Faculty of Electrical Engineering
Department of Computer Science

Master's Thesis

Visual Localization with HoloLens

Pavel Lučivňák

Supervisor: doc. Ing. Tomáš Pajdla Ph.D.

Field of study: Artificial Intelligence
Study programme: Open Informatics
August 2020

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Lučivňák Pavel** Personal ID number: **435627**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence**

II. Master's thesis details

Master's thesis title in English:

Visual Localization with HoloLens

Master's thesis title in Czech:

Vizuální lokalizace pro HoloLens

Guidelines:

1) Review the state of the art in indoor visual localization, see [1,2] and references therein.
2) Adjust method [2] to local environment and image acquisition using HoloLens. Create new 3D data set for the local environment and evaluate the accuracy of the localization w.r.t. a ground truth in that environment.
3) Apply InLoc localization method on data from HoloLens, evaluate behavior and inaccuracies of the localization on this data. Investigate a possibility of using multiple images for improving the localization.
4) Demonstrate and evaluate the improved method for HoloLens localization.

Bibliography / sources:

[1] Arandjelović, R.; Gronat, P.; et al. NetVLAD: CNN architecture for weakly supervised place recognition. In IEEE Conference on Computer Vision and Pattern Recognition, 2016.
[2] Taira, H.; Okutomi, M.; et al. InLoc: Indoor Visual Localization with Dense Matching and View Synthesis. In 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, June 2018, ISSN 1063-6919, pp. 7199–7209, doi:10.1109/CVPR.2018.00752.
[3] Garg, R.; Kumar, B. V.; et al. Unsupervised CNN for single view depth estimation: Geometry to the rescue. In European Conference on Computer Vision, Springer, 2016, pp. 740–756.
[4] Zhang, Y.; Funkhouser, T. Deep Depth Completion of a Single RGB-D Image. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.
[5] Van Gansbeke, W.; Neven, D.; et al. Sparse and Noisy LiDAR Completion with RGB Guidance and Uncertainty. In 2019 16th International Conference on Machine Vision Applications (MVA), IEEE, 2019, pp. 1–6.

Name and workplace of master's thesis supervisor:

doc. Ing. Tomáš Pajdla, Ph.D., Applied Algebra and Geometry, CIIRC

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **04.02.2020** Deadline for master's thesis submission: **14.08.2020**

Assignment valid until: **30.09.2021**

doc. Ing. Tomáš Pajdla, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgments

I would like to thank my family for their support and motivation; especially during the times when I was struggling to make progress. My supervisor Tomáš Pajdla was very helpful with his suggestions and leadership. Special thanks to Torsten Sattler, who took the time to clarify my questions about the use his open-source software; and Anna Zderadičková whose experience with HoloLens allowed me to capture the raw sequential data quickly.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 14. 8. 2020 .....

Abstract

Visual localization is a common computer vision problem of estimation of the camera pose that took a particular RGB image. The pose is estimated relative to a certain coordinate system. One particular instance of this problem occurs in HoloLens mixed reality. In a mixed reality settings, we are projecting virtual objects into the real world environment. In order to maintain the objects as the user navigates around a room, we need to keep track of the device pose. HoloLens already does this, however there is a room for improvement. A new indoor visualization datasets, consisting of 2 rooms and 3 query sets, has been created. Two of these query sets are sequential images (from HoloLens). Reference poses are also provided (although not for all queries). We have designed new methods that aim to merge InLoc [1] approach to indoor visual localization with the data from HoloLens. My implementation outperformed the original InLoc paper on the task of sequential localization from RGB images. However, our approach turned out to perform significantly worse than the pose estimation from HoloLens itself. I provide an overview of sources of errors in the new and InLoc methods for potential future improvement.

Keywords: HoloLens, localization, Matterport, Vicon

Supervisor: doc. Ing. Tomáš Pajdla Ph.D.
CIIRC ČVUT,
Jugoslávských partyzánů 1580/3,
Praha 6 - Dejvice,
160 00

Abstrakt

Vizuální lokalizace je často řešená problematika v počítačovém vidění. Typicky chceme určit pózu (polohu a orientaci) fotoaparátu, který pořídil daný RGB snímek. Odhadnutá póza se vztahuje k nějakému námi definovanému souřadnicovému systému. Tento problém se například řeší ve smíšené realitě v HoloLens. Promítáme zde virtuální objekty to reálného prostoru. Abychom mohli udržet tyto objekty na správném místě, zatímco se uživatel brýlí pohybuje, je potřeba vědět, kde se HoloLens nachází. HoloLens jako takové umí sledovat svou vlastní pózu, ale výsledek není perfektní. Vytvořil jsem novou sadu dat, která obsahuje skeny dvou místností a tři množiny query obrázků. Dvě z nich pochází právě z HoloLens. Obsahem datové sady jsou i referenční pózy fotoaparátu (u některých zatím chybí, ale dají se v případě potřeby vygenerovat). Navrhl jsem nové algoritmy, které kombinují metodu InLoc [1] s daty, co nám dává HoloLens. Má implementace je na sekvenčních obrázcích přesnější, než původní InLoc. Mé metody jsou ale výrazně méně přesné, než lokalizace ze samotných HoloLens. V práci shrnuji mé poznatky, proč vnikají určité chyby související s novými metodami nebo s InLocem. V budoucnu je možné na práci navázat a chyby zredukovat.

Klíčová slova: HoloLens, lokalizace, Matterport, Vicon

Překlad názvu: Vizuální lokalizace pro HoloLens

Contents

1 Introduction 1

2 Literature review 3

2.1 InLoc 3

2.2 NetVLAD 5

2.3 P3P 5

2.4 Camera coordinate system 6

2.5 Multi-camera pose estimation . . . 7

2.6 Procrustes analysis 8

2.7 Devices used 9

2.8 InLoc improvement 10

2.9 Single View Depth Estimation . . 10

2.10 Deep Depth Completion 10

2.11 Marker-based HoloLens localization 11

2.12 Augmenting Microsoft's HoloLens with vuforia tracking for neuronavigation 11

2.13 Magnetic field and Visual Sensors for Indoor Localization 11

2.14 Visual Indoor Localization in Known Environments 12

2.15 Multi-sensor-based Indoor Localization System 12

3 Dataset 13

3.1 Reference poses 16

3.2 Habitat 26

3.2.1 Usage 26

4 Implementation 27

4.1 Source code and dataset structure 31

4.2 Pseudocode 34

4.3 `MultiCameraPose` 38

4.3.1 Introduced changes 39

4.3.2 Usage 39

5 Evaluation 41

5.1 Experiment design 41

5.2 s10e query set 42

5.3 HoloLens1 query set 48

5.3.1 Summary 48

5.3.2 Best custom method 49

5.4 Sources of errors 52

5.4.1 Previous queries have meaningful correspondences but current query does not have any correspondences 52

5.4.2 Bad input `score` matrix 52

5.4.3 Hard to pick top 10 combinations for non-trivial segments 53

5.4.4 No HoloLens poses 53

5.4.5 Geometric verification fails .. 53

5.5 Computational complexity 54

6 Conclusion **55**

6.1 Future work 55

A Bibliography **57**

Figures

2.1 Camera coordinate system 7

3.1 Coordinate systems in use 17

3.2 Camera and Marker 18

3.3 HoloLens and Vicon timelines . . 19

3.4 Reprojection error on optimized s10e parameters 21

3.5 FOV quality comparison 25

4.1 Organization of sub-projects . . . 31

4.2 Core project structure 32

4.3 Dataset construction tool structure 32

4.4 Dataset structure 33

4.5 `MultiCameraPose` structure 33

5.1 s10e query pipeline 44

5.2 Translation error threshold vs accuracy on s10e query set 45

5.3 s10e B-315 top-view 46

5.4 s10e B-670 top-view 47

5.5 Translation error threshold vs accuracy on HoloLens1 query set . 49

5.6 HoloLens1 query pipeline 50

5.7 HoloLens1 B-315 top-view 51

Tables

3.1 Evaluation of optimized HoloLens1 reference poses 20

3.2 Evaluation of non-optimized HoloLens1 reference poses 22

3.3 Optimal HoloLens delay parameters 22

3.4 Statistics of the InLocCIIRC dataset 24

3.5 Handling of reflective surfaces . . 24

4.1 Input parameters of `multiCameraPose` MATLAB function 40

5.1 s10e pose estimation errors 43

5.2 InLoc and InLocCIIRC performance on non-sequential queries 45

5.3 s10e pose estimation error statistics 45

5.4 InLocCIIRC performance on HoloLens1 query set 48

5.5 HoloLens1 pose estimation error statistics 48

5.6 Queries affected by inaccurate `score` 53

5.7 Processing times of the experiments 54

Chapter 1

Introduction

Visual localization is a common computer vision problem where, given an RGB image, we want to estimate the camera pose. Such a camera pose can be specified by 6 parameters - 3 of which describe its position in space and the other 3 represent its orientation in the space. In case of outdoor visual localization, the problem can be simplified by making use of GPS for approximate position localization. Visual localization is a problem that also needs to be addressed indoors, however. It has use cases in e.g. augmented and mixed reality applications. Of course, the GPS signal is unusable in a building. In this thesis, I am going to focus on indoor visual localization with HoloLens. HoloLens is a mixed reality device; providing a powerful tracking of the camera as user navigates around a room. The accuracy is already high, however the idea of this thesis is to improve it further. Imagine a use case where we place virtual objects into the mixed reality. As user navigates around the room, we need to track the pose of HoloLens in order to maintain the object placements. The indoor environment is problematic for several reasons. One of them being that there are a lot of similar areas - the same type of windows, doors, textureless walls. Furthermore, the environment can change easily as people interact with it.

The objectives of this work are as follows:

1. State of the art in indoor localization must be reviewed; in particular the NetVLAD [2] and InLoc [1] papers.
2. A new indoor dataset based on the InLoc dataset [1] shall be created. The new dataset must also contain query images that were taken in a sequence (as an user with HoloLens walks in the room).

3. Make InLoc run on the newly created dataset, by processing the query images in non-sequential fashion.
4. Implement an improvement that takes the HoloLens data into account. One of the improvements should include taking multiple historical camera data into account (InLoc currently only uses a single camera pose, because it does not deal with sequential data).
5. At last, the performance of the newly implemented algorithms shall be evaluated.

This work is organized the following way. Chapter 2 contains related work on the topic of indoor visual localization. Background that my software and algorithms rely on is also described. The newly acquired dataset is described in Chapter 3. It contains statistics of the dataset, its structure and how it was created. An implementation of the techniques described here are covered in Chapter 4. Note that it is only a proof-of-concept implementation, unsuitable for deployment out of the box (just as the InLoc implementation). Chapter 5 evaluates the newly developed methods and compares them with some baseline methods. Sources of errors are also noted. Finally, the Chapter 6 is a summary of this work, whether it fulfilled the assignment and possible future work.

Chapter 2

Literature review

This chapter provides a review of relevant work. It also provides a theoretical or technical background on topics we are dealing with within this thesis.

2.1 InLoc

InLoc [1] a powerful method (state-of-the-art in 2018) for indoor visual localization. All the newly developed algorithms in this work use InLoc or its modification at its core. Not necessarily because there is no better method out there, but because it is the topic of this thesis (and the thesis supervisor is a co-author of the InLoc paper, which is beneficial for getting familiar with the method quickly). To understand the methods developed in this work better, it is useful to learn about InLoc first.

InLoc operates on top of an InLoc dataset. The dataset contains:

- 256 query RGB photos,
- 10 000 cutout RGBD images,
- 277 reference panorama poses (determined using [3]),
- reference query poses,
- query-cutout similarity matrix (known as `score`).

The dataset has been acquired at the Washington University in St. Louis. Five floors (at two buildings) were used to build the dataset. The InLoc method assumes existence of a 3D map. In the InLoc dataset, the 3D map was constructed using a high-end Faro 3D scanner. Each scan with that device produces an RGBD panorama image. The data from the various scans was then merged to create a single 3D model for each floor (using [3]). In addition, database images, also called *cutouts* were produced. The cutout RGBD images are a result of perspective view extraction from the RGBD panorama images. The panorama images were captured across all five floors using a high-end Faro 3D scanner. The query RGB images represent the images for which we aim to determine camera pose. These were taken using a smartphone camera with no depth information. Queries were taken only at two floors; the other floors serve as a confusion for InLoc.

The query photos were taken at a different time of the day, to take illumination and interior changes into account. The query reference poses are needed, to attest how well InLoc performs on the pose estimation task. These reference poses were determined using (paraphrased from [1]):

1. Selection of the visually most similar cutout images.
2. Automatic matching of query images to selected cutout images.
3. Computing the query camera pose and visually verifying the reprojection.
4. Manual matching of difficult queries to selected cutout images.
5. Quantitative and visual inspection.

Understanding the details of reference pose generation is not necessary for understanding this work. This is because, as we will see in Chapter 3, our new dataset:

- Already contains reference panorama poses from Matterport.
- Uses a pose-tracking system Vicon to estimate the reference query poses.

Let's take a look at a high-level overview of how the InLoc method operates. The following is a simplified and paraphrased (from the InLoc paper [1]) description:

1. For every query image: find top N similar cutout images using the `score` similarity matrix.

2. For every query-cutout pair: find tentative pixel-to-pixel correspondences using matching of NetVLAD 2.2 features. This step is called geometric verification.
3. Re-rank the top N cutout lists according to the highest number of tentatives found (if there is a draw, the original query-cutout score decides the order).
4. Choose top $M \leq N$ cutouts in the lists.
5. For all query-cutout pairs, construct a pose estimate using P3P-RANSAC[1]. This is possible since the pixel-to-pixel correspondences can be converted into 2D-3D correspondences (cutouts are RGBD).
6. Project the estimated poses and evaluate their similarities to the query images (pose verification step).
7. For every query: choose the cutout for which we have a synthesized query image with highest similarity to the query image (using DenseRootSIFT [5] [6]).

The computational requirements are missing from the InLoc paper. However, the authors mention the need for about 14 GB RAM in their experiment, to hold the image descriptors in memory.

2.2 NetVLAD

NetVLAD [2] is a convolutional neural network[2] (CNN) architecture for visual place recognition. The input to this network is an RGB image and the output is a feature representation of that image. Given two images and their corresponding features, we can compute to what extent they resemble the same place. In this work, as well as in InLoc [1], we use a VGG-16 [9] + NetVLAD model that is pre-trained on Pitts30k [2] dataset.

2.3 P3P

The P3P problem [4] is a problem of estimating a calibrated camera pose using at least three 2D-3D correspondences. A calibrated camera is a camera for which we know its calibration matrix (see section 2.4). RANSAC [4] can be used to

[1] P3P is covered in section 2.3; refer to [4] for RANSAC description.
[2] See the original paper [7] or a Deep learning survey [8].

further improve estimation accuracy, if some of the correspondences are imprecise or incorrect.

2.4 Camera coordinate system

Figure 2.1 shows a camera coordinate system γ, that defines the camera pose.

Calibration matrix. A camera calibration matrix K is a linear transformation that converts points in camera coordinate system γ into points in image coordinate system β. It is defined by five parameters [10]:

$$\mathrm{K} = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ 0 & k_{22} & k_{23} \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.1}$$

They represent focal length, sensor dimensions, origin of the image coordinate system and more.

Our implementation, however, only requires a subset of those parameters. Thus the calibration matrix K can be constructed as:

$$\mathrm{K} = \begin{bmatrix} f & 0 & w/2 \\ 0 & f & h/2 \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.2}$$

where w and h are width and height of the camera sensor in pixels. Focal length f is also in pixel units.

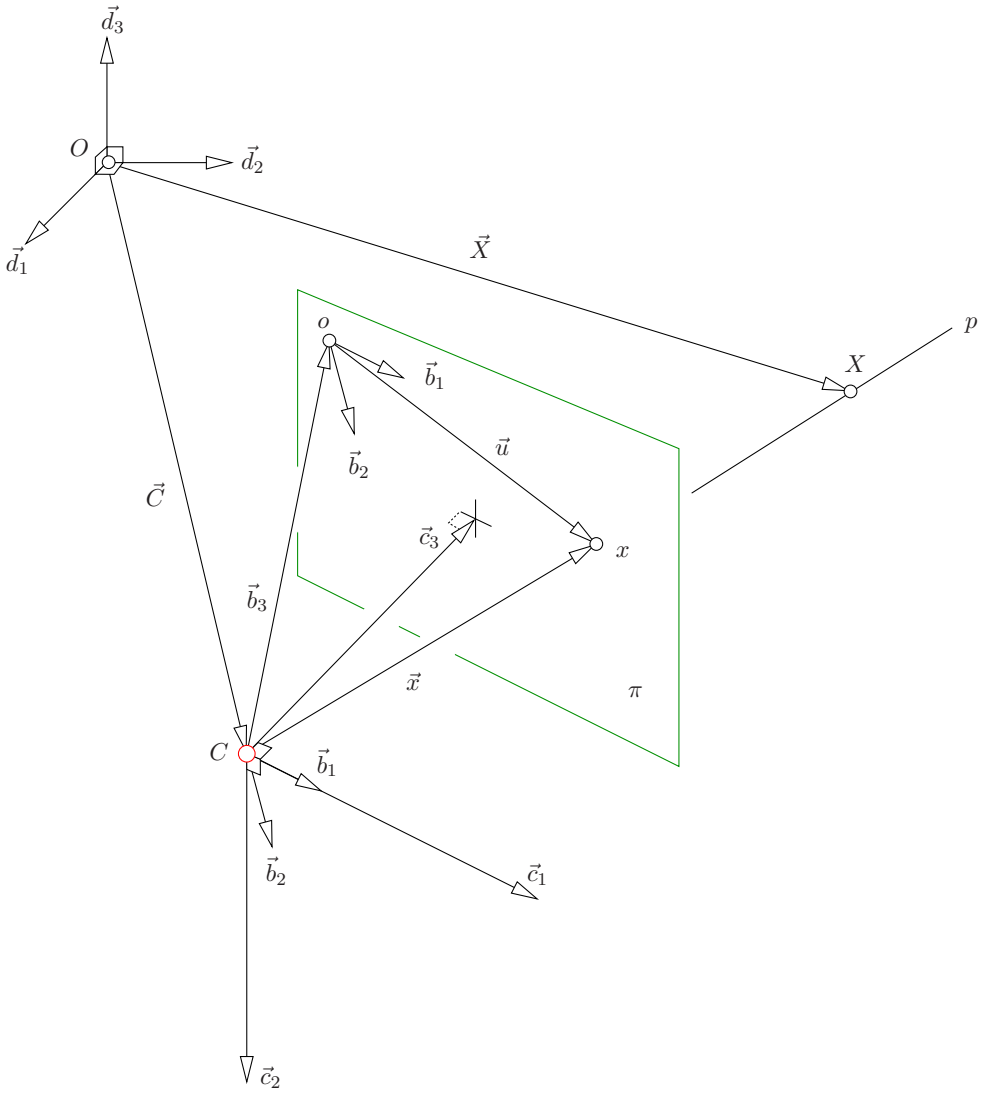


Figure 2.1: Camera coordinate system γ with bases $\vec{c}_1, \vec{c}_2, \vec{c}_3$; and with origin at $\vec{C}$ with respect to some World coordinate system γ. Camera points the $\vec{c}_3$ direction, having $\vec{c}_1$ on its right. $\vec{b}_3$ defines the origin of image coordinate system (pixel at 0,0). Vectors $\vec{b}_1, \vec{b}_2$ are considered to be orthogonal, as we are dealing with a rectangular sensor in this thesis. Point X with 3D coordinates $\vec{X}$ projects onto the image plane to a point x with image coordinates $\vec{u}$. The two points form a 2D-3D correspondence. Figure is from page 36 of [10].

2.5 Multi-camera pose estimation

Throughout this work, we often operate on query images that were taken in a sequence. Imagine a person walking around a room and taking a picture every once in a while. Considering the sequential nature of the queries can help us improve

the localization performance. To achieve this, we need to be able to estimate poses of multiple cameras in the sequence. In general, these cameras are referred to as a rig of cameras. The generalized pose-and-scale problem (GP4Ps/gsP4P), defined in [11] describes such a problem in general and provides an efficient solution. A particular implementation of the solver is provided as an open source software, named `MultiCameraPose`. The project is a result of a recent work [12]. For my use-case, I modified the implementation slightly and it is available at [13]. The implementation is very fast, computing the pose estimates of a rig with 5 cameras in about 60 milliseconds. A concrete usage of the `MultiCameraPose` program and the changes I have made are described in the Implementation chapter 4.

2.6 Procrustes analysis

Consider two k-tuples containing points in n-dimensional space:

$$X = (\vec{x}_1,\ \vec{x}_2,\ ...,\ \vec{x}_k), Y = (\vec{y}_1,\ \vec{y}_2,\ ...,\ \vec{y}_k). \tag{2.3}$$

We wish to find an (approximate) linear transformation for the corresponding pairs of points:

$$\vec{x}_i \approx \mathrm{T} \cdot \vec{y}_i, \quad \forall i \in [1, k]. \tag{2.4}$$

This is especially meaningful if:

1. Points in X are all with respect a common coordinate system, call it α.
2. Points in Y are all with respect a common coordinate system, call it β.
3. The points $\vec{x}_i$ and $\vec{y}_i$ represent (with possible noise) a common point in α.

Procrustes minimizes the sum of squared errors of points:

$$\arg\min_{\mathrm{T}} \sum_{i=1}^{k} (\vec{x}_i - \mathrm{T} \cdot \vec{y}_i)^2. \tag{2.5}$$

We will be using an implementation in MATLAB called `procrustes`; it is available as part of the Statistics and Machine Learning Toolbox [14].

2.7 Devices used

Matterport. Matterport is a device capable of creating a 3D map of indoor environments. Compared to the Faro 3D scanner used in InLoc dataset [1], Matterport is a cheaper alternative. Matterport operating time is also lower, and the resulting point cloud is of lower quality [3].

Vicon. Vicon is a stationary system capable of tracking an object's pose with high accuracy [15]. The tracked object, Marker, contains spheres which have a reflexive surface. Vicon is used for reference pose determination in the newly created dataset within this work.

HoloLens. HoloLens is a mixed reality device. Mixed reality (MR) is the blending of virtual and real environment, such that the user of MR can interact with both of these environments; see [16] for a difference between augmented reality, virtual reality and mixed reality. HoloLens is a head-worn device capable of mapping the real environment and localizing itself within it [17]. In this thesis, we are making use of the 1st generation HoloLens [18], referred to as *HoloLens* for simplicity.

HoloLens contains the following sensors [17] [18]:

- main RGB camera,
- 4 environment-understanding grayscale cameras,
- a depth sensor,
- 4 microphones,
- other sensors.

The environment mapping and localization are done directly on the device in real-time in a SLAM[3] -like manner [17]. In an experiment conducted in [17], the

[3]See [19], [20], [21].

poses estimated by HoloLens had the following mean accuracy with respect to the ground truth poses:

- 1.6 ± 0.2 cm translation error,
- $2.2 \pm 0.3°$ orientation error.

2.8 InLoc improvement

The paper called *Is This the Right Place? Geometric-Semantic Pose Verification for Indoor Visual Localization* [22] suggests an improvement to the original InLoc. It focuses on significantly improving the the pose verification step. Neither the source code nor the data have been published as of August 9, 2020; and there exists no other open-source implementation to my knowledge. Implementing the methods in this paper would be very time consuming; thus, the results of this paper are not used in our work.

2.9 Single View Depth Estimation

The authors of [23] provide a method for learning a convolutional neural network (CNN). The network's task is to, given an input RGB image, estimate the depth of each pixel. The impact of their work is that they managed to do so in unsupervised fashion; eliminating the cost of manually labelling the data [23]. The results of that paper could be useful to us, as we could use it to create depth data to improve localization accuracy. However, HoloLens already includes a depth sensor [17]. Although the sensor provides a limited field of view (FoV), it shall be first tested, whether the sensor data are sufficient for our purposes.

2.10 Deep Depth Completion

The paper [24] also provides a solution to the problem of depth estimation from a single RGB image. However, it makes use of the device's depth sensor, builds upon it and improves its accuracy. The problem with many depth cameras is that they

often fail to sense depth for shiny, bright, transparent, and distant surfaces [24]. This work, while promising, shall only be considered after we find that the HoloLens depth camera is not sufficient for our purposes.

2.11 Marker-based HoloLens localization

In the paper [25], authors suggest a method for improving the localization capability of 1st generation HoloLens. They do so by placing markers (2D QR code-like objects) across the environment. The method then allows them to accurately place large virtual models into the environment within a spatial accuracy of few centimeters [25]. The problem of this approach is that it requires the knowledge of the location of the markers with respect to some model coordinate system.

2.12 Augmenting Microsoft's HoloLens with vuforia tracking for neuronavigation

Paper [26] describes a significant improvement of HoloLens accuracy in a medical scenario. It does so by using proprietary Vuforia SDK. The proprietary aspect of it is, however, not ideal for open-source projects.

2.13 Magnetic field and Visual Sensors for Indoor Localization

Paper [27] shows how to improve localization accuracy by using both visual sensors and a magnetometer. The magnetometer is only present in 2nd generation HoloLens, which we do not have access to. We could use a custom sensor and attach it to the device, however.

2.14 Visual Indoor Localization in Known Environments

The article [28] suggests a different approach from InLoc to tackle the indoor localization problem. Instead of constructing a 3D model of the environment in the first stage, visual features are detected in a video sequence and SURF features [29] are extracted. According to the article: "the sequence must cover all the areas in which localization will be needed. ... each frame is manually labeled with positional information with a time-consuming procedure". During the localization, query video frames are matched to frames in the reference video sequence.

This article may be worth considering when we do not have access to a dedicated 3D-environment scanning device.

2.15 Multi-sensor-based Indoor Localization System

The authors of [30] propose a method for robust indoor localization integrating multiple sensors and a visual localization from a single RGB camera. Promising results are shown, but I was unable to find a reference implementation. Creating my own based on the paper would be very time consuming.

Chapter 3

Dataset

The original InLoc implementation is using the InLoc dataset [1], which is based on data taken at the Washington University in St. Louis (WUSTL dataset [3]). The InLocCIIRC dataset aims to keep the same structure as the InLoc dataset. The new dataset was created at the Czech Institute of Informatics, Robotics and Cybernetics (CIIRC).

The dataset is a result of scanning two rooms at CIIRC: the B-670 lecture hall and a room B-315. For scanning the environments, a Matterport 3D scanner is used. Let's call the environments *spaces*. Compared to the Faro 3D scanner used in InLoc dataset [1], Matterport is a cheaper alternative. Matterport operating time is also lower, and the resulting point cloud is of lower quality [3]. Matterport creates a point cloud and a mesh model of each space. This is made possible by scanning the area at various locations. Let's call each such scan a *sweep*, to match the Matterport API terminology. To construct the models, RGBD panoramas are taken around the rooms. In B-670, I have taken 31 such panoramas. In B-315, I have taken 27 panoramas. Overall, there are 58 RGBD panoramas taken by Matterport 3D scanner. The scanner was mounted on a tripod at height of approximately 1.52cm and I tried to avoid walls and objects in 60cm radius.

When creating an RGBD panorama, the Matterport scanner has to revolve around yaw axis in order to capture the scene in 360°. For each RGBD panorama, we are given the pose of the Matterport scanner at the moment right before the rotation started. These poses are provided by Matterport, so we don't have to go through the hurdles of estimating them ourselves as in [3].

Another outcome of the sweeps are RGB panoramas. Matterport does not support

automatic gathering of these panoramas, so they have to be downloaded manually for every sweep. Another problem is that these downloaded RGB panoramas are not pointing the same direction as is the initial orientation of the Matterport camera. Therefore, I have created a tool to semi-automatically find the proper orientations. This is done by

1. projecting the point cloud model according to the sweep pose,
2. sampling the RGB panoramas around the yaw axis and picking such a sample that best matches the projection. The matching is done by picking such a sample for which the amount of edges in a differential edge image is minimal.

This approach works well, however it may still fail in an exceptional case. Then, a user is encouraged to try the 2nd lowest amount of edges, 3rd least amount and so on. Alternatively, one may try to increase the point size of projected the model. As a last resort, one can manually find the RGB panorama sample by observing all perspective projections of that panorama, generated via a provided script.

Once we have the RGB panoramas which are pointing the same direction as the RGBD panoramas, we can move onto the next stage. Here we construct cutouts, which are perspective projections of the RGB panoramas at a specific orientation. As in InLoc, I am sampling around the yaw axis per 30°, under the pitch direction of $\{-30, 0, 30\}$ degrees. The cutouts also contain information about the depth (not provided by Matterport).

The dataset contains sets of query images (queries). The first set, called s10e, was taken by a smartphone camera — via Samsung Galaxy S10e's wide angle rear facing lens (i.e. the main lens). I have taken 40 query images in a restricted area of room B-315. This room was chosen to be in the dataset, because it contains a pose estimation system called Vicon. The other two sets of queries were obtained using 1st generation HoloLens; and were also taken in that restricted area. The sets are named HoloLens1 and HoloLens2 — the suffix number indicates the sequence number. The major difference between s10e and HoloLens query datasets is that the queries from HoloLens form a sequence of images, as the user walked around the room. The sequential nature of those query datasets shall be leveraged, and data from multiple cameras may be used for a higher-precision pose estimation of a current frame.

All of the query images were taken in this specific area of room B-315, so that their reference pose is known. No queries were taken in room B-670, as it would be time consuming to estimate the reference poses manually (or creating a program that does this). Hence, its only purpose is to serve as a confuser.

The queries in the s10e set have a pixel resolution 4032×3024. InLoc implementation requires the knowledge of focal length of the camera that was used when taking the query images. I found conflicting information about the S10e's field of view (FoV) online, and the focal length didn't add up. I ended up computing the focal length manually with the help of a tripod and a ruler. The focal length turned out to be 3172 pixels. The IDs of query images are sorted in a non-decreasing difficulty, e.g. queries with IDs 1 to 10 were taken such that the camera's direction vector is roughly parallel with the floor. Queries with higher IDs have the camera rotated on a tripod under any direction.

The HoloLens queries have a pixel resolution of 1344×756 pixels and according to the official documentation, the horizontal FoV is 67°. Because the results were not very precise at some point of development, I began to question whether the documented FoV of 67° is indeed true. Looking at the data generated while capturing the sequences, HoloLens provides a `cameraProjectionTransform` matrix. According to an article [31], the effective hFoV can be computed as

$$\texttt{hFoV} = 2 \cdot \arctan\left(\frac{1}{\texttt{cameraProjectionTransform.m11}}\right), \tag{3.1}$$

which gives the value of 65.83 degrees. This is the more accurate result, as a reprojection error was lower with this redefined constant.

The sweeps, used to construct the point cloud model, were taken on Thursday/Friday midnight. The s10e query images were taken on a Monday morning 3 days later. Note that there was a weekend between the two time frames, meaning the scene didn't change a lot during that time. The reason the query images were taken later was to test what happens when items such as chair, lighting and people move around or change.

The two HoloLens sequences were captured about three weeks later. This means the environment was more challenging to work with, because it has changed from the state in which it was scanned by Matterport.

Alignments define the pose of individual sweeps within the space they are in. Because the poses are given to us from Matterport, we do not need to perform the generalized iterative closest point (GICP) step, as in InLoc. Because Matterport gives us an entire model (point cloud and mesh) of each scanned space, we do not need to consider alignments at all. They were useful in InLoc, where there were individual point clouds for sweeps and thus the 3D coordinates of the points projecting onto cutouts were with respect to (wrt) the sweep coordinate system.

In InLoc, there are point cloud models for every sweep. On the contrary, in InLocCIIRC we have a model for each space.

The InLoc implementation requires the knowledge of scores between every pair of a query image and a cutout image. An individual score describes similarity between the two images. When the software is run, InLoc chooses, for each query, top N cutouts with the highest scores. The other cutouts will not be considered. It is thus quite important that these scores are relevant. NetVLAD [2] descriptors are computed for both cutouts and query images. The features are the output of the L2 normalization layer. A score between a query image and a cutout is computed using a dot product between the two feature vectors. Note that the similarity scores of cutouts for a query do not represent a probability distribution, and thus don't need to sum up to one. The code for doing so was not provided in InLoc, so I came up with an implementation that reuses existing InLoc MATLAB components. The resulting scores seem to be meaningful, but a reference implementation would have been better.

3.1 Reference poses

We need to know the reference pose of each query, in order to evaluate how accurate the pose estimation algorithms are. The pose of the cameras used to take the query pictures in query sets was also being tracked by a pose estimation system – Vicon. Figure 3.2a shows the s10e camera (thus also its coordinate system) and a coordinate system that is being tracked by Vicon. Let the latter coordinate system be called Marker.

Let's now focus on a more difficult scenario, which is the reference pose determination of the HoloLens queries. There are three reasons why the reference poses cannot be simply taken from the Vicon tracking:

1. camera pose and Marker are widely different,
2. the Vicon coordinate system differs from the World[2] coordinate system,
3. Vicon started tracking before HoloLens was run, as visualized in figure 3.3.

[1]For exceptions caused by delays take a look at table 3.3.

[2]The World coordinate system is a coordinate system of the point cloud and mesh models provided by Matterport.

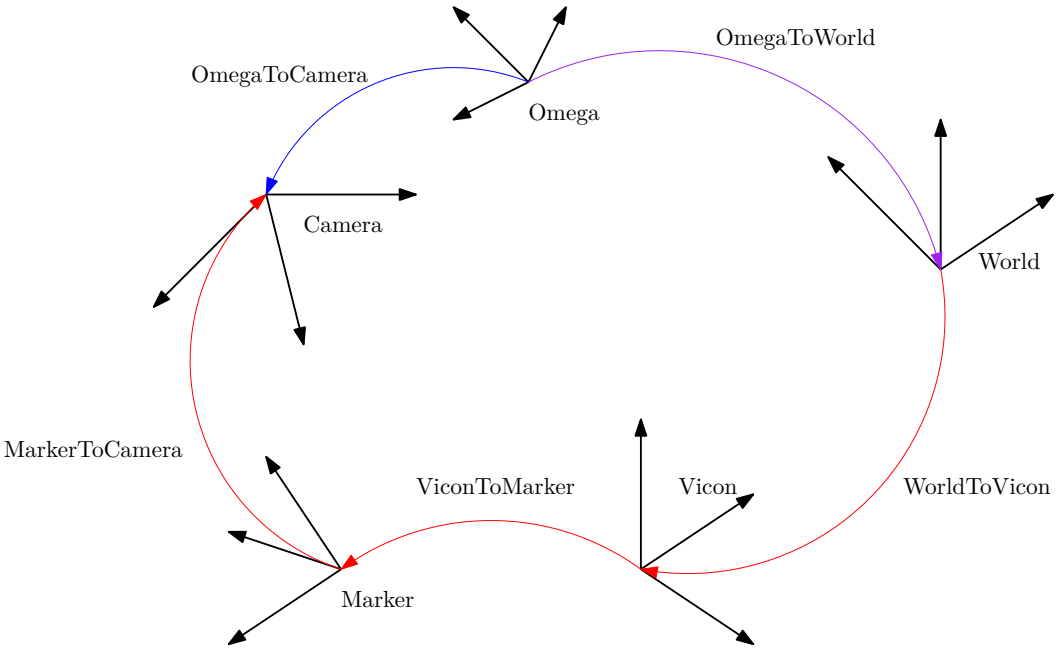
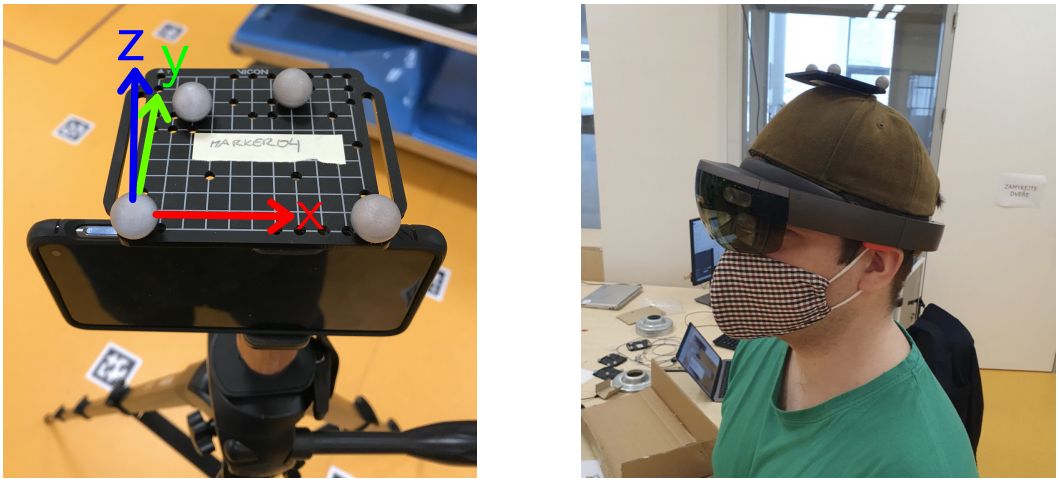


Figure 3.1: Visualization of the coordinate systems we are dealing with. Omega is the initial unknown HoloLens CS (see section 2.4 for a an example of a coordinate system). Notice that Omega has a scaling independent of the World CS scaling. Linear transformations are shown by the arrows. There are in fact two slightly different Camera coordinate systems – one that is estimated from HoloLens and another one (reference pose) that is estimated using Vicon. OmegaToCamera is known for most[1]of the HoloLens queries, because the data comes from HoloLens. ViconToMarker is provided from Vicon tracking. WorldToVicon has been manually determined. MarkerToCamera has been approximated by an algorithm described in the Reference poses section 3.1.

Luckily, the second issue turned out to be easily mitigated. I have been told where the origin of the Vicon coordinate system is. And by experimentation, the rotation matrix that converts Vicon bases to World bases was found. Because the Vicon bases and World bases are aligned to the room (i.e. a basic vector is parallel with the floor or the walls), the rotation matrix can be represented by a simple rotation.

The transformation from Marker to camera is considered to be a constant (for all queries in a query set), because the tracking device is securely attached to the camera. One could manually estimate that transformation and visually evaluate how close the model projection is to the original query image. However, this approach is prone to errors. Instead, a quantitative approach was employed, which I describe next.

For a particular query set, we need to manually set up the reference poses for a small number of queries. I used 6 of them in HoloLens1. Let these queries be called *interesting* queries. For such a query, we manually find nine 2D-3D correspondences. The 2D correspondences are carefully chosen, such that they actually represent the



(a) : s10e query set. **(b) :** HoloLens2 sequence.

Figure 3.2: The camera and marker (the object tracked by Vicon). Marker coordinate system is visualized in subfigure 3.2a by the `xyz` arrows.

same 3D point – because the 3D points were captured up to three weeks earlier than the query images and the environment has changed. For each query with the correspondences, we compute its initial reference pose using P3P. The pose returned by P3P may not be completely accurate, however.

Given reference poses for 6 queries and corresponding poses from Vicon, we can almost compute individual Marker to camera transformations. The last piece missing is a synchronization constant, to match the correct Vicon pose taken at Vicon time with a particular query taken at HoloLens time. I created a script, `findOptimalParamsForInterestingQueries.m`, which computes the Marker to camera transformations and evaluates the reference poses quality both quantitatively (reprojection error) and visually (manually investigated by the user). Currently, user must guess a synchronization constant. Finding a reasonable synchronization constant does not take long. Alternatively, one could implement a brute-force search, where various synchronization constants are guessed and the one with lowest quantitative error is chosen. In my case this was not necessary. At the end of the script, a generic transformation is suggested, which is an average of the individual transformations. The quality of the generic transformation is again evaluated on all the 6 queries. This generic transformation and the synchronization constant are used as a baseline and are further optimized, described next.

A brute-force search is employed to find an improved version of the baseline transformation and synchronization constant in nearby space. First, an improved synchronization constant is estimated, by simply evaluating the interesting queries on the same transformation but for different synchronization constants, that are close to the baseline constant. Then, different transformations are being tried. An Marker to camera transformation is described by a 3D translation vector and a

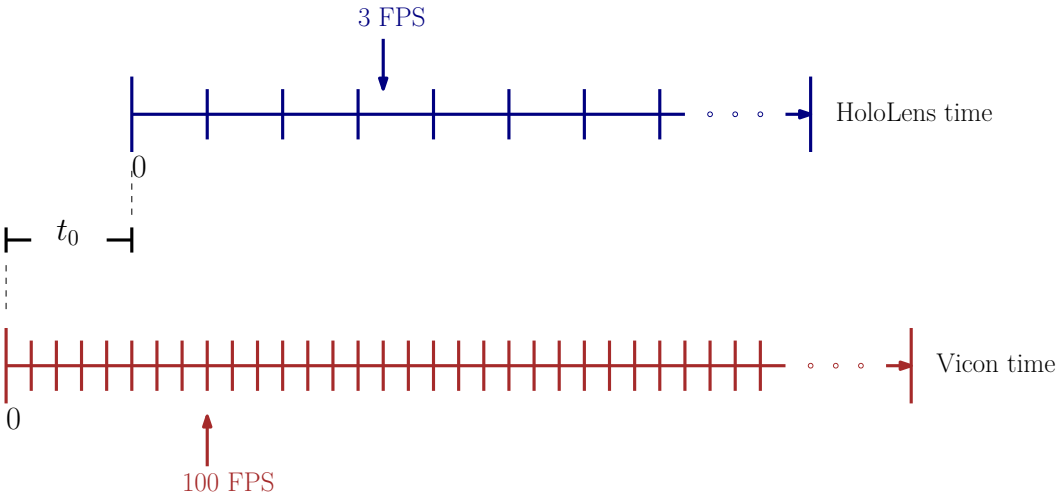


Figure 3.3: Visualization of the HoloLens and Vicon timelines. The synchronization constant must be found. Note that the sampling frequencies are vastly different. However, given a query image from HoloLens taken at some point in time (HoloLens sampling frequency), we find the corresponding reference pose that has the nearest timestamp (after taking the synchronization constant into account; Vicon sampling frequency).

3x3 rotation matrix. Note that this rotation matrix can be represented by three parameters (yaw, roll and pitch). Thus, the code iterates over predefined values of the 6 parameters, such that every combination is tried. For each combination, the reprojection error is computed and stored for later. The parameters are continuous, but I try a sequence of values nearby the baseline value, with a constant offset. When it comes to the translation parameters, I have had good experience with trying 17 values, where the middle value is the baseline. The offset was 0.023 Matterport meters. Each orientation parameter was evaluated on 11 values with even offsets, where the middle value was the baseline. The offset was 0.5°. The brute-force search is very time consuming, taking about 20 hours on a machine capable of processing 45 threads at once. Optionally, one can iterate over 5 synchronization constant values, for even more optimal parameters to be found. Of course, by doing that, the search will take asymptotically 5 times as much time and memory resources.

Table 3.1 shows quantitative evaluation of the quality of reference poses, after the brute-force optimization. Table 3.2 shows the same statistics for parameters prior to the optimization (baseline transformation). The improvement is not significant: 1 cm lower translation error and 0.14° lower orientation error. Figure 3.4 shows an example of the 9 manually defined correspondences and their reprojection errors.

The resulting reference poses are not perfectly matching ground truth poses, which can be seen when projecting the reference poses and comparing the results with the query images. I have created the following procedure in order to estimate the mean translation and orientation error (reference vs ground truth poses). Although we do not know the true ground truth poses, one can use the poses from HoloLens.

Query ID	Average projection error [px]	Sum of projection errors [px]
1	3.47	31.24
94	9.80	88.19
237	10.06	90.52
281	3.83	34.48
155	5.07	45.63
198	3.23	29.10
Sum	N/A	319.16

(a) : Reprojection error.

	Mean errors	Standard deviation of errors
Translation [m]	0.15	0.08
Orientation [m]	2.09	1.69

(b) : Estimate of reference vs ground truth poses errors. All the queries in the sequence were considered, with two kinds of exceptions. Queries, for which we do not have a reference pose (Vicon got lost) are not considered in the statistics. Queries for which we do not have a corresponding pose from HoloLens (due to the delay) are also not included in the statistics. Ground truth poses are estimated from the poses provided from HoloLens, after conversion to World coordinate system.

Table 3.1: Quantitative evaluation of reference poses quality. HoloLens1 sequence shown. Parameters describing the Marker to camera transformation were **optimized** using brute-force search.

According to [17], the poses estimated by HoloLens have the following mean accuracy with respect to the ground truth poses:

- 1.6 ± 0.2 cm translation error,
- $2.2 \pm 0.3°$ orientation error.

Notice that namely the the translation error is very low. To estimate the quality of my reference poses with respect to (wrt) ground truth poses, I consider the HoloLens poses as the ground truth poses. However, because the poses from HoloLens are wrt some unknown initial HoloLens coordinate system (Omega), I first need to convert those poses to be wrt World. To achieve this, I use procrustes [32], which finds a linear transformation from one coordinate system to another (translation, rotation, scale), given corresponding 3D points. In my case, the 3D points are simply the camera centers. Procrustes minimizes the sum of squared errors of points in the same coordinate system. After the conversion, we would have ground truth estimates.

Unfortunately, there was another hidden problem that had to be dealt with, prior the reference vs ground truth pose errors could be computed. The problem is that the poses provided from HoloLens do not correspond to the query they are associated

Figure 3.4: Query 94 of HoloLens1 and its reprojections errors. The optimized transformation params were used. The same image on non-optimized parameters is not shown, because the average improvement of reprojection error of a the correspondences is about 2 pixels. Therefore a naked eye can barely tell which image has lower reprojection error. Green points: optimal location of 2D correspondences. Red dots: location of the 3D correspondences (projected onto 2D image plane), under the generic parameters (that aim to work across all queries in the sequence).

with in the data. It turns out the poses are delayed. To make matters worse, both the translation and orientation that are used to construct the camera pose are delayed by a different amount! To resolve this issue, the pose from HoloLens associated to a query is computed to be based on translation and orientation data, that comes from the future queries. I found that the best results were achieved with the delays in table 3.3.

A consequence of the data being delayed is that, for some of the queries at end of the sequence, we do not have the poses from HoloLens available. Recall also that some reference poses are blacklisted, because Vicon got lost.

Using these delays and the `procrustes` method[3], we can compute the mean reference vs ground truth pose errors, which is:

- 15 cm translation error,
- 2.09° orientation error.

These errors may be either an upper bound (the data being delayed may still cause trouble) of the real mean errors, but they can also be approximately the true

[3]See section 2.6.

Query ID	Average projection error [px]	Sum of projection errors [px]
1	3.38	30.42
94	11.96	107.66
237	9.22	82.98
281	3.62	32.57
155	5.99	53.91
198	3.08	27.68
Sum	N/A	335.22

(a) : Reprojection error.

	Mean errors	Standard deviation of errors
Translation [m]	0.16	0.08
Orientation [m]	2.23	1.62

(b) : Estimate of reference vs ground truth poses errors. All the queries in the sequence were considered, with two kinds of exceptions. Queries, for which we do not have a reference pose (Vicon got lost) are not considered in the statistics. Queries for which we do not have a corresponding pose from HoloLens (due to the delay) are also not included in the statistics. Ground truth poses are estimated from the poses provided from HoloLens, after conversion to World coordinate system.

Table 3.2: Quantitative evaluation of reference poses quality. HoloLens1 sequence shown. Tables show performance on the parameters, describing the Marker to camera transformation, **prior** using brute-force search optimization.

Type	Number of frames
Translation delay	6
Orientation delay	4

Table 3.3: We are using a program [33] for fetching data from HoloLens. It provides a CSV file containing information on the query images it took, when they were taken (timestamp), estimated poses and more. The camera pose estimates are represented by translation and orientation parameters, which are in Omega coordinate system. However, these parameters are wrongly assigned, as they are in fact delayed by a number of frames. The time difference between two consecutive frames is about 333 milliseconds. Optimal delays for HoloLens1 sequence are shown.

mean errors. As you can see, the translation error is significant. This is concerning, because it will not clear whether our newly developed localization methods are better than the poses provided by HoloLens themselves. Note that in case of s10e queries, the reference poses seem to have a lower error wrt ground truth. However, because we do not know the ground truth and no HoloLens poses are available here, I cannot quantitatively evaluate their quality wrt ground truth (but the reprojection error on the queries that were manually assigned 2D-3D correspondences can be computed).

The query images can be split into two categories — InMap and OffMap. An InMap query is such a query, for which we have a cutout that has a similar pose. I have defined the pose similarity as:

- the translation difference is less than 1.3 meters,
- the angular distance between the reference query and cutout rotation matrices is at most 10 degrees,

Where we define the angular distance between two 3×3 rotation matrices R_1 and R_2, representing two orientations, as:

$$\left| \arccos\left(\frac{\mathrm{Tr}(R_2 * R_1^{-1}) - 1}{2}\right) \right|. \tag{3.2}$$

The set of s10e queries consists of 5 InMap queries and 35 OffMap queries. The set of HoloLens1 queries consists of 111 InMap queries and 239 OffMap queries. The HoloLens2 does not have up to date reference poses. According to an outdated result, it contains 48 InMap and 570 OffMap queries.

The entire dataset, excluding the generated output, occupies about 130 GB of disk space.

The dataset statistics are depicted in table 3.4. Notice that the horizontal field of view of database cutout images is widely different from the query horizontal FoVs. When I tried to generate the dataset, such that the cutouts have horizontal FoV of 60 degrees, the resulting pose estimation accuracy became 0%. I had spent a significant time investigating why this is happening, and came to the conclusion that the problem was in the data. When one creates a cutout of a lower FoV, smaller portion of the 360° panorama gets rendered. This also means that the visual quality of the image decreases. I believed that the quality of such cutouts is not good enough for the convolutional neural network to generate reasonable feature descriptors. Figure 3.5 illustrates this problem. It seemed that there was nothing we can do about it, since the pixel density of each 360° panorama is determined by Matterport. It is, however, true that one could experiment with other FoV values. Such experiments were not conducted here, as regenerating the dataset and then uploading it to an evaluation server takes a lot of time (one day is not an exception). Why have I used a past tense? The thing is that the experiment, where cutouts with hFoV 60° are used, took place a while ago. Since then, a couple of bugs in my implementation were fixed. The hFoV 60° set-up shall be tried again (I did not have the resources to do it).

Handling of reflective surfaces. The scanned rooms contain a couple of types of objects that can confuse the Matterport scanner; luckily, the Matterport Capture iOS

Type	Amount	Without ref. pose	Image size [px]	HFoV [°]
Query - s10e	40	0	4,032×3,024	64.86
Query - HoloLens1	350	24	1344×756	65.83
Query - HoloLens2	618	299	1344×756	65.83
Cutout	2,088	0	1,600×1,200	106.26

Table 3.4: Statistics of the **InLocCIIRC dataset**. Note that some queries are without a reference pose assigned to them. This occurs when Vicon gets lost (returns a non-sense pose for a certain period of time). Such queries are ignored in performance evaluation. Note that the HoloLens2 sequence contains a lot of queries, for which Vicon failed. This may be related to the fact that I moved slighly faster around the room in that sequence, making it harder for Vicon to keep track of the marker. Cutout poses are provided from Matterport and because of their quantity, not all of them were manually verified. For I never discovered a problem with the cutout poses, I consider their poses to be flawless. HFoV stands for the horizontal field of view.

app contains tools designed to deal with such problems. After some experimentation, I obtained the highest-quality 3D model with the following settings:

Object type	**Handling in Matterport**
Outer window	Window
Indoor window	Window
TV	Default
Door with glass elements	Window

Table 3.5: Handling of reflective surfaces in the Matterport Capture app. The options are: Default, Window, Mirror. Best results were obtained using the values in this table.

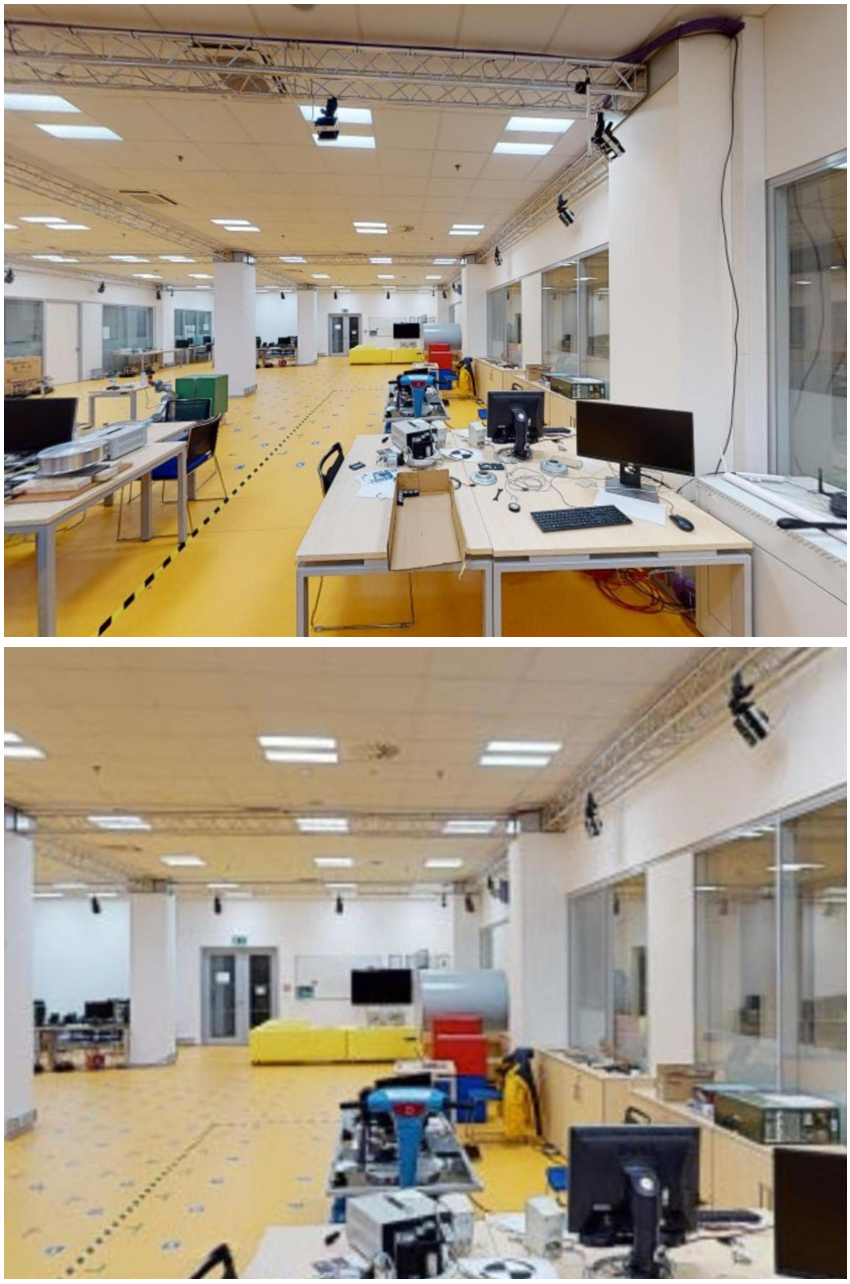


Figure 3.5: Visual quality comparision of the same cutout under different FoV. Top: horizontal FoV: 106.26°. Bottom: horizontal FoV: 60.00°. The image with a lower FoV contains a lot of artifacts and is of lower visual quality.

3.2 Habitat

We provide an (incomplete) tool for generating synthetic datasets in InLoc dataset-like format. The tool is a modification of the AI Habitat platform [34]. It currently supports creation of sequential query sets in digitalized environments, such as the Matterport3D dataset [35] and our InLocCIIRC dataset[4]. The implementation is available at [36]. It can be used for creation of large datasets (for evaluation of indoor localization methods) without an expensive 3D scanning equipment. However, evaluating an improved HoloLens tracking accuracy is not very meaningful here, as the data from HoloLens would need to be simulated.

3.2.1 Usage

1. Navigate to `habitat-api/examples/capture_sequence.py`.
2. Choose an environment in the `habitat-api/configs/datasets/pointnav/` folder.
3. Adjust the `environmentConfigPath` variable to the selected environment.
4. Adjust the `outputPath` variable.
5. Run the script.
6. Navigate around the environment using `WASD` and arrow keys.
7. After each movement, a query image is saved and its pose within that environment stored.
8. Press `F` to exit.

A future work could add support for the creation of the panorama/cutout images.

[4]Disclaimer: I am the author of the dataset and the modified Habitat source code. But in order to load the InLocCIIRC models into Habitat, a transformation must be performed to comply with the Habitat format. The transformation was developed and performed by Martina Dubeňová. If you need to load custom Matterport models into Habitat, feel free to browse the AI Habitat documentation or contact Martina at `dubenma1@fel.cvut.cz`.

Chapter 4

Implementation

InLoc [1] authors provide a demonstration in MATLAB that operates on the InLoc dataset. I have taken this demonstration and adjusted it, so that it works on the InLocCIIRC dataset instead. I have added an evaluation script, that was missing from the original code. Although the evaluation of InLoc is handled by `http://www.visuallocalization.net`, this tool of course doesn't handle the newly created InLocCIIRC dataset yet.

The entire InLocCIIRC implementation should run on a multi-core machine with a GPU. The number of processing CPU threads can be up to 45 at a time. In order to do this, I was running the program on a CMP[1] server. However, the GPU node prohibited the use of more than 8 CPU threads per user. Therefore the implementation was split into 2 parts: in the first run, the GPU is used; in the latter run, no GPU is required, but a CPU with a lot of cores is used. The need for a GPU comes from the fact that we are using inference of NetVLAD-based neural network, which would take much longer on a CPU. This GPU restriction is present in InLoc implementation as well.

The original InLoc implementation uses point cloud projection in the pose verification step. However, the code for point cloud (PC) projection did not support variable point size. Because the models in my datasets are not dense (compared to those taken with the Faro 3D scanner), the projection can sometimes see through pillars or objects that are close to the camera. This is not desirable, as seeing what is behind the object can result in a different NetVLAD descriptor that is not similar to the query image. At first I have implemented PC projection with a point size parameter,

[1] Center for Machine Perception at Czech Technical University in Prague.

but the problem is that it does not support headless[2] rendering. I ended up using a mesh model projection instead of a point cloud projection in the point verification step. I am using existing software packages to achieve this (`pyrender`, `trimesh`, `open3D`). My `projectMesh` method supports headless rendering. Unfortunately, it is very demanding - requires 14 GB RAM and it also takes time to load the dataset into memory. Of course, one would cache the model in memory and then just call the render functions. However, this would require non-trivial implementation changes, because the implementation is in MATLAB and the `projectMesh` routine is in Python.

A major change to the implementation was adding support for sequential queries. Currently the code[3] supports specifically sequential queries from HoloLens. To estimate poses (wrt World) of sequential queries from HoloLens, poses from HoloLens (wrt Omega) must be provided. The latter poses are computed by HoloLens itself. There are two approaches how the sequential nature of query sets is leveraged. Both approaches depend on a parameter k. We want to estimate the camera pose for each query in the query sequence. At each such query, we consider a segment of queries, such that the last query in the segment is the currently processed query. Constant k defines how long the segment is.

The first approach is called SequentialPV. It only leverages the other queries (in the segment) in the pose verification step. This approach aims to be more robust than the non-sequential one, by providing more evidence: the projection quality for all queries in the segment is considered and compared to the input query images. How is this done? We have top 10 camera poses (given by P3P in the pose estimation step). These poses are the estimated poses of the current query. Next, we have camera pose estimates for every query in the segment, provided by HoloLens. Those poses are wrt Omega. Therefore, we convert the poses from HoloLens from Omega to World, by aligning the two poses of the last query in the segment. The two poses are:

- the camera pose estimate (wrt World) provided from pose estimation step,
- the camera pose estimate (wrt Omega) provided from HoloLens.

To match the two poses, we just need to compute a linear transformation (rotation, translation). With this transformation, the other poses from HoloLens are converted from Omega to World. With all camera pose estimates being with respect to

[2]Headless rendering is rendering on a computer where the rendering program is not attached to a physical display.

[3]The repository implementing the new pipeline is called `InLocCIIRC_demo`, but the name is in fact not very accurate, because it is not really an implementation of the InLoc paper [1]. However, the pose estimation algorithms are indeed based on InLoc.

World coordinate system, we run the pose verification step. The pose verification step returns a score, symbolizing the quality of the input query image and the reconstructed query image. All the scores in the segment are summed up. Note that there are other ways of integrating the scores together: e.g. mean, maximum; I haven't tested them yet. This is done for those top 10 poses from the pose estimation step. At the end, I choose the candidate with highest score to represent the final camera pose estimate. This approach is a basic way to leverage the fact that the queries were taken in a sequence (and captured with HoloLens).

Approach two is called MultiCameraPose. We want to estimate pose of each query in the sequence by taking into account all poses and correspondences in the current segment. The camera pose estimates (wrt Omega) are taken from HoloLens. The 2D-2D correspondences are computed using the geometric verification step and are subsequently converted into 2D-3D correspondences. Given these data, an external program called `MultiCameraPose` [13] processes them and returns the camera pose estimates wrt World. The program contains an implementation of gsP4P [11]. For details on the `MultiCameraPose` program and gsP4P, please see Chapter 2. We then store all returned camera rig poses, as the main result of the pose estimation step. In the pose verification step, all the estimated poses within a particular segment are evaluated (score is computed). Again, the candidate segment with the highest cumulative (summed up) score is selected. The last pose from the estimated poses in the segment is selected to be the final camera pose estimate for current query.

There is an important change when MultiCameraPose is used, compared to processing non-sequential queries. To understand that, let me first describe how the pose estimation step works in the non-sequential case:

1. We are given top 100 candidate cutouts for each query. These cutouts aim to be visually similar to the query. They were constructed using the input `score` matrix.
2. Query and cutout features are extracted.
3. Geometric verification is executed for all query-cutout pairs. This gives us 2D-2D correspondences called "inliers" (some of which are inaccurate).
4. The top 100 candidates are re-ranked and sorted, so that query-cutout pairs with the highest number of inliers are preferred. If the number of inliers is the same, the original input `score` is used on top of it (floating point value between zero and 1).
5. Top 10 candidate cutouts for each query are chosen.
6. Each query-cutout pair and its 2D-2D correspondences are processed. Because one of 2D corresponding point sets lies in the cutout image, we can extract its

corresponding 3D points (the dataset provides depth and 3D point of every cutout pixel). The query-cutout 2D-3D correspondences are then passed into P3P. The camera pose is estimated.

7. We now have 10 candidate pose estimates for every query.

In the MultiCameraPose approach, the segments have length $k > 1$. We need to decide how to choose, for each query, top 10 *query-cutout segments*. The candidates will then be processed further using pose estimation and verification. Recall that the last query in the segment is always the one currently being processed (the one for which we want the camera pose). My current implementation does the following:

1. We have the re-ranked and sorted top 10 candidate cutouts for each query, as described in step 5 of the non-sequential pose estimation approach.
2. Generates all possible query-cutout segments of length k. There are 10^k possibilities.
3. Because k is expected[4] to be no more than 5, we can easily generate all the combinations.
4. Every query-cutout has a score assigned, as described in step 4 of the non-sequential algorithm. I simply choose the combinations which have the cumulative (summed up) score the highest. Top 10 combinations are selected.

The algorithm in step 4 may be a bit problematic for two reasons. First, some query-cutout pairs may naturally have more inliers (on average) than others. It might be sub-optimal to sum those scores. Instead, e.g. an average or a median should be considered. The second issue is that selecting 10 combinations from 10^k is not enough. But increasing the number of chosen top combinations is currently not possible, because pose verification is so slow. The second issue is described in more detail in subsection 5.4.3.

Another problem not considered in the reference InLoc implementation is related to the different aspect ratios of HoloLens queries and cutouts. Recall that HoloLens query images have a 1344×758 pixel resolution, whereas the cutouts have resolution of 1600×1200. This makes the HoloLens queries have $16 : 9$ aspect ratio and the cutouts have a $4 : 3$ aspect ratio. Why is that a problem? The methods used in geometric verification (GV) would break - the tentative correspondences would not be computed properly. Therefore, we need to provide the GV step with the same

[4]The higher the k, the higher the chance of data associated to a particular query in the segment is corrupted. This would negatively impact the resulting pose estimate precision.

aspect ratio (and also the same resolution by scaling). Of course, if we rescale the query image to match the cutout image dimensions, we will deform the query view. Therefore my solution is to add padding[5] on top and bottom of the query image; where the added pixels share the same constant value. The padding is added so that the aspect ratio matches the cutout aspect ratio. Then we can also rescale without deformation. In the pose estimation step, we have to undo the process on the query inliers, so that they correspond with the properties of the camera that took those images.

4.1 Source code and dataset structure

An umbrella repository referencing all the sub-projects has been created. It is available at [37]. A brief structure overview is provided here.

```
VisualLocalizationWithHoloLens
├── masters-thesis
│   ├── ⋮ ..... thesis source in LaTeX format
│   └── masters-thesis.pdf
├── InLocCIIRC_demo
├── InLocCIIRC_dataset (repo)
├── InLocCIIRC_dataset
├── InLocCIIRC_utils
├── MultiCameraPose
├── Habitat
│   ├── habitat-api
│   └── habitat-sim
```

Figure 4.1: Sub-projects listed. Only those I have been working on are listed.

[5] Another approach would be to crop the query image to achieve the same aspect ratio; then rescale without deformation. Of course, the problem with this approach is that we would lower the horizontal field of view, which would mean less RGB data to work with in the transformed query image.

```
InLocCIIRC_demo
├── functions
│   ├── InLocCIIRC_utils
│   ├── ⋮ ..... other external dependencies
│   └── wustl_function
│       ├── parfor_denseGV.m
│       ├── parfor_densePE.m
│       └── parfor_densePV.m
├── evaluate.m
├── inloc_demo.m. ..... main entry-point
├── ht_top10_densePV_localization.m
├── ht_top100_densePE_localization.m
├── evaluate.m
├── P3P_vs_MCP.m
└── PE_stability_check.m
```

Figure 4.2: The organization of the source code of the core project. Only notable items shown.

```
InLocCIIRC_dataset (repo)
├── buildCutouts
├── buildFileLists
├── buildScores
│   ├── buildFeatures.m ..... step 1
│   └── buildScores.m ..... step 2
├── functions ..... external dependencies
├── query
│   ├── buildRawPoses.m ..... optimize reference poses
│   ├── holoLensPoses.m ..... converted to be wrt World
│   └── transformPoses.m ..... create reference poses
└── README.md ..... detailed description
```

Figure 4.3: The structure of the repository containing the dataset construction tool. It also contains other useful scripts. Only notable items shown. Some of the code depends on a `PanoBasic` project [38] [39].

```
InLocCIIRC_dataset
├── cutouts/
├── evaluation-*/
├── inputs-*
│   ├── cutout_imgnames_all.mat
│   ├── query_imgnames_all.mat
│   ├── scores.mat
│   └── features/
├── models/
├── outputs-*
│   ├── densePE_top100_shortlist.mat
│   ├── densePV_top10_shortlist.mat
│   ├── gv_dense/
│   ├── PnP_dense_inlier/
│   └── synthesized/
├── panoramas/
├── query-*
│   ├── 1.jpg
│   ├── 2.jpg
│   ├── ⋮
│   ├── HoloLensPoses/ ..... if applicable; wrt World
│   ├── poses/ ..... reference poses
│   └── projectedPoses/ ..... visual quality of ref. poses
├── sweepData/ ..... from Matterport API
├── HoloLens sequences/ ..... data from the two HoloLens sequences
└── Habitat[a]/ .. data for AI Habitat experiments on top of our dataset
```

[a]Disclaimer: I am the author of the dataset and the modified Habitat source code. But in order to load the InLocCIIRC models into Habitat, a transformation must be performed to comply with the Habitat format. The transformation was developed and performed by Martina Dubeňová. If you need to load custom Matterport models into Habitat, feel free to browse the AI Habitat documentation or contact Martina at `dubenma1@fel.cvut.cz`.

Figure 4.4: The structure of the InLocCIIRC dataset. Only notable items shown.

```
MultiCameraPose
└── src
    ├── multi_camera_pose.cc
    └── common.h
```

Figure 4.5: The structure of the modified `MultiCameraPose` repository. Only notable items shown.

4.2 Pseudocode

```
mode = 'non-sequential' or 'sequentialPV' or 'MultiCameraPose'
segmentLength = 3 # aka 'k'; considered in 'sequentialPV', 'MultiCameraPose' modes
querySet = 's10e' or 'holoLens1' or 'holoLens2'
topRetrieval = 100
topGV = 10
topPE = 10
topPV = 1
neuralNet = NetVLAD()
coarseLayer = 'conv5'
fineLayer = 'conv3'

def main():
  score, queryNames, cutoutNames = initialize()
  assertNonSequentialModeUsedIfQuerySetIsNonSequential()
  # score represents query-cutout score matrix
  ImgList = retrieval(score, queryNames, cutoutNames)
  ImgList = poseEstimation(ImgList)
  ImgList = poseVerification(ImgList)
  evaluate(ImgList)

def initialize():
  return implementationDetail()

def addSecondaryQueries(ImgList, score, queryNames, cutoutNames):
  # primary query is a query user requested to perform pose estimation on.
  # secondary queries are part of the k-segments of primary queries.
  # they need to be added in 'MultiCameraPose' mode to be processed by
  # poseEstimation() onward
  implementationDetail()

def retrieval(score, queryNames, cutoutNames):
  ImgList = list()
  for i in len(queryNames):
    queryName = queryNames
    ImgList[i].queryname = queryName
    sortedScores, ind = sort(score[queryName].scores, 'descend')
    ImgList[i].topNname = cutoutNames[ind[0:topRetrieval]]
    ImgList[i].topNscore = sortedScores[0:topRetrieval]
  if mode == 'MultiCameraPose'
    addSecondaryQueries(ImgList, score, queryNames, cutoutNames)
  return ImgList

def extractFeatures(image):
  image = neuralNet.averagingImageNormalization(image)
  allLayerResults = neuralNet.forward(image)
  features = allLLayerResults # we need features from different layers
  return features
```

```
48
49  def loadQueryImageCompatibleWithCutouts(queryImage):
50    queryImage = padImageByAddingRowsToMatchCutoutAspectRatio(queryImage)
51    queryImage = scaleImageToMatchCutoutDimensions(queryImage)
52    return queryImage
53
54  def adjustInliersToMatchOriginalQuery(queryTentatives, queryDimensions,
        cutoutDimensions):
55    # reverts loadQueryImageCompatibleWithCutouts(...)
56    return implementationDetail(...)
57
58  def buildFeatures(ImgList):
59    features = list()
60    for i in range(len(ImgList)):
61      queryName = ImgList[i].queryname
62      thisQueryFeatures = list() # query image features followed by
            'topRetrieval' cutout features
63      queryImage = loadImage(queryName)
64      queryImage = loadQueryImageCompatibleWithCutouts(queryImage)
65      thisQueryFeatures.append(extractFeatures(queryImage))
66      for j in topRetrieval:
67        cutoutName = Imglist[i].topNname[j]
68        cutoutImage = loadImage(cutoutName)
69        thisQueryFeatures.append(extractFeatures(cutoutImage))
70      features.append(thisQueryFeatures)
71    return features
72
73  def coarseToFineMatching(queryFeatures, cutoutFeatures):
74    queryCoarseFeats = getFeaturesAtLayer(queryFeatures, coarseLayer)
75    cutoutCoarseFeats = getFeaturesAtLayer(cutoutFeatures, coarseLayer)
76    queryFineFeats = getFeaturesAtLayer(queryFeatures, fineLayer)
77    cutoutFineFeats = getFeaturesAtLayer(cutoutFeatures, fineLayer)
78    f1 = queryFineFeats
79    f2 = cutoutFineFeats
80    match12 = findNearestMatches(queryCoarseFeats, cutoutCoarseFeats)
81    return f1, f2, match12
82
83  def sortImgListRowByHighestScores(ImgListRow):
84    for i in len(queryNames):
85      sortedScores, ind = sort(ImgListRow[i].topNscore, 'descend')
86      ImgListRow[i].topNname = ImgListRow[i].topNname[ind]
87      ImgListRow[i].topNscore = ImgListRow[i]topNscore[ind]
88    return ImgListRow
89
90  def geometricVerification(ImgList, features):
91    NewImageList = ImgList.copy()
92    for i in range(len(ImgList)):
93      thisQueryFeatues = features[i]
94      queryName = ImgList[i].queryname
95      parfor j in range(topRetrieval):
96        cutoutName = Imglist[i].topNname[j]
97        queryImgFeatures = thisQueryFeatures[0]
```

```
98        cutoutImgFeatures = thisQueryFeatures[1+j]
99        match12, f1, f2 = coarseToFineMatching(queryImgFeatures,
              cutoutImgFeatures)
100       inls12 = denseRansac(f1, f2, match12)
101       save(queryName, cutoutName, f1, f2, match12, inls12)
102       NewImgList[i].topNscore[j] += len(inls12) # NOTE: the previous
              scores were between zero and one
103     NewImageList[i] = sortImgListRowByHighestScores(NewImgList[i])
104   return NewImageList
105
106 def getActualSegmentLength(idx, desiredSegmentLength, ImgList):
107   return
          getSegmentLengthSuchThatSegmentQueriesAreWithinSequenceBounds(idx,
          desiredSegmentLength, ImgList)
108
109 def getCandidatesForQueries(ImgList):
110   # for each query, we have multiple candidate solutions.
111   # parfor_densePE and parfor_densePV functions must be executed on
112   # all of those candidates
113   return implementationDetail()
114
115 def poseEstimation(ImgList):
116   features = buildFeatures(ImgList)
117   ImgList = geometricVerification(ImgList, features)
118   treatQueriesSequentially = mode == 'MultiCameraPose'
119   if not treatQueriesSequentially:
120     desiredSegmentLength = 1
121   else:
122     desiredSegmentLength = segmentLength
123   ImgListSequential = keepPrimaryQueriesOnly(ImgList)
124   for i in range(len(ImgListSequential)):
125     actualSegmentLength = getActualSegmentLength(i, desiredSegmentLength,
          ImgListSequential)
126     combinations = permuteIndices([0:topGV], actualSegmentLength)
127     queryName = ImgListSequential[i].queryname
128     scores = computeScoresForSegmentCombinations(combinations, ImgList,
          queryName, 'cumulative-sum')
129     ind = findBestCombinations(scores, topPE)
130     updateTopCutoutsAndScoresInTheSegment(ImgListSequential[i], scores,
          ind)
131   if treatQueriesSequentially:
132     posesFromHoloLens = getPosesFromHoloLens()
133   else:
134     posesFromHoloLens = list()
135
136   parfor queryName, candidateIdx in
        getCandidatesForQueries(ImgListSequential):
137     parfor_densePE(ImgListSequential, queryName, posesFromHoloLens,
          candidateIdx)
138
139   for i in len(ImgListSequential)
```

```
140       ImgListSequential[i].Ps = list(size=topPE) # estimated poses in the
              segment, for topPE combinations
141       for j in topPE:
142         ImgListSequential[i].Ps[j] = load_parfor_densePE_segment_poses(i, j)
143
144     return ImgListSequential
145
146   def parfor_densePE(ImgList, parentQueryName, posesFromHoloLens,
          candidateIdx):
147     actualSegmentLength = getActualSegmentLength(parentQueryName,
            implementationDetail(), ImgList)
148     Ps = list(size=actualSegmentLength)
149     useP3P = segmentLength == 1
150     if invalidPosesDueToDelay(posesFromHoloLens):
151       useP3P = True
152     for j in segmentLength:
153       queryName = getQueryNameBasedOnParentQueryName(j, parentQueryName)
154       f1, f2, match12, inls12 = load(queryName, cutoutName, candidateIdx)
155       queryTentatives = f1[inls12[0]]
156       cutoutTentatives = f2[inls12[2]]
157       queryTentatives = upscale(queryTentatives, cutoutSize)
158       queryTentatives = adjustInliersToMatchOriginalQuery(queryTentatives,
              queryDimensions, cutoutDimensions)
159       correspondences = build2D3DCorrespondences(queryTentatives,
              cutoutTentatives)
160
161     if useP3P:
162       P, inls = P3P(correspondences)
163       Ps[end] = P
164       save(parentQueryName, candidateIdx, inls)
165     else:
166       Ps = multiCameraPose(correspondences, posesFromHoloLens)
167
168     save(parentQueryName, candidateIdx, Ps)
169
170   def convertHLPosesToBeWrtCurrentQueryPoseEstimate(posesFromHoloLens):
171     # it should be clear how to do this from my textual description in the
            Implementation Chapter 4
172     return implementationDetail()
173
174   def parfor_densePV(ImgList, parentQueryName, candidateIdx):
175     queriesInSegment = getQueriesInSegment(parentQueryName)
176     cutouts = getCutoutsInSegment(parentQueryName)
177     Ps = load(parentQueryName, candidateIdx)
178     for i in range(len(queriesInSegment))
179       queryName = queriesInSegment[i]
180       cutoutName = cutouts[i]
181       P = Ps[i]
182       queryImage = loadQueryImage(queryName)
183       synthQueryImage = projectPose(P)
184       error = compute_DSIFT_error(queryImage, synthQueryImage)
```

```
185       save(parentQueryName, candidateIdx, queryName, cutoutName, error,
              synthImage)
186
187   def poseVerification(ImgList):
188     PV_list = setUpListForPoseVerificationProcessing(ImgList)
189     if mode == 'sequentialPV':
190       posesFromHoloLens = getPosesFromHoloLens()
191       posesFromHoloLens =
              convertHLPosesToBeWrtCurrentQueryPoseEstimate(posesFromHoloLens)
192       addPosesFromHoloLensForPoseVerificationProcessing(PV_list,
              posesFromHoloLens)
193
194     parfor queryName, candidateIdx in getCandidatesForQueries(PV_list):
195       parfor_densePV(PV_list, queryName, candidateIdx)
196
197     PV_list = reRankSortAndChooseTop(PV_list, topPV)
198     return PV_list
199
200   def evaluate(ImgList):
201     # chooses top 1 poseVerification results for each query
202     visualEvaluationQueries(ImgList)
203     visualEvaluationQuerySegments(ImgList)
204     computeTranslationAndOrientationErrorsWrtReferencePoses(ImgList)
205     showLocalizationAccuracyGivenThresholds()
206     showErrorStatistics()
```

Algorithm 4.1: `InLocCIIRC_demo` pseudocode.

Note that in the current version of the actual source code, I do not have the 'non-sequential' mode. Instead, it is determined by choosing 'MultiCameraPose' mode and setting segmentLength to 1.

4.3 MultiCameraPose

The `MultiCameraPose` project has been slightly modified and used as an external dependency. The modified source code is available at [13].

`MultiCameraPose` estimates the poses of the cameras in the rig. It is given a set of poses of those cameras wrt to some (unknown) coordinate system. For each camera, 2D-3D correspondences are provided. The resulting pose estimates will be wrt the coordinate system in which the 3D correspondences were provided.

4.3.1 Introduced changes

- Comments for easier code understanding.
- The core multi-pose estimation procedure runs `num_global_iterations`-times.
- At the end of each global iteration, the new estimate is considered the best so far, if it matches the following criteria:
- Criteria 1: The median translation error is not higher than the median translation error associated with the previous best estimate,
- Criteria 2: The median orientation error is not higher than the median orientation error associated to the previous best estimate.
- Fixed a bug regarding translation error computation.
- Added a build script.
- Other small changes.

4.3.2 Usage

The `MultiCameraPose` project is written in C++. It must first be compiled so that executable programs are created. An example procedure on how to build the project is in `make_cmp.sh` file. The project contains several executables, of which we are only interested in the `multi_camera_pose` one. It requires a set of command line arguments and input files, which will not be described here. However, I have created a function in MATLAB, that:

1. Sets-up the necessary command line arguments.
2. Sets-up the necessary input files.
3. Executes the executable file.
4. Fetches the results.
5. Gives the user relevant results.

The MATLAB function is present at `InLocCIIRC_utils/multiCameraPose/multiCameraPose.m`. Its usage is described in table 4.1.

Parameter	Data type	Description
`workingDir`	string	Path to a directory where to create auxiliary files.
`queryInd`	$n \times 1$ integer	The IDs of the queries we are processing.
`allCorrespondences2D`	$n \times 1$ cell array	Each element contains the 2D query correspondences. Each element is a $2 \times l$ double array, where l is the number of correspondences found.
`allCorrespondences3D`	$n \times 1$ cell array	Each element contains the 3D cutout correspondences. Each element is a $3 \times l$ double array, where l is the number of correspondences found.
`inlierThreshold`	double	Unused in `multi_camera_pose`.
`numLoSteps`	integer	Number of steps in internally used Locally Optimized RANSAC.
`invertYZ`	boolean	Multiplies the YZ coordinates of the 3D correspondences by -1.
`pointsCentered`	boolean	If not, the 2D correspondences are transformed, so that their origin is at (`imageWidth`/2, `imageHeight`/2).
`undistortionNeeded`	boolean	Corrects the impact of lens distortion[6] on the 2D correspondences.
`imageWidth`	integer	How wide the camera sensors are [px].
`imageHeight`	integer	The height of the camera sensors [px].
`K`	3×3 double	The camera calibration matrix (See section 2.4).
`params`	struct	Contains experiment-specific parameters. See `InLocCIIRC_utils/params/setupParams.m`.

Table 4.1: The input parameters of the `multiCameraPose` MATLAB function. The function acts as an interface to the `multi_camera_pose` executable program.

The function has a single output - `posesWrtModel`. It is a $1 \times n$ cell array. Each element is a 3×4 double. The 3×3 sub-matrix on the left is a rotation matrix R; it converts World bases into camera bases. The remaining 3×1 vector on the right is World origin wrt camera coordinate system.

[6] Consider reading [40] to learn more about what lens distortion is.

Chapter 5

Evaluation

5.1 Experiment design

Performance of the implemented solution had to be evaluated quantitatively for all the three main methods: Non-sequential method, sequentialPV method and the MultiCameraPose method. The two latter methods are designed to work with segments of queries, therefore different segment lengths, denoted by constant $k > 1$ were evaluated. Some of the promising methods were also visualized for human-friendly qualitative evaluation. The main point of the visualizations is to better understand the sources of errors (if any), which are described in section 5.4.

In order to measure how the InLocCIIRC algorithm is performing, the percentage of correctly localized poses within a threshold from a reference pose has been measured. Absolute position difference threshold is one of the following values, with decreasing difficulty: 0.25m, 0.50m, 1.00m. Angular threshold is set to 10°.

However, we first need to compute the translation and orientation errors for individual queries. An example of such data can is in table 5.1. This table shows the content of `evaluation-s10e/errors.csv` file. In general, if a row with `NaN` entries is present in an `evaluation/errors.csv` file, it is the consequence of one of the following:

a) `parfor_densePE` returned the cell array `Ps` with a `NaN` P matrix.

a. Data from HoloLens are missing or their poses contain `NaN`; (if applicable).
b. The last query-cutout in a current segment ($k > 0$) has an insufficient amount of correspondences.
c. The result of P3P or `multiCameraPose` contains `NaN`.

b) We do not have a reference pose for the query.

c) The estimated pose was in a different space than the reference query pose.

Another quantitative evaluation to consider is computing the statistics on the translation and orientation errors. For this, the mean, median, and standard deviation (std) were chosen.

A descriptive way to compare multiple methods is to compare percent of correctly localized queries, as the translation threshold increases (the orientation threshold is fixed).

I provide two kinds of visualization. The first one shows, for a subset of queries, how they are processed - the closest cutout found, the inliers used to reconstruct the camera pose, and an error map. This is also useful when determining why InLocCIIRC performs poorly on certain queries. The second kind of visualization shows a top level view of a scanned environment, including some localization data: the queries, estimated query poses and sweeps are drawn.

5.2 s10e query set

Evaluation results of the non-sequential s10e query set are shown in this section. Table 5.1 shows the errors in pose estimation for individual queries.

Table 5.2 shows the performance under the various thresholds. The InMap/OffMap performance is also shown. Error statistics are shown in table 5.3. Figure 5.2 shows how the localization accuracy changes given increasing translation error threshold.

Figure 5.1 shows example queries, how they are being processed and what is the localization result.

We say that InLocCIIRC *got completely lost* when the pose estimate of a query is `NaN` or a wrong space was estimated.

Query ID	InMap	Translation [m]	Orientation [°]
1	Yes	0.0880	1.5386
2	Yes	0.1832	1.2987
3	No	0.1709	1.4430
4	Yes	0.1065	1.2635
5	Yes	0.2185	0.4119
6	No	0.1324	1.1850
7	Yes	0.0752	0.9260
8	No	0.2015	1.2524
9	No	0.1167	1.0074
10	No	0.1302	0.4764
11	No	0.1152	0.7651
12	No	0.2927	1.0733
13	No	0.9610	13.8282
14	No	0.1324	2.0652
15	No	0.1320	1.6251
16	No	0.3284	4.1566
17	No	0.0745	1.6222
18	No	0.1053	0.6402
19	No	0.0259	1.4572
20	No	0.0788	0.3069
21	No	0.1652	1.5283
22	No	0.2209	1.3378
23	No	0.1552	2.8651
24	No	0.6788	2.7907
25	No	0.0944	1.0374
26	No	0.4796	1.6020
27	No	0.1465	2.4818
28	No	0.0779	0.8901
29	No	0.0538	1.4261
30	No	0.0305	2.0371
31	No	0.1258	1.1690
32	No	0.1369	1.9361
33	No	0.2868	2.6586
34	No	0.2834	3.3971
35	No	2.3826	0.4984
36	No	0.1939	2.0936
37	No	0.1471	2.6406
38	No	0.0761	1.4153
39	No	0.2338	2.6006
40	No	7.8370	153.0940

Table 5.1: Pose estimation errors on s10e query images.

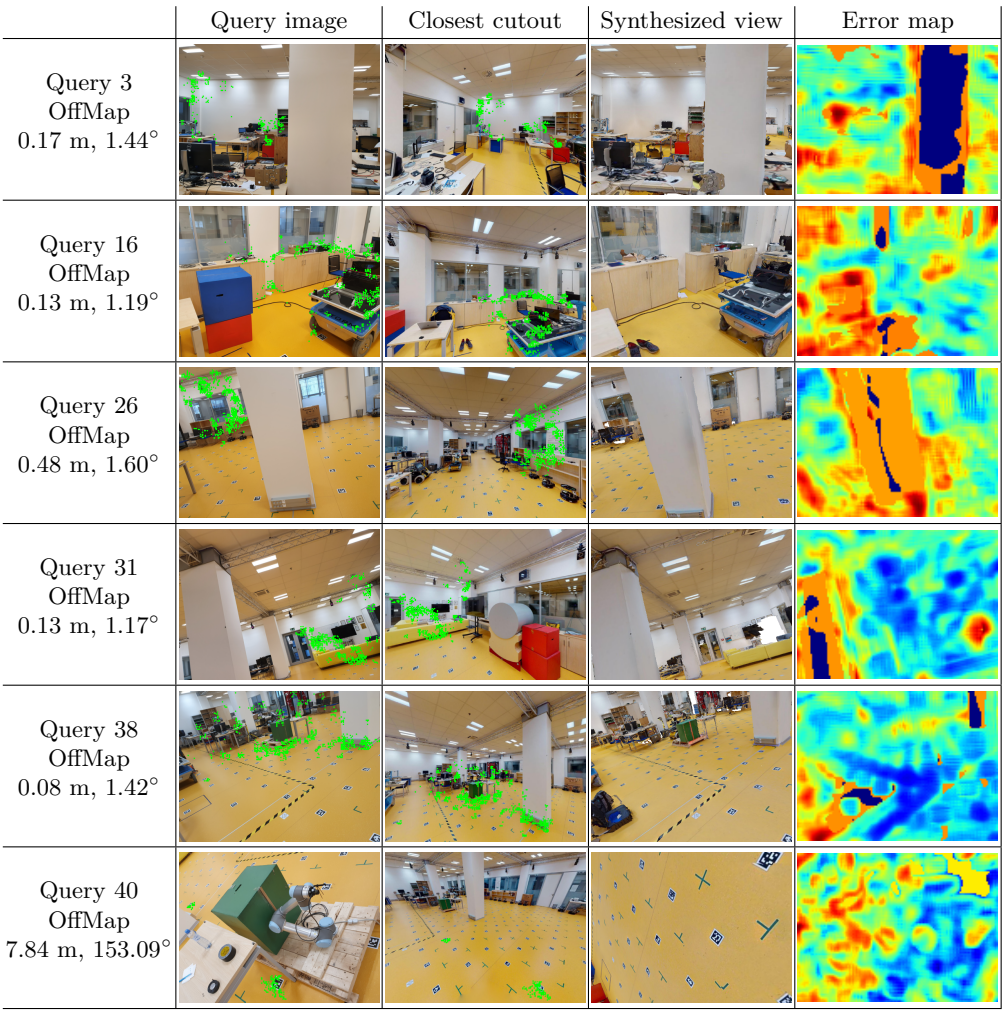


Figure 5.1: Qualitative comparison of s10e queries localization. From left to right: Query name and localization error (meters, degrees), query image, the best matching database image, synthesized view at the estimated pose, error map between the query image and the synthesized view. Green dots are the inlier matches obtained by P3P-LO-RANSAC. The majority of query images shown here are well localized within 0.5 meters and 5.0 degrees. All of the shown queries are OffMap, to test challenging estimation scenarios. InLocCIIRC struggles to find correct inliers on query 40, see subsection 5.4.5 for an investigation.

Figures 5.3 and 5.4 depict the dataset including the localization results.

Threshold	InLoc	**InLocCIIRC**	InMap	OffMap
0.25m	38.9%	**77.50%**	100.00%	74.29%
0.50m	56.5%	**90.00%**	100.00%	88.57%
1.00m	69.9%	**92.50%**	100.00%	91.43%

Table 5.2: Evaluation of performance of localization methods. The method in the first column was run on InLoc dataset. The second column method was run on the s10e query set of the InLocCIIRC dataset. Percentage rate of correctly localized queries within given threshold is shown. Angular threshold is equal to 10° in every row. The last two columns belong to InLocCIIRC method. InMap queries are queries for which we have a similar cutout in the dataset.

Statistics \ Error type	Translation [m]	Orientation [°]
Mean	0.44	5.70
Median	0.14	1.45
Standard deviation	1.26	24.00

Table 5.3: Statistics of the s10e pose estimation errors. InLocCIIRC got completely lost 0 out of 40 times. Not included in the mean/median/std errors. Errors are computed by comparing InLocCIIRC pose estimates with reference poses. Notice that the deviations are high. This is caused by the query 40 performing extraordinarily poorly.

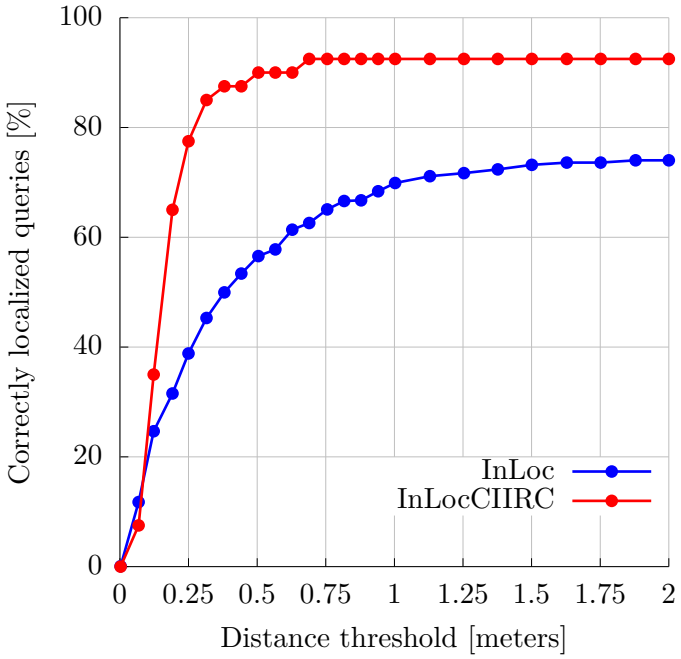


Figure 5.2: Comparison between InLoc and InLocCIIRC on their respective datasets. The s10e query set was used for InLocCIIRC. The independent variable describes the maximum allowed translation error. The angular threshold is set to 10°.

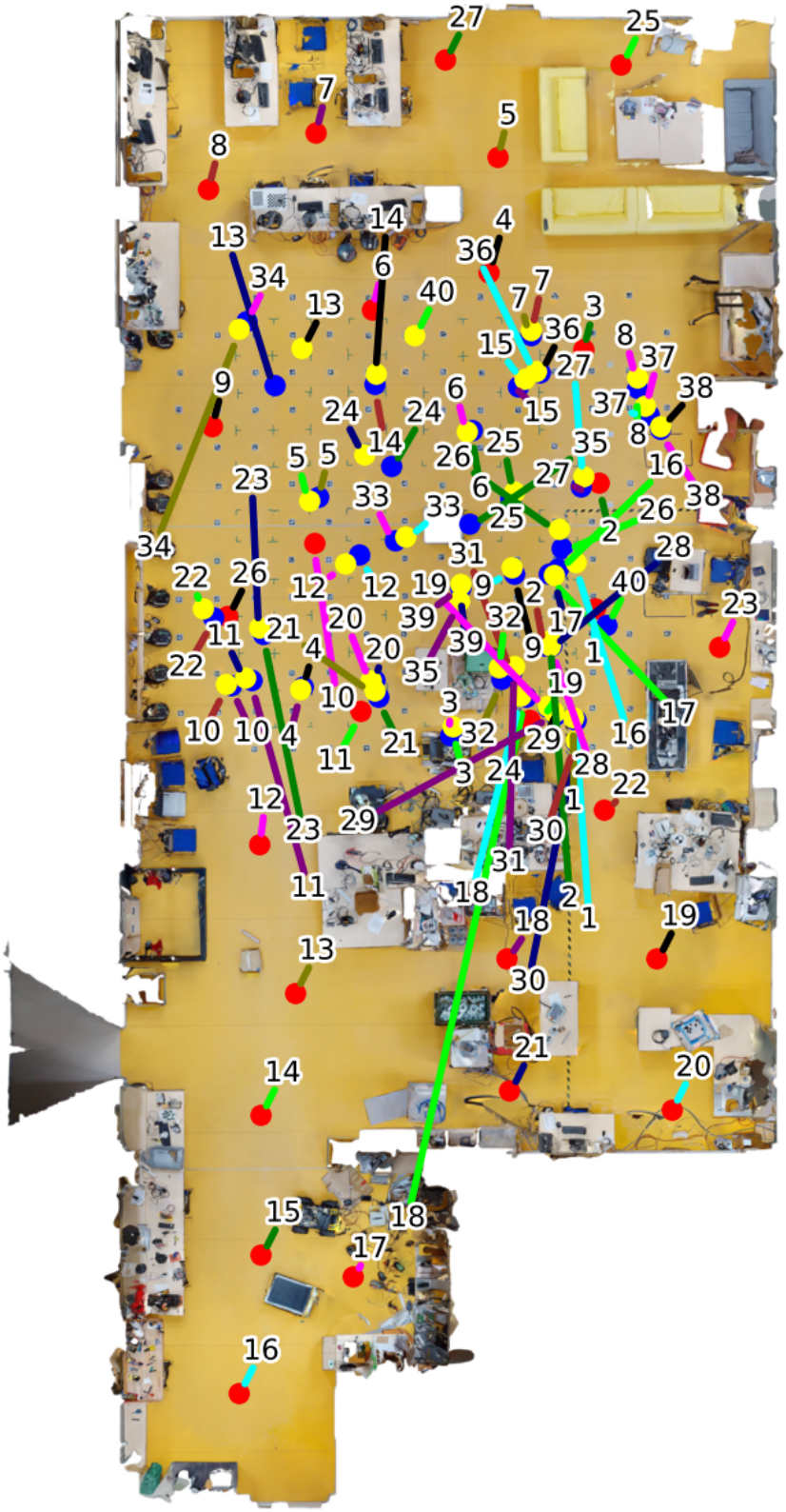


Figure 5.3: View on the floor plan of room B-315. Red dots: sweeps. Blue dots: queries. Yellow dots: estimated query poses. The s10e query set was used.

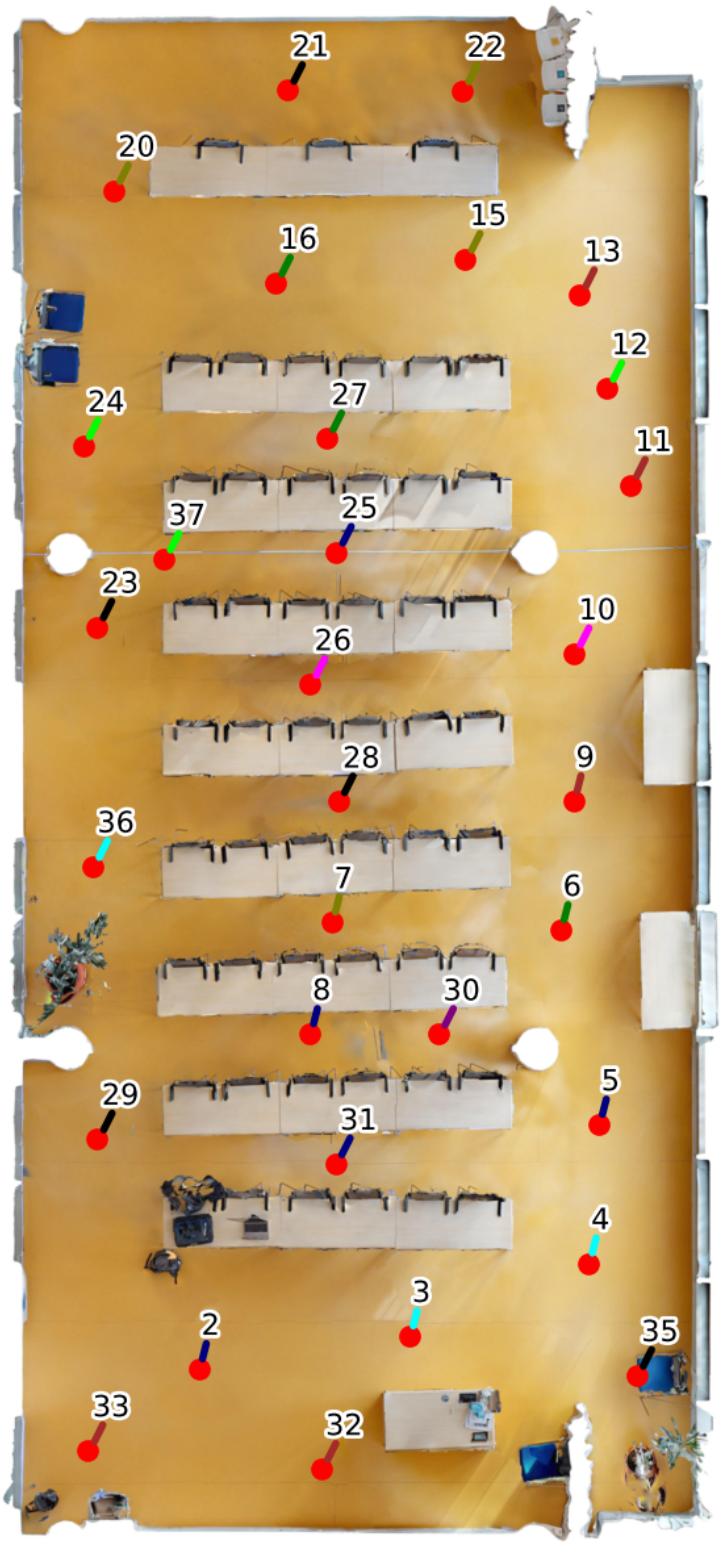


Figure 5.4: View on the floor plan of room B-670. Red dots: sweeps. Blue dots: queries. Yellow dots: estimated query poses. No s10e queries were incorrectly localized to this room.

5.3 HoloLens1 query set

5.3.1 Summary

Method \ Threshold	0.25m	0.50m	1.00m
k=1 (non-sequential)	63.80%	81.90%	85.89%
sequentialPV, k=2	63.80%	82.52%	86.50%
sequentialPV, k=3	63.80%	83.44%	86.50%
sequentialPV, k=4	61.96%	82.52%	85.28%
MultiCameraPose, k=2	**68.41%**	**83.74%**	**87.12%**
MultiCameraPose, k=3	**68.41%**	81.60%	86.20%
MultiCameraPose, k=5	67.18%	80.67%	85.58%
HoloLens	**84.36%**	**97.55%**	**97.55%**

Table 5.4: Evaluation of performance of localization methods on HoloLens1 query set (part of the InLocCIIRC dataset). Percentage rate of correctly localized queries within given threshold is shown. Angular threshold is equal to 10° in every row. The HoloLens method are the poses provided by HoloLens tracking itself, after being converted to be wrt World coordinate system. As it can be seen, it is superior to all the custom methods I have tried.

Method \ Statistics	Mean		Median		Std	
k=1	0.52m	3.62°	0.18m	2.01°	1.64m	11.33°
sequentialPV, k=2	0.52m	3.05°	0.17m	2.06°	1.64m	5.21°
sequentialPV, k=3	**0.45m**	3.04°	0.18m	2.02°	**1.36m**	5.21°
sequentialPV, k=4	0.60m	3.61°	0.19m	2.02°	1.93m	11.47°
MCP, k=2	0.53m	2.76°	**0.16m**	**1.85°**	1.84m	4.41°
MCP, k=3	0.54m	**2.68°**	**0.16m**	1.99°	1.85m	**3.06°**
MCP, k=5	0.60m	3.59°	0.17m	2.02°	1.92m	11.26°
HoloLens	**0.15m**	**2.09°**	**0.14m**	**1.51°**	**0.08m**	**1.69°**

Table 5.5: Statistics of the HoloLens1 pose estimation errors. InLocCIIRC got completely lost 29 out of 350 times for all methods (except the HoloLens method). The HoloLens method got completely lost 6 out of 350 times, which is caused by the HoloLens delay (see table 3.3). The *completely lost* cases are not included in the mean/median/std errors. Errors are computed by comparing InLocCIIRC pose estimates (or the pose estimates from HoloLens converted to be wrt World CS) with reference poses. The errors in [m] units are translation errors and the errors in [°] units are orientation errors. Lowest errors are highlighted in bold. MCP stands for MultiCameraPose. The original HoloLens method is superior to all the custom methods I have tried. Note that if the estimated poses were compared to the (unknown) ground-truth poses, the errors would likely be even lower, as discussed in the Reference poses section 3.1.

5.3.2 Best custom method

Judging from the results above, the best performing custom method is MultiCameraPose with sequence length $k = 2$.

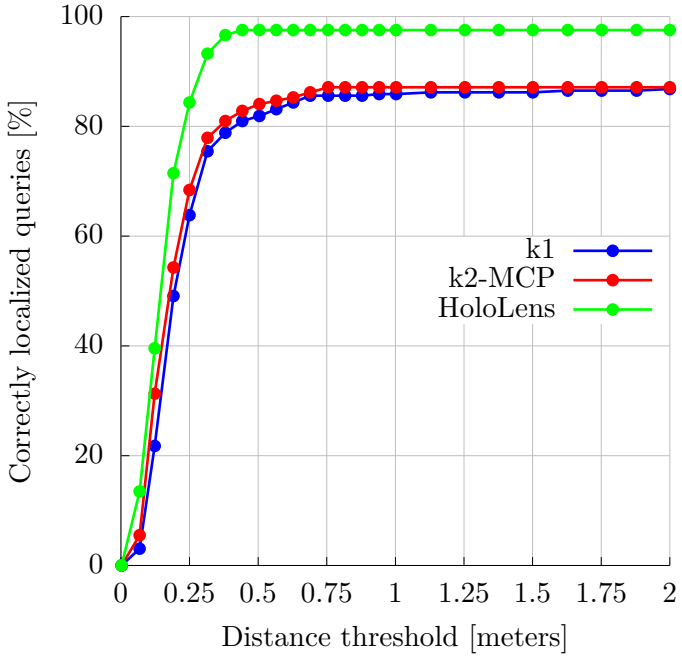


Figure 5.5: Evaluation of methods on the HoloLens1 query set. Comparison between the baseline method ($k = 1$, i.e. non-sequential) with the best performing custom method ($k = 2$, MultiCameraPose). The original HoloLens method, that we are aiming to surpass is also shown. The independent variable describes the maximum allowed translation error. The angular threshold is set to 10°.

Figure 5.7 depicts the top-view of B-315 including a subset of localization results (every 20th HoloLens query is rendered). View of room B-670 is not shown, as for this subset of results, it looks the same as in case of s10e query set, see figure 5.4. This means that none of the queries in the subset were incorrectly localized in room B-670.

Figure 5.6: Qualitative comparison of HoloLens1 queries localization. From left to right: Query name and localization error (meters, degrees), query image, the best matching database image, synthesized view at the estimated pose, error map between the query image and the synthesized view. Green dots are the inlier matches obtained by geometric verification. The pose estimation of query 84 is not completely wrong by human standards. InLocCIIRC matched the query image with a very similar cutout image, that is, however, at another location. Although this query is InMap, the chosen cutout is not the one that forms the InMap propery. Note that the query images have a different aspect ratio than the cutout images. The error maps not shown to save space.

Figure 5.7: View on the floor plan of room B-315. Red dots: sweeps. Blue dots: queries. Yellow dots: estimated query poses. Every 20th HoloLens1 query rendered.

5.4 Sources of errors

5.4.1 Previous queries have meaningful correspondences but current query does not have any correspondences

This was observed on MultiCameraPose, k=2 experiment. This results in InLocCIIRC completely being lost (returning `NaN` estimated pose), thus limiting the number of correctly localized queries given translation/orientation thresholds. In this scenario, the queries in the segment prior to the current query being processed have 2D-3D correspondences (found using geometric verification). Furthermore, those queries look meaningful upon manual inspection. However, we are interested in the current query, which does not have any correspondences. `MultiCameraPose` does not support a rig containing a camera for which there are no correspondences. Of course, we cannot use P3P on the current query, without knowing the query-cutout correspondences. Potential solution to this problem is: use the last estimated non-`NaN` pose in a sequence of queries ending with the current query. Limit the number as to how far into history to go. I did not have time to implement it. Known affected queries in HoloLens1 query set: 88, 122, 148, 231, 233, 236, 315, 319, 341. Why did we find no correspondences at those affected queries? For query 122 it is understandable - there was a very fast movement. For query 174 - it is somehow difficult, even the preceding queries 170-173 were hard to estimate (resulting poses were not `NaN`, but the errors from reference poses were high). However, for some affected queries, namely query 88, 148 and 231, a problem was discovered. The next subsection describes the problem.

5.4.2 Bad input `score` matrix

This was observed on non-sequential (k=1) and MultiCameraPose, k=2 experiments. Known affected queries: 88, 148, 231[1]. This issue probably affects more queries than is currently known by me. It causes no 2D-3D correspondences to be found. It is a problem, that currently causes `NaN` pose estimate for the affected queries. But it can also lower estimation accuracy for the successor queries, if the affected query is considered within its segment. This is because currently, P3P is used (non-sequential pose estimation), if some of the queries in a segment have no correspondences. For those 3 known affected queries, investigation revealed that the chosen cutout (i.e. the top one in pose verification output) from the previous query was not even considered in the top 100 cutouts in the pose estimation step. The reason the previous query's

[1] This query is somewhat blurry, which may also have an impact.

chosen cutout was picked is that the queries have not changed much during the two frames. The score for those cutouts was:

Query ID	Ranking of the previous query's chosen cutout
88	714
148	138
231	518

Table 5.6: The ranking after sorting all cutouts for a given query by highest score. Only top 100 make it to the pose estimation step, others are not considered. This suggests that the scores are not completely correct.

5.4.3 Hard to pick top 10 combinations for non-trivial segments

Geometric verification step chooses top 10 cutouts for each query based on the highest number of inliers. The wrong ones would normally be filtered out by pose verification. However, if segments of length $k > 1$ are used, the top 10 combinations (representing the current segment) don't necessarily contain only the reasonable query-cutout pairs. This is because we are only choosing top 10 combinations from 10^k possible combinations[2]. There is no easy solution to this problem. Making pose estimation return significantly more than top 10 candidates for each query will have a performance impact, because the pose verification step is already time consuming.

5.4.4 No HoloLens poses

Due to the delay (see table 3.3), some of the queries by the end of the HoloLens1 and HoloLens2 sequences do not have a pose estimated from HoloLens. In such a case we have to resort to using standard P3P, which performs (on average) worse than MultiCameraPose. Hopefully the delay was only caused by the software extracting the data from HoloLens and the delay is not actually present in real use. If it is present, it is a problem as the techniques based on InLoc described in this paper would not work in real-time.

5.4.5 Geometric verification fails

Our approach produced a completely wrong pose estimate in Query 40 of the s10e query set. Upon debugging the issue, it turns out that there actually were several

[2]We are not talking about combinations in a mathematical sense, but rather as a way of expressing a number of possibilities.

viable cutouts, similar to the query image. One would expect the correspondences to be found there. However, the geometric verification step (GV) found zero correspondences. This issue shall be investigated further.

5.5 Computational complexity

The redesigned and improved pose estimation pipeline is a fairly complex piece of software. It is hard to compute the asymptotic complexity of processing a query image or a set of query images. The computational requirements are missing from the InLoc paper [1]. However, the authors mention the need for about 14 GB RAM in their experiment, to hold the image descriptors in memory.

Table 5.7 shows the processing times I have measured. They are not necessarily accurate, as sometimes, the experiment was re-run while keeping some previously computed data.

Experiment	Step	Processing time
s10e	GPU	11 min
s10e	CPU	1h 46 min
HoloLens1-k1	GPU	11 min
HoloLens1-k1	CPU	about 20 hours
HoloLens1-k5-MCP	GPU	10 min
HoloLens1-k5-MCP	CPU	about 48 hours
HoloLens1-k4-sequentialPV	GPU	5 min
HoloLens1-k4-sequentialPV	CPU	about 40 hours

Table 5.7: The experiment is split into two parts; the first part runs on a GPU (feature extraction) and the rest runs on a CPU. For the CPU instance, a machine with 45 threads and K8 2000 CPUs (or similar) was used. GPU instance used a single NVIDIA 1080Ti GPU with 8 threads. The processing time is a rough estimate.

The preprocessing step shall also be taken into account. It takes about 20 minutes to create the `score` matrix for an s10e query set. It is done by executing `buildFeatures.m` followed by `buildScores.m`. The same task takes 22 minutes on the HoloLens1 query set. Note that both tasks share the same number of cutouts that require processing by `buildFeatures.m`.

The recommended amount of RAM for re-running the experiments is 90 GB.

Chapter 6

Conclusion

I have created a new dataset suitable for indoor visual localization; either on single RGB images or on a sequence of query images and localization data from HoloLens. I have adjusted the original[1] InLoc implementation and made it work on the newly acquired dataset. The performance on the non-sequential s10e query set is very good (compared to results in InLoc paper). This is likely caused by the fact that our dataset is much smaller than the InLoc dataset. I have also implemented two novel methods that are based on InLoc [1] - the sequentialPV method and the MultiCameraPose method. It was expected that the sequentialPV method would not to perform very well compared to HoloLens tracking. The MultiCameraPose method is more accurate than both the baseline InLoc method and the sequentialPV method. The resulting estimated poses are usable. However its performance is still significantly below the precision of HoloLens tracking itself. It is not clear why the new MultiCameraPose method is not performing that well. In the previous chapter, I have described known sources of errors, a lot of which can be targeted in a future work. This will certainly improve the evaluation performance.

6.1 Future work

Improve the accuracy of the MultiCameraPose method by fixing known sources of errors. Spend extra time to analyze why there are inaccurate poses for certain queries and suggest an enhancement. The work on the HoloLens2 sequence should be continued - we need to compute reference poses. The code is there, but currently

[1] Original InLoc implementation available at [41].

we are missing more manually set-up 2D-3D correspondences. Also, the work on synthetic dataset generation using AI Habitat shall be continued. Although it cannot give us data from HoloLens tracking, it can be used to generate new indoor localization datasets, without the need for expensive equipment (such as a Matterport scanner). There are extra parameters such as setting up the cutout horizontal field of view, `dslevel`[2], `MultiCameraPose` software [13] parameters and more. The accuracy of the reference poses wrt ground truth poses shall be also improved.

The paper [22] provides an improvement of the InLoc pose verification step. Once a reference implementation is available, I recommend incorporating the changes into our implementation.

The cutouts shall be regenerated with horizontal field of view equal to 60°, to see if it improves performance. This hFoV would also match the InLoc cutouts' hFoV.

[2]Determines how much to downsample images in the pose verification step.

Appendix A

Bibliography

[1] Taira, H.; Okutomi, M.; et al. InLoc: Indoor Visual Localization with Dense Matching and View Synthesis. In *CVPR*, 2018.

[2] Arandjelović, R.; Gronat, P.; et al. NetVLAD: CNN architecture for weakly supervised place recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[3] Wijmans, E.; Furukawa, Y. Exploiting 2D Floorplan for Building-scale Panorama RGBD Alignment. In *Computer Vision and Pattern Recognition, CVPR*, 2017. Available from: `http://cvpr17.wijmans.xyz/CVPR2017-0111.pdf`

[4] Fischler, M.; Bolles, R. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. In *Readings in Computer Vision*, Morgan Kaufmann, 1987, ISBN 978-0-08-051581-6, pp. 726 – 740, doi:https://doi.org/10.1016/B978-0-08-051581-6.50070-2.

[5] Arandjelović, R.; Zisserman, A. Three things everyone should know to improve object retrieval. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 2911–2918.

[6] Liu, C.; Yuen, J.; et al. SIFT Flow: Dense Correspondence across Scenes and Its Applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 33, no. 5, 2011: pp. 978–994.

[7] Fukushima, K. Neocognitron: a self organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, volume 36, no. 4, 1980: p. 193—202, ISSN 0340-1200, doi:10.1007/bf00344251. Available from: `https://doi.org/10.1007/bf00344251`

[8] Schmidhuber, J. Deep learning in neural networks: An overview. *Neural Networks*, volume 61, 2015: pp. 85 – 117, ISSN 0893-6080, doi:https://doi.org/10.1016/j.neunet.2014.09.003.

[9] Simonyan, K.; Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*, 2015.

[10] Pajdla, T. *Elements of Geometry for Computer Vision*. 2020.

[11] Kukelova, Z.; Heller, J.; et al. Efficient Intersection of Three Quadrics and Applications in Computer Vision. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[12] Wald, J.; Sattler, T.; et al. Beyond Controlled Environments: 3D Camera Re-Localization in Changing Indoor Scenes. In *Proceedings IEEE European Conference on Computer Vision (ECCV)*, 2020.

[13] Sattler, T.; Lučivňák, P. MultiCameraPose. [online], 2020, [cit. 2020-08-14]. Available from: `https://github.com/lucivpav/MultiCameraPose`

[14] The MathWorks, Natick, MA, USA. MATLAB Statistics and Machine Learning Toolbox. [online], 2019, [cit. 2020-08-09]. Available from: `https://www.mathworks.com/help/releases/R2019b/stats/index.html`

[15] Merriaux, P.; Dupuis, Y.; et al. A Study of Vicon System Positioning Performance. *Sensors*, volume 17, 07 2017: p. 1591, doi:10.3390/s17071591.

[16] Milgram, P.; Kishino, F. A Taxonomy of Mixed Reality Visual Displays. *IEICE Trans. Information Systems*, volume vol. E77-D, no. 12, 12 1994: pp. 1321–1329.

[17] Hübner, P.; Clintworth, K.; et al. Evaluation of HoloLens Tracking and Depth Sensing for Indoor Mapping Applications. *Sensors*, volume 20, 02 2020: pp. 1021:1–23, doi:10.3390/s20041021.

[18] Zeller, M.; et al. HoloLens (1st gen) hardware. [online], 2020, [cit. 2020-08-09]. Available from: `https://docs.microsoft.com/en-us/hololens/hololens1-hardware`

[19] Smith, R. C.; Cheeseman, P. On the Representation and Estimation of Spatial Uncertainty. *The International Journal of Robotics Research*, volume 5, no. 4, 1986: pp. 56–68, doi:10.1177/027836498600500404.

[20] Smith, R.; Self, M.; et al. Estimating Uncertain Spatial Relationships in Robotics. 01 1986, pp. 435–461, doi:10.1109/ROBOT.1987.1087846.

[21] Leonard, J. J.; Durrant-Whyte, H. F. Simultaneous map building and localization for an autonomous mobile robot. In *Proceedings IROS '91:IEEE/RSJ International Workshop on Intelligent Robots and Systems '91*, 1991, pp. 1442–1447 vol.3.

[22] Taira, H.; Rocco, I.; et al. Is This the Right Place? Geometric-Semantic Pose Verification for Indoor Visual Localization. 10 2019, pp. 4372–4382, doi: 10.1109/ICCV.2019.00447.

[23] Garg, R.; B G, V. K.; et al. Unsupervised CNN for Single View Depth Estimation: Geometry to the Rescue. 03 2016, doi:10.1007/978-3-319-46484-8_45.

[24] Zhang, Y.; Funkhouser, T. Deep Depth Completion of a Single RGB-D Image. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[25] Hübner, P.; Weinmann, M.; et al. Marker-based localization of the Microsoft HoloLens in building models. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, volume 621, 2018: pp. 195–202.

[26] Frantz, T.; Jansen, B.; et al. Augmenting Microsoft's HoloLens with vuforia tracking for neuronavigation. *Healthcare Technology Letters*, volume 5, no. 5, 2018: pp. 221–225.

[27] Liu, Z.; Zhang, L.; et al. Fusion of Magnetic and Visual Sensors for Indoor Localization: Infrastructure-Free and More Effective. *IEEE Transactions on Multimedia*, volume 19, no. 4, 2017: pp. 874–888.

[28] Piciarelli, C. Visual Indoor Localization in Known Environments. *IEEE Signal Processing Letters*, volume 23, no. 10, 2016: pp. 1330–1334.

[29] Bay, H.; Ess, A.; et al. Speeded-Up Robust Features (SURF). *Computer Vision and Image Understanding*, volume 110, no. 3, 2008: pp. 346 – 359, ISSN 1077-3142, doi:https://doi.org/10.1016/j.cviu.2007.09.014, similarity Matching in Computer Vision and Multimedia.

[30] Xu, S.; Chou, W.; et al. A Robust Indoor Localization System Integrating Visual Localization Aided by CNN-Based Image Retrieval with Monte Carlo Localization. *Sensors*, volume 19, no. 2, Jan 2019: p. 249, ISSN 1424-8220, doi: 10.3390/s19020249. Available from: `http://dx.doi.org/10.3390/s19020249`

[31] Southworth, M. Calculating the Practical Field of View of the HoloLens. [online], 2018, [cit. 2020-08-09]. Available from: `https://www.linkedin.com/pulse/calculating-practical-field-view-hololens-michael-southworth/`

[32] Kendall, D. A Survey of the Statistical Theory of Shape. *Statistical Science*, volume 4, no. 2, 1989: pp. 87–99, ISSN 08834237.

[33] Zderadičková, A.; Schönberger, J.; et al. HoloLensDataAcquisition. [online], 2020, [cit. 2020-08-09]. Available from: `https://github.com/lucivpav/HoloLensDataAcquisition`

[34] Savva, M.; Kadian, A.; et al. Habitat: A Platform for Embodied AI Research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[35] Chang, A.; Dai, A.; et al. Matterport3D: Learning from RGB-D Data in Indoor Environments. *International Conference on 3D Vision (3DV)*, 2017.

[36] Lučivňák, P.; Steidl, S.; et al. Habitat. [online], 2020, [cit. 2020-08-10]. Available from: `https://github.com/lucivpav/Habitat`

[37] Lučivňák, P. VisualLocalizationWithHoloLens. [online], 2020, [cit. 2020-08-14]. Available from: `https://github.com/lucivpav/VisualLocalizationWithHoloLens`

[38] Zhang, Y.; Song, S.; et al. Panocontext: A whole-room 3d context model for panoramic scene understanding. 2014: pp. 668–686.

[39] Zhang, Y.; Song, S.; et al. PanoBasic: Toolbox for panorama image processing. [online], 2017, [cit. 2020-03-24]. Available from: `https://github.com/yindaz/PanoBasic`

[40] Grigonis, H. Understanding Lens Distortion in Photography (And How To Fix It). [online], [cit. 2020-08-13]. Available from: `https://expertphotography.com/what-is-lens-distortion`

[41] Taira, H.; et al. InLoc_demo. [online], 2017, [cit. 2019-10-05]. Available from: `https://github.com/HajimeTaira/InLoc_demo`