Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Tracking of real objects in VR

Filip Suchý

Supervisor: Ing. David Sedláček, Ph.D.
Field of study: Open Informatics
Subfield: Human-Computer Interaction
May 2021

Acknowledgements

I would like to thank my supervisor Ing. David Sedláček, Ph.D. for all his support and patience during the creation of this Master's thesis. I think it was his belief that allowed me to finish this Master's thesis in time.

I would like to thank CTU in Prague for providing me the best possible study facilities.

Also, I would like to give a huge thanks to my parents who gave me the possibility and all support necessary to study and make my dreams come true.

Last but not least thanks to my friends who lead me through my life and support me in my crazy ideas.

Declaration

Abstract

The goal of this Master's thesis is to provide easy to use real-time tracking system for casual objects in the real world such as pen and ruler. This tracking system has to be integrable into VR applications created in the Unity game engine. The performance and precision of the tracking system should be suitable for VR applications. Users should feel real objects in positions where they see them in VR.

Keywords: VR, Tracking, Depth camera, Unity

Supervisor: Ing. David Sedláček, Ph.D.
Praha 2, Karlovo náměstí 13, E-425

Abstrakt

Hlavním cílem této diplomové práce je poskytnout snadno použitelný trackovací systém pro běžné objekty v reálném světě jako jsou pero a pravítko. Tento trackovací systém musí být integrovatelný do VR aplikací vytvořených v herním enginu Unity. Výkon a přesnost trackovacího systému by měly být vhodné pro VR aplikace. Uživatelé by měli cítit objekty reálného světa na pozicích, kde je vidí ve VR.

Klíčová slova: VR, Trackování, Hloubková kamera, Unity

Překlad názvu: Sledování reálných objektů ve VR

Contents

Contents v

Figures vii

Tables

Project Specification 1

Introduction 3

1 Analysis 5

1.1 Requirements 5
1.1.1 Real-time processing 6
1.1.2 Precision 6
1.1.3 Real objects without adjustments 6
1.1.4 Compatibility with Unity 7
1.2 Depth Camera 7
1.2.1 Microsoft Azure Kinect 8
1.3 Similar works 10
1.3.1 Tangible VR Book 11
1.3.2 EfficientPose 11
1.4 Summary of Lazarek's work 12
1.5 Neural Networks 14
1.5.1 PointVoteNet 15
1.6 Iterative Closest Points 16
1.7 VRPN and WebSockets 17
1.7.1 VRPN 18
1.7.2 WebSockets 18

2 Suggested solution 21

2.1 Tracking system 23
2.1.1 Segmentation 23
2.1.2 6DoF pose estimation 25
2.2 VR Application 27
2.3 Communication between Unity and Tracking system 28
2.3.1 Tracker 29

2.3.2 Consumer 29

3 Implementation 31

3.1 Tracking system 31

3.1.1 Program overview 31

3.1.2 Segmentation 33

3.1.3 6DoF pose estimation 34

3.1.4 Networking 35

3.2 VR Application 36

3.2.1 TrackingManager script 37

3.3 Communication server 39

4 Testing 43

4.1 The scene 43

4.2 Performance of Tracking system 45

4.3 User experiance 46

4.3.1 Developing VR application 47

4.3.2 Experience with VR and Tracking 49

4.4 Stability 52

5 Conclusion 53

A Contents of enclosed SD card 55

Figures

1.1 Example of outputs from RGB and Depth cameras 7
1.2 Example of Point Cloud output from Depth camera 8
1.3 Schema of Microsoft Azure Kinect 8
1.4 FOV schema of Kinect cameras 10
1.5 FOV of Kinect cameras at distance of 2 meters 10
1.6 Prototypes in Tangible VR Book paper 11
1.7 Example of prediction from EfficientPose 12
1.8 The photo of tracked objects from Lazarek's work 13
1.9 Visualization of the pose estimation process 15
1.10 Object model and resulting segments for each point 16
1.11 Example of ICP from Open3D library [7] 17

2.1 Created models of Pen (left) and Ruler (right) 22
2.2 3D printed Pen (left) and Ruler (right) 22
2.3 RGB color space as 3D cube 24
2.4 HSV color space 24
2.5 Example of segmented points in point cloud 25
2.6 Very bad segmentation of detected object with outliers 26
2.7 Server-Client communication schema 29

3.1 Morphological closing operation 33
3.2 Morphological transformation using dilate operator 34
3.3 Example of the Scene hierarchy 37
3.4 Properties of TrackingManager 38
3.5 Communication between Tracker and Server 40
3.6 Communication between Consumer and Server 41

4.1 Photos of the scene setup 44
4.2 Another photos of the scene setup 44
4.3 Screenshot from the Tracking system 45
4.4 The scene in VR application 48
4.5 The calibration point in VR application 48
4.6 Screenshots from the VR application scene 49

4.7 Photos from testing of VR application 49
4.8 Photos from testing of VR application 50
4.9 Drawings in VR application 50
4.10 Drawings in VR application 51
4.11 Drawings in VR application 51
4.12 Drawings in VR application 51

Tables

1.1 Available modes for RGB camera in Kinect 9
1.2 Available modes for Depth camera in Kinect 9

4.1 Measured average FPS 46
4.2 Measured deviations in position and rotation 52

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Suchý**	Jméno: **Filip**	Osobní číslo: **420097**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra počítačové grafiky a interakce**		
Studijní program:	**Otevřená informatika**		
Specializace:	**Interakce člověka s počítačem**		

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Sledování reálných objektů ve virtuální realitě

Název diplomové práce anglicky:

Real object tracking in Virtual Reality

Pokyny pro vypracování:

Prostudujte problematiku sledování reálných objektů ve virtuální realitě (např. objekty běžné denní potřeby). Jedná se o metody založené buď na tradičních algoritmech počítačového vidění jako RANSAC, ICP nebo aktuální trend využívající neuronové sítě. Předpokládejte, že sledované objekty jsou snímány jednou RGB nebo RGBd kamerou.
Navrhněte možnost sledování základních kancelářských potřeb (pravítko, tužka), za účelem jejich využití ve virtuální realitě pro realizaci zážitku připomínajícím kreslení na papír. Metody tedy vyberte hlavně s ohledem na rychlost estimace pozice a orientace a jejich přesnost.
V případě potřeby navrhněte úpravy sledovaných objektů za účelem zlepšení rozpoznání (vlastní 3D tisk tvaru, barevné úpravy), tyto úpravy musí být replikovatelné pro případ ztráty původních předmětů.
Implementujte vybrané metody a změřte rychlost a kvalitu sledování např. v závislosti na zakrytí předmětu, změnách osvětlení, rychlosti pohybu, množství předmětů ve scéně.
Navrhněte a implementujte propojení s herním engine Unity pro vizualizaci sledovaných objektů, vytvořte demonstrační scénu a otestujte s minimálně pěti uživateli (hodnoťte věrohodnost práce s nástroji a zpoždění systému).

Seznam doporučené literatury:

1] Joseph J. LaViola, Jr. et all. 3D User Interfaces: Theory and Practice, second edition. 2017. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.
2] Practical Augmented Reality: Steve Aukstakalnis, 2017, Addison Wesley
3] Dieter Schmalstieg, and Tobias Hollerer, Augmented Reality: Principles and Practice (Usability), Addison Wesley 2016
4] T. Nakamura, 'Real-time 3-D object tracking using Kinect sensor,' 2011 IEEE International Conference on Robotics and Biomimetics, Karon Beach, Thailand, 2011, pp. 784-788
5] Virtual Reality Peripheral Network, https://github.com/vrpn (online)

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. David Sedláček, Ph.D., katedra počítačové grafiky a interakce FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **10.03.2021** Termín odevzdání diplomové práce: ______________

Platnost zadání diplomové práce: **19.02.2023**

[illegible] [illegible] [illegible]

Ing. David Sedláček, Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

Introduction

Nowadays VR (Virtual Reality) is growing and we can see it more often than before. It's visible mainly game/entertainment industry, but we can see VR used for training, education, health therapy, and much more. But to make VR successful technology used in many areas we need to pull users into the VR world without disturbing their experience. There are many aspects that influent the VR experience and one of them is interaction - how can users interact with objects in VR, how they can move, etc.

To provide basic interaction in VR we usually use controllers. They are specialized devices with a tracking system, buttons, joystick, and motors for haptic feedback. We can use these controllers in VR to track the position of hands and provide interaction like grabbing for example. With haptic feedback, we can feel if we hit something in VR or if we grab some object. This kind of interaction is good enough, but you can't feel actual objects in your hands, their weight, surface, etc. Also, you need to have these controllers still in your hands. In some areas, this kind of interaction can be limiting and cause problems with adaptation VR in these areas. One example can be the training of surgeons, where we need precise work with a scalpel, we need to have a real scalpel in our hands, feel it, and get used to it.

One way how to push the VR experience further is to add a possibility to interact with objects in the real world and bringing them into the virtual world, so you can naturally interact with objects in your real world and see them in the virtual world.

The goal of this Master's thesis is to try to find the solution to this type of interaction - natural interaction with real objects and bring them into VR. The biggest challenge in this interaction is to track the position and rotation of these objects without any special tracking things on them. Moreover, we need to track these objects in real-time so users can manipulate and see them naturally.

Chapter 1

Analysis

The goal of this Master's thesis is tracking of objects in real world and estimate **6DoF** pose (position/translation and rotation) of them. These information will be used in Unity [13] game engine to render virtual objects in VR at correct position and rotation. Because we want to use this tracking in VR application, one of the constraint will be real-time processing. The same problem was solving in Bachelor's thesis by Bc. Jan Lazarek [10] and this project will continue from his work.

For our tracking system, we will need to obtain some kind of information about the scene - a space where real objects and users are located. Because we want to track these objects without any adjustments or with just some small color/shape adjustments, there is probably only one solution on how to obtain necessary information - the visual image of the scene. We will use a camera to obtain a visual image of the scene, but estimating full 6DoF pose from just visual image could be challenging. Because of that, we will also use a depth camera to obtain information about distances of points in the visual image from the camera. With depth information, we can obtain 3D point cloud of the scene.

The whole process of obtaining 6DoF pose from visual image and depth image can be divided into two main parts:

1. Recognize required object in visual and depth data
2. Estimation of 6DoF pose from these data

1.1 Requirements

Because this project will be used in a real application, we have also some requirements. The most important and most challenging requirement is real-time processing. There are also other requirements but they aren't so important.

1.1.1 Real-time processing

What does real-time processing mean in this context? It depends on experiments.

We simply use the term **FPS** (**F**rames **P**er **S**econd) to define how many times in a second we obtain 6DoF pose about an object. For example, 3FPS will mean that we obtained 3 times 6DoF pose about object in one second. This allows us to measure performance in terms of real-time processing. More FPS will give us smoother movement of objects so more FPS means better performance in our case.

Many VR headsets such as Oculus Rift S [18] have a refresh rate of around 90Hz. If we have enough computing performance, we will render our VR application in 90FPS to match this refresh rate and make the user experience smooth enough. From this fact we can say that 90FPS will be real-time processing because for every rendered frame we will have correct 6DoF pose. But this number of FPS is very hard to achieve.

In this project, we will use a depth camera that has a 30FPS output, so we know, that our maximal performance (from point of view of FPS) will be 30FPS due to the limitation of depth camera sampling rate.

What's important is the user. We want for users a smooth feeling of moving with real objects in VR. So to verify if we satisfy real-time processing requirements, we need to test it with users. From experience, we can tell that less than 10FPS can't satisfy the real-time requirements.

1.1.2 Precision

Another important requirement is precision. Because we will use this tracking in VR application, our main goal for precision is feeling of a user that object is at position where he feel the object might be at. Another aspect of precision is stability - if there are some vibrance of objects or some driftring of position over time. This type of precision will be tested as an experience of user in VR application with tracking of the objects.

Less important is absolute precision. If we will know correct position and rotation of the object, we can measure how precisely we can estimate these information. This is what I called absolute precision and if we can achieve just small errors in this type of precision, previous precision based on user's feeling will be satisfied.

1.1.3 Real objects without adjustments

This requirement isn't as important as others. We need objects without any adjustments such as marks, because of the simplicity of using them in a real application. But if this will means that we can't achieve sufficient real-time performance, than some color coding or marks will be possible.

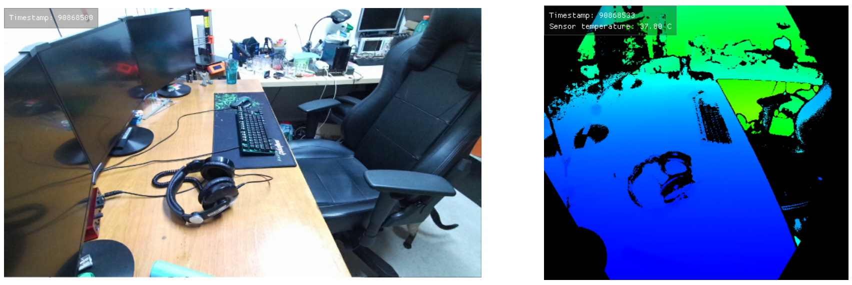
1.1.4 Compatibility with Unity

VR application which will use this tracking system will be make in Unity [13] game engine. This means that we need somehow connect output of our tracking system into Unity and also make sure, that we can correctly setup what will be needed to correctly use 6DoF pose from tracking system in Unity. It's possible that tracking of real objects will run on different machine than VR application, so we need to handle this case.

Also there can be more than just one instance (client) of running VR application and we will need 6DoF pose of tracking objects in all running client, so we also need to satisfy communication of tracking system with more clients.

1.2 Depth Camera

Because we want to track real objects, we need to see them in scene. There are probrably only one way how to do that - using camera. Camera gives us 2D picture of our scene - this will be very useful for detecting real objects in scene. To calculate 6DoF pose about detected object in scene, we need probably more information than just 2D picture. For this purpose depth camera gets handy. In short, depth camera is special device, which gives us information about distance between camera and object in font of this camera. We can imagine output from this type of camera as 2D picture, but instead of colors (like normal RGB camera has) we get distances. Example of output from normal RGB camera and Depth camera is shown on pictures (1.1).



(a) : RGB Camera **(b) :** Depth Camera

Figure 1.1: Example of outputs from RGB and Depth cameras

Each color in picture (1.1b) represents distance of this point from depth camera. This type of information helps us to calculate 6DoF pose of given object. Depth camera can give as also output in form of Point Cloud. It's a set of points in 3D space which camera can see. Positions of these points are relative to the camera and it's another type of information which we can get from this type of camera and it can help us in calculation of 6DoF pose. Example of point cloud given from depth camera is shown on picture (1.2)

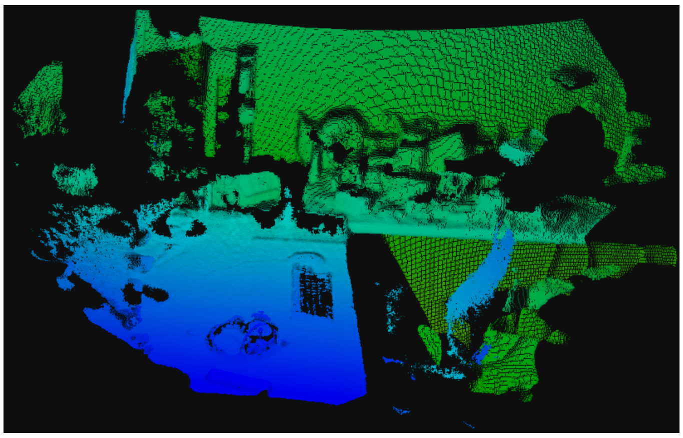


Figure 1.2: Example of Point Cloud output from Depth camera

Because this Master's thesis continues on work of Bc. Jan Lazarek, I will use the same depth camera as He did - Microsoft Azure Kinect [22].

1.2.1 Microsoft Azure Kinect

Microsoft Azure Kinect (hereinafter referred to as Kinect) is complete solution for developers with RGB camera, Depth camera, System of microphones, and IMU (**I**nternal **M**easurement **U**nit - used to track movement of device). Also one advantage of Kinect is that we can choose between several modes of cameras, so we can find mode that fits our requirements best. Simple schema of Kinect device is in the picture (1.3).

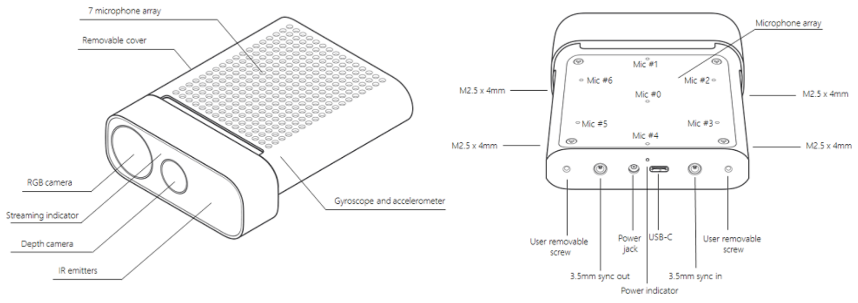


Figure 1.3: Schema of Microsoft Azure Kinect

For our purpose we will use only RGB and Depth camera. Depth camera in Kinect is based on ToF (Time-of-Flight) technology. This technology measure the time of flight of light signal to calculate distance. We can simply imagine it as light ray send from the camera. This ray will flight to the nearest surface where it will be reflected back to the camera. When camera detect this ray, it calculates time from sending ray to detecting ray and then calculate distance using simple equation $d = (t * c)/2$, where d is calculated distance in meters, t is time from sending ray to detecting ray in seconds, and c is constant of light speed in meters per second. We need divide it by 2 because ray travel to the object and back, so it's double the distance.

In table (1.1) we can see which modes of RGB camera we have available in Kinect.

Resolution	**Aspect Ratio**	**FOV**	**FPS**
3840 x 2160	16:9	90°x 59°	0, 5, 15, 30
2560 × 1440	16:9	90°× 59°	0, 5, 15, 30
1920 × 1080	16:9	90°× 59°	0, 5, 15, 30
1280 × 720	16:9	90°× 59°	0, 5, 15, 30
4096 × 3072	4:3	90°× 74,3°	0, 5, 15
2048 × 1536	4:3	90°× 74,3°	0, 5, 15, 30

Table 1.1: Available modes for RGB camera in Kinect

We can see that there are lots of options for RGB camera. Because we need real-time processing, modes with very high resolution such as **3840 x 2160** will be unusable for us. In table (1.2) we can see available modes of Depth camera in Kinect.

Mode	**Resolution**	**FoI**	**FPS**
NFOV unbinned	640 x 576	75°x 65°	0, 5, 15, 30
NFOV 2x2 binned (SW)	320 x 288	75°x 65°	0, 5, 15, 30
WFOV 2x2 binned	512 x 512	120°x 120°	0, 5, 15, 30
WFOV unbinned	1024 x 1024	120°x 120°	0, 5, 15

Table 1.2: Available modes for Depth camera in Kinect

In tables (1.1) and (1.2) we can see some unknown terms, so let me explain them:

- **NFOV** - **N**arrow **f**ield-**o**f-**v**iew depth mode
- **WFOV** - **W**ide **f**ield-**o**f-**v**iew depth mode
- **FOV** - **F**ield-**o**f-**v**iew
- **FoI** - **F**ield **o**f **I**nterest

How we can see in table (1.2) we can choose between 2 modes (NFOV, WFOV) with different FoI. This possibility is very handy and we can decide which mode will be better for our purpose on-the-fly without changing the actual camera.

In the picture (1.4) we can get better idea of different FOV/FoI settings and how they compare.

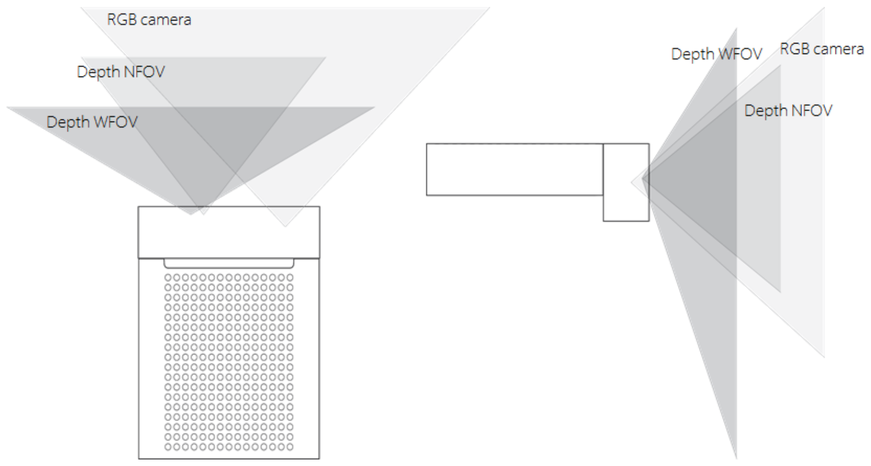


Figure 1.4: FOV schema of Kinect cameras

Another useful picture (1.5) demonstrates the camera's FOV as seen from the front at a distance of 2 meters.

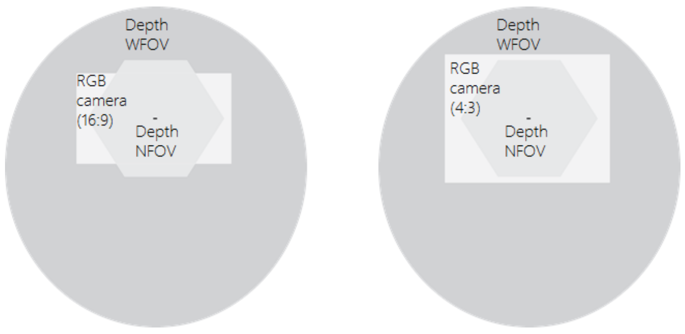


Figure 1.5: FOV of Kinect cameras at distance of 2 meters

Another features like microphones or IMU of Microsoft Azure Kinect aren't important, because we will not use them.

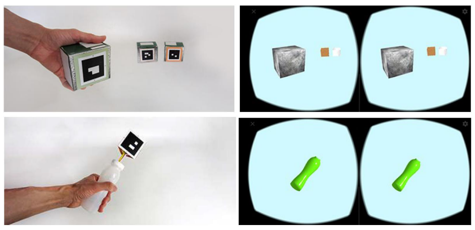
1.3 Similar works

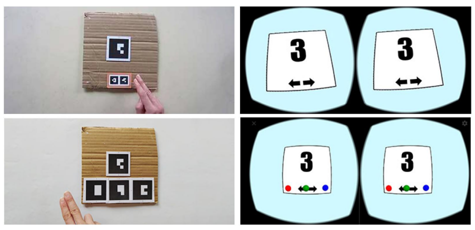
Tracking of real objects for using them in VR applications is nothing new. There were many works before that aims similar goals as this project. Because I'm continuing on work of Bc. Jan Lazarek I do not mention lots of other works in this thesis. I will detail describe Lazarek's work and also one paper that I found very interesting because of how close it is to my case and it's one of the state-of-the-art methods.

I also want to mention some other works and very briefly describe why I do not analyze them more in this thesis and why I have no more similar works there.

1.3.1 Tangible VR Book

One interesting paper is Tangible VR Book [12]. In this work authors also try to track real objects and they were prototyping the whole TUI (Tangible User Interface) framework. They created elements such as buttons, sliders, or books and also tries some objects like boxes. It was just prototyping, but it works well and pictures from their work are below.


(a) : boxes at the top, bottle in the bottom


(b) : button at the top, gestural slider in the bottom

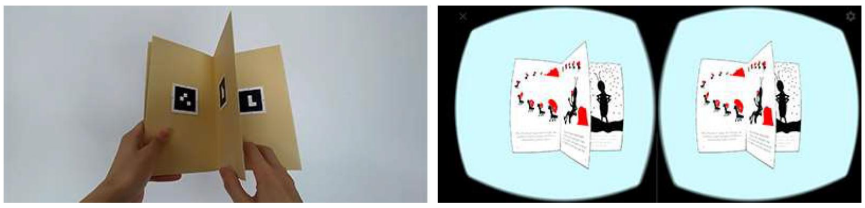

(c) : book

Figure 1.6: Prototypes in Tangible VR Book paper

The reason why I didn't describe details of this works is because of used technologies. They were using a smartphone camera and special markers to detect these objects. In our case, we want to detect objects without any special markers and be able to detect the 6DoF pose even with a partially covered object. We want to try to estimate the 6DoF pose from a depth camera where markers are no needed.

1.3.2 EfficientPose

EfficientPose [9] is a great example of the state-of-the-art 6DoF pose estimation. Their work is based on EfficientDet [11] which is the state-of-the-art neural-network-based detection of objects in RGB images. In their work, they extend EfficientDet by their own neural networks which can also determine 3D rotation and translation from a detected object from RGB image. Details of this work are beyond this thesis and for details of these neural networks, I rather recommend reading the original paper of EfficientPose [9].

With their neural networks they was able to precisely estimate 6DoF pose from two different datasets and also they achived performance around 26 FPS. Example of output from their algorithm is shown in the picture (1.7).

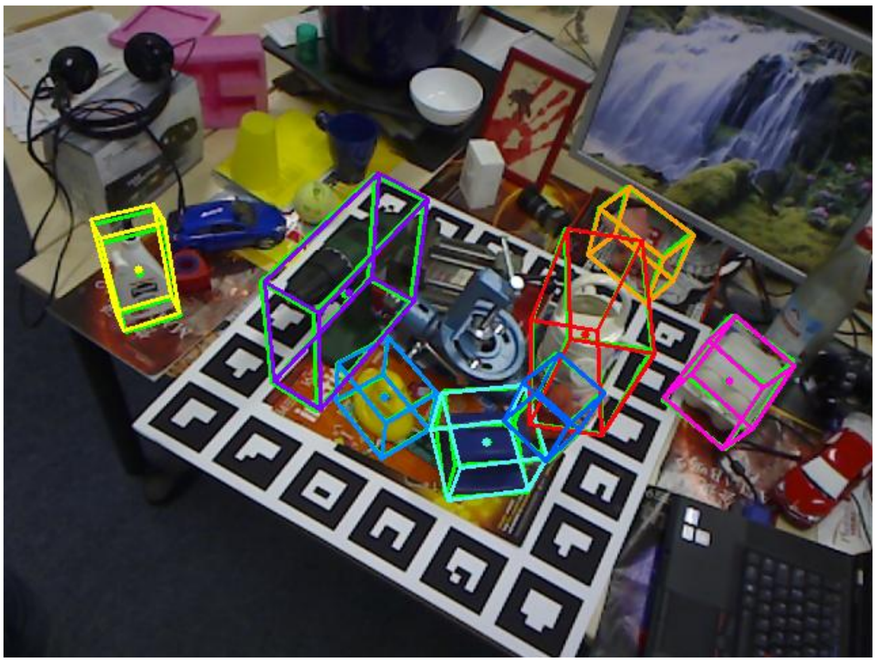


Figure 1.7: Example of prediction from EfficientPose

In the picture (1.7), green boxes represent ground truth and colored boxes represent the output of the EfficientPose algorithm. As we can see their results are impressive. The reason why I'm not describing works similar to this is that they are using only RGB images to directly estimate the 6DoF pose. Moreover, EfficientPose is based on neural networks, and to estimate the 6DoF pose of some object we need to train this algorithm to it. This means that we need lots of labeled data (images with 6DoF pose of required objects).

Compare to this approach we are trying to estimate the 6DoF pose of objects without learning, directly from depth data. This is the reason why we are using a depth camera.

1.4 Summary of Lazarek's work

Bc. Jan Lazarek did a great job in analyzing appropriate technologies for this task. You can learn more about depth cameras, why he choose Microsoft Azure Kinect and much more about image processing. More important for us is his solution to this task.

He describes how he prepare the scene. In his setup there was one depth camera Microsoft Azure Kinect and for VR he uses HTC Vive headset. Also he added LeapMotion on headset so he can track hands in VR application. To get correct position of depth camera in VR (and to properly calculate 6DoF pose about objects for VR application) he add HTC Vive tracker to the depth camera, so his VR headset can track depth camera position such as position of controllers for example. This was setup to track and run VR application and now what about tracked objects.

His task was to track objects like pen and triangle ruler. For his scene he prepared models of these object from colored paper. Colors on objects had an important role - color coding for detection in images using filtering pixels with the same color. This allows him to quickly detect objects in scene and filter depth data to area, where objects are located. In picture (1.8) you can see how these objects looks. The picture is from his work.

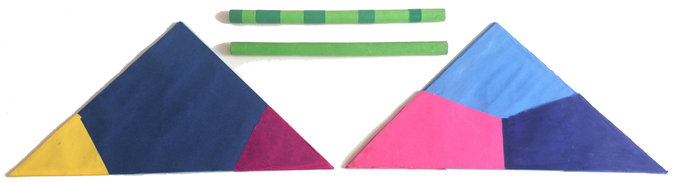


Figure 1.8: The photo of tracked objects from Lazarek's work

In his work, he used proved Open Source libraries for Image Processing and Point Cloud processing. These libraries allowes him quickly verify functionality of his algorithm without writting code which was already written by someone. These libraries (OpenCV [3] for example) are very handy and will be propably used also in this project.

The Lazarek's algorithm itself can be divided into two parts:

1. Find object in the scene

   The first step is to find our tracked object in the scene. He can use the fact, that he knows exactly the color of the object, so he used color information to filter depth data to only those, which fits the color. Because of light conditions color in the picture can be slightly different from the defined color. For this purpose, he used HSV color space to define the color range which will be accepted as the defined color of the object.

2. Fit object into depth data

   The second step is to fit object into filtered point cloud from the previous part. For this purpose he used Point Cloud Library and RANSAC algorithm. Point Cloud Library has predefined simple objects like cylinder and plane. Cylinder was used for pen and plane was used for triangle ruler. RANSAC algorithm can fit these objects into point cloud and specify, which points represents the objects and which are outliers. After RANSAC algorithm, he calculate center of gravity for those points, which was marked using RANSAC algorithm as points of given object. This center of gravity was used as position of the object. Rotation was used from RANSAC algorithm. For the triangle ruler, there was a problem of detecting which side is on top. For this purpose he has different color code for each side, so after the first part he knows which side of the triangle ruler is on top. In the picture (1.8) you can see two different triangle rulers - each one represents colors for each side. With this information and rotation from RANSAC algorithm he can calculate final rotation of the triangle ruler.

This algorithm gives him very good results, but also he mentions problems with calibration. Because he uses colors to find objects in the scene, there have to be good light conditions and colors have to be calibrated.

1.5 Neural Networks

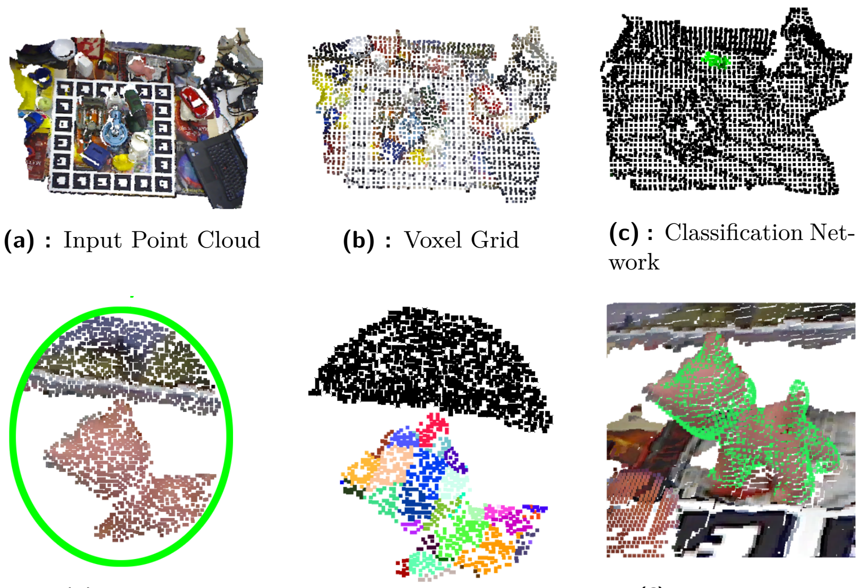
In past years neural networks get very popular and show us their power. They are often used for image processing/recognition, data prediction, and so on. The task of estimating 6DoF pose of objects isn't an exception. One example that I found is PointVoteNet [8].

It's learning-based method so it's trained on some set of data. There are also many others learning-based methods that estimate 6DoF pose of objects in the scene, but they use only RGB images without depth information. Some of them use depth information to refine estimated position from RGB image. PointVoteNet aims to find object in the scene and estimate its 6DoF pose directly from depth data (point cloud) even without RGB information. If there is also RGB information it's used at the end to extend depth information and estimate 6DoF pose more precisely.

How I wrote above, there are others learning-based methods for estimation 6DoF pose, but I do not describe them in this work, because they estimate 6DoF pose directly from RGB image without using depth information. One example of such algorithm is EfficientPose [9] described above. In our case we will be using depth infromation as primary source to estimate 6DoF pose of objects, so it looks unnecessary to describe different approach. I found PointVoteNet interesting to describe how it works, because it's algorithm, which uses depth information to estimate 6DoF pose.

1.5.1 PointVoteNet

The best way to understand how PointVoteNet works is to explain each steps of estimation process. In the picture (1.9) we can see process of estimating 6DoF pose in 6 pictures.



(a) : Input Point Cloud **(b) :** Voxel Grid **(c) :** Classification Network

(d) : Sorting **(e) :** Segment Prediction **(f) :** Pose Voting

Figure 1.9: Visualization of the pose estimation process

In picture (1.9a) we can see how input data looks like. It's unsorted point cloud (in this case also with RGB information) propably from some depth camera. To limit number of points, the scene is uniformly downsampled using voxel grid in PCL (**P**oint **C**loud **L**ibrary). This downsampling limits number of points to approximately 3000-5000 anchor points (It's showen in the picture (1.9b)).

In the next step they find the object using PointNet [5]. They sample 2048 points in the spherical neighborhood around each anchor point from the previous step and pass them into PointNet with only a single logistic output neuron. This gives him probability of the presence of the object. This is shown in the picture (1.9c) where green points shows presence of the object. In this point they found the object in the scene.

Next they sort anchor points by probability from the previous step and uses only the 16 highest scored. They use these 16 anchor points and their neighborhood and pass them into Neural Network inspired by PointNet to associate them to either background or to the corresponding point on the given object surface. After this step they have segmentetion of points. These segments correspond to segments on the input object shown in the picture (1.10).

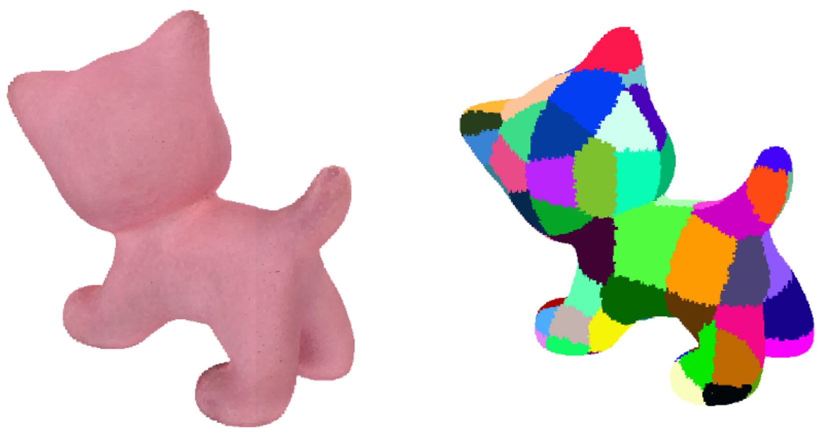


Figure 1.10: Object model and resulting segments for each point

Segmentation is shown in the picture (1.9e). After this step the problem is reduce to estimation of relative pose between the two point sets. For this purpose they use the rotational subgroup voting algorithm [6] to calculate full 6DoF transformation between object and the scene. After that the 16 poses for each of the processed anchors are refined using a coarse-to-fine ICP [1].

At the end of the whole process the final position of the object is estimated from these 16 poses of anchor points from the previous step. At first they transform object into the scene using estimated pose. After that they cut off points lying behind the scene data relative to the camera viewing axis and remaining points are paired with the closest points in the scene. They compute geometric and color (if RGB data are available) loss and then they find transformation which minimize this loss. Details are in PointVoteNet [8].

1.6 Iterative Closest Points

Iterative Closest Points (ICP) is widely used algorithm of rigid registration. We have two point clouds, one is called source and the second is called target and we want to estimate transformation (translation and rotation) from source to target such that if we apply this transformation on the source point cloud it will match the target point cloud. ICP solves this problem and it is guaranteed that it will converge. Moreover one big advantage of ICP is that we don't need to know correspondences between points in source and target point clouds.

In the picture (1.11) we can see two point clouds before using ICP and after using ICP to find transfromation between two point clouds.

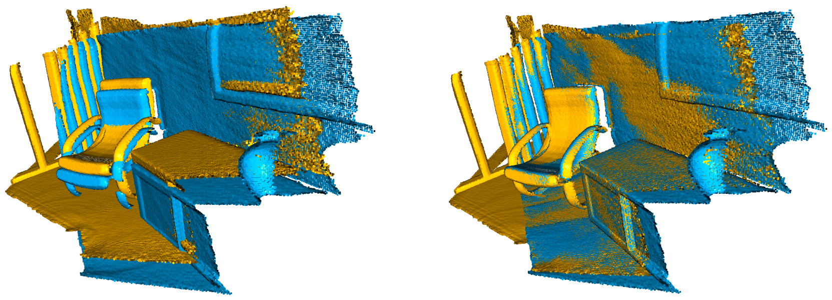


Figure 1.11: Example of ICP from Open3D library [7]

How ICP algorithm works can be described in two simple steps:

1. Find correspondences

   First step of iteration is to find correspondences between source and target point clouds, whereas source point cloud is transformed using current transformation matrix. This step give use set of correspondence points between two point clouds.

2. Update transformation

   Using set of correspondence points from previous step update transformation matrix in the way that updated transformation matrix minimizes given objective function defined over set of correspondence points.

These two steps described above are repeated until convergence or some constraints are satisfied. Constraints could be for example maximal number of iterations or minimal difference between two point clouds. Difference between two point clouds can be defined using RMSE (Root Mean Square Error). There are many variants of ICP algorithm, which can differ between each other using different objective function. Two common ICP variants are **Point-to-Point ICP** [1] and **Point-to-Plane ICP** [2].

ICP is very powerful algorithm that's reason why it is used in so many cases. But ICP is only local optimization algorithm. This means that if you have two point clouds, you have to find rough alignment of them before using ICP. From the description above, in the first step of ICP it's trying to find set of correspondence points on two point clouds using current transformation matrix, so some initial transformation matrix have to be given.

1.7 VRPN and WebSockets

One of a requirement for this project is compatibility with Unity game engine. Because of character of this requirement there have to be probably some network based communication between tracking system and VR application created in Unity.

There are many possibilities how to solve this communication, but from my point of view there are two simple-enough options - **VRPN** [23] and **WebSockets** [4].

1.7.1 VRPN

Virtual **R**eality **P**eripheral **N**etwork (VRPN) is a device-independent, network-based interface for communication with peripherals for virtual reality in VR applications and games. Purpose of VRPN is to provide unified interface for input devices like trackers and controllers with these features:

1. Time-stamping of data
2. Multiple simultaneous access to peripheral devices
3. Automatic re-connectin of failed servers
4. Storage and playback of sessions

VRPN consist of server which communicates with devices and clients which communicates with server to get information about devices. VRPN is written in C++ programming language, but has also wappers in C#, Python and Java.

One advantage of VRPN is that it's supported by Unity game engine out-of-box and no 3rd party plugins are needed. Unity provides access to VRPN devices using interface called **Cluster Input** and it's very easy to use. On the other hand, one disadvantage is that for our tracking system, we have to extend VRPN server by writing device driver in C++ to communicate with our tracking system and obtain data in correct format.

1.7.2 WebSockets

WebSocket is communications protocol providing full-duplex communication over TCP connection. WebSocket protocol is supported by web browsers, but can be used in any client-server application. Libraries for WebSocket can be found in many different programming languages such as C/C++, Python, JavaScript, C#, Java, etc. For our purpose and simplicity, there is library called Socket.IO [14] which uses WebSocket protocol as a transport, but gives us more simple approach than WebSocket.

Socket.IO is event-based communication which allows us create custom events (you can imagine these events as messages of different type) and use JSON as payload of these events. For example in plain WebSocket protocol, we can send message/data. If we want to send two different messages like *position* and *rotation*, we have to somehow encode this into one general message, which will be send over WebSocket. On the other side we have to decode this general message and decide if it's *position* or *rotation* message. Socket.IO library solved this for us, so we can emit message *position* and *rotation* and on the other side we simply add event listeners for *position* message and *rotation* message separatly.

Chapter 2

Suggested solution

This Master's thesis continues on work of Bc. Jan Lazarek so I wanted to improve his solution using some inspiration from other works, mentioned in chapter 1. Some papers mentioned in chapter 1 uses neural networks, which are complicated and requires lots of training data. For example PointVoteNet also uses neural networks, but these are used for recognition of object in 3D scene and not for final 6DoF pose estimation. This brought me an idea, that for this project I want to try find some simple solution just for 6DoF pose estimation - simple solution, which could be fast because of simplicity. One solution which was used in many other works is using ICP. So I decided to use core idea of Bc. Jan Lazarek's work but with ICP at the end of 6DoF pose estimation.

One assumption which I could used in this project was that I will be tracking two different objects - **Pen** and **Ruler**. Because ICP is used to align two point clouds, I wanted to have accurate 3D models of pen and ruler which I will be tracking, so I can align 3D model (represented as a point cloud) to real corresponding object. Bc. Jan Lazarek uses approximation of cylinder for pen and plane for ruler.

Picture (2.1) shows created models of **Pen** (left) and **Ruler** (right) which will be used in this project as objects which we will be tracking. These models are simple, but also have some details which can help us correctly align 3D model with point cloud. For example pen has no body in shape of cylinder but has slightly triangular shape, so rotation in axis of pen cannot be arbitrary. In case of ruler, one side is completely flat, but the other side has some shape on borders to distinguish what is top side and what is bottom side of ruler.

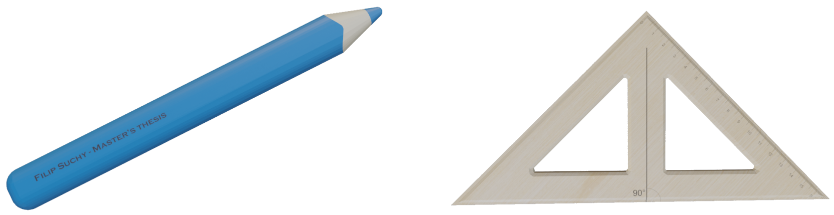


Figure 2.1: Created models of Pen (left) and Ruler (right)

Because we need accurate 3D models to real objects which we will be tracking, I used these created models from picture (2.1) and print them on 3D printer from plastic material. This method gives us very accurate real objects to 3D models. Also because we use 3D printing to create real objects, we can use different materials (filaments) with different color to help detect these objects in the scene based on color. I choose blue color for pen and green color for ruler. 3D printed objects of pen and ruler are shown in the picture (2.2). As we can see colors of these objects are specific and could be pretty well detected based on their colors.

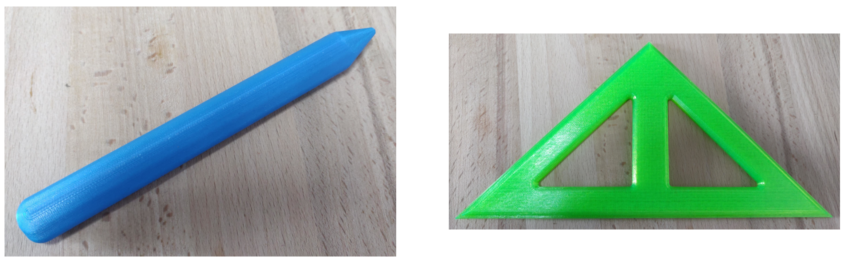


Figure 2.2: 3D printed Pen (left) and Ruler (right)

I can divide whole solution of this project into 3 main pieces:

1. Tracking system
2. VR Application
3. Communication between Unity and Tracking system

Tracking system and VR Application are independent parts which are connected together with Communication between Unity and Tracking system. There are some reasons for having Tracking system independent from VR Application. One of this reason performance. Processing depth data to estimate 6DoF pose could use lots of processing power. To get stable and high performance of Tracking system, we could run it separate machine dedicated just for this purpose. Another reason is multi-client desing described in requirements in chapter 1. If we will have more instances of the same VR Application (multiplayer) we want only one tracking system which will be common for all instances. Last but not least reason to having Tracking system independent is possibility to use different technologies/programming languages. We can use Unity and C# for creating VR Application while using different technologies for Tracking system which will suites given problem better.

2.1 Tracking system

Whole tracking system can be split into two main parts:

1. Segmentation
2. 6DoF pose estimation

2.1.1 Segmentation

For the next part of Tracking system (6DoF pose estimation) we need only depth data which correspond to the real object which we are tracking. This is problem of segmentation where we split data into multiple segments. In our case we want to segment data into 3 main segments - **Pen**, **Ruler** and **Uninteresting**. For this purpose I will use RGB image from camera to find objects (pen and ruler) and then use information about segmentation in RGB image to segment also depth data - point cloud.

Segmentation is process-intensive algorithm which I'll simplify as much as I can to save some processing power for 6DoF pose estimation. So in my case I will use only segmentation based on color. Because I can create objects which I will be tracking, I can create them in some specific color. Then I'll use this information about color to filter only pixels in RGB image which correspond to color of tracking objects.

Image in RGB color space is not good for color segmentation. We can imagine RGB color space as cube where each axis correspond to one color (red, green, blue). Figure (2.3) shows how RGB color space looks like. Problem with RGB color space is how colors are represented. If we want to select only one color (for example green) in image representing object in a real world, this color will be in some range (green object will be represented with some shades of green). Problem is that if you simply select color by range on each color axis in RGB color space, you will select sub-cube of whole color space, but this sub-cube will contains also colors, which are far from required color range.

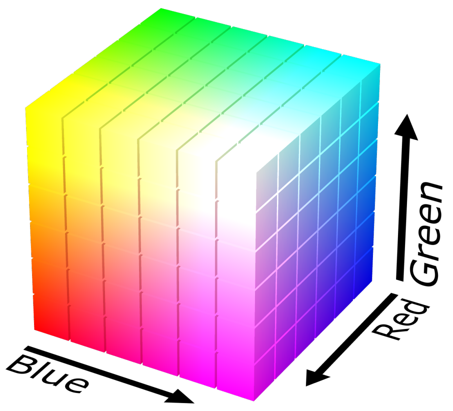


Figure 2.3: RGB color space as 3D cube

For this purpose HSV (**H**ue **S**aturation **V**alue) color space was created. In this color space, each color is represented by color hue (red, blue, etc), saturation of this color, where lower saturation means more "grayness" color, and value which represents brightness of the color. How HSV color spaces looks like is shown in the picture (2.4).

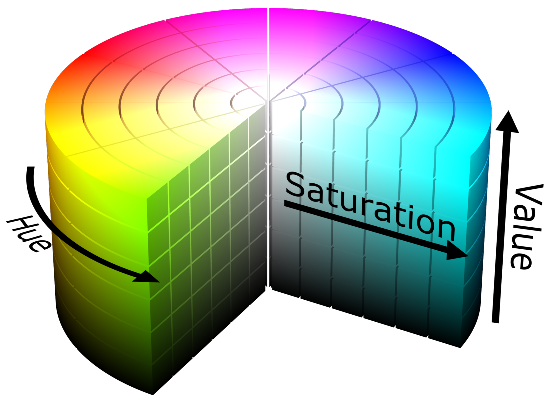


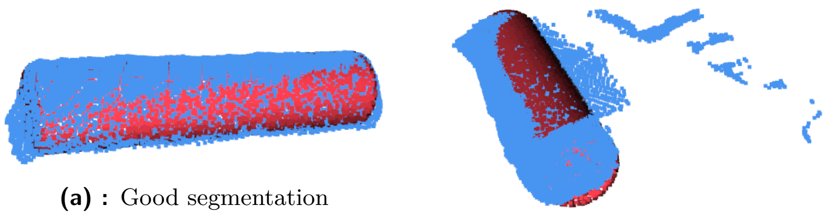
Figure 2.4: HSV color space

This color space is much more suitable for color segmentation, because we can use simple ranges in each color axis and it will give us exactly color range, which we want. So for segmentation of objects in image I will convert this image into HSV color space and then I will use simple range in HSV to select colors coresponding to pen or ruler.

This type of image segmentation is not so process-intensive and to save even more processing power I will downscale image to smaller resolution. In our purpose it's not necessary to have segmentation in high resolution and it can save some processing time. After segmentation in RGB image I can calculate corresponding 3D points in point cloud and use only these corresponding points in the next part of Tracking system - 6DoF pose estimation.

2.1.2 6DoF pose estimation

From the previous part we obtain point cloud, which correspond to detected object. Segmentation in the previous part is not good enough to obtain only 3D points of required object. There are some outliers which causes problems in 6DoF pose estimation. A good example of such segmentation is in the picture (2.5). Blue points represents segmented point cloud and red points represents model of detected object. In the picture (2.5a) we can see pretty good segmentation, but in the picture (2.5b) we can see very bad example of segmentation. How I wrote in chapter 1, very popular algorithm for point cloud alignment is ICP, but this algorithm is only for local optimization. If we look at picture (2.5a), we can immediately see, that center of segmented point cloud will represent center of model which we want to align, so we can align centers of two point clouds and we have initial transformation necessary for ICP algorithm. But if we look at picture (2.5b), we can see that this method will not work correctly, because center of the blue point cloud is actualy outside of required object.


(a) : Good segmentation

(b) : Segmentation with outliers

Figure 2.5: Example of segmented points in point cloud

In the picture (2.6) we can see very bad segmentation where there is lots of outliers and these outliers are far away from required object. In this picture we can also see that model represented by red points is not correctly aligned.

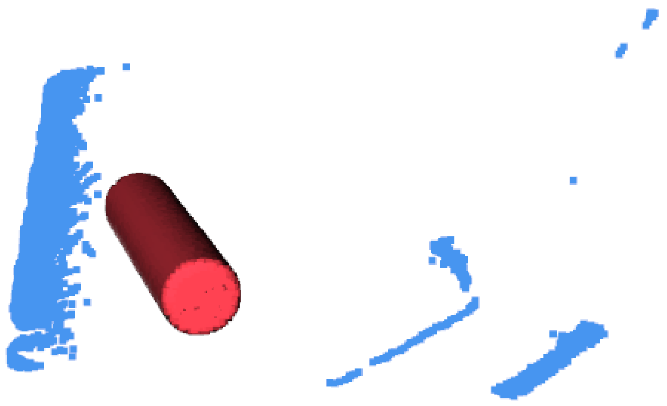


Figure 2.6: Very bad segmentation of detected object with outliers

This type of bad segmentation is very often if we use only segmentation from RGB image for segmentation of point cloud data. Bc. Jan Lazarek used in his work RANSAC algorithm, which can detect outliers and not use them. Problem with RANSAC is time to process such 6DoF pose estimation.

Because of character of our usage of Tracking system, we can put some assumptions to solve this issue in a simple way. Here are two assumptions which I'll use:

1. Shape of tracking objects
2. Continuous movement

We will be tracking **Pen** and **Ruler**. These objects are very simple, small in size and we can minimize area of interest (AoI) to sphere with radius of half of the maximum dimension of this object. AoI is an area where we know that object is in. If we use such AoI as described before, we can minimize outliers in such way, that center of points in AoI will be close to center of required object. This is very simple way how to improve our segmentation in point cloud.

Another assumption is in movement of objects. We know that these objects will be used on table to draw pictures, so we can say that speed of movement of these objects will be limited to some small value and also it is very unlikely that objects will disappear in some position and shows up on a different position. So we can use estimeted 6DoF pose from previous frame to obtain AoI in the current frame using sphere around previous center of estimated object position.

If we combine segmentation from RGB image with AoI obtained from previous frame, we get pretty good segmentation of tracking objects in point cloud. How I describe above, if we have good segmentation and simple objects like pen and ruler, we can calculate initial transformation for ICP algorithm just by align centers of two point clouds. This type of initial transformation is missing rotation, it's just translation. To obtain even better initial transformation we can use rotation from previous estimated 6DoF pose of tracking object. After obtaining such initial transformation we can simply use ICP algorithm to fine-tune 6DoF pose of detected object.

2.2 VR Application

The subject of this work is not VR application itself, but we need to integrate our tracking system into VR application and make it work correctly. To get data from tracking system we will use some type of communication, in our case it will be simple communication based on Socket.IO library. Using this communication we will get information about which objects our tracking system see and what is 6DoF pose of these objects. One problem is that this 6DoF pose information is relative to the camera.

I want to allow more tracking systems (for example because of performance) in one VR application and make them work very simple for developer of VR application. For this purpose I will create some type of Tracking manager in Unity which will handle all necessary things for the developer. This manager will automatically create objects which are detected from trackers and also update their position. To handle relative 6DoF pose information, for each tracking system manager will create empty object which will be representing tracking camera. All tracked objects will be then placed relative to this "virtual" tracking camera, so we can use directly 6DoF pose from tracking system.

One problem to solve is position and rotation of "virtual" tracking camera in the scene. For this purpose I will define one point (calibration point) in the scene, which I will use for calculating position of "virtual" camera. This point is defined by 3D position and normal vector. In the tracking system, I will define this point in a real world and then use it to calculate position and rotation of "virtual" camera in the scene. To obtain calibration point from a real world, I will use the fact, that I want to track objects on the table. I can grap 3 points on the table from depth camera to calculate normal vector of the plane which represents top of the table. Then I can grap point form depth camera which is our "calibration" point and use normal vector of calculated plane as a normal vector of the calibration point. This calibration point will be send from tracking system into our VR application and then we will position "virtual" camera to match calibration points from tracking system and VR application. This is a very simple way how to correctly position objects from tracking system without using another device, such as tracking device of VR headset.

Another problem to solve is position of VR headset. We can now correctly place tracking objects from our tracking system into the scene in a place where we want to have them. But we also need to place user to the correct place in the scene in such way, that he can feel real objects in a place where he see it in the scene. Fortunately this can be solved very easily. We can use one of controllers of VR headset, place it in real world on our calibration point and then match position of controller with calibration point in our scene. For this purpose we can place whole scene into one empty object and then move this object to align whole scene even with our tracking system. This is a simple way how to align scene with VR headset and how to align tracking system into scene.

2.3 Communication between Unity and Tracking system

Communication between Unity and Tracking system will be solved using Socket.IO library. This library gives us very simple, event based, bidirectional network communication and we can only focus on messages which we want to send. How I wrote before I want to enable multi-client communication and also enable more than one tracking system. For this purpose I will create standalone server, which will accept connection from both - tracking system and VR application. VR application will be called **consumer**, because it only consumes data from tracking systems. Tracking system will be called **tracker**. So our server will recognize two type of clients (meaning Socket.IO clients) - consumers and trackers. Each client, after connection to server will send message where he specify which type of client he is.

Advantage of scheme of one server and multiple clients is that we can handle lost connections of clients and also very easily synchornize data between more trackers and consumers, because everything goes over one point - server.

In the picture (2.7) we can see simple schema of communication. **Tracker** will be our tracking system and **consumers** will be VR applications using information from **tracker**. We can see that clients communicate only over server, so if some clients lost connection, other clients doesn't need to handle it and it will not cause crash of them.

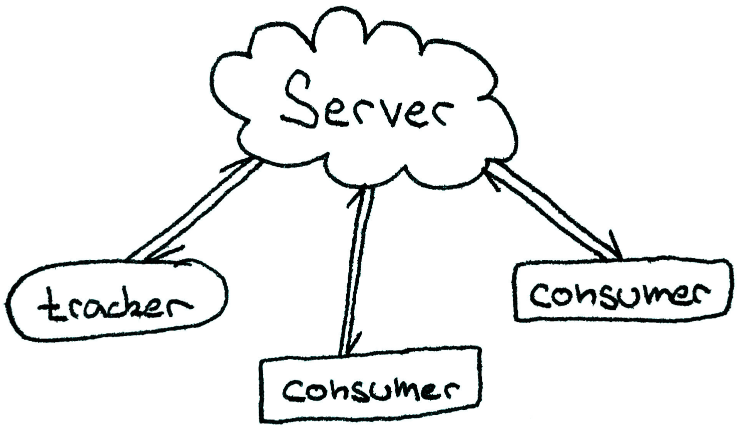


Figure 2.7: Server-Client communication schema

2.3.1 Tracker

Communication of Tracker is very simple. After connection to the server, Tracker register itself as **tracker client** and then it will sends data about itself to the server. No information about consumers or other trackers is needed, so communication with server is simply informing about current state of tracker itself.

One information which tracker can send to the server is calibration point. After tracker setup or change calibration point, sends message about this point (position and normal vector) to the server. Server will save this information so if some consumer will need this information, server sends it.

Tracker will send to the server another two types of information. The first type is registration of object which tracker will be tracking. This object is called **trackable** and for example in our case, tracking system can inform server about two trackables - pen and ruler. Each trackable has unique ID and some name, which will be used in VR application to assing some prefab for objects of given name. The second type of information is update of trackable. Whenever tracking system obtain some update for given object, it will send this information to the server. It could be update of translation and rotaion, but also information that trackable was lost and it's no longer tracked.

2.3.2 Consumer

Communication of Consumer is more bidirectional than in case of Tracker. After connection to the server, Consumer register itself as **consumer client**. After that, consumer will ask for trackables which server know about and also for calibration points from trackers. Server will reply to consumer with list of trackables and calibration points. This is initial "conversation" between server and consumer. After this point, just updates from server will be send to consumer. If consumer losts some information like list of trackables or calibration points, it can again asks server for sending these information.

After initial "conversation", if server receives some kind of information from trackers, automatically inform all consumers about this information. So all consumers will be receiving updates of 6DoF pose of trackables, information about lost trackables, etc. It's up to consumer, which information he want to use and which he will be ignoring.

Chapter 3

Implementation

As I describe in chapter 2 whole solution of this Master's thesis is divided into 3 pieces and each of them is implemented separately, so in this chapter I will describe implementation of these pieces separately.

3.1 Tracking system

For implementation of Tracking system, I choose the Python programming language. It's a high-level language, which allows me to focus on the algorithm itself instead of the implementation of structures, memory management, etc. Price for that is performance, but in Python, there are lots of very powerful libraries, which are written in C/C++ and has only bindings to Python. So if I use functions from these libraries instead of implementing everything by myself in Python, I can obtain very good performance.

For general array manipulation and calculation, I used Numpy [24] and PyTorch [15] libraries. For image processing, some visualization, and image segmentation I used the OpenCV library [3], which was also used in Lazarek's work. Finally, for point cloud processing, I used the Open3D [7] library.

Because of using libraries in Python, I will not describe some details of used algorithms/calculations. It's beyond the scope of this work and some details can be found in the documentation of these libraries. Also, my goal is to continue on work of Bc. Jan Lazarek and try to improve performance in a simple way and no to dive into mathematical/algorithmic details.

3.1.1 Program overview

Whole tracking program is divided into 8 Python files: hsv_calibration.py, main.py, network.py, player.py, pose.py, recorder.py, segmentation.py and utils .py. In files pose.py and segmentation.py there are classes for pose estimation and for image segmentation. In file network.py we can find class for communication with **Communication server**. File hsv_calibration.py is a simple util program which allows you to find correct range values for color segmentation using real-time visualization of results.

Files player.py and recorder.py are simple util programs for recording and playing data from depth camera. This could be used to record some situation and use it as an example. Tracking system can also use player instead of real depth camera, so you can use this for showcase with recorded scene or for measurement of performance on different machines with same input.

Important file is main.py, where you can find the whole implementation of this project. In this file, you can find method main, which is the entry point of the program. There are also another methods which are used in main method to simplify reading of this method. Some utility methods are placed in file utils.py.

At the beginning of main method, there is initialization of Network communication, Microsoft Azure Kinect camera, visualization, also models which will be tracked are loaded and some constants are defined here. After some initialization program shows image from camera and try to obtain calibration point by clicking using mouse into this image. After calibration point is gathered, program will send it to the server. Important part of main method is while loop. In each cycle of this loop one frame from camera is processed. At the beginning of this loop we capture one frame from the camera. There are RGB and Depth data in this frame. After capturing frame, we downscaled RGB image, convert it to the HSV color space and call method for image segmentation. This is done for each tracked object separately - so for pen and ruler there are 2 segmentations. In pseudo-code (3.1) this is represented by methods downscale_frame_and_convert_to_hsv and get_segmentation_from_image. After segmentation we use it to obtain depth data from frame and then try to estimate 6DoF pose from these data. This is represented by pose_estimation method in pseudo-code (3.1). This process also use filtering in depth data using previous pose of object and sprehe how it was described in chapter 2. After 6DoF pose is estimated, we calculate translation and rotation and send them as update information to the server. There is also detection of losing object from the scene. If object is lost, then information about lost object is send to the server instead of new translation and rotation. At the end of while loop there are visualization of estimated pose and check for pressing **Esc** key on keyboard to break while loop and end the program. After while loop, there are some cleaning code such as disconnecting from server, closing camera session, etc.

Whole program pseudo-code is shown in code (3.1).

```
def main():
  initialization()
  obtain_calibration_point()
  send_calibration_point()

  while True:
    frame = capture_frame_from_camera()
    downscaled = downscale_frame_and_convert_to_hsv(frame)

    segmentation = get_segmentation_from_image(downscaled)
    pose = pose_estimation(frame, segmentation)

    translation = get_translation(pose)
    rotation = get_rotation(pose)
    send_to_server(translation, rotation)

    visualization()
    if exit:
      break

  cleanup_work()
```

Listing 3.1: Pseudo-code of whole program

3.1.2 Segmentation

Segmentation of required object from image in HSV color space is implemented in class Segmentation in file segmentation.py. Implementation is very easy thanks to OpenCV library which was used for this purpose. Class Segmentation has only two methods. The first method is constructor of this class where you specify range of HSV segmentation. This range can be obtained using program in hsv_calibration.py. The second method is process which expects image in HSV color space as input and returns image with the same resolution as input image called mask, where black pixels represents pixels not corresponding to detected object and white pixels represents pixels corresponding to detected object.

To obtain such mask from image, inRange method from OpenCV library is used. Because there can be some noise in picture, I apply morphological transformation using closing operator. In short, this operation removes noise and make segmented objects nice and closed, which is necessary for our purpose. In figure (3.1) we can see example of this operation.

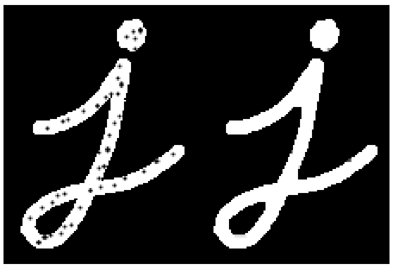


Figure 3.1: Morphological closing operation

Because detected area could be smaller due to noise and light conditions than area which correspond to the detected object, I use also dilate morphological transformation to dilate this area to make sure, that edges of detected object will be present in our final mask. In figure (3.2) we can see example of dilate morphological transformation.

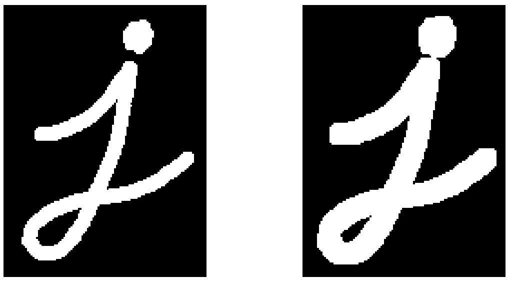

(a) : before dilate **(b) :** after dilate

Figure 3.2: Morphological transformation using dilate operator

3.1.3 6DoF pose estimation

6DoF pose estimation of detected object from depth data is implemented in class PoseEstimation in file pose.py. In this class I'm using Open3D library to manipulate with point clouds and run ICP algorithm to estimate 6DoF pose. In this class we can find 5 methods. The first method is constructor of this class where you specify parameters for pose estimation:

1. voxel_size

   This parameter is used to resample point cloud to the same voxel size as models (which we trying to align with obtained point cloud) and ideally to reduce amount of points in point cloud by using large voxel_size, but also small enough to keep necessary detail of models. This parameter depend on model size and detail.

2. icp_threshold

   This parameter is used to define maximal distance between correspondence points in ICP algorithm. ICP algorithm tries to find correspondecne points and then minimize some objective function between these points using transformation.

3. icp_max_iterations

   This parameter represents maximum iterations of ICP algorithm. If algorithm do not converge in icp_max_iterations it will end after icp_max_iterations and return current transformation.

4. filter_radius

   This parameter is used to define radius of sphere with center in previous detected pose which will be used to filter points in point cloud which are outside of this sphere.

There are also two simple methods get_target which returns current point cloud used to 6DoF pose estimation and is_lost method which returns True if object is lost for tracking (and estimating of 6DoF pose) or False if estimated 6DoF pose is correct and object is visible. Another simple method is filter_points_around_center which is used by process method to filter points in point cloud around sphere defined by previous 6DoF pose and filter_radius.

The main method of this class is process which expects two input parameters:

1. source

   Which is Open3D's Point Cloud instance representing model of tracked object. This model will be aligned with given point cloud.

2. points

   Which is array of points from depth camera representing point cloud of detected object.

First step of process method is filtering points in point cloud if there is previous estimation of 6DoF pose to remove outliers from point cloud. After this we use Open3D library to create Open3D's Point Cloud from these points and initial transformation using previous pose is created. If distance between previous pose and current center of point cloud is too large, we set flag representing that object is lost to True. This is simple detection of losing object.

After these steps ICP algorithm (Point-to-Point variant of ICP) from Open3D library is called and found transformation is returned.

3.1.4 Networking

Communication with server is implemented in class Network in file network.py. This class is using Socket.IO library and implement a few simple method. First method is constructor of this class, where you specify host and port of running server. To connect or disconnect from server, there are simple methods connect and disconnect.

Three other methods are present in class Network:

1. add_object

   This method expects id and name as input parameters and its purpose is to inform server about object which will be tracked with this tracking system.

2. update_object

   This methods expects name of object, which pose we want to update, position which is 3D vector of current position, rotation which is Quaternion of current rotation and lost which is flag if tracked object is currently lost or not.

3. send_calibration

   This method is used to send calibration point to server which will be used to correctly place tracked objects into the scene. Method expects 3 input parameters n which is normal vector to the plane on which calibration point is placed, r0 which is one point used to calculate normal vector of the plane and c which is our calibration point on the plane.

3.2 VR Application

As I mentioned many times before, I will be using Unity [13] game engine for creating VR application and also my goal is to make Tracking system compatible with Unity game engine. No other engines are supported/tested. To create VR application in Unity I will be using SteamVR [17] library. Because I need to "synchronize" position of the scene, VR headset and Tracking system, I will be using controller and so implementation of using controller is fit to SteamVR library. With another library for VR my implementation will not work.

Another library which I will be using is Socket.IO for C# [20]. This library is available using NuGet packaget manager [21] and I will use it for network communication with **Communication server**.

Whole implementation of integration with Unity is written in single C# script TrackingManager.cs. There are also two prefabs connected with this script - **TrackingManager** and **TrackingCamera**. Both of these prefabs are empty objects. **TrackingManager** prefab has TrackingManager script bind on it and it's the most important prefab for developers. If you want to integrate Tracking system into your scene in Unity, you just add **TrackingManager** prefab into the scene. The second prefab - **TrackingCamera** - is used by TrackingManager script to create "virtual" camera of tracking system. All tracked objects of this tracking system will be under **TrackingCamera** object as childs and positions and rotations from tracking system will be updated relative to this **TrackingCamera**. Position of **TrackingCamera** in the scene is determined using calibration point from tracking system and given origin in the scene - it is point in the scene which is represented by calibration point in the real world.

Because we need to synchronize positions of the scene, VR headset and Tracking system, we have one requirement for the scene hierarchy. Whole scene has to be under one object called **Room** (it's not necessary to name it Room in scene hierarchy). This room will be moved to correct. Prefab **TrackingManager** has to be at root of hierarchy. In the picture (3.3) we can see how such a scene hierarchy could looks like.

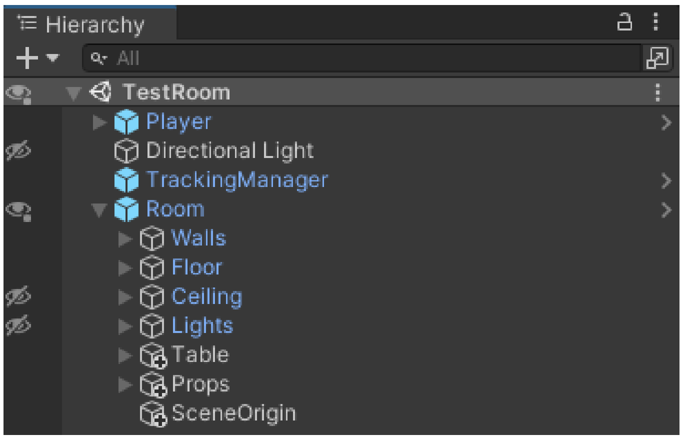


Figure 3.3: Example of the Scene hierarchy

3.2.1 TrackingManager script

TrackingManager script has a few properties which are configured in an Inspector window in Unity editor. If you put **TrackingManager** prefab into the scene, you will find these properties in an Inspector window under TrackingManager and it's necessary to configure these properties correctly to make Tracking system works.

All properties and their description is in the list below

1. host

   This is host (IP address) where Communication server is running. Default is http://localhost

2. port

   This is port of the Communication server. Default value is 3000

3. trackingCameraPrefab

   This is prefab used to create "virtual" camera of the tracking system. Defaul prefab is **TrackingCamera** which is empty object, but you can use you own prefab if you want (for example with some camera object in it to show where camera is placed in the scene).

4. sceneOrigin

   This is important property. This property is a Transform object of the calibration point in the scene. This point will be aligned with the calibration point given by tracking system. You can use some empty object placed and rotate in a correct position in the scene.

5. trackableDefinition

   This is another very important property. It's a list of trackable objects. For example lets assume, that our tracking system will track Pen and

Ruler. Our tracking system will register this objects under names pen and ruler. If we want to show these objects in our VR application, we add two elements into trackableDefinition property. Each element has another two properties - name and prefab. name is the same name as tracking system used to determine type of tracked object - in our example it's pen or ruler. prefab is Prefab representing this object in our VR application. If tracking system sends information about pen or ruler object, TrackingManager automatically instantiate prefab for given object and place it into correct position.

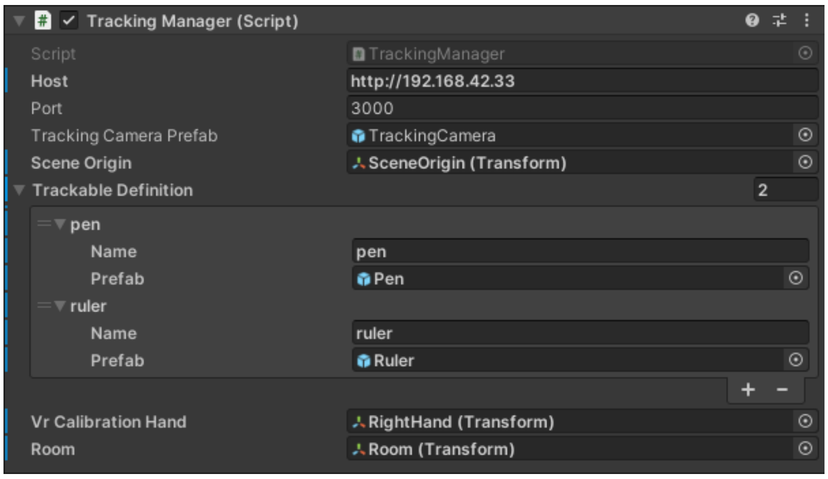
6. vrCalibrationHand

   This is object of SteamVR hand. It will be used to determine current position of the controller to properly move Room object into correct place.

7. room

   This is Room object which contains all scene objects as childs. This object will be moved to align scene with VR headset (in other words to move player exactly to the place, where he can feel objects from tracking system in the place where he see it in the scene).

In the picture (3.4) we can see example of configured TrackingManager using Inspector window in Unity editor.



Tracking Manager (Script)	
Script	TrackingManager
Host	http://192.168.42.33
Port	3000
Tracking Camera Prefab	TrackingCamera
Scene Origin	SceneOrigin (Transform)
Trackable Definition	2
pen	
Name	pen
Prefab	Pen
ruler	
Name	ruler
Prefab	Ruler
Vr Calibration Hand	RightHand (Transform)
Room	Room (Transform)

Figure 3.4: Properties of TrackingManager

There are also some important methods in the TrackingManager script - Awake, OnUpdate and OnDestroy. Method OnDestroy simple disconnects from connected server - nothing special here. Method Awake will setup communication with server. It will connect to the server and bind all events (messages types) to handle responses from server. After connection to the server, this method register itself as **consumer** client for server and ask for list of **trackables** from server and **calibration point**. Last important method is OnUpdate.

At the beginning of this method, I check VR controller if user press button to "align" scene with calibration point. After alignment is done, user has to reset it with another button to make possible a new alignment. Button for align scene is **GrabPinch** and button for reseting is **GrabGrip**.

After checking of alignment of the scene, there are three important sections of update

1. Remove trackables

   If tracking system stop tracking some trackable or if tracking system is shutdown, server will inform consumer about trackables that was removed. These trackables are marked as *to be removed* and in OnUpdate method they are removed. In each update cycle I check if there are some objects to remove.

2. Create trackables

   If tracking system start tracking some trackable or tracking system starts, server will inform consumer about new trackables which are tracked. These trackables are marked as *to be created* and in OnUpdate method they are created if we have prefab for given type of trackable. In each update cycle I check if there are some new objects to create.

3. Update trackables

   Last section is update of trackables. In each cycle of update I check if there is some update from server for trackables and new updates (position, rotation and infromation about lost trackables) are applied to instantiated objects assigned to these trackables.

At the end of the OnUpdate method I check information about calibration point. If there is new information from server about calibration point, I instantiate new "virtual" camera for it and position it to the correct place using sceneOrigin property specified in TrackingManager script. If "virtual" camera for given calibration is already exists I just update position of already instantiated "virtual" camera.

In TrackingManager there are also other methods which are binded to messages from the Communication server and just process these messages. Details of these methods are not important.

3.3 Communication server

For implementation of Communication server I choosed TypeScript programming language and NodeJS [19]. Combination of these two technologies is perfect for creating such servers and it's very fast and easy to implement it. Used library Socket.IO is primarily developed for NodeJS.

Implementation of whole server is divided into two files: main.ts and server.ts.

File main.ts is an entry point of server and only define port on which server will be listening and then create instance of Server class from server.ts file with defined port and then just start listen on defined port.

Important implementation of server is under Server class in server.ts file. In the constructor of this class I setup Socket.IO server and bind connection event. This event is fired when new client is connected to the server. Setup of newly connected client is done in addClient method.

In addClient method I push new client into array of clients and then I bind all events for this client. There are 3 types of events:

1. Common events

   These events are commond for both clients (trackers and consumers). Only two events are here: disconnect and register. disconnect event handle disconnection of client and inform about this change others if necessary (for example if tracker will disconnect, than we remove all trackables of this tracker). register event is to specify if client is consumer or tracker.

2. Events for trackers

   These events are specific for trackers. Event register_object is to inform about trackable which will be tracked by tracker. update_object is to send updated infromation (position, rotation and lost flag) about trackable to the server. And last event is calibration which is used to send calibration point to the server.

3. Events for consumers

   Consumers has only two specific events: get_trackables which is request to obtain all trackables which are registered in this server and get_calibration to obtain calibration point of given tracker.

Communication between tracker and server is shown in the picture (3.5). We can see that only tracker sends messages to the server, because tracker itself doesn't need any infromation about other trackers or consumers.

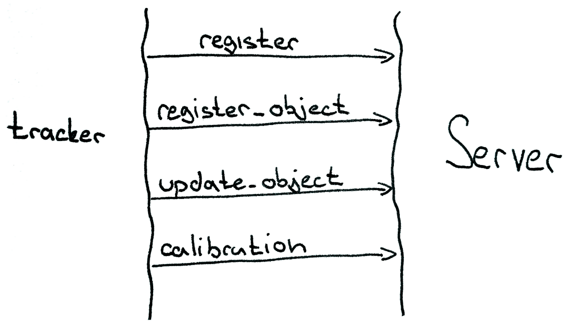


Figure 3.5: Communication between Tracker and Server

Communication between consumer and server is shown in the picture (3.6).

Unlike tracker to server communication we can see that consumer and server have bidirectional communication where consumer asks for information and server sends these information to the consumer. Also messages like update_object are send without request. Message like update_object is send after tracker send update.

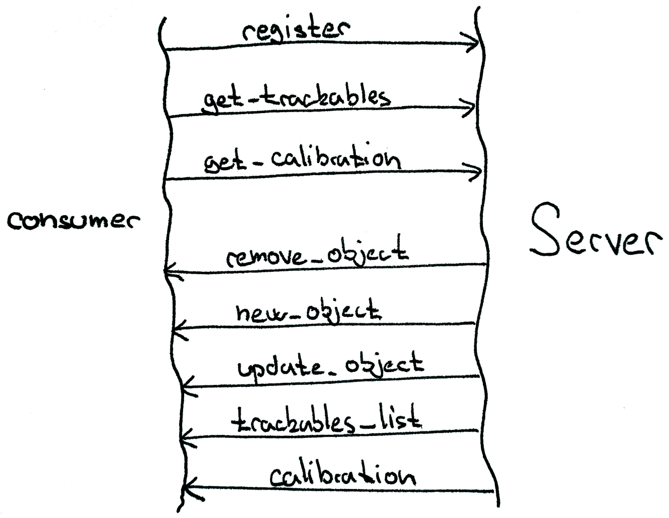


Figure 3.6: Communication between Consumer and Server

Chapter 4

Testing

In the beginning I want to say that my primary goal of testing wasn't to measure precision of estimated 6DoF pose from the Tracking system. I had no device to measure precisely the distances or positions (and rotations) of objects in the scene to measure such precision of estimated 6DoF pose. How I wrote before in my case precision is subjective feeling of user. If user feels position of tracked object in position where he see it in VR application, than it's precise enough for our case.

Whole testing in this work can be divided into two main directions. The first direction is ease of use of this Tracking system. How difficult is to setup the scene and how difficult is to develop VR application using this Tracking system. The second direction is user experience in VR application. For this purpose I prepared simple VR application with room where table is placed and on top of this table there is white paper. With the pen tracked using my Tracking system you can draw something on this paper. It's very simple scene, but drawing on paper is task dificult enough to see if precision of tracking system is usable.

Also because we are using this tracking system with VR application I will be using some VR headsets. For this project I choosed **HTC Vive** headset. How I will be mentioning later in testing chapter I also used **Oculus Quest** connected to the PC. The reason why I choose HTC Vive is very simple - because this Master's thesis continues on Bc. Jan Lazarek's work and he also choosed HTC Vive.

4.1 The scene

The scene setup was very easy. How I wrote above I want to also test how easy is to setup the scene, so I setup whole scene in my workplace. Because I'm using color segmentation for detecting position of objects I was thinking about black textile covering the scene to make it easy for tracking system to segment objects. But I wanted to test simplicity, so I just put small box on floor in the room and then position depth camera on stative to look at this box. This box was representing table in VR application room. Also I connected depth camera to the laptop and then just run Communication server and Tracking system on it. Very simple setup done in a few minutes.

In the picture (4.1) you can see how this setup was looking. Messy room with simple box and camera in "middle".

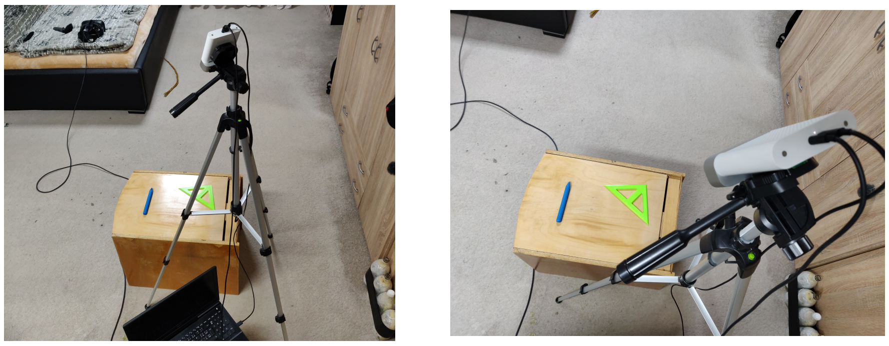


Figure 4.1: Photos of the scene setup

In the picture (4.2) you can see scene setup from more angles of view to make better idea of how setup of the scene was done.

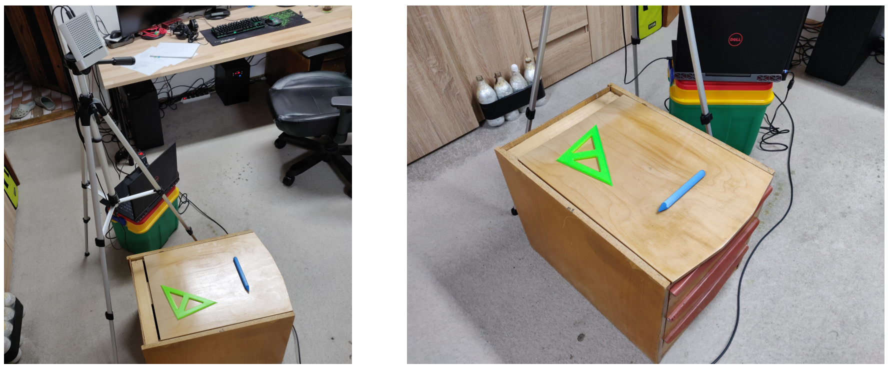


Figure 4.2: Another photos of the scene setup

Also you can see in the pictures (4.1) and (4.1) that on the box there are our tracked objects - pen and ruler. Detail of these objects si in te picture (2.2). How you can see this setup is very easy and everybody can do it in a few minutes. It's probably simplier to setup then **HTC Vive** headset which was used for testing as VR headset.

After you run the Tracking system on a laptop, you will see two windows. The first window shows what camera see in 3D and also add tracked objects (they have brown color). So you can see if estimated 6DoF pose of objects are correct. The second window shows RGB image of what camera see and adds red circles around detected objects. In the picture (4.3) we can see screenshot form Tracking system of our scene. The big window is showing 3D scene and the small one in the right bottom corner shows RGB image. I also added 3 green points to show, where I clicked to specify 3 points of the plane, which is used for calibration point. Between these 3 green points I also added the red point which represents the fourth point which I clicked in the image and this point is our calibration point. After you run the Tracking system, these four points has to be specified by clicking with mouse into RGB image. After this whole tracking system is setup.

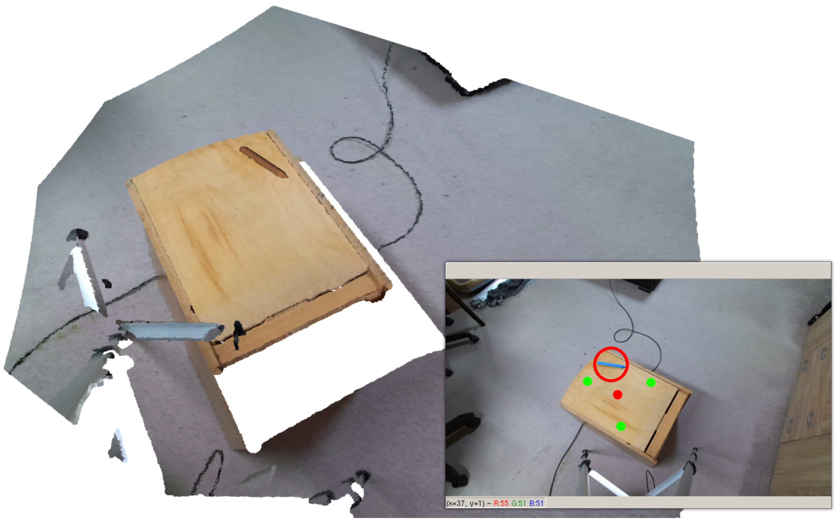


Figure 4.3: Screenshot from the Tracking system

I also setup HTC Vive headset, but it's not subject of this testing so I do not describe how to setup VR headset.

4.2 Performance of Tracking system

I tested this tracking system on two mechines - desktop and laptop. Specifications of both mechines are listed below.

Desktop specification

- CPU: **Intel Core i5 6600K @ 3.9GHz**
- RAM: **16GB DDR4 3000MHz CL17**
- GPU: **nVidia GeForce RTX 2070 SUPER**
- OS: **Arch Linux, Kernel version: 5.12.5**

Laptop specification

- CPU: **Intel Core i7 6700HQ @ 2.6GHz**
- RAM: **16GB DDR4 2400MHz CL17**

- GPU: **nVidia GeForce GTX 960M**
- OS: **Arch Linux, Kernel version: 5.11.16**

How I wrote in chapter 1 I will be measuring performance of this tracking system using FPS. I obtained FPS by simply calculating in the program by measure running time and counting frames that were processed. In the end, FPS was simply calculated by formula (4.1).

$$fps(f,t) = \frac{f}{t} \tag{4.1}$$

where f is a count of frames and t is the elapsed time in seconds to process f frames. Table (4.1) shows the average FPS in different situations and on different machines.

Situation	**FPS on laptop**	**FPS on desktop**
Only reading from camera	30	30
Tracking only pen	15	19
Tracking only ruler	18	26.5
Tracking pen and ruler	13	14.5

Table 4.1: Measured average FPS

As we can see in the table (4.1), desktop has better performance than laptop. It's nothing suprising. What is suprising is that if we are tracking only ruler object, we have much higher FPS than if we are tracking only pen. Performance of the Tracking system on desktop when tracking only ruler is realy good. If we are tracking both objects FPS will drop, but in case of desktop we get 14.5 FPS which is not bad.

4.3 User experiance

The first part of user experience is developing VR application with using Tracking system. Because I needed some VR application for testing of user experience of using VR application with Tracking system, I have also my own experience with Tracking system as a developer of VR application.

Another part of user experience is VR experience itself, where user is transferred to the virtual world using VR headset and interacts with real objects to control virtual objects which he can see. In my case user will be drawing with a pen on a paper.

4.3.1 Developing VR application

There is no much to say. Developing VR application for testing purpose with Tracking system was very easy. It's similar to developing classic VR application. To enable tracking system in this VR application you have to do 3 steps:

1. Add tracking package

   To add tracking package you install Socket.IO library, add two prefabs and one script. That's all.

2. Add tracking manager into scene

   After you have tracking package in you project, you just add **TrackingManager** prefav into the scene and setup correctly properties of TrackingManager script which is add on this prefab.

3. Add room object and calibration point

   Last step is to move all object of the scene (visible objects, no objects like some event managers or similar) under one "root" object which I named **Room**. After that I just put empty object into the scene in place where I want to have mapped calibration point from tracking system. In my case it is a place where is a paper.

After these 3 steps tracking system was integrated in VR application and when I run it, it just worked. From my point of view it's very simple to integrate tracking system and it represents no overhead.

In pictures (3.3) and (3.4) in chapter 3 you can see how scene hierarchy and TrackingManager properties looks in my VR application.

How the scene in VR application looks is shown in the picture (4.4). We can see white paper on the table. We want to align this table and paper on it with a box from the real world which was used in the scene setup described early in this chapter.

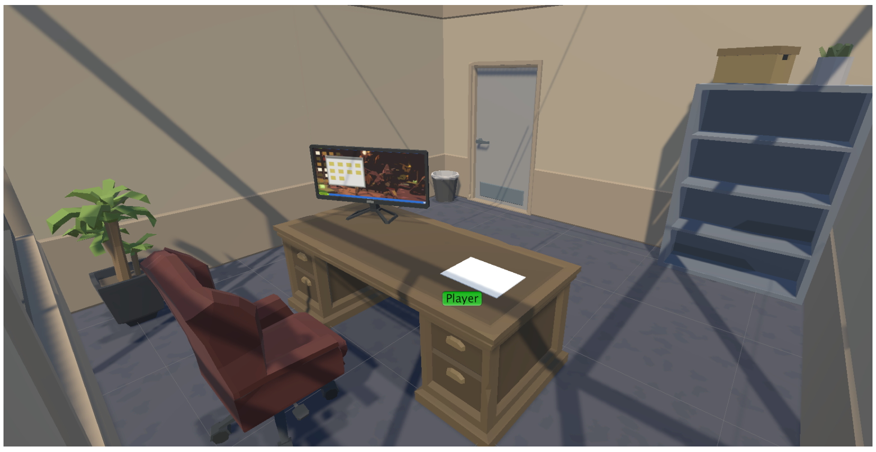


Figure 4.4: The scene in VR application

In the picture (4.3) I mentioned the red point representing calibration point of the tracking system. We want to have this point in the middle of white paper in our scene in VR application. In the picture (4.5) we can see point roughly in the middle of paper representing calibration point. This is way how we align tracking system into our scene where we want.

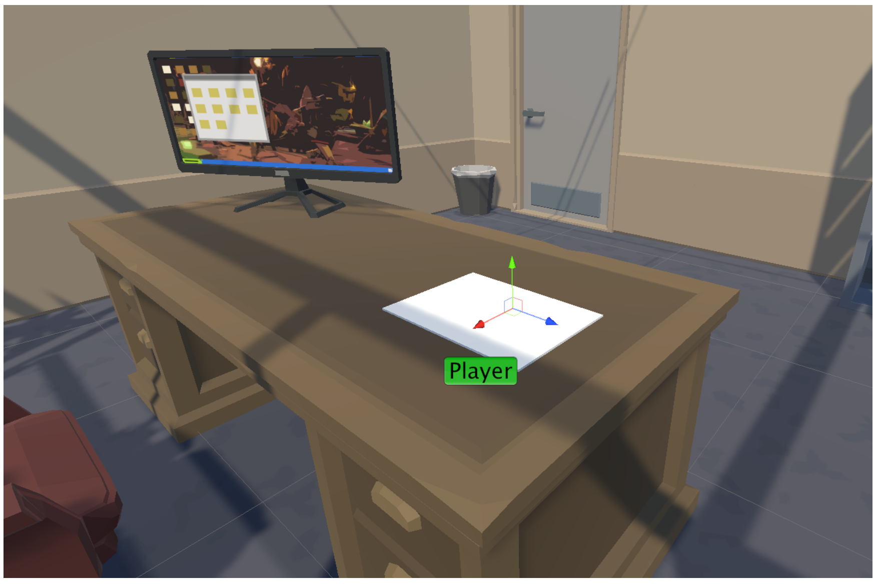


Figure 4.5: The calibration point in VR application

I also add two more screenshots from the scene to show how the scene in VR application looks. These screenshots are in the pictures (4.6).

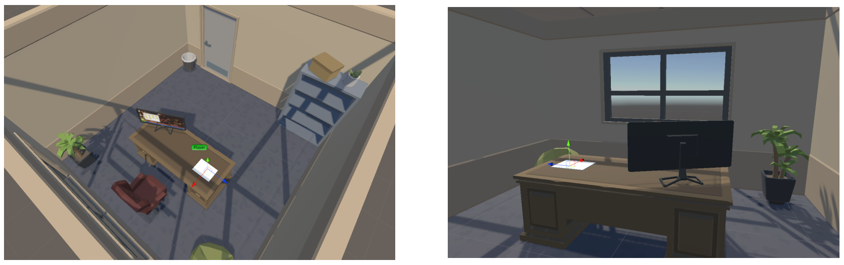


Figure 4.6: Screenshots from the VR application scene

4.3.2 Experience with VR and Tracking

There was total of 3 participants testing VR experience including me. Tests was performed on the tracking setup described early in this chapter. VR application was running on different machine than tracking system. So network communication was also tested that it's working. Moreover in one test there was runing two separate clients of VR application to see if we can run multiplayer VR application and still get correct information from Communication server and all clients of multiplayer will see tracked objects. This was working without any problems.

One of participants agreed with capturing some photos of testing so I'm including a few photos of using VR application with tracking system. In pictures (4.7) and (4.8) we can see how participant sits at the box representing table in VR application.

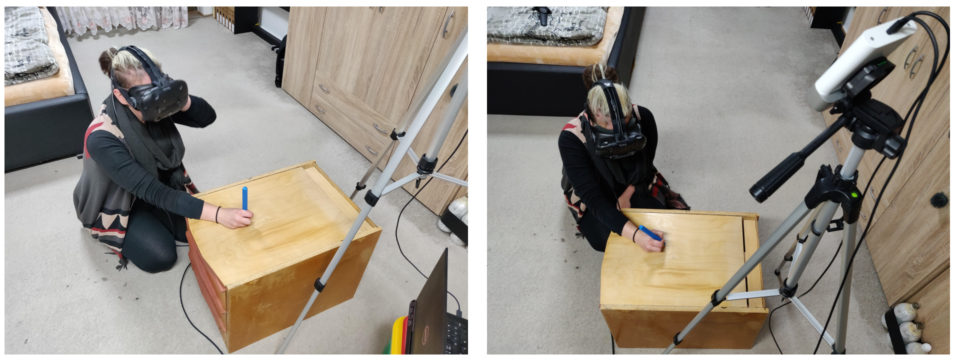


Figure 4.7: Photos from testing of VR application

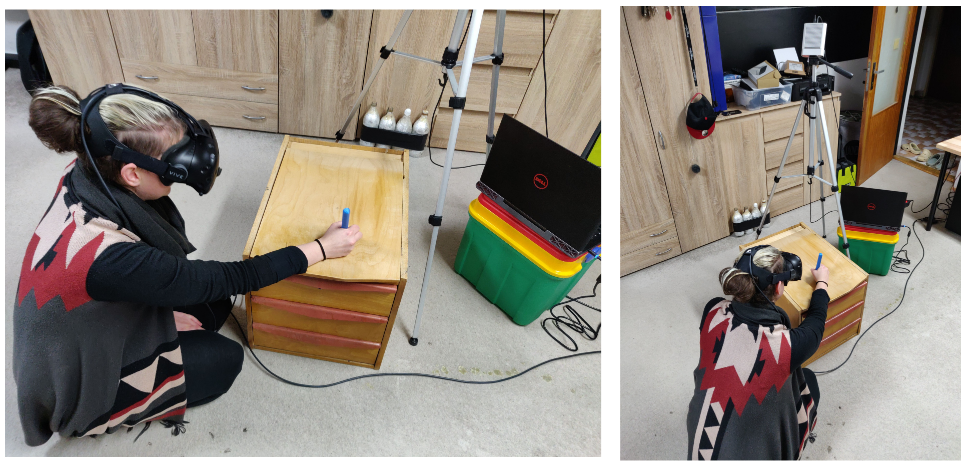


Figure 4.8: Photos from testing of VR application

Using of pen in real VR application was not so good as I expected. Performance of tracking system in term of FPS is probably no problem at this point. Any of participants was feeling no big problems with speed of tracking. One problem with using this tracking system in real VR application is overall usability. It tooks some time to used to using pen, mainly because of some unstable vibrations of object. Another problem was that pen sometimes lost from tracking. This could happen if camera can't see pen, but it was happening when pen was visible by camera.

Another big problem was with HTC Vive itself. Experience with this headset was bad. It was drifting sometimes from position, sometimes it has lags and sometimes image was flashing (like if you lost tracking and headset shows you blank screen). With longer time of using VR application users felt that pen is shifted in position. Over time pen was slowly shifting position.

But one good news is that using of this tracking system was not totally usable at all. If you accepted that it's not perfect and tried a little bit, you was able to draw something on paper. In pictures (4.9) and (4.10) we can see some drawing which can demonstrate how precise and usable tracking system was.

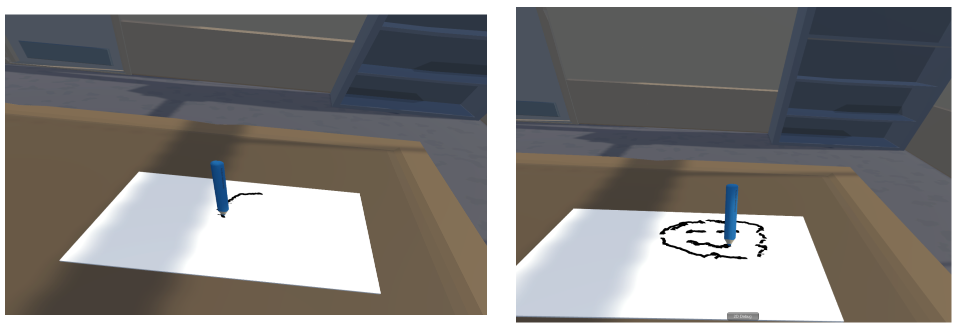


Figure 4.9: Drawings in VR application

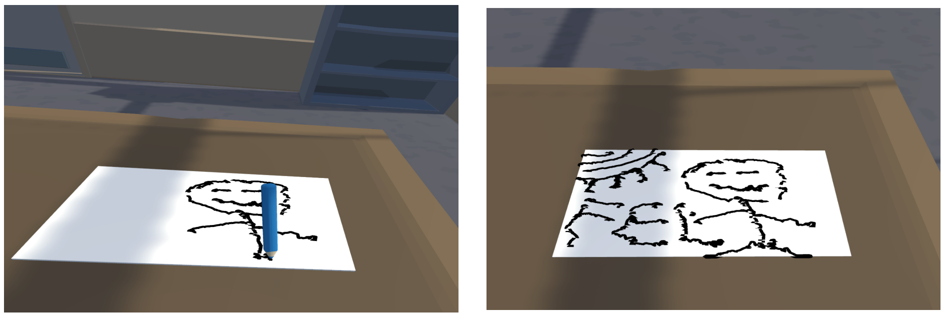


Figure 4.10: Drawings in VR application

Pictures above was done by one participant. Implementation of drawing line on paper in VR application was not good and you can see some weird artefacts. Line was represented as segments of sprites. So with better implementation of drawing lines it will be look better, but some curves in lines are caused by vibrations of pen. This could be probably solved by some type of filtering.

In the picture (4.11) we can see another drawing done by another participant.

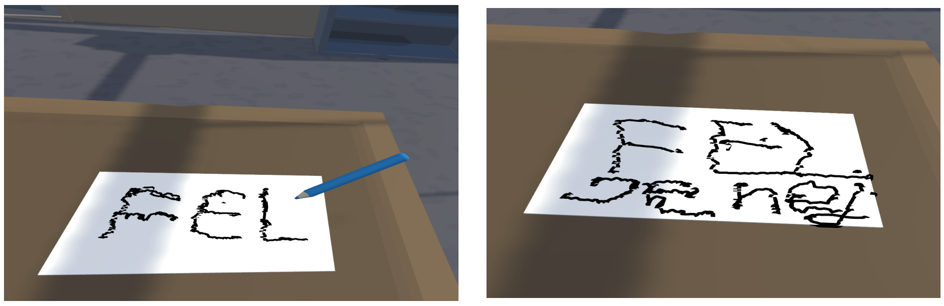


Figure 4.11: Drawings in VR application

The last participant was trying to draw flower. The last participant was also looking at text on the pen. Screenshots from this are shown in the picture (4.12).

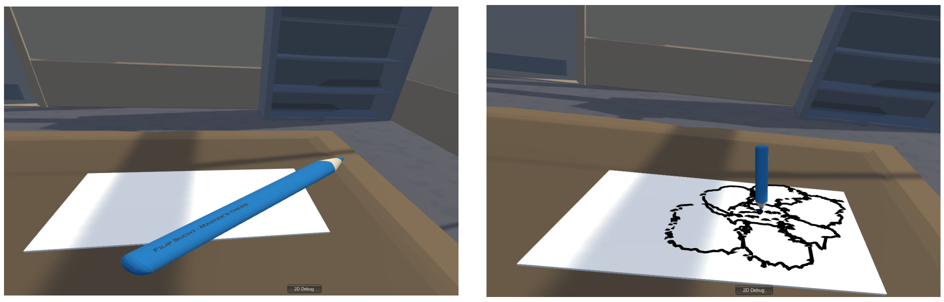


Figure 4.12: Drawings in VR application

After these tests I was trying to test VR application also with another headset. This test was done with the last participant and for this test we was using

Oculus Quest connected to the PC using Oculus Link. In this setup Oculus Quest can be used with SteamVR as HTC Vive. This last test was very surprising.

Oculus Quest uses inside-out tracking and have no base stations like HTC Vive has. When we turned off HTC Vive and use only Oculus Quest, experience with VR application was immediately better.

Whole experience was much more stable. Tracking of headset was precise and stable, no drifting of position was present as well as flashing caused by lost tracking. In subjective feeling also tracking of pen was more stable. No lost of tracking was present and vibrations of object was smaller. Drawing was little bit easier and lines looked more fluent.

After the last test I realized that IR lights used in depth camera (in my case Microsoft Azure Kincet) and tracking of HTC Vive headset are interference together somehow. This was probably causing lost of tracking of HTC Vive as well as some unstable behiever in tracking system.

4.4 Stability

In VR application it felt that the pen was vibrating/trembling. This could be a problem in cases like drawing lines. For this purpose, I tried to measure how much is pen vibrating/trembling by simply leaving the pen in a stable position and then capturing position and rotation for some time. On the captured data I then calculated mean, standard deviation, and maximal deviation.

Results of this measuring is shown in the table (4.2)

Position	**X [mm]**	**Y [mm]**	**Z [mm]**
Max	2.5883	1.4377	0.9513
Std	0.8355	0.3790	0.3324
Rotation	**X [deg]**	**Y [deg]**	**Z [deg]**
Max	1.7491	9.6340	2.5302
Std	0.5397	3.6054	0.8499

Table 4.2: Measured deviations in position and rotation

As we can see a maximum deviation in position is 2.5mm. It's not bad at all, it's usable even for drawing lines. But the rotation is another story. A maximum deviation in the rotation is 9.6 degrees and a maximal standard deviation is 3.6 degrees. These values are too high for stable drawing lines.

Chapter 5

Conclusion

The project overall revealed some problems and possibilities connected with tracking of real objects in VR. In this Master's thesis I showed that solution for 6DoF pose estimation could be done in pretty simple way and get at least usable tracking system although it's not good enough for industry use.

I get reasonable performance of tracking especially if we are tracking only single object. Tracking system, Communication server and Unity integration with single script and two prefabs shows how easy is to setup Tracking system. This is very important for real usage and it was done well in this work. Developers of VR applications can integrate this tracking system with almost no overhead. What's also very good property of this tracking system is possibility to connect more tracking systems and clients together and make them work seamlessly. You just plug them together and it works. Precision and usability of tracking of object is questionable, but for tasks where precision is not needed it's probably usable.

Very unpleasant are problems with using HTC Vive and Microsoft Azure Kinect together. Interference of IR lights causes problem to tracking system as well as to VR headset and experience with it is not good. Another problem especially in case of drawing lines with pen is vibrating/trembling of tracked object. Because of I'm using segmentation of object based on color it could causes problems of losing tracked objects in bad light conditions or if there is "color noise". The last problem which was mentioned by participants in testing is connected with loosing of tracked objects. Because I'm using last known position of object to estimate new position, sometimes is problematic to restore tracking after object was lost.

There are lots of possibilities for future work on this project. First of all it will be good to try another depth cameras like Intel RealSense [16] for example and also another VR headset. Also with objects like pen, which are symetric around some axis, it will be good to add also difference in color into objective function of ICP algorithm. This could also stabilize 6DoF pose estimation and make it more precise even with some covering of objects. Another posible improvement is to move calculations to GPU. In this project there are calculations which could be massively parallelized and GPU can brings much more processing power than CPU. But it have to be tested if there will not be a big overhead caused by moving data between GPU and CPU.

Appendix A

Bibliography

[1] P. J. Besl and N. D. McKay. “A method for registration of 3-D shapes”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.2 (1992), pp. 239–256. DOI: `10.1109/34.121791`.

[2] Yang Chen and Gérard G. Medioni. “Object modelling by registration of multiple range images.” In: *Image Vis. Comput.* 10.3 (1992), pp. 145–155. URL: `http://dblp.uni-trier.de/db/journals/ivc/ivc10.html#ChenM92`.

[3] G. Bradski. “The OpenCV Library”. In: *Dr. Dobb’s Journal of Software Tools* (2000).

[4] Alexey Melnikov and Ian Fette. *The WebSocket Protocol.* RFC 6455. Dec. 2011. DOI: `10.17487/RFC6455`. URL: `https://rfc-editor.org/rfc/rfc6455.txt`.

[5] Charles R. Qi et al. “PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation”. In: *arXiv e-prints*, arXiv:1612.00593 (Dec. 2016), arXiv:1612.00593. arXiv: `1612.00593 [cs.CV]`.

[6] Anders Glent Buch, Lilita Kiforenko, and Dirk Kraft. “Rotational Subgroup Voting and Pose Clustering for Robust 3D Object Recognition”. In: *arXiv e-prints*, arXiv:1709.02142 (Sept. 2017), arXiv:1709.02142. arXiv: `1709.02142 [cs.CV]`.

[7] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. “Open3D: A Modern Library for 3D Data Processing”. In: *arXiv:1801.09847* (2018).

[8] Frederik Hagelskjær and Anders Glent Buch. “PointVoteNet: Accurate Object Detection and 6 DoF Pose Estimation in Point Clouds”. In: *arXiv e-prints*, arXiv:1912.09057 (Dec. 2019), arXiv:1912.09057. arXiv: `1912.09057 [cs.CV]`.

[9] Yannick Bukschat and Marcus Vetter. “EfficientPose: An efficient, accurate and scalable end-to-end 6D multi object pose estimation approach”. In: *arXiv e-prints*, arXiv:2011.04307 (Nov. 2020), arXiv:2011.04307. arXiv: `2011.04307 [cs.CV]`.

[10] Jan Lazarek. “Sledování reálných objektů ve virtuální realitě”. Bachelor’s thesis. České vysoké učení technické v Praze, May 2020.

[11] Mingxing Tan, Ruoming Pang, and Quoc V. Le. “EfficientDet: Scalable and Efficient Object Detection”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2020.

[12] Jorge C. S. Cardoso and Jorge M. Ribeiro. “Tangible VR Book: Exploring the Design Space of Marker-Based Tangible Interfaces for Virtual Reality”. In: *Applied Sciences* 11.4 (2021). ISSN: 2076-3417. DOI: `10.3390/app11041367`. URL: `https://www.mdpi.com/2076-3417/11/4/1367`.

[13] Unity Technologies 2020. *Unity*. URL: `https://unity3d.com` (visited on 01/09/2021).

[14] Damien Arrachequesne. *Socket.IO*. URL: `https://socket.io/` (visited on 05/18/2021).

[15] Torch Contributors. *PyTorch*. URL: `https://pytorch.org/` (visited on 05/19/2021).

[16] Intel Corporation. *Intel® RealSense™ Computer Vision - Depth and Tracking cameras*. URL: `https://www.intelrealsense.com/` (visited on 05/21/2021).

[17] Valve Corporation. *SteamVR Unity Plugin | SteamVR Unity Plugin*. URL: `https://valvesoftware.github.io/steamvr_unity_plugin/` (visited on 05/20/2021).

[18] LLC. Facebook Technologies. *Oculus Rift S: VR headset pro počítače připravené na VR | Oculus*. URL: `https://www.oculus.com/rift-s/` (visited on 01/10/2021).

[19] OpenJS Foundation. *Node.js*. URL: `https://nodejs.org/` (visited on 05/20/2021).

[20] HeroWong. *doghappy/socket.io-client-csharp*. URL: `https://github.com/doghappy/socket.io-client-csharp` (visited on 05/20/2021).

[21] Patrick McCarthy. *GlitchEnzo/NuGetForUnity*. URL: `https://github.com/GlitchEnzo/NuGetForUnity` (visited on 05/20/2021).

[22] Microsoft. *Azure Kinect DK – vývoj modelů AI | Microsoft Azure*. URL: `https://azure.microsoft.com/cs-cz/services/kinect-dk/` (visited on 01/10/2021).

[23] Virtual Reality Peripheral Network. *vrpn*. URL: `https://github.com/vrpn/vrpn` (visited on 05/18/2021).

[24] NumPy. *NumPy*. URL: `https://numpy.org/` (visited on 05/19/2021).

Appendix B

Contents of enclosed SD card

```
/
├── models/ ........................... models for tracked objects
│   ├── 3d_print/ ......................... models for 3D printing
│   └── vr_models/ ..................... models for VR application
├── server/ ...................... code for Communication server
├── thesis/ ....... directory of LaTeX source code of this thesis
│   ├── master-thesis.pdf ............ pdf version of this thesis
│   └── master-thesis.tex .......... main file of LaTeX source code
├── tracking/ ..................... directory of tracking system
│   ├── code/ ..................... source code of tracking system
│   ├── models/ .................. models used in tracking system
│   └── environment.yml ................ environment for anaconda
├── unity/ ........................... directory of Unity project
│   ├── Build/ ............... build of Unity project for Windows
│   ├── TrackingVR/
│   │   └── Assets/
│   │       ├── Scripts/
│   │       │   └── Tracking/ ........... scripts for tracking system
│   │       │       └── TrackingManager.cs
│   │       └── Prefabs/ ............... prefabs for tracking system
│   │           ├── TrackingCamera.prefab
│   │           └── TrackingManager.prefab
│   ├── NuGetForUnity.3.0.2.unitypackage .......... NuGet package
│   └── socket.io-client-csharp-2.2.0.zip ...... SocketIO package
└── README.md/ ....... file with basic information about project
```