSV version of the Camshift Algorithm showed some great results, both with the user wearing gloves and without. When the gloves were used, the algorithm managed to correctly detect the hands 100% of the time, however, without the gloves, it was only able to do so at the highest light value.

### 5.2.4. OpenPose Neural Network

This algorithm appeared to be the slowest of the ones tested, reaching an average of 0.28 FPS in the *Empty* scene and 0.27 FPS in the *World* scene. This is much lower than the average 30 FPS of the color-based methods, and is basically unusable, as the application turns into a slideshow instead of a VR experience.

Thus, only judging by speed, we determined that this method is not ready to be used in a real application since modern smartphones do not have enough processing power to run this neural network smoothly. Because the method was discarded based on speed alone, we did not test its accuracy, as that would be redundant and take a very long time.

### 5.2.5 YOLOv3 Neural Network

Two versions of this neural network were tested - YOLOv3 and YOLOv3Tiny. During the speed test the larger network, YOLOv3, performed slightly better than the OpenPose network, achieving 1.06 FPS in the *Empty* scene and 1.05 FPS in the *World* scene. While being faster than OpenPose, 1 FPS is still not enough for the

application to feel responsive, so, with the current processing power of modern smartphones, YOLOv3 is not a viable solution for a real application.

The smaller version, YOLOv3Tiny, performed much better, getting 13.92 FPS in the *Empty* scene and 13.15 FPS in the *World* scene. While not being as smooth as the 30 FPS achieved by the other methods, YOLOv3Tiny still felt responsive enough for it to be considered as a hand tracker in a real application.

When measuring the accuracy, we noticed that when running on artificial footage, these networks were performing visibly worse than if they were running on actual real video footage. Since both versions of the network do not require anything extra for the user to wear, we also tested these networks on the same prerecorded real-world video. The results confirmed the suspicions - YOLOv3 correctly detected 51% of the frames in an artificial video at 50W, 31% at 20W and 0% at 0W, but at the real-world video it managed to correctly detect the hand in 97% of the frames. YOLOv3Tiny had similar imporvements, getting 39% at 50W, 33% at 20W, 0% at 0W and 47% at the real-world video.

Overall, this method works well in terms of user experience, since the user does not have to wear gloves or special markers. However, while the larger network correctly detects most of the time, it is too slow to be used in a real app, and while the smaller network is faster, it only manages to detect the hand about half of the time. So, while this method has potential, it is not a viable option with the current state of the neural network and the smartphone processing power.

# 6. Discussion

The experiments provided a lot of data and allowed us to see the strong and weak points of each of the approaches. ArUco was the only one to support full 3D hand tracking, but it had troubles with accuracy in low light and with fast-moving objects. The Color Thresholding algorithm performed well in the HSV color space and with the user wearing gloves, but it was still outperformed by the Camshift algorithm in the same conditions. Camshift, however, only supports one hand to be tracked, and tends to lose tracking if the hand leaves the camera frame.

We did find that the L$\alpha\beta$ versions of the algorithms generally performed worse than the HSV ones. The reason behind that is not entirely clear, but it might be related to the fact that the testing videos were constructed in a way that tested the tracked object under different values of light (in shadow and in light), and HSV was designed specifically to represent the colors under light.

The neural network algorithms generally performed worse than the color-based and marker-based ones. OpenPose was too slow to even consider using in a real app, YOLOv3 achieved decent results but was also slow, and YOLOv3Tiny was fast enough but underperformed in accuracy.

| Tracker | Provides full 3D tracking | Able to track multiple hands | Does not need extra hand attachments | Fast enough to be used in a real app |
|---|---|---|---|---|
| ArUco | Yes | Yes | No | Yes |
| Threshold | No | Yes | Both | Yes |
| Camshift | No | No | Both | Yes |
| OpenPose | No | Yes | Yes | No |
| YOLOv3 | No | Yes | Yes | No |
| YOLOv3Tiny | No | Yes | Yes | Yes |

Table 1: Capabilities of the hand tracking algorithms.

Overall, all algorithms have their advantages and disadvantages. Table 1 outlines the capabilities of the algorithms, and the best one to use highly depends on the requirements of the app. The main candidates for the best approach are:

- ArUco, if the distance from the camera and the 3D rotation of the hand are important and need to be detected. However, it will be vulnerable to low light and fast-moving objects, and it requires the user to wear special markers.
- Threshold, if the distance from the camera and rotation are not as important, and the application requires multiple hands to be tracked. It is, however, vulnerable to changes in lighting conditions and requires the user to wear bright gloves for best results.
- Camshift, if the distance from the camera and rotation are not important and if tracking one hand is enough. It does have more restrictions and also requires bright gloves for best results, but it achieves the best accuracy score compared to the other methods.
- YOLO, if the device has enough processing power to run it efficiently, and it is preferable that the user does not need to use any extra hand attachments like markers or gloves. Current smartphones do not seem to be able to run it fast enough yet, but it has the potential to provide consistent predictions on bare hands.

Of course, the verdict is subject to change in the future. The growing processing power of smartphones and quality of the neural networks will give OpenPose, YOLO and other networks a better chance. The conditions set on the color-based algorithms could be overcome with more development - this is discussed more closely in the Future Work chapter. Finally, there are many other approaches that could be applied to hand tracking that might prove to be better.

But the main outcome still holds - the proof of concept works. With the current technologies it is possible to create a mobile Virtual Reality experience with full headset and hand tracking in 6 Degrees of Freedom. While not being nearly perfect, the experience is comparable to the standalone headsets, and with more work it could be turned into a real competitor in the current space of VR headsets.

# Conclusion

The work presented here shows that while it is possible to create a mobile VR experience with both 6 Degrees of Freedom headset and hand tracking, it is still far from being perfect. The speed, accuracy and overall user experience is not a match for what the commercial standalone VR headsets provide. After all, they are optimized for VR and have many more sensors than just one camera, making their tracking more robust.

However, this thesis shows that the concept is feasible, and, with the current growth rate of the smartphones' processing power, in the near future the methods presented here (or other ones) could be used to create real competitors for the standalone VR headsets.

## Future work

There are three main things in the thesis that could be worked on in the future:

- *Trying new tracking algorithms*. There exist many more algorithms for object tracking, and chances are they might perform well when applied to hand tracking.
- *Improving the current algorithms to support depth detection*. All the algorithms presented in the thesis, apart from ArUco, only detect the hands in 2D, and project the detection a fixed distance away from the camera. Instead of being fixed, this distance could be calculated by the algorithms. Potential approaches include markers, working with shapes (e.g., deducing the distance from the camera based on the distance between fingers), or neural networks.
- *Applying various post-processing algorithms to the trackers*. At the moment, hand trackers provide the raw detected positions and apply them to the in-game objects, which results in some amount of noise and lost frames. Various post-processing algorithms could be applied to the raw position data in order to smooth it out and make it more accurate.

# Bibliography

[1] AR Foundation (2021, July). *https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@4.1/manual*

[2] OpenCV (2021, July). *https://docs.opencv.org/master/d1/dfb/intro.html*

[3] Francisco J.Romero-Ramirez, Rafael Muñoz-Salinas, Rafael Medina-Carnicer, "Speeded up detection of squared fiducial markers", *in Image and Vision Computing, vol 76, pages 38-47, 2018*.

[4] S. Garrido-Jurado, R. Muñoz Salinas, F.J. Madrid-Cuevas, R. Medina-Carnicer, "Generation of fiducial marker dictionaries using mixed integer linear programming", *in Pattern Recognition:51, 481-491, 2016.*

[5] Wikipedia, "HSL and HSV" (2021, July), *https://en.wikipedia.org/wiki/HSL_and_HSV*

[6] Wikipedia, "CIELAB color space" (2021, July), *https://en.wikipedia.org/wiki/CIELAB_color_space*

[7] Shrivastava, R. (2013, February). "A hidden Markov model based dynamic hand gesture recognition system using OpenCV". *In 2013 3rd IEEE International Advance Computing Conference (IACC) (pp. 947-950). IEEE.*

[8] G. R. Bradski. "Computer vision face tracking for use in a perceptual user interface", 1998

[9] Y. Cheng. "Mean shift, mode seeking, and clustering". *IEEE Trans. Pattern Anal. Mach. Intell., 17(8):790–799, 1995.*

[10] D. Exner, E. Bruns, D. Kurz, A. Grundhöfer and O. Bimber, "Fast and robust CAMShift tracking", *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops, 2010, pp. 9-16, doi: 10.1109/CVPRW.2010.5543787.*

[11] Z. Cao, G. Hidalgo Martinez, T. Simon, S. Wei, & Y. A. Sheikh (2019). "OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields". *IEEE Transactions on Pattern Analysis and Machine Intelligence.*

[12] Tomas Simon, Hanbyul Joo, Iain Matthews, & Yaser Sheikh (2017). "Hand Keypoint Detection in Single Images using Multiview Bootstrapping". *In CVPR.*

[13] Zhe Cao, Tomas Simon, Shih-En Wei, & Yaser Sheikh (2017). "Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields". *In CVPR.*

[14] Shih-En Wei, Varun Ramakrishna, Takeo Kanade, & Yaser Sheikh (2016). "Convolutional pose machines". *In CVPR.*

[15] Redmon, J., & Farhadi, A. (2018). "YOLOv3: An Incremental Improvement". *arXiv.*

[16] Florian Bruggisser (2020). "YOLO-Hand-Detection". *https://github.com/cansik/yolo-hand-detection*

[17] OpenCV for Unity (2021, July). *https://enoxsoftware.com/opencvforunity*

[18] Shaoqing Ren, Kaiming He, Ross Girshick, & Jian Sun. (2016). "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks".

[19] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". *arXiv preprint arXiv:1704.04861, 2017.*

[20] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, & Piotr Dollár. (2018). "Focal Loss for Dense Object Detection".

[21] OpenCV, "Detection of ArUco Markers" (2021, July). *https://docs.opencv.org/4.5.2/d5/dae/tutorial_aruco_detection.html*

# List of Tables

| Method | Correct frames | Incorrect frames | Missed frames | Total frames |
|---|---|---|---|---|
| ArUco (50w) | 460 | 0 | 39 | 499 |
| ArUco (20w) | 455 | 0 | 44 | 499 |
| ArUco (0w) | 216 | 0 | 283 | 499 |
| Threshold HSV Gloves (50w) | 442 | 55 | 2 | 499 |
| Threshold HSV Gloves (20w) | 447 | 47 | 5 | 499 |
| Threshold HSV Gloves (0w) | 499 | 0 | 0 | 499 |
| Threshold Lαβ Gloves (50w) | 455 | 14 | 30 | 499 |
| Threshold Lαβ Gloves (20w) | 420 | 79 | 0 | 499 |
| Threshold Lαβ Gloves (0w) | 0 | 499 | 0 | 499 |
| Threshold HSV Skin (50w) | 0 | 499 | 0 | 499 |
| Threshold HSV Skin (20w) | 0 | 499 | 0 | 499 |
| Threshold HSV Skin (0w) | 458 | 41 | 0 | 499 |
| Threshold Lαβ Skin (50w) | 0 | 499 | 0 | 499 |
| Threshold Lαβ Skin (20w) | 0 | 499 | 0 | 499 |
| Threshold Lαβ Skin (0w) | 0 | 499 | 0 | 499 |
| Camshift HSV Gloves (50w) | 499 | 0 | 0 | 499 |
| Camshift HSV Gloves (20w) | 499 | 0 | 0 | 499 |
| Camshift HSV Gloves (0w) | 499 | 0 | 0 | 499 |
| Camshift Lαβ Gloves (50w) | 100 | 399 | 0 | 499 |
| Camshift Lαβ Gloves (20w) | 100 | 399 | 0 | 499 |
| Camshift Lαβ Gloves (0w) | 0 | 499 | 0 | 499 |
| Camshift HSV Skin (50w) | 499 | 0 | 0 | 499 |
| Camshift HSV Skin (20w) | 269 | 230 | 0 | 499 |
| Camshift HSV Skin (0w) | 162 | 337 | 0 | 499 |
| Camshift Lαβ Skin (50w) | 0 | 499 | 0 | 499 |
| Camshift Lαβ Skin (20w) | 0 | 499 | 0 | 499 |
| Camshift Lαβ Skin Lab (0w) | 0 | 499 | 0 | 499 |

Table 2. Accuracy of the ArUco, Color Thresholding and Camshift algorithms.

| Method | Correct frames | Incorrect frames | Missed frames | Total frames |
|---|---|---|---|---|
| YOLOv3 (50w) | 254 | 0 | 245 | 499 |
| YOLOv3 (20w) | 154 | 0 | 345 | 499 |
| YOLOv3 (0w) | 0 | 0 | 499 | 499 |
| YOLOv3Tiny (50w) | 196 | 0 | 303 | 499 |
| YOLOv3Tiny (20w) | 165 | 0 | 334 | 499 |
| YOLOv3Tiny (0w) | 0 | 0 | 499 | 499 |
| YOLOv3 Real footage | 292 | 0 | 8 | 300 |
| YOLOv3Tiny Real footage | 142 | 0 | 158 | 300 |

Table 3. Accuracy of the YOLOv3 network.

| Method | FPS - Empty | FPS - World |
|---|---|---|
| None | 29.9 | 29.83 |
| ArUco | 29.86 | 29.48 |
| Threshold HSV | 29.81 | 29.78 |
| Threshold Lαβ | 29.84 | 29.65 |
| Camshift HSV | 29.75 | 29.73 |
| Camshift Lαβ | 29.88 | 29.81 |
| OpenPose | 0.28 | 0.27 |
| YOLOv3 | 1.06 | 1.05 |
| YOLOv3Tiny | 13.92 | 13.15 |

Table 4. FPS of the hand tracking methods in the two scenes.

# List of Abbreviations

1. **VR** - Virtual Reality
2. **AR** - Augmented Reality
3. **DoF** - Degrees of Freedom
4. **OpenCV** - Open Source Computer Vision Library
5. **YOLO** - You Only Look Once
6. **HSV** - Hue, Saturation, Value
7. **Camshift** - Continuously Adaptive Mean Shift
8. **ROI** - Region of Interest
9. **Dnn** – Deep neural network

# Appendix

This chapter provides information on how to install and use the built application, how to run the source code and how to replicate the experiments.

## Installing and using the built application

The build of the application is the *mobile-vr-with-hand-tracking.apk* file located in the top-level directory of the attached zip archive. It can be installed on a device running on Android 7.0 or later using the default Android package installer.

After the application is installed and started, it will display the starting menu screen. This screen contains general information on how to use the application and allows the user to select the hand tracker (using the dropdown on the left) and the virtual world (using one of the two buttons on the right) they want to use.

Once the hand tracking method and one of the virtual worlds are selected, the app will proceed to the next screen. If the selected tracker requires a color range to operate, the application will show a “Continue” button at the bottom right, and tapping anywhere else on the screen will reveal the color picker and sample the color range from the selected point. The hand tracking algorithm will show the results it is getting with the color range (contour for the *ThresholdTracker,* bounding box for the *CamshiftTracker*). It is possible to adjust the color range to improve the detection preview. Once the preview looks good, clicking “Continue” will send the application to the next screen.

The next screen is for World Instantiation. Slightly moving the device and rotating its camera view around will help AR Foundation to find surfaces in the real world. The detected surfaces will be displayed as grids of semi-transparent dots. It is usually enough to wait until the floor surface is detected. Once a surface is detected and displayed, tapping anywhere within the surface will create an instance of the virtual world at that point and move the application into the next screen.

The final screen is the virtual world itself. The user can move their device around and use their hands (in the representation required by the selected algorithm) to interact with the world.

At any point in time, apart from the starting menu screen, it is possible to access the settings menu using an icon at the top of the screen. This allows toggling debug information and split screen, as well as going back to the starting menu.

As mentioned before, different algorithms work best with different hand representations. OpenPose, YOLOv3 and YOLOv3Tiny do not require anything extra on the hands. Threshold and Camshift can work without anything extra, but they perform much better if the user wears bright gloves. A cheap and easy-to-get solution is to use blue surgical gloves. Finally, the ArUco algorithm requires ArUco markers placed on the user's hands. The markers can be any marker from the standard 4X4 marker dictionary, and their size needs to be 5 cm by 5 cm. Figures 14 and 15 provide the first two markers from the 4X4 dictionary as an example.

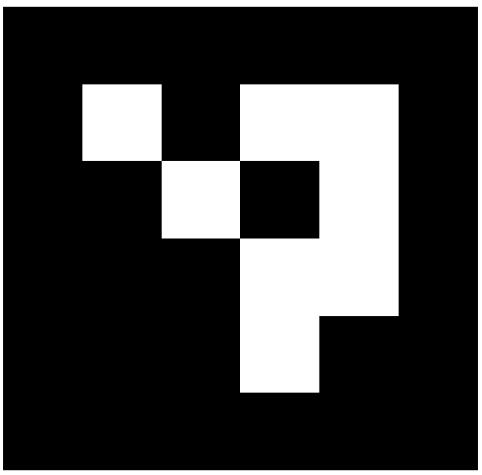

Figure 14. 4X4 ArUco marker with id = 0

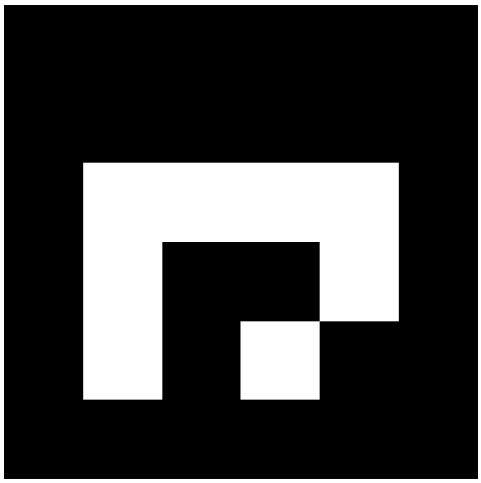

Figure 15. 4X4 ArUco marker with id = 1

## Setting up the project with the source code

The source code for the application is located in the *source* folder of the attached zip archive. It can be opened directly using Unity, and it is recommended to use the Unity version that the application was developed in - 2020.3.2f1.

The project needs the following packages from the Unity Asset Store and the Unity Package Registry to function:

- AR Foundation (4.0.12)
- ARCore XR Plugin (4.0.12)
- ARKit XR Plugin (4.0.12)
- TextMeshPro (3.0.4)
- OpenCV for Unity (2.4.3)
  *https://assetstore.unity.com/packages/tools/integration/opencv-for-unity-21088*

The following assets are not essential but required to run the *Main* scene.

- Outdoor Ground Textures (1.2.1)
  *https://assetstore.unity.com/packages/2d/textures-materials/floors/outdoor-ground-textures-12555*
- Nature Starter Kit 2 (1.0)
  *https://assetstore.unity.com/packages/3d/environments/nature-starter-kit-2-52977*
- Free HDR Sky (1.0)
  *https://assetstore.unity.com/packages/2d/textures-materials/sky/free-hdr-sky-61217*
- Plank Textures PBR (1.0)
  *https://assetstore.unity.com/packages/2d/textures-materials/wood/plank-textures-pbr-72318*

Once the project is set up and all the assets are installed, the application can be run in the Unity Editor. It is worth mentioning that when loading up the project for the first time, Unity can show a warning related to missing packages. It can be ignored as the packages above are indeed missing and need to be added manually.

## Recreating the experiment

There are two main experiments that can be recreated - the speed benchmark in FPS and the accuracy test.

The speed of an algorithm can be tested in the built application itself. In the application, select a hand tracker to be tested and proceed to one of the two virtual worlds. Then, access the settings menu and turn on the *Show debug information* toggle. This will enable the FPS monitor that will benchmark the speed of the algorithm.

The accuracy experiments can be replicated in the Unity Editor. Once the project is installed and set up following the steps from the previous subchapter, the experiment setup can be accessed by loading the *MainExperiment* scene located in *Assets/Scenes/Experiments.* The scene contains the *HandPositionEstimator* object, and the experiment can be tweaked by changing the public variables of the script attached to that object (it can be done directly in the editor by viewing the object in the Inspector). There are three values that can be changed:

- *Hand Tracker Type* - determines the hand tracker to be tested
- *Filename* - the filename of the video file that the tracker will be executed on. All of the video files are located in the *Assets/StreamingAssets* folder. When specifying the filename its file extension must be included.
- *Manual Frame By Frame* - a boolean that determines the way the experiment will run. If unchecked, the algorithm will run on the video feed without interruptions. This can be useful for visual analysis of the algorithm's accuracy. If checked, on the other hand, the algorithm will pause after every frame and wait for user input. The user will need to visually inspect the frame and the detection results and classify it as *correct* by pressing Q, *incorrect* by pressing W, and *missed* by pressing E. The logs in the console will keep track of the total amount of each of the frame types. This is useful for more precise testing of the algorithm's accuracy.

Once the desired values are set, the experiment can be started by pressing the *Play* icon in the Unity Editor.